POLITECNICO DI TORINO

Master's Degree in
COMPUTER ENGINEERING

Master Thesis

ISA extensions in the Snitch Processor for Signal Processing

Developed at

ETH zürich

SUPERVISORS
Prof. Luca Benini
Prof. Alberto Macii

ADVISORS
Samuel Riedel
Matheus Cavalcante

CANDIDATE
Sergio Mazzola

Academic Year 2020/2021

Abstract

To tackle the large computational loads of multimedia applications, specialized platforms called image signal processor (ISP) have been developed to meet the demanding requirements of power- and timing-constrained platforms thanks to their highly-parallel architectures and domain-specific instructions. MemPool is a novel 32-bit many-core system with 256 Snitch cores sharing a L1 scratchpad memory pool through a low-latency interconnect. Snitch is a tiny RV32IMA core based on the RISC-V open instruction set architecture (ISA), paired with an application-tunable accelerator. In this work we present Xpulpimg, an extension of the RISC-V instruction set including domain-specific instructions for digital signal processing (DSP) carefully selected from the Xpulp custom extension to exploit the potentialities of the MemPool system as an ISP. In particular, the Xpulpimg extension introduces in Snitch new addressing modes for load and store instructions, single-instruction-multiple-data (SIMD) operations and additional arithmetic utilities for DSP. With the aim to fully support the Xpulpimg extension and make it compliant with the open, modular and extensible nature of the standard RISC-V ISA, we also propose a complete framework for opcode space management, ISA modeling and simulation, verification and compilation support. To evaluate the proposed extension we benchmarked the MemPool cluster in several configurations with DSP algorithms optimized for Xpulpimg, measuring a speed-up of up to 4.6× with respect to the initial baseline design. Post-synthesis figures, very much taken into account for the micro-architectural design exploration, have been obtained from the modern GlobalFoundries' 22FDX Fully-Depleted Silicon-Over-Insulator (FD-SOI) technology, and demonstrated an energy efficiency increased of 3.8× at the tile level.

Acknowledgments

Throughout the whole development of this project I have received significant support and assistance from my supervisors and the community of the Integrated Systems Laboratory at ETH Zürich.

I would first like to thank my supervisor, Professor Luca Benini, who guided my focus with his precious advice and very much encouraged my integration in the research group. I truly appreciate the experience to work at ETH Zürich that you made possible for me; I felt, as a student, very much valued and taken into consideration.

I would also like to thank my advisors, Samuel Riedel and Matheus Cavalcante, for their relentless support and meticulousness. I greatly enjoyed learning everything from you, from the small details of our project to the big picture of the research process.

Furthermore, I thank my supervisor Professor Alberto Macii, for his helpful support on the side of Politecnico di Torino.

Finally, I would like to thank all the brilliant and enthusiastic members of the PULP research group, who always made me feel involved in the group activities and part of a community. I would particularly like to single out Davide Schiavone and Florian Zaruba. Your support as gurus of the two processor cores I have mainly dealt with have been crucial for a deeper understanding of the whole project.

In addition, I would like to acknowledge the immeasurable support of my family and friends in my achievements and throughout the whole academic career that led me here. In particular, I thank my parents for unconditionally believing in me, constantly supporting and encouraging my choices. I also thank my lifelong friends Ignazio, Gianvito, Luca, Gianluca and Nicoló for sharing the highs and lows of this road, always with the same intensity, no matter the kilometres apart. Lastly, I thank Linda for being there with her heart and soul, enriching these last steps of mine with even greater meaning and purpose.

Contents

Chapter 1

Introduction

The last decades have seen a growing interest for data processing in power-constrained environments with strict timing requirements. The leading example of such a trend is represented by mobile devices, particularly striving for high performance in multimedia applications [1], with their main drivers being fields such as computer vision, augmented reality, computational photography.

Smartphones cameras feature image sensors with tens of millions of pixels, imposing huge image and video processing loads to be handled with tight power budgets, often with real-time requirements. However, the nature of such loads and their inherent degree of data parallelism can be leveraged to meet power and timing constraints. To this end, specialized platforms called image signal processors (ISPs) have increasingly gained attention, with their highly parallel architectures, often organized with domain-specific processing models, and domain-specific instructions.

A primarily important point of highly parallel architectures is the memory sharing among the processing units, usually done at L1-cache level for better performance, power efficiency and programmability [2]. Several architectures for L1-shared clusters have been developed. However, they either do not scale beyond tens of cores [3] or solve memory sharing with a deep memory hierarchy [4], leading to a significant degradation in power consumption and a difficult programming model. Specialized architectures that achieve both are dedicated to a specific family of algorithms due to their over-restrictive interconnect, like systolic arrays [5][6].

MemPool is a 32-bit many-core system that scales up to 256 cores [7]. The cores in the cluster share a large pool of scratchpad memory (SPM) as L1 cache through a low-latency, hierarchical interconnect. Despite its general-purpose architecture and the high core count, the MemPool cluster reaches very competitive performance and efficiency with respect to state of the art designs. Its smallest unit of repetition is the MemPool core complex (CC), featuring a fast RV32IMA Snitch core. *Snitch* is a in-order, single-issue processor core based on the RISC-V open instruction set architecture (ISA), optimized for area, efficiency and flexibility [8]. It is paired with an application-tunable accelerator, whose pipeline is fully decoupled from the core, making the system very modular and

extensible. It supports outstanding transactions and out-of-order write-back with very little control area overhead, inexpensively reaching high instruction per cycle (IPC) ratios.

In this work we present *Xpulpimg*, an extension of the RISC-V instruction set including domain-specific instructions for digital signal processing (DSP). Many DSP instructions are particularly useful for image processing: by combining signal processing functionalities with Snitch, with its high performance and extreme replication capabilities due to its small control area overhead, the potential of the MemPool system as an ISP can be exploited, leveraging its already high level of parallelism. We carefully selected the introduced DSP instructions from the *Xpulp* custom RISC-V instruction set extension for DSP [9], based on their impact on software of interest, particularly concerned with image processing. We also kept MemPool CC post-synthesis figures into consideration during the register-transfer level (RTL) implementation, as feedback from the technology for the micro-architectural design exploration.

In particular, the Xpulpimg extension introduces in Snitch new addressing modes for load and store memory accesses, single-instruction-multiple-data (SIMD) operations for 16-bit and 8-bit sub-words and additional arithmetical instructions generally useful for DSP purposes, such as multiply-accumulate, clips, comparisons.

Due to the open and modular nature of the standard RISC-V ISA [10], a main concern of our project has been to keep the whole environment as modular and extensible as possible, also granting *full support* for the new extension. In doing so, we propose a framework for opcode space management, ISA modeling and simulation, verification and compilation support.

To evaluate the proposed extension we benchmarked the MemPool cluster in several configurations with DSP algorithms that we specifically optimized for Xpulpimg, measuring a speed-up of up to 4.6× with respect to the initial design. We synthesized the MemPool CC for the modern GlobalFoundries' 22FDX Fully-Depleted Silicon-Over-Insulator (FD-SOI) technology, aiming to obtain a Pareto-optimal design in terms of area and frequency. In typical conditions, we measured a maximum operating frequency degradation of 3.6%. At the MemPool target frequency of 500 MHz, the extended core complex synthesized in the worst-case corner accounts for 59.7 kGE, for an area increase of 18 kGE. The Xpulpimg implementation is able to reach an energy efficiency 3.8× higher than the baseline.

To sum up, the main contributions of this project are:

- the definition of a new DSP-oriented instruction set extension, Xpulpimg, with particular focus on image processing, originated from the study and analysis of the Xpulp specification and reference implementation, CV32E40P;
- the development of a full support for the Xpulpimg extension starting from basic tools of the standard RISC-V ecosystem, including verification, implementation and compilation tools;
- the implementation of the Xpulpimg extension in the Snitch processor, extending its philosophy for small control area overhead and high performance to the field

of signal processing, to exploit MemPool's general-purpose parallelism in the ISP domain;

- the evaluation of the post-synthesis results in terms of performance, area and power for the new design with respect to the initial baseline in the context of MemPool in an advanced GlobalFoundries 22FDX FD-SOI technology.

The remainder of this report is organized as in the following. Chapter 2 poses the foundation of the whole project, illustrating the state of the art of ISP and the already existing tools on which we based Xpulpimg and its environment. Chapter 3 goes in depth in the description of the Xpulpimg framework we developed, highlighting our contribution with respect the state of the art; this chapter can also be considered as a reference guide for further extension of the Xpulpimg extension. Chapter 4 is dedicated to the hardware design step of the workflow we established and describes, item by item, the architecture the implemented Snitch extensions. Finally, Chapter 5 details the evaluation methodology we employed, in terms of both software benchmarking and synthesis results collection, along with the results that we obtained. Chapter 6 draws the final conclusions of the project paving the way for further work. Additionally, Appendix A documents the whole Xpulpimg ISA extension, listing all of its instructions along with their behaviors.

Chapter 2

Background and Related Work

The architectures, the tools and the design methodology developed in this project have a strong foundation in the state of the art, which we extended with our contribution.

This chapter gives an overview of the background of our project, consisting of the tools and the platforms that we employed as baseline for our contributions. Related works about DSP and ISP are also pointed out, to put our work in perspective and evaluate it.

2.1. RISC-V open ISA

Up until few years ago, all popular commercial ISAs were proprietary. This tendency still persists in the mostly dominating commercial domains of electronics, such as personal computer, mobile devices, workstations. However, an increasing number of companies, organizations and research groups have been embracing the philosophy of an open ISA, particularly in the form of the *RISC-V ISA*.

ISAs embody the very core of hardware-software interface, certainly the most important interface in a computer system. There are no actual technical reason to keep such an interface proprietary [11]: innovation is obstructed, proprietary ISAs are complex and difficult to implement, simpler ISA subsets are not encouraged, microprocessors cost increases [12].

RISC-V ISA is a completely free and open instruction set architecture born in 2010 as an educational project at the University of California, Berkeley. In the following years, it has been developed to overcome all the formal and technical limitations of commercially available proprietary ISAs, to obtain an open and modular ISA, fully customizable in terms of extensions and implementation. With these characteristics, RISC-V is thought not only to be suitable for nearly any computing device, but also to revolutionize the market and the research [12].

The RISC-V ISA standard describes a reduced instruction set computer (RISC) load-store architecture. There are three base ISAs: RV32I, RV32E and RV64I [10]. The RV32I is the base 32-bit integer ISA: it has 31 general-purpose integer registers, named `x1-x31`

(`x0` is used to specify the constant zero), and 47 instructions among system instructions, computational operations, control flow and memory accesses. The RV32E is a variant of RV32I with fewer registers, thought for embedded devices, while the RV64I ISA is its 64-bit variation.

The basic RISC-V ISA includes the following six types of instructions, different in the kind of encoding and inputs:

- *R-type* – register-register;
- *I-type* – short immediates and loads;
- *S-type* – stores;
- *B-type* – conditional branches, a variation of S-type;
- *U-type* – long immediates;
- *J-type* – unconditional jumps, a variation of U-type.

In order to make the ISA suitable for both low-end and high-performance devices, the RISC-V provides flexibility in the form of *extensions*. Several extensions are already ratified, while others are just drafted or in development. Examples of common standard extensions are: **M**, for integer multiplication and division, **A**, for atomic instructions, **F** and **D**, respectively for single-precision and double-precision floating-point operations.

A standard specification has also been defined for privileged execution modes, in particular describing a *machine level* (with the highest privileges) and a *supervisor level* [13].

A main concern of the RISC-V project has also been to provide *support* for the designed ISA and all of its extensions, to encourage the free and open development of implementations and customizations. The following sections describe the main tools of the RISC-V ecosystem.

2.1.1. Instructions encoding management

The RISC-V ISA has many extensions and can also feature custom instructions. The tool `riscv-opcodes` is useful to enumerate all of them in a human-readable form, listing the operands that they use and the encoding of their opcodes. Additionally, standard control and status registers (CSRs) are enumerated [14].

The tool is also used to convert the high-level description of the instructions encoding into several formats (e.g. C header, SystemVerilog package, LaTeX documentation). As a matter of fact, `riscv-opcodes` is not meant to stand alone; rather than that, it is integrated with other tools in the RISC-V ecosystem and is able to provide them with the encoding of all the implemented instructions in the format that they need. While doing this, the tool also checks for the consistency of the employed instructions opcode space looking for encoding overlapping.

The output files generated by the tool mainly contain the description of RISC-V CSRs and the information to recognize instructions, namely the masks of their opcodes and the

related values to be matched. As an example, Spike, the RISC-V ISA simulator described in Section 2.1.3, needs a description of the encoding for the implemented instructions as a C header, to be able to read a binary file, match its instructions and execute the related C++ behavioral model. The `riscv-opcodes` tool automatically installs its C header output in Spike, providing it with such a description.

2.1.2. GNU compiler toolchain

To offer full support for the defined standard ISA, RISC-V also provides its user with a C and C++ cross-compiler [15]. The components of the toolchain, including the GCC compiler and the GDB assembler and disassembler, can be finely instructed to decide which ones of the RISC-V extensions to enable to generate the target binary. In this way, the flexibility of the ISA is reflected on the software, which can be easily adapted to any platform.

2.1.3. Spike simulator

Spike is the RISC-V ISA simulator. It implements the functional model of one or more RISC-V cores [16] and provides a C++ golden reference for software simulation. It supports multiple ISAs extensions and the user, supervisor and machine privilege levels.

Spike serves as a starting point for running software on a RISC-V target, but it is also useful for the development and test of new custom instructions, due to its high extensibility.

2.1.4. Unit tests suite

A suite of ISA-level unit tests for RISC-V ISA is made available with the `riscv-tests` tool [17]. The tool defines an environment to automatically compile the tests for a RISC-V target with the RISC-V GNU toolchain and simulate them with Spike.

The test suite is composed of functional tests comparing the output of the tested instructions at given inputs with a known gold standard. With the aim to maximize the reuse of each test, the test programs of the `riscv-tests` suite are constrained to only use features of a given test virtual machine (TVM); TVMs hide differences among different test implementations and easily allow the same test program to be compiled and run on different target environments.

Each test program for the RISC-V ISA is written within a single assembly language file, which is passed through the C preprocessor. To start and end the execution of the test, to determine its success or failure and to define the data section, the macros provided by the TVM are used. The definition of the TVM macros is given within the environment selected for the compilation.

In addition to the set-up of the TVM, each unit test contains a series of test cases for the instruction under test. The `riscv-tests` tool also includes a set of macros, based on the type of the instruction, to easily generate test cases by only providing the input

data and the expected output. The test cases macros in `riscv-opcodes` contain self-checking assembly code to test the instructions under test; self-checks rely on the correct functioning of the processor instructions used to implement the self checks themselves (e.g. branches), so a complete verification of those is needed in the first place.

2.2. Open-source DSP extensions

Due to the increasingly widespread employment of the RISC-V ISA also in embedded and Internet of things (IoT) contexts, there has been a large interest for further extension oriented towards DSP. This section sums up the main contribution in this direction.

2.2.1. RISC-V P draft extension

During the last RISC-V workshops, an interest developed for packed-SIMD fixed-point operations for use in the integer registers of small RISC-V implementations [10].

Packed-SIMD, also known as sub-word parallelism, is an approach to sub-word computation (i.e. data smaller than the size of a word, such as 8-bit bytes or 16-bit half-words). It allows to perform vectorial operations in a scalar environment with specific datapath extensions, considering scalar registers as arrays of sub-words [18].

At date, the official RISC-V packed-SIMD extension is only a draft [19], with its definition being still in a preliminary state and with an incomplete support.

2.2.2. Xpulp custom DSP extension

Along with the official extensions, the RISC-V standard offers some encoding space for custom extension of the ISA. The *Xpulp* instruction set is a DSP-oriented RISC-V ISA extension [20]. It has been developed with the aim to achieve similar performance and code density to the state-of-the-art microcontroller units (MCUs) based on a proprietary ISA, specifically targeting the field of IoT applications, with sensors data processing and near-threshold (NT) parallel operation [9].

Its most recent version, the Xpulp instruction set includes the following extensions:

- extended addressing modes for post-increment load and store instructions, including register offset and post-increment of the base address (the base address is always contained in a register);
- hardware loops;
- bit manipulation operations;
- general arithmetic logic unit (ALU) operations;
- multiply-accumulate (MAC) operations;
- fixed-point instructions for addition, subtraction, multiplication, MAC;

- 8-bit and 16-bit packed-SIMD extension, with three addressing modes: vector-vector with registers, vector-scalar with registers and vector-scalar with an immedaite (generic arithmetic operations, dot-product, comparisons, manipulation operations and complex numbers operations).

The Xpulp extension is complete of C and C++ cross-compiler support [21]. The PULP GCC compiler has been derived from the original GCC RISC-V version, while the binutils have been extended to support the additional instruction set [9].

Xpulp also comes with a reference implementation in the CV32E40P core, previously known as *RI5CY* [9][20][22]. CV32E40P is a small and efficient 32-bit RISC-V processor. Its 4-stage pipeline implements the RV32IMC instruction set architecture, along with the optional Xpulp and single-precision floating-point extensions.

The CV32E40P core has been designed for ultra-low-power signal processing applications, targeting NT operation to achieve higher power efficiency while recovering performance through parallelism. In particular, CV32E40P is employed in tightly coupled multi-core clusters but also in single-core microcontrollers such as PULPissimo [23].

2.3. Snitch processor core

The main object of our research is the *Snitch* core [8][24], which implements the RISC-V concepts described so far but lacks the domain-specific DSP capabilities we long for. Snitch is a general-purpose, single-stage, single-issue, in-order RISC-V core tuned for simplicity, energy efficiency and minimal area footprint. Snitch is highly configurable and can be paired, as a tiny control core, with an application-tunable accelerator to off-load RISC-V instructions, which makes the system very modular and extensible.

With its coprocessor, Snitch supports the RV32G (i.e. RV32IMAFD) ISA, with the possibility of fine-tuning the proposed extensions. The Snitch core itself mandatorily implements the basic RV32I instruction set, or its RV32E smaller variation.

Snitch has a dedicated instruction fetch port, a data port with a valid-then-ready decoupled request and response path and a generic accelerator offloading interface. The accelerator interface, supporting an AXI-like handshake, has two independent decouple channels: one for offloading an operation with up to three operands and a response channel for the write-back of the result. The interface can offload an entire 32-bit instruction, with the accelerator supporting the same RISC-V instructions encoding. The block diagram of the described Snitch baseline is depicted in Fig. 2.1.

Snitch supports outstanding loads, which are useful in the context of shared-memory many-core systems, and the offloading of multi-cycle instructions to the coprocessor. The results of load operations and multi-cycle instructions can possibly be retired out-of-order. Such functionalities are supported by a simple scoreboarding mechanism which keeps track of each one of the 31 integer registers with a single busy bit. Snitch is thus able to reach high IPC ratios with only a small control area overhead, avoiding expensive hardware for queuing, reordering or register renaming.

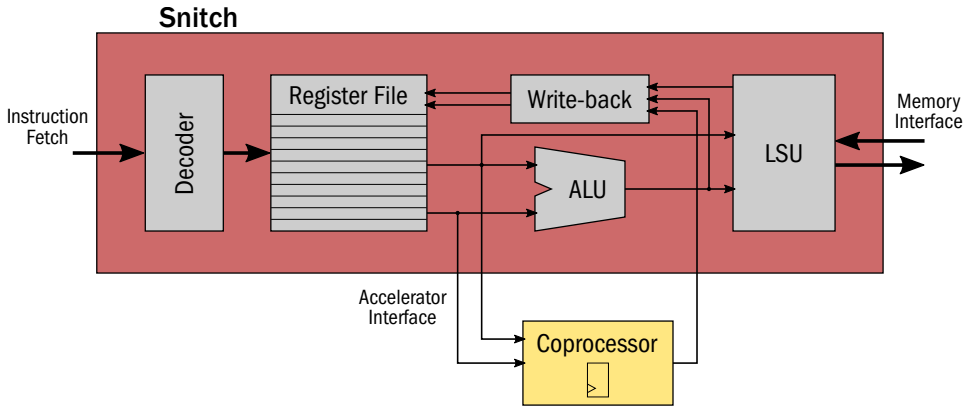


Figure 2.1.: Block diagram of the Snitch baseline; thinner arrows stand for individual data transfers, while thicker arrows also include the flow of control signals.

2.4. ISPs and MemPool

MemPool is a 32-bit many-core system with 256 cores [7]. Each processing element, named *MemPool core complex (CC)*, contains an RV32IMA Snitch core paired with an accelerator customized to perform integer multiplications and divisions. The tiny 21 kGE area of this Snitch implementation allows for massive replication within MemPool, while obtaining anyway high performance due to its architecture.

A common architectural pattern for building highly parallel systems suitable for multimedia applications is having a cluster of simple cores sharing a L1 memory through a low-latency interconnect [2]. It is generally thought that the core count of a L1-shared cluster is limited within the low-tens of units, as in the *streaming multiprocessors* of Nvidia Ampere GPUs, with 32 floating-point cores sharing 192 KiB of L1 memory [3]. To achieve a core count above the hundreds, memory sharing is usually achieved with deep memory hierarchy, for example sharing the main memory among clusters with a private address space, as it is the case for Kalray's MPPA-256 many-core system [4]. This generally compromises not only efficiency, but also programmability of the system.

Many-core architectures achieving high-core count, efficiency and good programmability usually result limited in terms of applicability, being extremely optimized for a family of algorithms by their over-restrictive interconnect. As an example, Google's Pixel Visual Core [6] is an ISP featuring specialized cores connected into a ring network; within each core, an array of 256 lanes communicate through a rigid read-neighbour network, which is highly efficient for systolic algorithms but reduces the architecture applicability to them.

MemPool overcomes these problems scaling up its core count into the hundreds, with its cluster sharing a common view of a large multi-banked pool of SPM through a low-latency, hierarchical interconnect.

The MemPool cluster is composed of four local groups, as in Fig. 2.2; local groups communicate with each other by means of 16×16 radix-4 butterfly networks. Each local

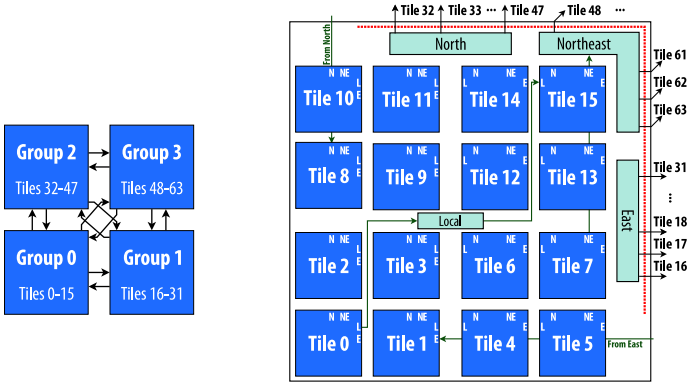


Figure 2.2.: On the left, the MemPool cluster, divided in 4 groups; on the right, a detailed view of the first local group, from [7]. Dashed lines represent register boundaries.

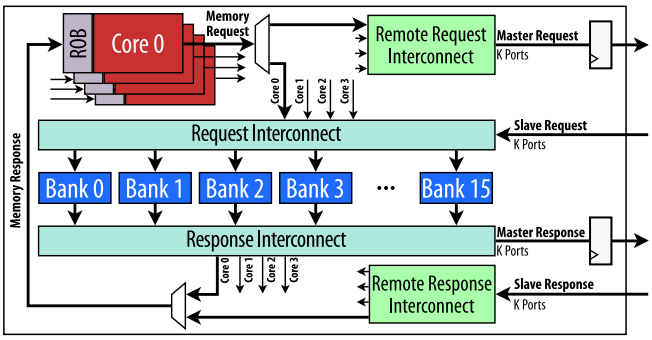


Figure 2.3.: Architecture of a MemPool tile with K request ports and K response ports, from [7].

group is composed of 16 tiles, communicating within their group with an additional 16×16 radix-4 butterfly network.

As in Fig. 2.3, each tile features four MemPool CCs, 16 banks of L1 tightly-coupled-data-memory (TCDM) and a 4-way L1 instruction cache. Each core has a dedicated port to access the TCDM banks within its tile in one cycle. An address decoder statically decides where to route the memory requests of the cores. Each tile has K master ports to access remote tiles and K slave request ports receiving memory request from remote cores. A register boundary cuts the master request and response paths.

With such an interconnect, cores can access any remote memory bank in the same local group with a zero-load access latency of 3 cycles. On the other hand, the latency for memory accesses to banks in remote groups is 5 cycles due to the register boundary on local groups' master interface.

In addition to the interconnect architecture, MemPool also employs an hybrid addressing scheme able to exploit the sequentially interleaved memory mapping across all memory banks to minimize banking conflicts, but also adding sequential regions in

which contiguous address target a single tile, not to waste the locality of private data.

The MemPool interconnect architecture, along with its hybrid addressing scheme for memory accesses, gives the system high efficiency and performance with respect to the state of the art while enabling an extremely high level of parallelism, without any degradation of its general-purpose capabilities and high programmability.

2.5. Image processing algorithms

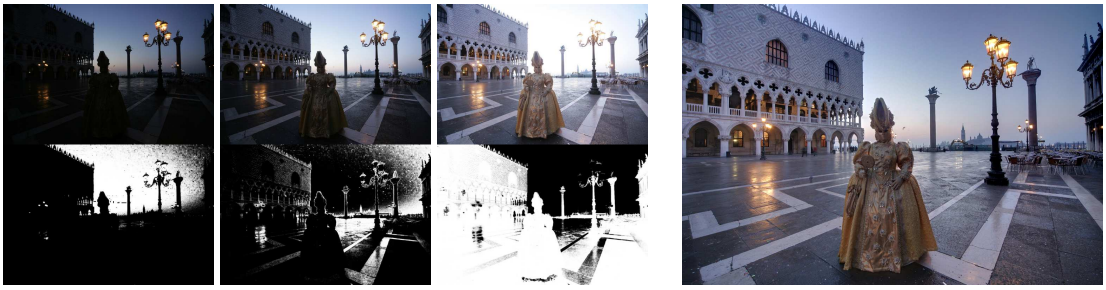
Image signal processors are gaining increasing focus mainly to address the problem of huge timing-constrained computational loads in low-power mobile devices, as it is the case for multimedia applications such as image processing, video processing, augmented reality, artificial vision [1].

A main field of interest whose presence is dominant in today's digital world is *computational photography* [25]. Computational photography refers to the techniques used to enhance and extend the capabilities of digital photography; it is thus naturally coupled with the increasingly powerful sensors of modern mobile devices. In this context, a very common algorithm implemented in every smartphone is *high dynamic range (HDR)* [26]. HDR imaging is used to increase the dynamic range of an image, collecting an higher quantity of details from several different exposures of the same scene [27].

Several implementations of the HDR pipeline area available in the literature [27][28][29], but most of them usually encompass the same steps:

- *alignment* – align the images with the different exposures such that the scenes correctly overlap: this step is needed to avoid artifacts in the merge of the exposures, such as blurring and ghosting due to motion;
- *camera response function (CRF) computation* – before processing the pixels of the different exposures in order to merge them, an estimation of the CRF is needed; indeed, the response of the camera sensor to the brightness of the scene is not linear, thus pixels values must be first adjusted by estimating the CRF;
- *merging* – knowing the exposure time of all the input exposures and the CRF, the pixels of each exposure can be merged in a single image filtering out too bright or too dark pixels, obtaining an output with an higher dynamic range featuring details from all the exposures; for higher precision and wider ranges, often floating-point or fixed-point arithmetic are employed;
- *tone mapping* – merge algorithms work with data types bigger than the usual 24-bit RGB format to collect the most out of the set of input images; hence, their output has to be mapped back to an 8-bit range for each channel to view it on usual displays.

An example implementation of the HDR pipeline is given in the OpenCV library [30]. In particular, OpenCV implements two algorithms for HDR imaging with exposures



(a) Input images with corresponding weight maps (b) Fused result

Figure 2.4.: Example of exposure fusion from [32]; the weight maps of each input exposure is computed by means of the quality figures and employed to fuse them in the final result.

sequences (Debevec merge [27] and Robertson merge [31]), and another lighter approach called *exposure fusion* [32][33].

This last approach blends multiple exposures, without any need of CRF estimation or tone mapping, by using simple quality measures such as saturation and contrast to determine the goodness of the pixels of each exposure, and thus their weight in the merge. An example is given in Fig. 2.4. This method actually circumvents the computation of the HDR image, obtaining an output which can be directly displayed in the common 8-bit format. The exposure fusion implementation heavily relies on the convolution of input images with given kernels: to extract one of the quality indexes employed by the algorithm, a Laplacian filter is applied on the exposures, corresponding to a convolution with a 3×3 kernel; Gaussian pyramids are also very much employed for the fusion step, hence resulting in a large number of convolutions with 5×5 kernels.

Chapter 3

Design Methodology

This chapter presents the design methodology which guided the development of the whole project; we established a series of precise steps, which we followed throughout every iteration of the design in order to grant the desired characteristics to the final implementation: compliance with RISC-V standards and tools, modularity, flexibility, quality of support and documentation.

Such properties are not limited to the standardization of the hardware design phase; on the contrary, they mainly concern the whole environment in which the project has been designed, developed, tested and simulated. We established such an environment and extended every one of its components, while simultaneously introducing new ISA extensions in Xpulpimg. In doing this, we also provided full support for our newly defined Xpulpimg extension, in terms of opcode space management, ISA modeling and simulation, verification and software compilation.

At every iteration, consisting of a new set of instructions selected to be added to our custom extension, we complied to the following steps, also sketched in Fig. 3.1:

1. *selection of new instructions* based on the speed-up of relevant kernels;
2. generation of the *encoding* of the selected instructions, dependency of all the other tools [14];
3. extension of the custom Xpulpimg *ISA subset* in the PULP RISC-V GNU toolchain [21] with the new instructions;
4. *behavioral implementation* of the instructions in the Spike RISC-V ISA simulator [16];
5. extension of the RISC-V *test suite for verification* [17] with the unit tests for the new instructions;
6. *verification* of the Spike behavioral implementation with unit tests;
7. refinement of the behavioral implementation to a *RTL implementation* in Snitch;

8. *verification* of the Snitch RTL implementation with unit tests;
9. *synthesis* of the design with the current extension and area-timing (AT) figures feedback to RTL implementation.

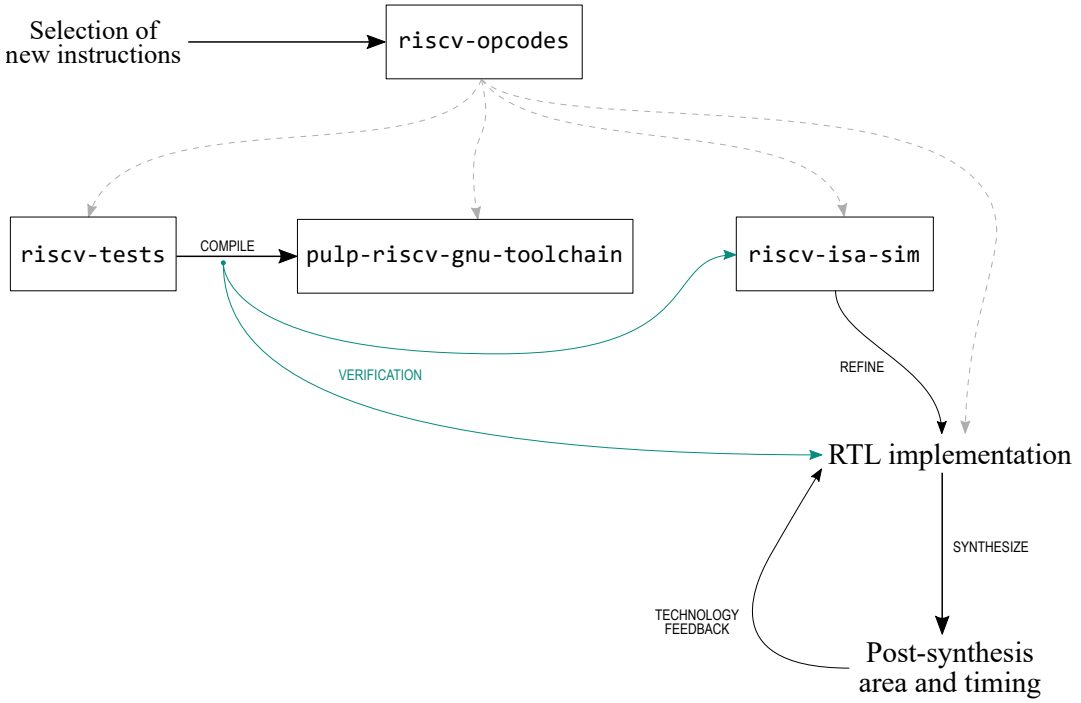


Figure 3.1.: Flow of the design methodology that we established through the customization of the RISC-V environment; the labels in a `monospaced` font represent the actual tools from RISC-V that we extended with Xpulpimg.

Each individual step is detailed in the following sections.

Note that the purpose of this chapter goes beyond the description of the employed design methodology. As a matter of fact, it can be considered a step-by-step guide to extending the framework established for the scope of this project for any further development of the Xpulpimg instruction set extension.

3.1. Selection of suitable instructions

To allow the highest degree of modularity, extensibility and compatibility with other extensions, we defined a stand-alone new instruction set extension, named *Xpulpimg*. We then used the defined instruction set extension to gather the instructions of interest for our purpose of extending the MemPool system with DSP functionalities, mainly focusing on image processing.

The starting point for selecting instructions to add to the Xpulpimg set has been the Xpulp custom instruction set extension, in particular its `Xpulpv2` version [20][9].

However, the Xpulpimg set is not to be considered a strict subset of `Xpulpv2`; even if, in its current version, it only encompasses Xpulp instructions, it is rather meant to be a custom DSP image-processing-oriented instruction set: the instructions of interest taken from Xpulp are a starting point of general usefulness for DSP, which will be extended in the future with additional custom instructions suitable for its purpose, most likely also outside the scope of Xpulp.

At each iteration of the design phase, we selected new useful instructions to introduce in Xpulpimg by inspecting the `Xpulpv2` documentation. Then we determined candidate instructions based on a preliminary analysis of their impact on relevant kernels of interest for DSP and image processing, introducing in Xpulpimg the ones promising the largest improvements in terms of throughput and code density. Other factors taken into account also encompassed the general usefulness of the candidate instructions for signal processing, including the reduction of registers utilization and of control flow instructions.

Practical examples of this design step are given in Chapter 4, when the instructions of Xpulpimg are presented along with their hardware architecture in the MemPool CC.

Overall, Xpulpimg accounts for 173 new instructions, mainly grouped in the following sections.

Generic arithmetic operations

Instructions of general usefulness for DSP to reduce control flow instructions and increase code density; they include comparisons, absolute value, clip operations for partial fixed-point support and immediate branching.

Extended L/S addressing modes

Extended addressing modes for load and store instructions, including register-register mode (offset coming from a register) and post-increment mode (auto-increment of the base address after the memory access).

MAC operations

Multiply-accumulate operations able to multiply the content of two registers and perform either an accumulation on the destination register or a subtraction from it.

Packed-SIMD extension

Packed-SIMD instructions for 16-bit and 8-bit sub-words, with vectorial mode and scalar replication mode, both with immediate and register operand; this extension includes dot-product operations, addition, subtraction, comparison, shifts, logicals and support instructions for packing and unpacking SIMD data.

The listed instruction sets also define the granularity and the order with which the extensions have been introduced in Xpulpimg and implemented in the Snitch core, with their architecture described in Chapter 4 and their implementation cost and performance

gain analyzed in Chapter 5. A complete list of the Xpulpimg instructions and their specification is given in Appendix A.

3.2. Instruction encoding generation

The first step to port the selected instructions to the Xpulpimg environment is to include a human-readable description of their encoding in the `riscv-opcodes` tool [14]. Its purpose is to establish a single source of truth for the instruction mnemonics and opcodes, generating the instructions encoding files needed by all the other dependencies in the framework in such a way that they are always aligned.

We developed a custom extension of the baseline `riscv-opcodes` tool which adheres to the way the official RISC-V tool manages the automatic instructions encoding generation and their integration in all the tools employed for the design and development phases. While a trial was already been made to integrate the tool in the Parallel Ultra-Low Power (PULP) environment, it results to be outdated and not very much maintained.

Several tools in our environment are dependent on `riscv-opcodes` outputs; the generated output types and the dependent tools are listed in the following:

- *C format* – C header containing the macros declaring the mask of each instruction opcode and its actual value, associated with the instruction mnemonic; such declarations are employed by:
  - the Spike simulator, to read compiled binaries and simulate the behavior of the corresponding instruction, and its disassembler;
  - the `riscv-tests` unit tests suite, in particular to access CSRs;
  - the software compilation framework for the MemPool system, in particular to access CSRs;
  - the assembler of the RISC-V GNU toolchain, in a manually-modified version, adapted to the specific features of the Xpulp extension;
- *SystemVerilog format* – SystemVerilog package containing the parameters defining the mask and the matching value of each instruction opcode, used in the hardware description of MemPool CC's decoders.

When new encodings are generated, all the mentioned dependencies are automatically updated.

For further flexibility, we also extended the tool with the possibility to select which extensions have to be considered for the generation of the encodings, and which ones have instead to be excluded. This functionality can be used to manage opcode spaces overlapping problems of custom extensions. Another way to address such an issue, at a much finer grain, is to make the overlapping instructions pseudo-instructions, making them aliases of the instructions with the same opcode.

Our main contribution to this tools consists, apart from its adaptation to our environment, in its extension to include several Xpulp instructions under the Xpulpimg

instruction set. To add new instructions to `riscv-opcodes`, their mnemonics have to be added to the textual file related to the extension of belonging. Then, a list of the instruction operands has to be associated to the mnemonic, along with the list of values of its opcode bit-fields. An example is given below.

```
p.clip   rd rs1 imm5   31..25=10 14..12=1 6..2=0x0C 1..0=3
```

Several of the instructions introduced in Xpulpimg from Xpulp are not compliant with standard RISC-V instruction types. For this reason, we extended the parser script, in particular to introduce three new possible operands:

- `prs3` – alongside `rs1` and `rs2`, it represent a third source operands (the `p` in `prs3` stands for PULP); it actually has the same encoding of `rd`, but we gave it a different name due to its different semantical meaning of source register rather than destination;
- `imm5` – 5-bit immediate of Xpulp instructions;
- `imm6` – 6-bit immediate, particularly employed by Xpulp SIMD instructions.

Note that the operand fields considered by the parser script have no actual numerical value; they are instead only relevant for their bit positions in the instruction encoding. For this reason, the signedness of the immediate fields is irrelevant at this step of the design.

3.3. GNU toolchain custom subset

The Xpulpimg instruction set is supported by the GNU toolchain, in terms of both compiler and assembler. As Xpulpimg started out as a subset of Xpulp, the PULP GNU toolchain has been used as a baseline for our custom extension.

We integrated Xpulpimg in the toolchain as an additional Xpulp version, due to it including many of the Xpulp instructions. This means that, when it is enabled, no other Xpulp versions can be enabled for compilation or assembly.

As far as the *GCC compiler* is concerned, we extended intrinsics and code generation support from Xpulp to the Xpulpimg instruction set. From the *GNU assembler* side, we defined the complete list of instructions to include in Xpulpimg, with their opcode masks and values to match, whose definitions get generated by the `riscv-opcodes` tool.

3.4. Spike simulator extension

With the environment aware of the encoding of the new instructions, the following step is to study their behavior and obtain a first, high-level implementation. With that, we aimed to study and correctly understand the instructions, their edge cases and finally verify their functional correctness. The official RISC-V tool `riscv-isa-sim` [16], Spike, has been employed to implement this phase.

Being based on C++, Spike empowered us to model instructions behavior at the highest possible level. Not only is this the best way to first approach and start to understand the behavior of an instruction and its interaction with the other components of the processor, but it also exposes most of the edge cases of its functionalities, hiding details unnecessary at this step. Spike also allowed us to perform simulations of new instructions from the very first stages of their design, which is optimal for verification purposes as it allowed to grant the functional correctness of the defined behaviors and a certain degree of edge cases covering directly from the beginning.

Apart from the C++ behavioral description of Xpulpimg instructions, Spike required additional modifications to support Xpulpimg:

- the decoding logic of the simulator has been extended to include the new fields of the Xpulpimg instructions encoding:
  - the third input register index `rs3`;
  - the 5-bit signed and unsigned immediates;
  - the 6-bit signed and unsigned immediates;
- several macros reflecting the original Spike implementation have been added to ease the description of Xpulpimg instructions behavior:
  - sign-extension and zero-extension of 8-bit and 16-bit data to the full data width of the architecture, for the extension instructions and SIMD extension;
  - read access to the new `rs3` source register, for the register-register stores;
  - write access to the `rs1` register, for the post-increment memory instructions;
  - subword-level (half-word and byte) read and write accesses for register operands, for the SIMD extension;
- the Spike disassembler, useful for debug purposes, has been extended to support and correctly interpret the new instructions;
- some instructions of the vector extension have been disabled both in the simulator and in the disassembler due to their overlapping with the `Xpulpv2` SIMD extension: the vector extension is deeply integrated in Spike, so that a quick and modular deactivation was not possible; as a result, such a custom Spike version cannot be correctly used for the simulation of a core extended with the official RISC-V vector instruction set.

The behaviors of the implemented instructions are presented in Appendix A.

3.5. Unit tests verification

Verification has been a main concern in the design methodology that we adopted, which enabled us to grant a high quality for both the behavioral modeling of Xpulpimg instructions and the RTL code implementing them in the processor core.

The unit test suite extension is tightly coupled to the previous step of behavioral Spike implementation: not only Spike extensions undergo a row of functional verification with the developed tests, but both steps support each other in the understanding of the correct behavior of the instructions and the study of their edge cases.

In an effort to be compliant with RISC-V verification framework and to keep the environment as flexible and modular as possible, we employed the already existing `riscv-tests` [17] as a verification tool extending it for our purposes, which were mainly:

- extend the unit tests suite with the tests for the instructions introduced in Xpulpimg;
- adapt the test framework to automatically run the unit tests on the Spike behavioral model and on the MemPool RTL implementation.

To integrate Xpulpimg unit tests in the test suite, a new TVM has been defined for the 32-bit user-level Xpulpimg instruction set. As the other unit tests in the test suite, each instruction has its own assembly file containing the individual test cases and the macros for the startup and the conclusion of the test, along with its data section.

Xpulpimg contains instructions of different types, in terms of operands use, with respect to the ones in the standard RISC-V ISA. For the sake of standardization, we developed new test case macros for each new type of instruction introduced in our custom set, as it was done in the standard tool; in particular, the new types of test cases we introduced are for:

- instructions with `rs1` and unsigned 5-bit immediate inputs, `rd` output – generic arithmetic instructions with immediate;
- instructions with `rs1` and unsigned 6-bit immediate inputs, `rd` output – generic arithmetic SIMD instructions with immediate;
- instructions with `rs1` and signed 6-bit immediate inputs, `rd` output – generic arithmetic SIMD instructions with immediate;
- instructions with only `rs1` input and `rd` output – scalar and SIMD absolute value, subword extract instructions;
- instruction with `rs1`, `rs2` and `rd` inputs, `rd` output – MAC, SIMD dot-product with accumulation, SIMD shuffle instructions;
- instruction with `rs1` and unsigned 6-bit immediate inputs, `rd` output – immediate SIMD instructions;
- instruction with `rs1` and signed 6-bit immediate inputs, `rd` output – immediate SIMD instructions;
- loads and stores with register offset;
- post-increment loads and stores with immediate offset;
- post-increment loads and stores with register offset;

- immediate branching instructions.

With the test case's structures defined, we then instantiated the actual test cases for each instruction by providing each macro with some input values, a corresponding expected value and a test ID, to return in the end-of-computation (EOC) register in case of test failure. The test cases' data has been mainly obtained from the already existing test cases developed for the PULPissimo platform [34], consisting of random input values whose corresponding output has been computed by a golden model written in Python. In addition to those, several other tests have been carefully developed for each instruction to cover the edge cases of their functional behaviors. Overall, 68 functional tests have been developed to test all of the Xpulpimg instructions, each targeting a specific one. In particular, they account for a total of 1428 individual test cases.

Due to the unit tests employing startup and EOC macros, the test cases of an instruction are independent from the platform running it, which is instead described by the definition of those macros. Thanks to this feature of the `riscv-tests` environment, we have been able to neatly integrate such a tool in an automatic framework to compile and run the tests on the Spike behavioral model and on the MemPool RTL implementation, with the aim to perform the verification of the design at these two different phases of the development. Note that, for the RTL simulation, only a single core of the system is active and runs the unit test, which is enough to grant the functional correctness of the Snitch implementation.

For the compilation of the tests, we used the custom PULP GCC toolchain extended with our Xpulpimg instruction set. For their RTL simulation on the 16-core MemPool, we employed Mentor Questa Sim 2019.3.

3.6. Snitch RTL implementation

The central phase of each design iteration is the implementation of the new set of candidate instructions in the RTL model of the MemPool CC, comprising a Snitch core and its integer processing unit (IPU). At this point, all the tools in the environment are aware of the new instructions mnemonics and encoding, and the new instruction set extension is fully implemented in the compiler; additionally, we also have a deep understanding of the instructions behaviors and of their corner cases, due to the prior development of the functional test suite and the implementation of the instructions in Spike.

The knowledge developed up to this point is thus employed to refine the Spike behavioral model of the instructions to an RTL implementation. SystemVerilog hardware description language (HDL) is used for this purpose. In particular, the SystemVerilog package containing the encodings of the instructions generated by `riscv-opcodes` is employed for the extension of the decoding logic of the processor.

The hardware architecture of the Xpulpimg instructions implemented in MemPool CC is presented in Chapter 4.

As a final step of the RTL implementation, the design undergoes a further row of verification, consisting of the RTL simulation of the unit tests from `riscv-tests` com-

piled with the extended PULP RISC-V GNU toolchain. At each design iteration, every instruction of the ISA implemented in Snitch is tested, to check for the correctness of the new extension and whether it perturbed the implementation of other instructions.

3.7. Synthesis

Synthesis has been a step of main relevance for the finalization of each design iteration. In this phase, we estimated the cost, in terms of area increase and maximum operating frequency decrease, of the speed-up promised for the analyzed kernels by the new instructions.

To this end, after the refinement of the MemPool CC RTL model from the Spike behavioral model and its verification with the developed unit tests, we synthesized it as an individual module fully enclosed in a register boundary. In doing so, our aim has been to collect precise area estimates with their increment only due to each new ISA extension, without the noise from caches and TCDM interconnect synthesis. We synthesized our designs for the GlobalFoundries' 22FDX FD-SOI technology using Synopsys Design Compiler 2019.12.

We also employed the post-synthesis results as a feedback from the employed GF22 technology, in a tight loop with the RTL design step, with the aim to explore the design space and achieve the architecture for the most efficient implementation in terms of area and timing, fine-tuning parallelism and hardware sharing.

Further details about the synthesis methodology are given in Section 5.1.2.

Chapter 4

Hardware Architecture

The central contribution of our work is the implementation of the Xpulpimg instruction set, gathering the main image processing functionalities from the Xpulp DSP extension, in the MemPool system. The processing element object of our work is the MemPool core complex (CC), multiply-instantiated and hierarchically-interconnected to form the MemPool system. Each MemPool CC is composed of:

- a Snitch integer core;
- a coprocessor for the Snitch core, the Snitch integer processing unit (IPU);
- spill registers to cut the request and the response paths between the core and the IPU.

Snitch is a tiny integer core able to achieve high IPC performance, thanks to its support for outstanding transactions, out-of-order write-back and interface to a generic application-tunable accelerator, with a small control area overhead, determined by the simple scoreboarding mechanism that it implements For these reasons, Snitch is extremely suitable for massive replication inside an highly-parallel cluster like MemPool. Our implementation of the Xpulpimg ISA extension in the Snitch processor aims to extend its philosophy for small control area overhead and high performance to the field of signal processing, to exploit MemPool's general-purpose parallelism in the ISP domain.

In this chapter, we describe the hardware architecture of our ISA extension in the MemPool CC, determined by the implementation of Xpulpimg. The chapter is organized hierarchically reflecting the organization of MemPool CC modules and sub-modules. For each module implementing a given extension, the instruction set of the extension is also described, pointing out the reasons bringing to its introduction and assembly examples of how it affects the DSP kernels of interest for our analysis.

Starting from the baseline Snitch architecture implementing the RV32IMA ISA, depicted in Fig. 2.1, we extended the core in the decoding, in the integer register file and in some functionalities of the control path to support the Xpulpimg extension. A

detailed analysis of how the implementation of each extension affected the cost of the implementation and the achievable performance is available in Chapter 5.

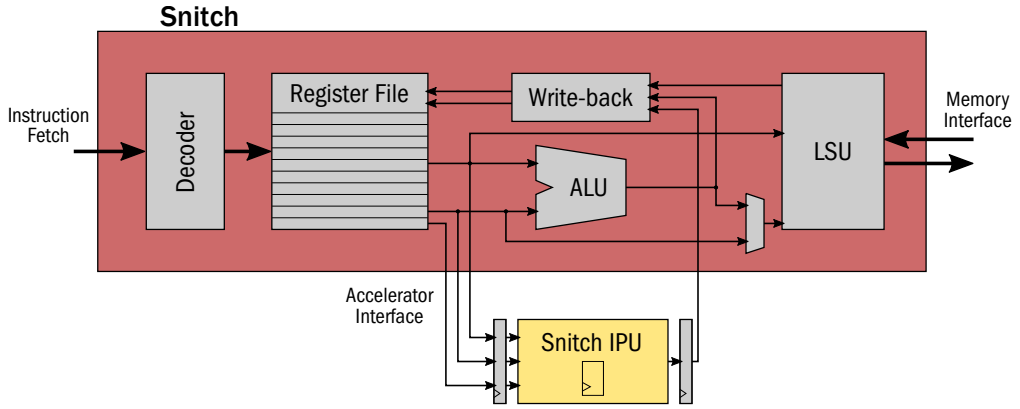


Figure 4.1.: Block diagram of the MemPool CC extended with Xpulpimg, focused on the differences with respect to the baseline of Fig. 2.1; the register boundary around the IPU is also highlighted, even if already present in the MemPool CC baseline.

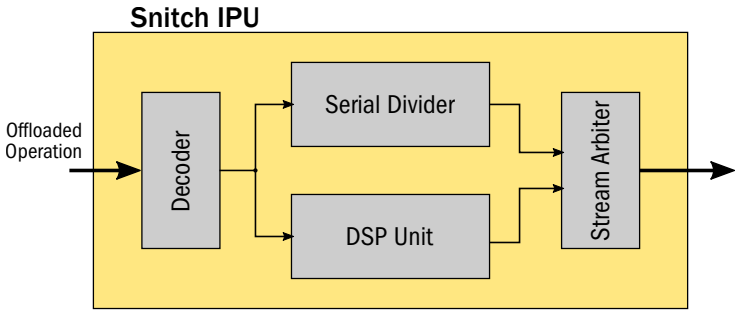


Figure 4.2.: Block diagram of the Snitch IPU coprocessor extended with Xpulpimg functionalities.

As far as the Snitch IPU is concerned, it has been inspired by the coprocessor used in the baseline Snitch CC implementing the RV32IMA ISA, previously employed to exclusively accelerate integer multiplications and divisions. With Xpulpimg, we instantiated a new sub-module in the Snitch IPU, a DSP unit, handling all the extensions introduced from Xpulp.

The block diagram of the MemPool CC architecture extended with Xpulpimg is shown in Fig. 4.1; the architecture of the IPU is detailed in Fig. 4.2.

4.1. Snitch architecture extension

The extension of the Snitch integer core to support Xpulpimg did not substantially modify its overall architecture, whose main characteristics remained the ones discussed in Section 2.3. For the sake of modularity and extensibility, we parameterized every modification related to the Xpulpimg extension, so that the original RV32IMA version of the core can be obtained by changing the related parameter in the core instantiation. With the Xpulpimg extension disabled, any issued Xpulpimg instruction causes an illegal instruction exception to be raised.

The architectural features which have been impacted the most by our modifications to the core are:

- the extension of the decoding logic to support Xpulpimg instructions (either executed inside Snitch or offloaded to the IPU);
- the modification of the input operands port on the accelerator interface;
- the number of read ports of the integer register file, brought from 2 to 3;
- the extension of the scoreboarding and stall control logic to additionally consider `rd` as a source register;
- the extension of the load/store mechanism to introduce the possibility of post-incrementing the base address with a register or an immediate;
- the extension of the scoreboarding, stall and write-back control logic to additionally consider `rs1` as a destination register;
- the decoding and sign-extension of a new immediate field for immediate branching instruction;
- the extension of the branching mechanism to also include immediate branching.

For the implementation of the Xpulpimg functionalities, we complied with the main idea behind Snitch: Snitch is a fast and small core, meant to efficiently carry out control operations with its single-stage, single-issue, in-order design. For this reason, we actually implemented most of the new instructions inside its coprocessor, the Snitch IPU, with the modifications introduced in the core only having a minor impact in terms of employed resources.

The instructions actually implemented inside the Snitch core are the plain and post-increment loads and stores, both with register and immediate offsets, and the immediate branching instructions. Their architecture is described in the following.

4.1.1. Post-increment and register-register loads and stores

The standard RISC-V instruction set only supports one addressing mode for memory access instructions, where the actual address is computed by adding an offset coming from the sign-extension of an immediate to a base address stored in a register [10].

Extended load and store instructions	
`p.l{b[u],h[u],w} rd, {iimm12s,rs2}(rs1)`	Load a value from address (`rs1` + {`iimm12s,rs2`}) into `rd`
`p.l{b[u],h[u],w} rd, {iimm12s,rs2}(rs1!)`	Load a value from address `rs1` into `rd`, then increment `rs1` by {`iimm12s,rs2`}
`p.s{b,h,w} rs2, {simm12s,rd}(rs1)`	Store the value from `rs2` at address (`rs1` + {`simm12s,rd`})
`p.s{b,h,w} rs2, {simm12s,rd}(rs1!)`	Store the value from `rs2` at address `rs1`, then increment `rs1` by {`simm12s,rd`}

Table 4.1.: All the extended load and stores instructions introduced in Xpulpimg with Extended L/S addressing modes. Note that `b`, `h` and `w` represent the data length of the memory access, standing respectively for byte, half-word and word. Load operations of sub-words can either be signed or unsigned. The immediates employed are `iimm12s`, the standard 12-bit signed I-type immediate, and `simm12s`, the standard 12-bit signed S-type immediate.

Load and store instructions introduced in Xpulpimg from `Xpulpv2` have two *additional addressing modes* with which:

- the actual memory address is obtained from the sum of an offset and a base address, both stored in registers;
- the actual memory address corresponds to the base address stored in a register; after the access, it is incremented with an offset stored in a register or sign-extended from an immediate (*post-increment* mode).

All the mentioned memory access instructions are available for word (32 bits), half-word (16 bits) and byte (8 bit) data lengths. The extended load and store instructions are summed up in Table 4.1; note that standard RISC-V loads and stores with immediate offset are also aliased by Xpulpimg for the sake of completeness.

The extended loads and stores are useful for every kind of data processing algorithm, in particular when regular memory access patterns occur, with a speed-up of up to 20% [9]. An example of how these extension is useful and why it has been selected is presented in Listing 4.1 and Listing 4.2: post-increment instructions can perform the increment of array pointers along with the memory access, not only reducing the code size, but also actually issuing two operations in the same cycle; in this way, less cycles are needed per useful computation, increasing the throughput of the operation. Note that thanks to the register-register addressing mode, the post-increment is possible even when the array elements have an offset which is unknown at compile time.

The architectural modifications to implement the extended load and store instructions are shown in Fig. 4.3; specifically, we describe them in the following.

Listing 4.1: Example of kernel with an element-wise multiplication operation on two arrays; the hot-loop is 8-instruction long and performs one multiplication per iteration, with a throughput of 0.125 multiplications/instruction.

```
<loop>:
    lw a6, 0(a5)      ; load element from A
    lw t3, 0(a7)      ; load element from B
    addi a5, a5, 4    ; increment A pointer
    mul a6, a6, t3    ; compute C = A * B
    add a7, a7, t5    ; increment B pointer
    sw a6, 0(a4)      ; store result in C
    addi a4, a4, 4    ; increment C pointer
    bne t4, a5, <loop>
```

Listing 4.2: Kernel of Listing 4.1 optimized with the extended loads and stores; saving the increments of the load and store addresses (3 instructions less), the throughput rises to 0.2 multiplications/instruction.

```
<loop>:
    p.lw a6, 4(a5!)   ; load element from A, increment pointer
    p.lw t3, t5(a7!)  ; load element from B, increment pointer
    mul a6, a6, t3    ; compute C = A * B
    p.sw a6, 4(a4!)   ; store result in C, increment pointer
    bne t4, a5, <loop>
```

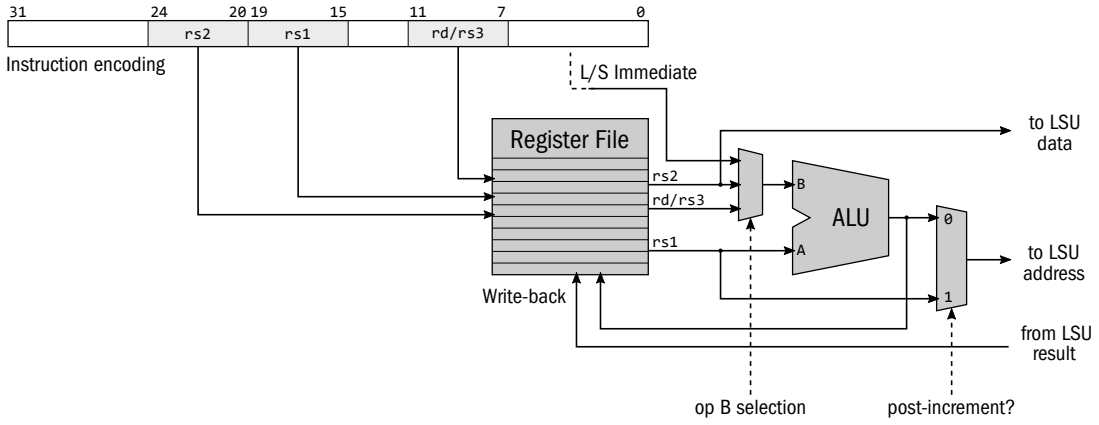


Figure 4.3.: Block diagram of the modifications we introduced to manage the third operand for the extended load and store operations and the post-increment mechanism.

Post-increment destination register

When post-increment loads and stores are performed, either with register or immediate offset, `rs1` is employed both as a source register, containing the base address, and as an additional destination register storing the incremented base address.

Due to this additional behavior, we extended the scoreboarding logic to flag the entry corresponding to `rs1` as busy each time a post-increment instruction is issued. In this way, read-after-write (RAW) hazards which might be generated by following instructions employing the same register as a source can be detected.

We also extended the write-back logic of Snitch to consider `rs1` as a possible destination. The write-back of `rs1` by means of a post-increment instruction always happens in the same clock cycle during which the instruction itself has been issued (unless a stall occurs), differently from what happens with load memory accesses. For this reason, the retirement of a post-increment on `rs1` and the retirement of a basic RISC-V integer operation executed inside the core are mutually exclusive: thanks to this, the write-back port dedicated to retirement of I instructions on `rd` can be muxed to also retire post-increments on `rs1`. With this approach, we managed to keep just two write ports for the register file.

Third source operand

The introduced register-register store operations need three input sources:

- `rs1` for the base address;
- `rs2` for the data to store in memory;
- `rs3`, which has the same encoding of `rd`, for the register offset.

Due to this need, we extended the integer register file to have three read ports; the first two read ports are dedicated to the instruction fields `rs1` and `rs2`, while the third one is indexed by the `rd` field, which semantically corresponds to `rs3`. The size of register files is proportional to the number of read and write ports, which makes this modification the most impacting in terms of area and critical path within the Snitch core, as we outline in Chapter 5.

When `rd` is used as a source register, its potential RAW hazards have to be considered before accessing it, so that we also extended the scoreboarding logic to tackle such issue.

In Snitch internals, the content of `rs2` is usually assigned to operand `B` of the ALU; when a register-register store is decoded, a crossing of signals is performed to assign `rd` to operand `B` instead, and use `rs2` as data source for the store. Due to this crossing, an additional extension of the control logic for `rs2` hazard detection has been needed.

Post-increment mechanism

For standard load and store instructions, the base address contained in the register `rs1` is assigned to the operand `A` of the ALU, with operand `B` being the load immediate or

the store immediate. The output of the ALU, which performs the addition of the two, is then used as effective memory address.

With the register-register addressing mode, the operand `B` can also be assigned with the content of `rs2` (for register-register loads) or `rd` (for register-register stores), to be summed up with the content of `rs1`.

As far as the post-increment mechanism is concerned, we extended the way the address for the memory access fed to the load-store unit (LSU) is managed: instead of hardwiring the address input to the output of the ALU, the actual address is muxed between the ALU output and the content of `rs1`, direct output of the register file. In this way, when post-increment instructions are issued, the base address can be directly used for the memory access, while the output of the ALU is written back to `rs1` as a post-increment operation. When basic memory access operations are instead issued, the output of the ALU, containing the sum of base address and offset, is selected as effective address.

4.1.2. Immediate branching

RISC-V standard ISA only support branch instructions comparing two registers [10]. With Xpulpimg, we introduce *immediate branching* instructions, summarized in Table 4.2: these operations compare the content of a register with an immediate to decide whether to take the branch. The 12-bit B-type immediate encodes the signed PC-relative offset in multiples of 2 bytes, with a branch range of ± 4 KiB.

Immediate branching instructions	
`p.beqimm rs1, pimm5s, bimm12s`	If `rs1` is equal to `pbimm5s`, branch to PC + (`bimm12s` << 1)
`p.bneimm rs1, pimm5s, bimm12s`	If `rs1` is not equal to `pbimm5s`, branch to PC + (`bimm12s` << 1)

Table 4.2.: Branching instruction with immediate comparison introduced in Xpulpimg with Generic arithmetic operations. The register `rs1` is compared against `pimm5s`, the 5-bit sign-extended immediate introduced from Xpulp; the program counter (PC) offset is instead given by `bimm12s`, the standard 12-bit signed B-type immediate.

Immediate branching instructions are useful when the value of the second operand in the comparison is already known at compile time, thus can be directly encoded into an immediate. This might be the case when the number of iterations of a loop is known and fixed (e.g., a convolution with a kernel of fixed size), or when the final value of a counter in a loop is already known, or also when we have an *if* statement with a constant comparison.

In the mentioned scenarios, immediate branches may help reducing the pressure on the integer register file since one register less is used: in case of functionalities with an intensive registers use, stack accesses might result reduced.

The architectural modifications needed to implement immediate branching instructions in Snitch impacted the core very lightly. The standard branching mechanism already present for standard RISC-V I branches is actually the same of Xpulp immediate branches; the only step affected by the new instructions is the comparison part.

As a matter of fact, we introduced the decoding of a new immediate field, the Xpulp 5-bit immediate. Its encoding is the same of `rs2`, and it gets sign-extended before being fed to the Snitch ALU, which compares it with the content of `rs1`. The output of the ALU, which is 1 or 0 based on the outcome of the comparison, is then used to decide whether to use the branch target or (PC + 4) as next program counter.

4.1.3. Instructions offloaded to the IPU

The implementation of the instructions on the control path is fully contained inside the Snitch core. On the contrary, data processing instructions, which are the remaining instructions in the Xpulpimg extension, are executed by the IPU coprocessor, with Snitch only dealing with their offloading to the accelerator interface.

Decoding logic

The decoding logic of data processing Xpulpimg instructions is divided into four parts, each one fine-tuning the set-up of the scoreboarding logic, based on the kind of register operands of the instructions:

- instructions with only a source register (`rs1`) and a destination register (`rd`);
- instructions with two source registers (`rs1` and `rs2`) and a destination register (`rd`);
- instructions with two source register (`rs1` and `rd`) and a destination register (`rd`);
- instructions with three source register (`rs1`, `rs2` and `rd`) and a destination register (`rd`).

Apart from the scoreboarding control logic, in the decoding phase the accelerator interface is also prepared for the AXI-like handshake: the accelerator is checked to be ready, the valid signal is raised when the instruction is valid and not stalling, the operands are fetched and fed to the operands ports, along with the whole encoding of the instruction.

Third source operand

Many Xpulpimg instructions (namely MAC operations, insert operations, SIMD sum dot-product instructions and SIMD shuffles) need to read the content of `rd` along with the other input registers or immediates.

For this reason, those instructions benefit from a third read port indexed by `rd` for the integer register file in the Snitch core, already added for memory accesses with register-register addressing mode. We extended the scoreboarding logic accordingly

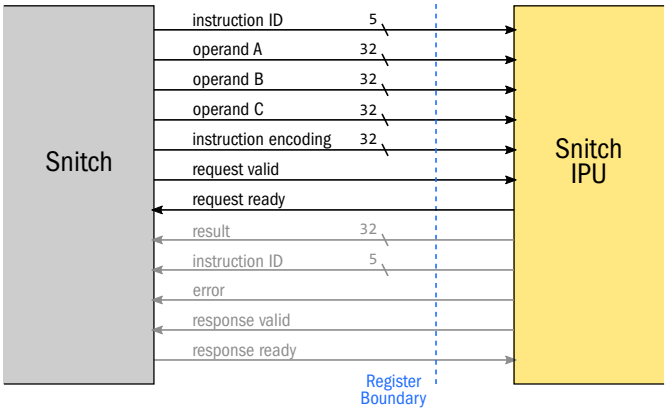


Figure 4.4.: Interface between the Snitch core and its IPU coprocessor.

to consider `rd` as a proper third source register, for RAW hazards detection. Note that `rd` may be used by the same instruction as both input and output register, without any problem from an architectural point of view.

Finally, since three potential registers are now in need to be sent to the coprocessor, we modified the operands fed to the accelerator interface by also offloading the content of `rd`.

The interface between Snitch and the Snitch IPU is depicted in Fig. 4.4.

4.2. Snitch IPU architecture

The architecture of the Snitch IPU stems from the previous version of the accelerator paired with Snitch in the MemPool CC. All the modifications to the coprocessor are parameterized in the same way of the core: if Xpulpimg is disabled, the baseline RV32M accelerator will be instantiated, instead of our custom IPU.

The coprocessor interface with the Snitch core, shown in Fig. 4.4, employs an AXI-like handshake. When an offloading request happens, the three operands are fed to the accelerator interface, along with the full 32-bit instruction, useful to decode opcodes and immediate operands. The ID of the instruction, corresponding to its destination register index, is also sent to the coprocessor: this is useful for the retiring of offloaded instructions. On the response port, the data result and the ID of the instruction are sent back to the core.

As in Fig. 4.2, the IPU is mainly composed of a decoder, two execution units and a stream arbiter. The decoder receives all the instructions offloaded to the IPU and exclusively decide to which one of the two execution units to offload them.

The execution units are a serial divider, which was already present in the original accelerator, and the newly developed digital signal processing unit (DSPU), executing the Xpulpimg instructions. The previously present integer multiplier has been embedded in the DSPU for higher hardware sharing.

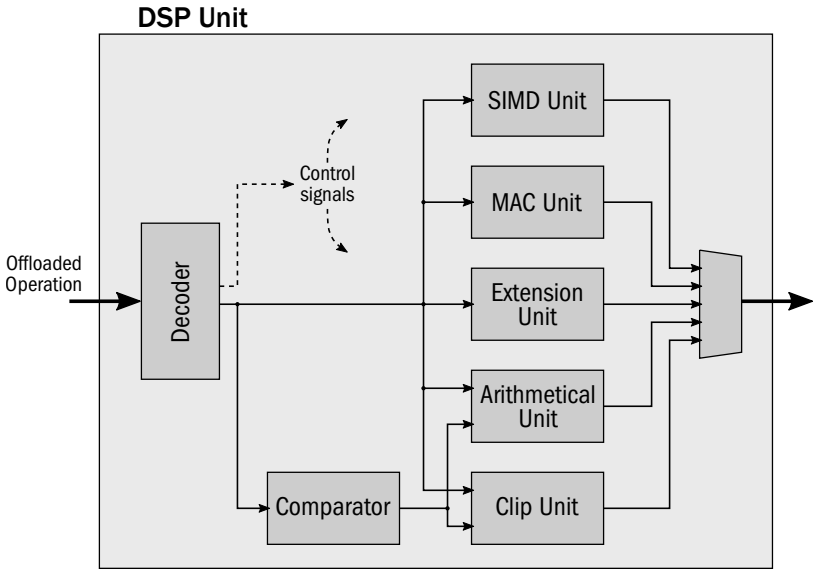


Figure 4.5.: Block diagram of the DSPU.

The DSPU is a single-stage execution unit containing a datapath organized as follows: a decoder reads the opcode of the offloaded instruction and generates an intermediate representation of the instruction to drive the internal units of the DSPU. Such an approach has been employed to obtain a better hardware sharing and a more flexible and extensible architecture in our custom unit, also allowing for some degree of power gating of the unused resources. The block diagram of the DSPU is shown in Fig. 4.5.

The main execution units of the DSPU are:

- a *shared comparator*, used by arithmetic instructions and clip operations;
- an *arithmetic unit*, performing some basic arithmetic operations such as absolute value and comparisons, based on the shared comparator;
- a *clip unit*, performing all the clip operations;
- an *extension unit*, which deals with sign- and zero-extensions;
- a *multiplier*, which computes all the M instruction set multiplications and, additionally, Xpulpimg MAC operations;
- a *SIMD unit*, which handles all the SIMD operations.

We describe the architecture of each unit and the datapath flow for the execution of their related instructions in the following sections.

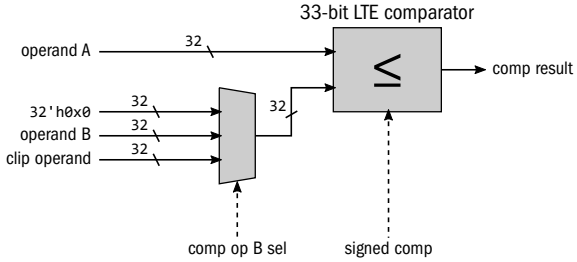


Figure 4.6.: Block diagram of the shared comparator in the DSPU.

4.2.1. Shared comparator

The DSPU is provided with a 33-bit *comparator* able to perform both signed and unsigned operations. The comparator is explicitly instantiated and shared among the functionalities of the arithmetic unit and the clip operations.

As in the block diagram of Fig. 4.6, the first operand of the comparator is always operand `A` of the IPU (i.e. the content of `rs1` from Snitch), while the second operand can be, based on the control signals generated by the decoder:

- zero, for the absolute value operation;
- operand `B` of the IPU (i.e. the content of `rs2` from Snitch), for the remaining arithmetic operations;
- an operand generated by the clip unit, used by clip operations.

The comparator is hardwired to perform a *less than or equal* operation between its two operands; for *greater than* comparisons, its negated output can be used. The comparator can perform both signed and unsigned comparisons on 32-bit data; the additional bit in the most significant position is needed to decide the signedness of the operation at runtime, by means of a control signal.

Generic arithmetic instructions	
`p.abs rd, rs1`	Compute the absolute value of `rs1`
`p.slet[u] rd, rs1, rs2`	If `rs1` is less or equal than `rs2` set `rd` to 1, otherwise to 0
`p.min[u] rd, rs1, rs2`	Store in `rd` the minimum between `rs1` and `rs2`
`p.max[u] rd, rs1, rs2`	Store in `rd` the maximum between `rs1` and `rs2`

Table 4.3.: Generic 32-bit arithmetic operations introduced in Xpulpimg with Generic arithmetic operations.

4.2.2. Arithmetic operations

Xpulpimg introduces some arithmetic operations generally useful for DSP. The ones discussed in this section are mainly based on the use of the shared comparator and are

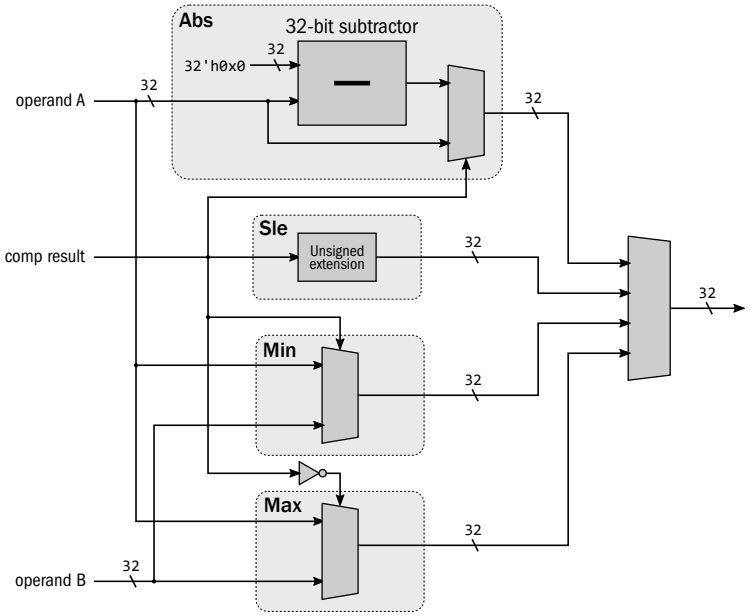


Figure 4.7.: Block diagram of the arithmetic unit of the DSPU.

listed in Table 4.3.

Such instructions are useful to reduce the amount of instructions needed to perform frequent operations employed in DSP, thus reducing the code size and the number of runtime cycles. In addition to this, minimum and maximum computations require control flow instructions: with Xpulpimg, the same result can be achieved without any alteration of the program flow, thus obtaining higher IPC values. An example of improvement due to this extension is given in Listing 4.3 and Listing 4.4.

The architecture of this unit, sketched in Fig. 4.7, is very simple and written in compliance with Synopsys Design Compiler guidelines to maximize hardware sharing: all the operations are described in the same *case* statement and depend on the same signals (i.e. the result from the comparator and the IPU input operands `A` and `B`).

Listing 4.3: Example assembly code, compiled for the baseline RV32IMA instruction set, to compute the absolute value of a variable and the minimum between two variables.

```
<start>:
    ; compute a0 = abs(a5)
    srai a0, a5, 0x1f
    xor a5, a0, a5
    sub a0, a5, a0
    ; compute a1 = min(a2, a1)
    ble a2, a1, <skip>
    mv a1, a2
<skip>:
    ...
```

Listing 4.4: Algorithms of Listing 4.3 for absolute value and minimum computation compiled with Xpulpimg extension; note that only one instruction is now needed for the absolute value, while program flow alterations are now absent for the minimum computation.

```
<start>:
    ; compute a0 = abs(a5)
    p.abs a0, a5
    ; compute a1 = min(a2, a1)
    p.min a1, a1, a2
```

4.2.3. Clip unit

The current version of Xpulpimg does not contain any dedicated fixed-point extension. Complete fixed-point functionalities can be anyway obtained with a combination of any instruction able to perform the following steps: integer arithmetic operation between the inputs, rounding, normalization (i.e. a shift to adjust the result to the correct Q format), saturation.

Fixed-point arithmetic has many applications in advanced image processing functionalities, as mentioned in Section 2.5. Since fixed-point numbers usually represent real quantities, it is often useful to check if a fixed-point result falls in a given range and saturate it to a minimum or a maximum bound otherwise. This operation is quite expensive in terms of clock cycles and program flow alteration; for this reason, we implemented clip operations in our Xpulpimg extension to, at least partially, support fixed-point functionalities. The clip operations from Xpulp support any Q-number format and allows to round and normalize the value before saturating, which provides higher precision [9].

The list of introduced clip operations is presented in Table 4.4. An example of how clips improve the code performance and the program flow is given in Listing 4.5 and Listing 4.6.

Clip instructions	
`p.clip rd, rs1, pimm5u`	Clamp `rs1` between $-2^{\texttt{pimm5u-1}}$ and $2^{\texttt{pimm5u-1}} - 1$
`p.clipu rd, rs1, pimm5u`	Clamp `rs1` between 0 and $2^{\texttt{pimm5u-1}} - 1$
`p.clipr rd, rs1, rs2`	Clamp `rs1` between $-$`rs2` $- 1$ and `rs2`
`p.clipur rd, rs1, rs2`	Clamp `rs1` between 0 and `rs2`

Table 4.4.: Immediate and register-register 32-bit clip operations from Generic arithmetic operations; the immediate employed by immediate clips is `pimm5u`, the 5-bit zero-extended Xpulp immediate.

Listing 4.5: Example of compiler-generated code from [9] where two arrays containing n Q1.11 signed elements are added together; the results if normalized between -1 and 1 and returned in the same Q1.11 format.

```
<start>:
    addi t5, x0, 0x800          ; load lower bound
    addi t4, x0, 0x7ff          ; load upper bound
    addi a3, x0, n              ; load elements number
<loop>:
    p.lh a4, 2(s0!)
    p.lh a5, 2(s1!)
    add a4, a4, a5              ; Q1.11 sum
    blt a4, t5, <lower_bound>   ; check for lower bound
    blt t4, a4, <upper_bound>   ; check for upper bound
    j <endL>
<lower_bound>:
    mv a4, t5                   ; saturate to lower bound
    j <endL>
<upper_bound>:
    mv a4, t4                   ; saturate to upper bound
<endL>:
    addi a3, a3, -1             ; update counter
    sw a4, 2(s2!)
    bne a3, x0, <loop>
```

Listing 4.6: Code from Listing 4.5 optimized with the support for clips; note that not only is the size of the hot-loop smaller, which increases the throughput, but also control flow instructions are not present anymore, achieving better IPC.

```
<start>:
    addi a3, x0, n      ; load elements number
<loop>:
    p.lh a4, 2(s0!)
    p.lh a5, 2(s1!)
    add a4, a4, a5      ; Q1.11 sum
    p.clip a4, a4, 12   ; clamp between -1 and 1 in Q1.11
    addi a3, a3, -1     ; update counter
    sw a4, 2(s2!)
    bne a3, x0, <loop>
```

We developed a clip unit to maximize the hardware sharing among the possible four clip operations. To generate the two bounds for the clamp range of immediate clips, we instantiated a small barrel shifter using the 5-bit Xpulp immediate as shift amount. For non-unsigned clips, the two bounds are the bitwise *not* of each other; for unsigned clips, the upper bound does not change, while the lower bound is zero.

From a behavioral point of view, the functionality of a clip needs two comparisons to clamp operand `A` between the two bounds: the input must be compared once against the lower bound and once against the upper bound. However, one comparison is enough if we select its second operand basing on the sign of the clip inputs: if at least one between the operand `A` of the clip and the maximum bound of the range is negative, then the lower bound is used as second operand for the comparison; otherwise, operand

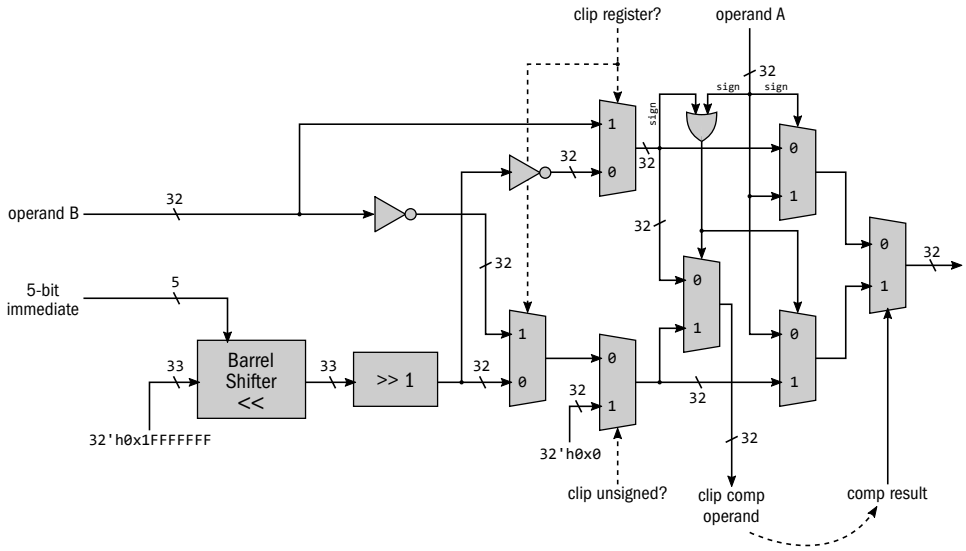


Figure 4.8.: Block diagram of the clip unit in the DSPU.

`A` is compared against the upper bound of the clip. To implement such behavior we employed a series of multiplexers, as in Fig. 4.8. The shared comparator is used to perform the comparison.

The comparison result, i.e. the relative magnitude of the operand `A` with respect to one of the clip bounds, is then used, in combination with the sign of operand `A`, for the selection of the final result, which can be one among: the lower bound, operand `A` and the upper bound.

4.2.4. Extension unit

Some other generally useful instructions, more involved with the management of different data widths as it is the case for SIMD operations, are represented by the extension operations. Xpulpimg introduces four instructions to extend the least significant sub-word contained in a register to the full 32-bit width of the destination register; the sub-word can have the size of either a byte or an half-word and the extension can be signed or unsigned. The instructions are listed in Table 4.5.

Extension instructions	
`p.exth{s,z} rd, rs1`	Sign- or zero-extend the lowest half-word of `rs1` to 32 bits
`p.extb{s,z} rd, rs1`	Sign- or zero-extend the least significant byte of `rs1` to 32 bits

Table 4.5.: 32-bit extension instructions for byte and half-word data introduced with Generic arithmetic operations.

This unit is implemented with a simple *case* statement containing the four possible extension operations described in a behavioral way, exploiting SystemVerilog signed and unsigned type casting functions and slicing of the input operand `A`. This maximizes the possibilities of optimization from the synthesizer side.

4.2.5. MAC unit

As already mentioned, the MAC unit of the Xpulpimg DSPU also implements the RISC-V multiplication instructions of the standard M extension. The multiplier is paired with an accumulator to also perform the MAC operations from Xpulp, listed in Table 4.6.

Multiply-accumulate instructions	
`p.mac rd, rs1, rs2`	Compute `rd` + `rs1` · `rs2` and store the result back in `rd`
`p.msu rd, rs1, rs2`	Compute `rd` − `rs1` · `rs2` and store the result back in `rd`

Table 4.6.: MAC operation for the multiplication of two 32-bit operands with 32-bit multiplication or subtraction, introduced in Xpulpimg with MAC operations. The lowest 32-bit of `rs1` · `rs2` are used for the accumulation operation.

Listing 4.7: Example assembly code of the inner loop for a naive 2D convolution algorithm implementation for 32-bit elements; the throughput of this algorithm is of 0.14 MACs/instruction.

```
<loop>:
    lw a5,0(a2)     ; load input element
    lw t4, 0(a4)    ; load coefficient
    addi a4, a4, 4
    mul a5, a5, t4  ; multiply the element and the coefficient
    add a1, a5, a1  ; accumulate the multiplication output on a1
    addi a2, a2, 4
    bne a0, a4, <loop>
```

Listing 4.8: Code from Listing 4.7 with MAC instructions support; the throughput of the algorithm is now 0.167 MACs/instruction, with an improvement of 17%.

```
<loop>:
    lw a5,0(a2)     ; load input element
    lw t4, 0(a4)    ; load coefficient
    addi a4, a4, 4
    p.mac a1, a5, t4  ; multiply and accumulate on a1
    addi a2, a2, 4
    bne a0, a4, <loop>
```

MAC operations are very common in DSP, especially in image processing; for example, they are largely used in digital filters, which boil down to convolution operations, and dot-product algorithms, also used to implement matrix multiplication. An intuitive example of the improvements determined by the MAC extension is given in Listing 4.7

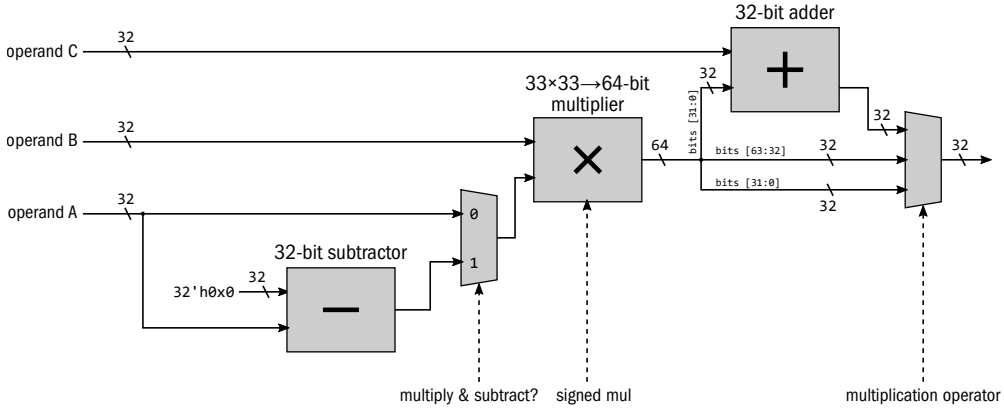


Figure 4.9.: Block diagram of the MAC unit in the DSPU.

and Listing 4.8; much higher benefit in terms of performance and power efficiency can be obtained with more sophisticated algorithms, as shown in Chapter 5.

With the help of the technology feedback, we found the best way to implement MACs being to remove the initial integer multiplier from the coprocessor and include its functionalities inside the new MAC unit in the DSPU. This allowed us to save some area cost, while paying in terms of hardware sharing. Anyway, this did not represent any issue due to the MAC unit not being timing-critical at all. The architecture of the MAC unit is depicted in Fig. 4.9.

The instructions from the standard M extension also performed by the MAC unit are `mul`, `mulh`, `mulhu` and `mulhsu`, a set of 32×32 multiplications with different signedness taking either the lower or the upper 32 bits of the result.

Since the signedness of the operation must be decided at runtime by the decoder control signals, the actually employed multiplier has two 33-bit inputs. One of them can be inverted in sign, so that multiply-subtract (MSU) instructions can be performed too.

A 32-bit adder is instantiated in series with the multiplier: when a MAC or a MSU are issued, the lowest 32 bits of the multiplier output are summed up to operand `C` of the IPU. On the other hand, when standard M multiplications are executed, the output of the multiplier unit can be chosen to be either the upper or the lower 32 bits of the multiplier output.

4.2.6. SIMD unit

When the full width of 32-bit operations is not needed, the environment of a scalar processor can be used, with few datapath extensions, to perform vectorial operations on sub-words in parallel. Xpulpimg introduces packed-SIMD instructions for generic arithmetic operations (Table 4.7), dot-product operations (Table 4.8) and sub-words management (Table 4.9). As in `Xpulpv2`, the instructions are implemented for 16-bit (half-word) and 8-bit (byte) operations in three addressing variations:

SIMD arithmetic instructions	
`pv.add[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD addition
`pv.sub[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD subtraction
`pv.avg[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD average
`pv.avgu[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	SIMD unsigned average
`pv.min[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD minimum
`pv.minu[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	SIMD unsigned minimum
`pv.max[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD maximum
`pv.maxu[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	SIMD unsigned maximum
`pv.srl[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD logical right-shift
`pv.sra[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD arithmetic right-shift
`pv.sll[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD logical left-shift
`pv.or[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD logical or
`pv.xor[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD logical xor
`pv.and[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD logical and
`pv.abs.{h,b} rd, rs1`	SIMD absolute value

Table 4.7.: Generic packed-SIMD arithmetic instructions, from Packed-SIMD extension; the employed immediate is the 6-bit Xpulp immediate, which is sign- or zero-extended based on the executed instruction. Note that shift operations use only the least significant 4 bits of the second operand for half-word-level parallelism, and the least significant 3 bits for byte-level parallelism.

Listing 4.9: Example assembly code of the inner loop for a naive 2D convolution algorithm implementation for 8-bit elements, compiled for the RV32IMA instruction set; its throughput is of 0.14 MACs/instruction.

```
<loop>:
    lb a5,0(a2)     ; load input element
    lb t4, 0(a4)    ; load coefficient
    addi a4, a4, 1
    mul a5, a5, t4  ; multiply the element and the coefficient
    add a1, a5, a1  ; accumulate the multiplication output on a1
    addi a2, a2, 1
    bne a0, a4, <loop>
```

SIMD dot-product instructions	
`pv.dotup[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	SIMD unsigned dot-product with reduction on `rd`
`pv.dotusp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD signed-unsigned dot-product with reduction on `rd`
`pv.dotsp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD signed dot-product with reduction on `rd`
`pv.sdotup[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	SIMD unsigned dot-product with reduction and accumulation on `rd`
`pv.sdotusp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD signed-unsigned dot-product with reduction and accumulation on `rd`
`pv.sdotsp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	SIMD signed dot-product with reduction and accumulation on `rd`

Table 4.8.: Packed-SIMD dot-product instructions, from Packed-SIMD extension; the employed immediate is the sign- or zero-extended 6-bit Xpulp immediate. All the dot-product instructions multiply the elements of the first operand with the second operand, basing on the addressing mode, and sum them up into a 32-bit destination.

Listing 4.10: Code from Listing 4.9 with packed-SIMD extension support; note that not only did the size of the hot-loop decrease, but also 4 elements are processed per iteration, instead of only one. `lw` is used instead of `lb` to load 4 sequential elements at a time, and 4 MACs are performed for each dot-product instruction. The throughput of the algorithm is now 0.67 MACs/instruction, with a speed-up of 3.7×.

```
<loop>:
    lw a5,0(a2)         ; load 4 input elements
    lw t4, 0(a4)        ; load 4 coefficients
    addi a4, a4, 4
    p.sdotsp a1, a5, t4 ; compute 4 multiplications and accumulate on a1
    addi a2, a2, 4
    bne a0, a4, <loop>
```

SIMD support instructions	
`pv.extract.{h,b} rd, rs1, pimm6s`	Extract from `rs1` the half-word indexed by `pimm6s[0]` or the byte indexed by `pimm6s[1:0]`, sign-extend it and store in `rd`
`pv.extractu.{h,b} rd, rs1, pimm6u`	Extract from `rs1` the half-word indexed by `pimm6u[0]` or the byte indexed by `pimm6u[1:0]`, zero-extend it and store in `rd`
`pv.insert.{h,b} rd, rs1, pimm6s`	Substitute in `rd` the half-word indexed by `pimm6s[0]` or the byte indexed by `pimm6s[1:0]` with the least significant half-word or byte from `rs1`; the remaining bits of `rd` are untouched
`pv.shuffle2.{h,b} rd, rs1, rs2`	Each sub-word of `rs2` is used to index the source register sub-word which should be copied in the respective sub-word of `rd`; the source register for each sub-word is selected with `rs2` sub-words and can be either `rs1` or `rd`.

Table 4.9.: Support instructions for packed-SIMD operations from Packed-SIMD extension; they represent a basic extension for the packing and unpacking of SIMD data.

1. *vectorial* – the two input registers `rs1` and `rs2` are considered vectors of two 16-bit or four 8-bit elements, and an element-wise operation is performed;
2. *scalar replication* – the lowest half-word or byte of `rs2` is considered a scalar and used for the operation with the array of half-words or bytes in `rs1`;
3. *immediate scalar replication* – this variation has the same behavior of the scalar replication mode but uses the 6-bit signed or unsigned immediate as scalar operand.

SIMD has a large utilization in DSP due to the trade-off that it enables between precision and performance, in terms of speed-up and energy efficiency. Sub-word parallelism, especially at byte-level, is one of the extensions with the highest impact on image processing: with pixels represented by 8-bit values, the speed-up theoretically determined by packed-SIMD is 4×. An example is given in Listing 4.9 and Listing 4.10.

On the other hand, SIMD operations need to have the data packed in a single register, thus some overhead to prepare input operands and to unpack outputs is needed; this is particularly the case when the elements are not stored sequentially in memory. However, such operations can be accelerated by several SIMD support instructions, listed in Table 4.9.

We developed the SIMD unit of the DSPU keeping all the operations within the same *case* statement, to give to Synopsys Design Compiler the highest potential of optimization.

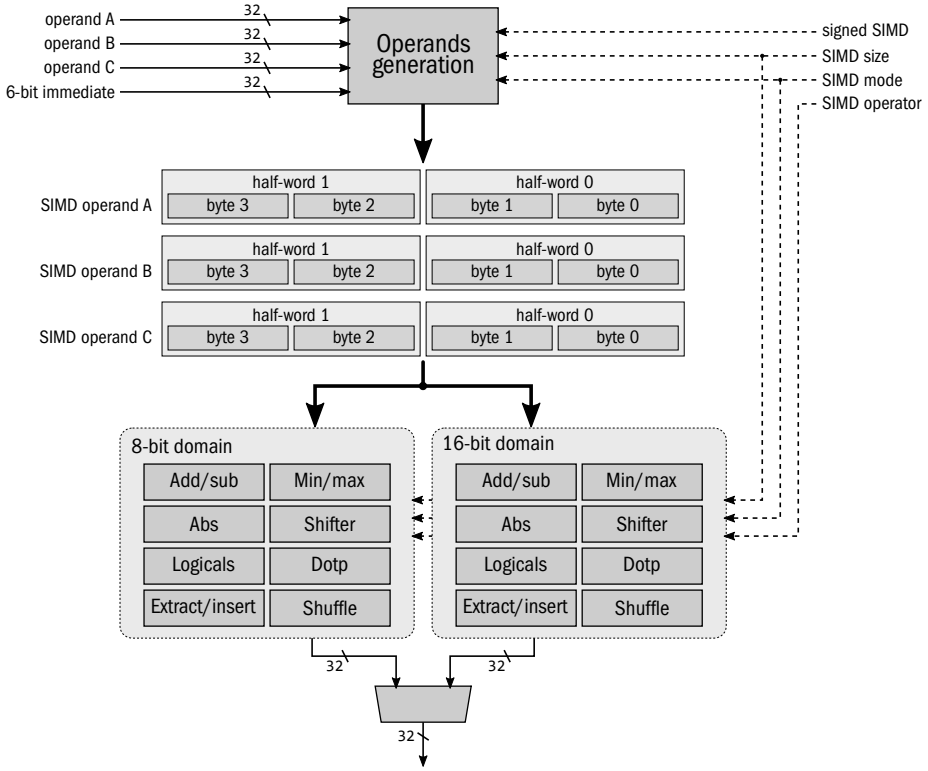


Figure 4.10.: Block diagram of the SIMD unit in the DSPU.

Before the execution, the three input operands `A`, `B` and `C` of the unit are adjusted into vectorial structures basing on the granularity of the issued SIMD operation. In particular, the second operand is generated basing on the employed addressing variation starting from the operand `B` of the IPU or the 6-bit immediate, after its sign- or zero-extension.

As in Fig. 4.10, the vectorized datapath of the SIMD unit is divided into two domains: one is segmented in two parts to perform half-word operations, the other is segmented in four parts and deals with byte sub-words. Such an organization maximizes the possible optimizations from the synthesizer side in terms of hardware sharing and timing inside the same domain, i.e. among operations with the same width, which results in more efficient implementations.

Chapter 5

Results

We implemented the Xpulpimg ISA extension in the Snitch processor core with the aim to exploit its desirable features of flexibility, efficiency and area to provide the MemPool system with an highly parallelizable processing element with DSP capabilities. This paved the way for MemPool to approach more efficiently the domain of image processing. In this chapter we show the results that we collected in terms of performance, area and power in comparison with the baseline system implementing the RV32IMA instruction set only, demonstrating the validity of our approach.

In particular, we firstly illustrate our evaluation methodology in terms of benchmarking and synthesis procedure. Then, we analyze in depth the implementation cost of each one of the extensions that we incrementally implemented in the MemPool CC, going through our design iterations and looking at the related code improvements of a kernel of interest. Subsequently, we analyze the benchmark results of the kernel and study the overall speed-up determined by Xpulpimg. We also illustrate the results obtained from the benchmark of other, more generic DSP kernels.

Finally, to characterize the impact of the Xpulpimg implementation from the broader perspective of the MemPool cluster, we analyze its implementation cost in a MemPool tile; to the same end, we also consider a power analysis to quantify the energy efficiency improvements.

5.1. Evaluation setup

This section details our evaluation setup in terms of benchmarking methodology and synthesis process.

5.1.1. Benchmarking methodology

For performance evaluation, we carried out cycle-accurate RTL simulations in Verilator 4.024 2019-12-08, recording traces for the processor cores with a SystemVerilog behavioral

tracer. The traces were parsed to a human-readable format by means of a Python script, exploiting Snitch CSR counters to keep track of the number of elapsed cycles.

We benchmarked MemPool with two algorithms of interest for DSP and image processing applications:

- *conv2d* – a 2D integer convolution of a 32×32 matrix with a 3×3 kernel;
- *matmul* – a matrix multiplication between two 64×64 matrices.

For simulation performance reasons, we benchmarked MemPool in a 16-core configuration with 64 memory banks, accounting for a total of 64 KiB for the shared L1 TCDM. We used such a cache to store the input and output matrices, hence avoiding long stalls due to potential L2 accesses while focusing on the performance of our extensions.

Basing on the size of the hot-loops of the employed kernels, we also optimized the MemPool configuration to the benchmark algorithm in terms of instruction cache size. With four cores per tile, each tile has an L1 instruction cache of 2 KiB. To avoid memory access stalls due to an hot-loop not entirely fitting in the private L0 cache of each core, we halved L1 associativity (moving to a two-way set-associativity) doubling cache line size (from 128 bits to 256 bits), to obtain twice the size for the L0 cache [35]. With this configuration, the L0 cache has four 256-bit lines, able to hold up to 32 instructions.

We developed the benchmark algorithms in C language and compiled them with the PULP RISC-V GCC toolchain extended with the Xpulpimg instruction set. In their compilation we took care of aligning all loops to the instruction cache line boundary of 32 bytes, to grant the hiding of potential instructions pre-fetcher accesses to L1. To harvest the most out of the new instruction set extensions, we optimized the 2D convolution and the matrix multiplication employing compiler intrinsics and manual assembly-level optimizations specifically suited for Xpulpimg. Optimized algorithms, in terms of parallelization and loop unrolling, have also been employed for the RV32IMA benchmarks simulated on the baseline system.

The 2D convolution algorithm with 8-bit data types, which can be thought as pixels of a grayscale image, has been run in a single-thread mode, to better study the speed-up of an individual MemPool CC processing element among the several design iterations, comparing it with the area and timing costs.

The matrix multiplication kernel has instead been simulated with different data widths and parallelized over multiple cores, with the aim to characterize the performance of the whole system in a more general way.

5.1.2. Synthesis methodology

We synthesized our system for GlobalFoundries 22FDX FD-SOI technology using Synopsys Design Compiler 2019.12. The synthesis aim has been twofold. As already mentioned in Section 3.7, we used post-synthesis AT figures as feedback for the RTL design exploration of Snitch extensions; to this end, we synthesized MemPool CC in typical conditions (TT, 0.80 V, 25 °C) at each incremental development iteration of Xpulpimg.

Note that the MemPool CC exclusively includes a Snitch core, its Snitch IPU coprocessor and the spill registers in between them, as shown in Fig. 4.1. Such a system has been synthesized as if enclosed in a register boundary. This approach allowed us to collect the exact incremental post-synthesis figures of each design iteration, without the synthesis noise coming from cache memories or the TCDM interconnect.

In this process, at each design iteration, we measured the maximum reachable operating frequency of the system and computed the set of Pareto-optimal points in terms of clock period and area, in a sweep from 2 ns (500 MHz) to 0.5 ns (2 GHz). The set of design points obtained with this approach is optimal in terms of area for each given operating frequency, in compliance with the need of Snitch to be massively replicated within MemPool.

On the other hand, we also synthesized the whole MemPool tile, including four MemPool CCs, targeting 500 MHz at worst-case conditions (SS, 0.72 V, 125 °C) and a 16-core configuration, to compare the final post-synthesis area, timing and power results between the baseline and our extended version from the perspective of the whole MemPool cluster. The tile has been synthesized with 2 KiB of instruction cache and 16 KiB of SPM.

For the power analysis, we extracted the switching activity by simulating a benchmark on the post-synthesis netlist of a tile with back-annotated post-synthesis delay information for the typical corner (TT, 0.80 V, 25 °C) in Questa Sim 2019.3; we performed the power estimation in Synopsys PrimeTime 2019.12 at typical conditions.

5.2. Design iterations evaluation

In this section we carry out an analysis of the incremental versions of the MemPool CC that we developed throughout its design iterations. Initially, we go through the individual design iterations, each consisting of a newly implemented subset of Xpulpimg with respect to the previous version. For each incremental extension, we analyze the improvement in the hot-loop size of a benchmark kernel and the related cost in terms of MemPool CC area and timing, highlighting the main criticalities. The legend of the analyzed design iterations is described in the following, and refers to the Xpulpimg extension sections already mentioned in Section 3.1:

- Baseline – the system implementing the starting RV32IMA instruction set;
- Generic ALU – generic **arithmetic instructions**;
- Post-increment L/S – extended **load and store** addressing modes;
- MAC – **multiply-accumulate** operations;
- SIMD – **packed-SIMD** extension.

Subsequently, we analyze the overall speed-up of a benchmark of interest comparing the baseline version and the final version including all the instructions from our Xpulpimg instruction set.

Note that, when the full Xpulpimg extension is referenced, it corresponds to the last version of the above mentioned design iterations, *SIMD*. Hence, in the following, we will refer to it with the label Xpulpimg.

For the benchmark simulation and the assembly code analysis of the next sections, we employed a single-core 3×3 2D convolution benchmark for 32×32 8-bit matrices, due to its relevance in the field of image processing as outlined in Section 2.5. Its hot-loop compiled from C language with the PULP RISC-V GNU toolchain, in its different variations based on the Xpulpimg version, is listed in Table 5.1. Listing B.1 shows the C function from which the *Baseline*, *Generic ALU*, *Post-increment L/S* and *MAC* versions of the kernel have been compiled; Listing B.2 shows the C convolution version optimized for Xpulpimg, exploiting the compiler intrinsics for the *SIMD* extension.

5.2.1. Incremental analysis of design iterations

In the following, an analysis of the incremental versions of the MemPool CC is carried out. Table 5.1 shows the hot-loop of the convolution kernel for each one of the detailed design iterations, highlighting the changes among them. Additionally, Fig. 5.1, Fig. 5.2 and Fig. 5.3 show the post-synthesis AT plots obtained from the sweep of the clock period for the final version of each design iteration; note that only design points which met the timing constraints have been plotted. As already mentioned, an area optimization has been performed at each given clock period; hence, from the resulting design points the Pareto-optimal curve in terms of area and timing can be extracted for the MemPool CC. Additionally, Table 5.2 shows the percentage increment of MemPool CC area and its sub-modules with respect to the baseline version, for each design iteration. Table 5.3 details how the maximum reachable operating frequencies of the MemPool CC has been affected by each extension.

Baseline

The baseline version of the MemPool system, implementing only the RV32IMA ISA, has been the starting point for the development of our extension. The hot-loop of the convolution binary generated with such an instruction set has a size of 47 instructions.

The highest operating frequency reached by this initial version is 1.85 GHz (clock period of 0.54 ns), with the most critical paths traversing the register file inside the core through its read port and going across the write-back path.

With the coprocessor positioned in between two spill registers, its execution units have one full clock cycle for their combinational logic. For this reason, most of the hardware in the IPU is not timing-critical at all, and is instead optimized mainly for area, even at higher frequencies.

Baseline	Generic ALU	Post-increment L/S	MAC	SIMD
lbu a4,0(a3)	lbu a4,0(a3)	lbu a4,0(a3)	lbu s11,0(a3)	pv.dotsp.b a5,a5,t5
lbu a6,1(a3)	lbu a6,1(a3)	lbu a6,1(a3)	lbu t2,1(a3)	pv.sdotsp.b a5,a7,t4
lbu a0,2(a3)	lbu a0,2(a3)	lbu a0,2(a3)	lbu s0,2(a3)	pv.sdotsp.b a5,a4,t3
mul a5,a5,a6	mul a5,a5,a6	mul a5,a5,a6	mul a5,a5,t2	divu a5,a5,t6
mul a2,a2,a4	mul a2,a2,a4	mul a2,a2,a4	p.mac a5,t6,s11	mv s7,t1
lbu a4,3(a3)	lbu a4,3(a3)	lbu a4,3(a3)	lbu t2,3(a3)	p.sw a5,s4(s6!)
mul s1,s1,a0	mul s1,s1,a0	mul s1,s1,a0	p.mac a5,t0,s0	addi a5,t1,-1
lbu a6,4(a3)	lbu a6,4(a3)	lbu a6,4(a3)	lbu t6,4(a3)	lbu a6,0(a5)
add a2,a2,a5	add a2,a2,a5	add a2,a2,a5	p.mac a5,a6,t2	p.lb s8,1(s7!)
lbu a0,5(a3)	lbu a0,5(a3)	lbu a0,5(a3)	lbu t0,5(a3)	andi a6,a6,255
mul a4,t6,a4	mul a4,t6,a4	mul a4,t4,a4	p.mac a5,t3,t6	lbu a5,0(s7)
add a5,s1,a2	add a5,s1,a2	add a5,s1,a2	lbu s11,6(a3)	pv.insert.b a6,s8,1
add s1,a4,a5	add s1,a4,a5	add s1,a4,a5	p.mac a5,a7,t0	pv.insert.b a6,a5,2
lbu a2,6(a3)	lbu a2,6(a3)	lbu a2,6(a3)	lbu s0,7(a3)	addi s5,s5,1
mul a6,t4,a6	mul a6,t4,a6	mul a6,t0,a6	p.mac a5,a2,s11	mv a5,a7
lbu a4,7(a3)	lbu a4,7(a3)	lbu a4,7(a3)	lbu t0,8(a3)	mv a7,a4
mul a0,t0,a0	mul a0,t0,a0	mul a0,t5,a0	mv t6,a4	mv a4,a6
add a6,a6,s1	add a6,a6,s1	add a6,a6,s1	p.mac a5,a0,s0	add t1,t1,a2
lbu a5,8(a3)	lbu a5,8(a3)	lbu a5,8(a3)	addi t2,a4,-1	pv.insert.b a4,zero,3
mul a2,t3,a2	mul a2,t3,a2	add s1,a0,a6	lbu t2,0(t2)	bne s5,t0,loop
add s1,a0,a6	add s1,a0,a6	mul a2,t1,a2	p.mac a5,t1,t0	
add a0,a2,s1	add a0,a2,s1	add a0,a2,s1	p.lb s0,1(t6!)	
mul a4,t5,a4	mul a4,t5,a4	mul a4,t3,a4	divu a5,a5,s1	
mul a6,t2,a5	mul a6,t2,a5	mul a6,t6,a5	lbu s11,0(t6)	
add a5,a4,a0	add a5,a4,a0	add a5,a4,a0	mv t0,a7	
add a2,a6,a5	add a2,a6,a5	mv a2,a7	p.sw a5,s9(t5!)	
divu s1,a2,s2	divu s1,a2,s2	addi s1,a7,-1	mv t6,a6	
lbu a6,-1(a7)	lbu a6,-1(a7)	add a0,a6,a5	mv a7,t1	
lbu a0,0(a7)	lbu a0,0(a7)	lbu a4,0(s1)	mv a6,a2	
lbu a4,1(a7)	lbu a4,1(a7)	divu a5,a0,s2	slli t1,s11,0x18	
mv a2,t6	mv a2,t6	p.lb a0,1(a2!)	slli a2,t2,0x18	
sw s1,0(t1)	sw s1,0(t1)	mv s1,t5	addi t4,t4,1	
mv a5,t4	mv a5,t4	lbu a6,0(a2)	mv a5,t3	
mv s1,t0	mv s1,t0	mv t5,t6	add a4,a4,a1	
mv t6,t3	mv t6,t3	p.sw a5,s10(s0!)	mv t3,a0	
mv t4,t5	mv t4,t5	mv a2,t4	srai a2,a2,0x18	
mv t0,t2	mv t0,t2	slli t6,a6,0x18	mv a0,s0	
slli t3,a6,0x18	slli t3,a6,0x18	mv t4,t1	srai t1,t1,0x18	
slli t5,a0,0x18	slli t5,a0,0x18	addi t2,t2,1	bne t4,s2,loop	
slli t2,a4,0x18	slli t2,a4,0x18	slli t1,a4,0x18		
addi s0,s0,1	addi s0,s0,1	mv a5,t0		
add a7,a7,a1	add a7,a7,a1	add a7,a7,a1		
add t1,t1,s4	add t1,t1,s4	mv t0,t3		
srai t3,t3,0x18	srai t3,t3,0x18	srai t1,t1,0x18		
srai t5,t5,0x18	srai t5,t5,0x18	mv t3,a0		
srai t2,t2,0x18	srai t2,t2,0x18	srai t6,t6,0x18		
bne s0,s3,loop	bne s0,s3,loop	bne t2,s3,loop		

Table 5.1.: Assembly code improvement of the 2D convolution hot-loop among Xpulpimg design iterations; the instructions highlighted in blue are the new ones with respect to the previous extension.

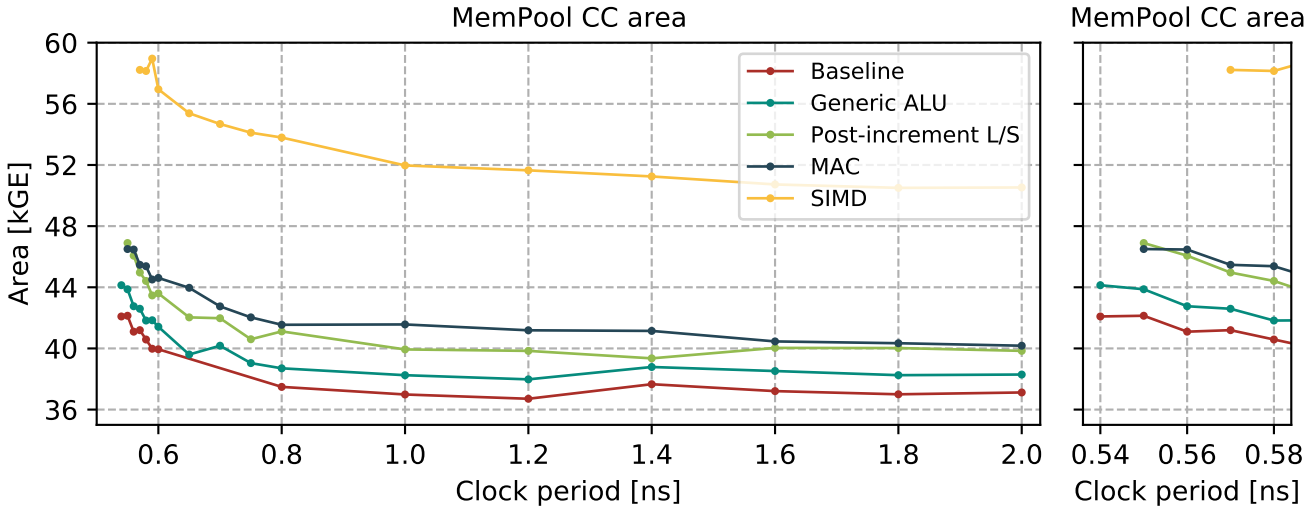


Figure 5.1.: Area figures of MemPool CC as a function of the clock period; the right plot focuses on the most critical frequencies.

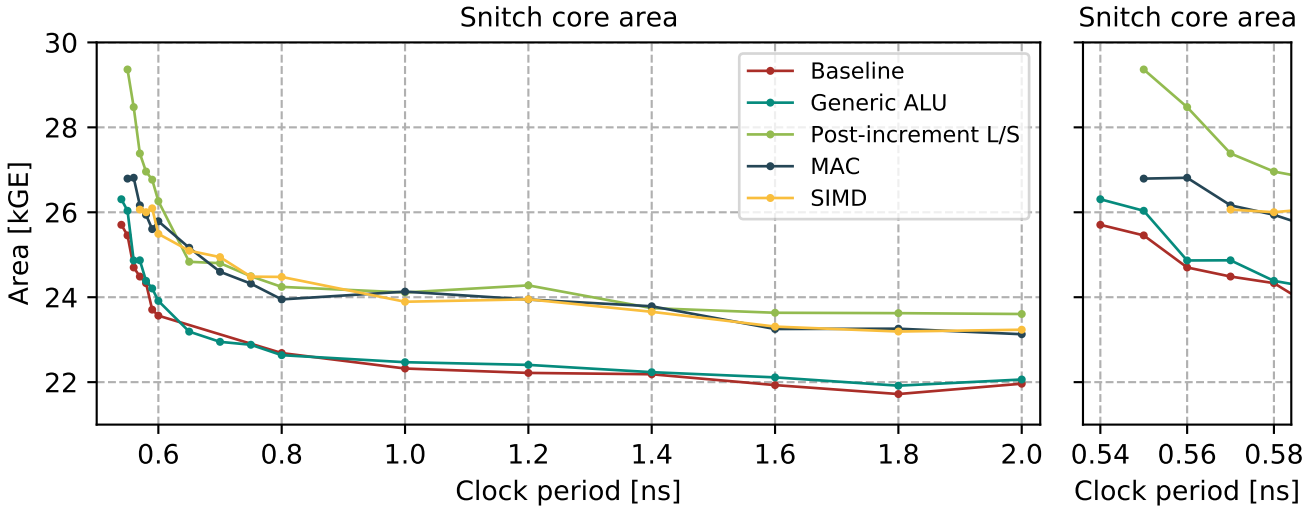


Figure 5.2.: Area figures of the Snitch core from MemPool CC synthesis as a function of the clock period; the right plot focuses on the most critical frequencies.

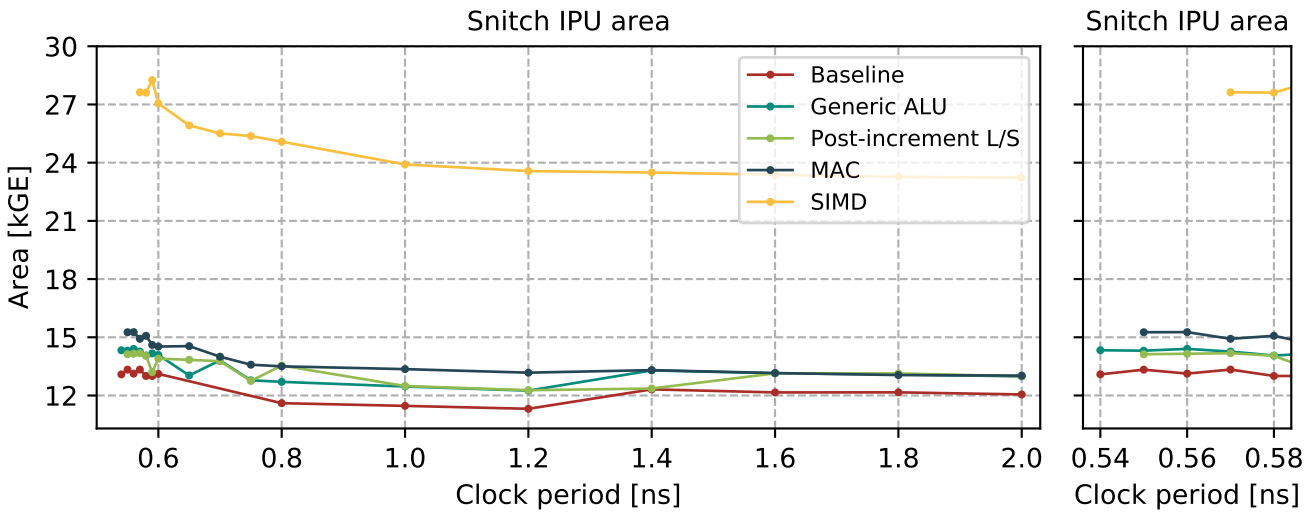


Figure 5.3.: Area figures of the Snitch IPU coprocessor from MemPool CC synthesis as a function of the clock period; the right plot focuses on the most critical frequencies.

	Generic ALU	Post-increment L/S	MAC	SIMD
MemPool CC	3.64%	8.74%	10.87%	40.40%
Snitch core	1.02%	10.22%	7.14%	7.25%
Snitch IPU	8.27%	7.23%	12.80%	103.92%

Table 5.2.: MemPool CC average post-synthesis area increase with respect to the baseline version; the average refers to the clock period sweep between 2 ns and 0.5 ns.

Baseline	Generic ALU	Post-increment L/S	MAC	SIMD
1.85 GHz (0.54 ns)	1.85 GHz (0.54 ns)	1.82 GHz (0.55 ns)	1.82 GHz (0.55 ns)	1.75 GHz (0.57 ns)

Table 5.3.: MemPool CC maximum operating frequency for each design.

Generic ALU

The instructions implemented in this design step are listed in Table 4.3, Table 4.4, Table 4.5 and Table 4.2; they include generic arithmetic operations for comparisons and absolute value, clip operations, signed and unsigned extensions and immediate branches.

Despite their general usefulness in the DSP domain, they are not of much help for convolution, which heavily relies on a sequence of additions and multiplications with a linear program flow. For this reason, the compiled convolution hot-loop for this design iteration does not show any change, with such extension being mostly useful for other families of algorithms.

The area of the Snitch core, which implements the extended branch instructions described in Section 4.1.2 and the decoding logic for the new instructions offloaded to the coprocessor, is not significantly affected by this extension, which determines an increase of just 100 to 600 GE. The remaining instructions, whose datapath is fully implemented in the IPU coprocessor, determine an average area increase of 8.3% (about 1 kGE), mainly consisting in: a 33-bit barrel shifter with 5-bit shift amount, employed by the clip unit described in Section 4.2.3; the shared 33-bit less-than-equal comparator of Section 4.2.1; a 32-bit subtractor used by the arithmetic unit for the absolute value computation, as in Section 4.2.2.

Such a small increase, despite the new sub-module in the Snitch IPU and the several different functionalities introduced, has been possible thanks to the feedback from the GF22 technology which brought us to implement the decoding architecture described in Section 4.2, the shared comparator and the optimized clip unit.

The highest operating frequency reached by this version is still 1.85 GHz (clock period of 0.54 ns), without any need of pipelining the coprocessor.

Post-increment L/S

The implementation of the extended load and store instructions, listed in Table 4.1, does not have a large impact on the considered hot-loop: the main memory accesses in this loop are needed to load the coefficients of the 3×3 filter, which have a fixed memory address, hence basic load-byte instructions are enough. Very few post-increment instructions are instead generated only to load the new row of the kernel window and to store the result in the output matrix.

The small influence of the extended memory operations on the 2D convolution kernel is mainly due to the specific optimization that we employed for this algorithm: since during each iteration only 3 elements are loaded to perform 9 MACs, the hot-loop is not very much dependent on the memory accesses optimizations.

While the new addressing modes for loads and stores do not represent a turning point for the 2D convolution hot-loop, they are anyway of very high interest for every other image processing and DSP algorithm. Additionally, load operations which can be optimized with the post-increment addressing mode are very much present in the outer loop of our convolution algorithm, which loads a new full 3×3 window of the input matrix.

In terms of cost, while the implementation of the extended memory operations does not affect the IPU at all, it determines an average increase of about 10% (2.4 kGE) in the area of the core. This is mainly due to the introduction of a third read port in the integer register file, pointed out in Section 4.1.1, accounting for an increase between 1.4 kGE and 1.7 kGE.

The introduction of a new port in the register file also increases the criticality of the timing path across it, decreasing the maximum operating frequency of MemPool CC to 1.8 GHz (clock period of 0.55 ns).

MAC

The MAC extension includes the instructions listed in Table 4.6. Due to its nature, they are of main relevance for the convolution hot-loop: many MACs instructions are indeed compiled to substitute the series of element-wise multiplications with accumulation between the window of the input image and the filter, reducing the size of the hot-loop to 39 instructions.

After the design exploration performed with the GF22 technology feedback, we found that the most efficient way to implement MACs in the Snitch IPU is to share their multiplier with the integer multiplications from the standard RISC-V M extension.

A 64-bit signed/unsigned integer multiplier was already present in the Snitch IPU to implement the M extension. By including it in the DSP unit, paired to a 32-bit adder and a 32-bit subtractor, we managed to save between 8 kGE and 9.5 kGE, avoiding the instantiation of a multiplier exclusively dedicated to MACs as discussed in Section 4.2.5.

With such an architecture, this extension determines an average area increase of 12.8% with respect to the baseline. Such an increases just consists of the hardware for the sharing of the multiplier, the subtraction for *MSU* operand inversion and the accumulation. The maximum operating frequency does not result worsened by the introduction of MACs operations.

SIMD

The packed-SIMD extension is composed of the instructions included in Table 4.7, Table 4.8 and Table 4.9. The main contribution to the convolution kernel is brought by the dot-product instructions, which are able to perform up to four 8-bit MACs per instructions, computing the output of a 3×3 window with just three instructions. Also, thanks to the better register occupation determined by the use of packed data, the coefficients of the filter do not need to be re-loaded at each iteration, causing a strong reductions in the number of loads. Thanks to SIMD, the hot-loop size results reduced to 20 instructions.

While being the extension with the most relevant contribution to performance improvement, SIMD also causes a significant increase in the area of the Snitch IPU, which almost doubled in the number of equivalent gates. As a matter of fact, we measured an area between 11 kGE and 14 kGE for the introduced SIMD unit, accounting for an average 40% increase in the MemPool CC area.

In particular, the SIMD unit introduces in the DSP unit several 8-bit and 16-bit adders, comparators, shifters and multipliers, as pointed out in Section 4.2.6. The most critical resources in the unit are the sub-word multipliers and adders for dot-product operations, which are optimized for both area and speed. The SIMD unit has indeed a detectable impact, even if negligible, on the critical path of the system, whose maximum operating frequency decreases to 1.79 GHz.

Nevertheless, post-synthesis timing reports identify the path through the register file in the Snitch core as one of the most critical ones. In a real environment, with the MemPool CC interfacing with the cache memory and the TCDM interconnect, the critical path through Snitch would be the actually timing-critical one, due to the Snitch IPU being enclosed in a register boundary, and thus not affected by external modules.

5.2.2. Convolution benchmark analysis

To compare our Xpulpimg implementation with the baseline RV32IMA MemPool CC, we characterized the speed-up of the baseline and extended system with three version of the 2D convolution benchmark:

- *Baseline conv2d* – the baseline version compiled from the code in Listing B.1 for the RV32IMA instruction set, simulated on the RV32IMA MemPool CC;
- *Xpulpimg conv2d* – the version compiled from the code in Listing B.2 for the RV32IMAXpulpimg instruction set, simulated on the MemPool CC extended with the full Xpulpimg extension;
- *Unrolled Xpulpimg conv2d* – the version compiled from the code in Listing B.2 for the RV32IMAXpulpimg instruction set with loop unrolling enabled, simulated on the MemPool CC extended with the full Xpulpimg extension.

Note that we did not benchmark an unrolled version of *Baseline conv2d* due to it being already saturated in terms of unrolling.

Table 5.4 shows the absolute figures resulting from the benchmarks simulation on the baseline MemPool CC and on the final version implementing the whole Xpulpimg extension, while Table 5.5 shows the performance figures with respect to MAC operations. Fig. 5.4 sums up the performance comparison among the three benchmarks in terms of MACs/cycle.

The *Xpulpimg conv2d* benchmark, whose compiler-generated hot-loop is under the *SIMD* column in Table 5.1, is only **1.6× faster**, in terms of MACs/cycle, than the baseline version, displayed under the *Baseline* column. The speed-up is due to the smaller size of the hot-loop, which increased the throughput: with SIMD, only 3 dot-product instructions are needed to perform 9 MACs, even if some overhead due to *insert* instructions is present to pack the loaded bytes into SIMD data structures. The number of load operations also results greatly decreased (of a factor of 3.7×) due to the decreased register occupation determined by the use of SIMD: with the baseline version, 9 registers are needed to hold all the coefficient of the 3×3 convolution filter; since Xpulpimg

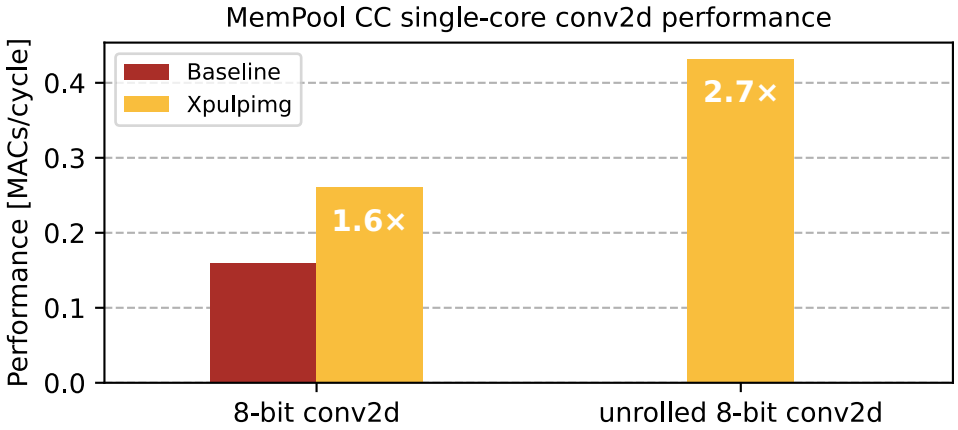


Figure 5.4.: Performance of the MemPool CC in terms of MACs/cycle measured simulating the 8-bit 3×3 2D convolution kernel.

Benchmark	IPC	Cycles	Issues	Loads	Stores	Stalls
Baseline conv2d	0.83	52561	43807	11121	913	3268
Xpulpimg conv2d	0.58	32794	19108	2998	910	8446
Unrolled Xpulpimg conv2d	0.90	20232	18256	3135	964	467

Table 5.4.: Absolute benchmark figures for single-core 2D convolution of 32×32 8-bit matrices.

Benchmark	Hot-loop size	MACs/iteration	MACs/cycle
Baseline conv2d	47	9	0.159
Xpulpimg conv2d	20	9	0.261
Unrolled Xpulpimg conv2d	75	36	0.432

Table 5.5.: Benchmark results with respect to MAC operations for single-core 2D convolution of 32×32 8-bit matrices.

only needs 3 registers to pack all the coefficients, they can be kept in the same registers without being reloaded at every iteration.

However, the size of the hot-loop reduced of a factor of 2.35× results in a very much reduced amount of independent instructions. With less scheduling flexibility for the compiler, a larger number of stalls due to data hazards occurs, with the IPC dropping from 0.83 to 0.58.

In other words, Xpulpimg exposes further possibility of optimization in terms of loop unrolling for the 8-bit *conv2d*; to understand the real performance improvement due to our extension, we also benchmarked *Unrolled Xpulpimg conv2d*, corresponding to the same algorithm of Listing B.2 but unrolled to process 4 3×3 windows during each

hot-loop iteration, instead of only one. The unrolled Xpulpimg algorithm hides most of the data dependencies and is **2.7× faster** than the baseline in terms of MACs/cycle, getting closer to the 4× theoretical limit determined by the use of 8-bit SIMD.

However, the algorithm employed for the described convolution benchmark has an actual theoretical limit for the speed-up determined by SIMD of just 3×: as a matter of fact, the SIMD dot-product instructions can perform 4 MACs as a single operation, but with 9 MACs needed for each 3×3 window, only 9 of the 12 available MAC *slots* are employed during each iteration of the hot-loop. To have an actual theoretical limit of 4× for the SIMD convolution speed-up, a different algorithm would be needed, able to interleave the computation of different windows. It would then exploit the full 12 MACs slots of each iteration, but at the cost of additional complexity and overhead.

This means that with the 2.7× speed-up of *Unrolled Xpulpimg conv2d*, determined for the MemPool CC by the Xpulpimg extension implementation, we nearly reached the speed-up upper bound for the 2D 3×3 convolution algorithm. A certain overhead is still present due to the processing needed to re-arrange the bytes in packed arrays for SIMD operations; also, load-word instructions cannot be used to load four elements at a time due to the kernel-based nature of the convolution algorithm, which might result in unaligned memory accesses (Snitch does not currently support unaligned memory accesses).

5.3. Additional benchmarks

For the sake of a more generic benchmarking, we also measured the performance of the matrix multiplication algorithm, in particular with 32-bit and 8-bit integer data types, comparing its performance between the baseline and the final version of the Xpulpimg extension. Their results are summed up in Fig. 5.5; we give a detailed analysis in the following. The baseline kernels and the versions that we optimized for the Xpulpimg-extended ISA are available in Appendix C.

5.3.1. 32-bit matrix multiplication

We benchmarked the multiplication between two 32-bit matrices with 64×32 and 32×64 dimensions, resulting in a 32-bit 64×64 matrix. The comparison between the two compiler-generated hot-loops, the baseline and the Xpulpimg versions, is shown in Table 5.6. The baseline version has been compiled from the C code in Listing C.1, while the Xpulpimg version has been compiled for the RV32IMAXpulpimg architecture from the C code in Listing C.2.

Both versions are equally parallelized over the 16 simulated cores and unrolled in the following way: the outer loop is unrolled to compute a 2×2 chunk of the output matrix for each iteration (i.e. for each complete run of the hot-loop); the inner loop (i.e. the hot-loop) is, in turn, unrolled to compute 2 MACs for each element of the 2×2 output chunk during each iteration. Thus, every iteration of the hot-loop loads 8 elements from the input matrices (4 elements from matrix A and 4 elements from matrix B) and

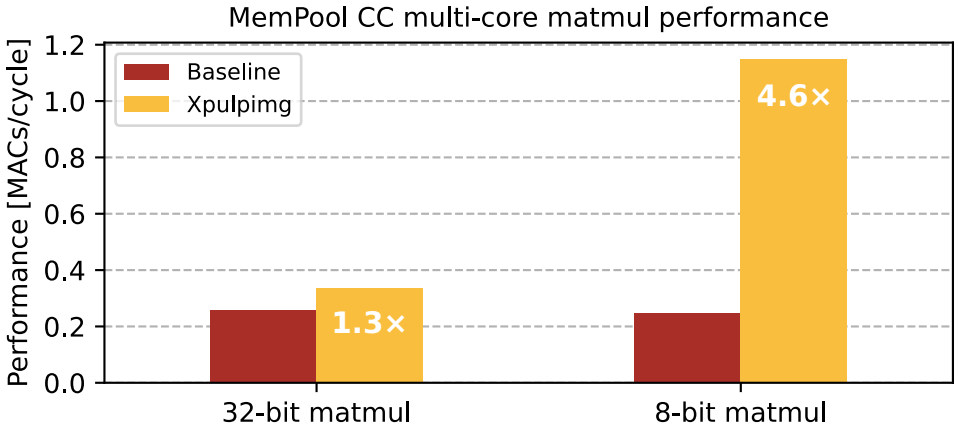


Figure 5.5.: Performance of the MemPool CC in terms of MACs/cycle measured simulating the matrix multiplication kernels.

performs 8 MACs. Additionally, for the Xpulpimg matrix multiplication, we performed specific optimizations at assembly level explicitly defining the post-increment load and store operations, to obtain an optimal register utilization for base addresses and offsets.

The results of the kernel benchmark averaged on the 16 cores are reported in Table 5.7; Table 5.8 reports the relative performance measured in terms of MAC operations. The kernel compiled for the RV32IMA version has a size of 30 instructions and its benchmark returned a performance of 0.259 MACs/cycle; in particular, 8 loads are needed to collect all the elements to multiply from the two matrices, along with their address incrementing operations. The kernel compiled with Xpulpimg not only is able to collapse all the additions and multiplications under single MACs, but it also cancels the need of incrementing the addresses thanks to the post-increment loads. With such improvements, the hot-loop reaches a size of 22 instructions and a MACs/cycle ratio of 0.335, with **speed-up of 1.3**$\times$ with respect to the baseline, as shown in Fig. 5.5.

5.3.2. 8-bit matrix multiplication

Finally, we benchmarked an 8-bit matrix multiplication between two 64×64 matrices, returning a 64×64 32-bit output matrix. The comparison between the two compiler-generated hot-loops, the baseline and the Xpulpimg versions, is shown in Table 5.9. The baseline version has been compiled from the C code in Listing C.3, while the Xpulpimg version has been compiled for the RV32IMAXpulpimg architecture from the C code in Listing C.4.

The C algorithms for the 8-bit *matmul* kernel have the same structure of the 32-bit version, described in Section 5.3.1. However, we developed several additional optimizations for the Xpulpimg version:

- as in the 32-bit kernel, we manually defined the sequence of load and store operations at the assembly level, for an optimal register utilization;

Baseline	Xpulpimg
`lw a1,0(t2)`	`mv a5,t6`
`lw t0,0(s0)`	`mv a3,t0`
`lw a3,0(s1)`	`p.lw t4,4(a5!)`
`lw a5,4(s1)`	`p.lw s2,t1(a5!)`
`lw t3,4(t2)`	`p.lw s4,4(a5!)`
`lw a2,4(s0)`	`lw s0,0(a5)`
`lw a0,0(s2)`	`p.lw t3,4(a3!)`
`lw a7,4(s2)`	`p.lw s3,a6(a3!)`
`mul t6,a1,a3`	`p.lw s1,4(a3!)`
`mul a1,a1,a5`	`lw t2,0(a3)`
`mul a3,t0,a3`	`p.mac a2,t4,t3`
`mul a5,t0,a5`	`p.mac a1,t4,s3`
`add t4,t6,t4`	`p.mac a0,s4,t3`
`add t5,a1,t5`	`p.mac a7,s4,s3`
`mul t6,t3,a0`	`addi t5,t5,2`
`add t1,a3,t1`	`p.mac a2,s2,s1`
`mul a0,a2,a0`	`p.mac a1,s2,t2`
`mul t3,t3,a7`	`p.mac a0,s0,s1`
`add a6,a5,a6`	`p.mac a7,s0,t2`
`mul a2,a2,a7`	`addi t6,t6,8`
`addi s3,s3,2`	`add t0,t0,s5`
`add t4,t6,t4`	`bltu t5,a4,loop`
`add t5,t3,t5`	
`add t1,a0,t1`	
`add a6,a2,a6`	
`addi t2,t2,8`	
`addi s0,s0,8`	
`add s1,s1,s4`	
`add s2,s2,s4`	
`bltu s3,a4,loop`	

Table 5.6.: Assembly code improvement of the 32-bit matrix multiplication hot-loop from the baseline to the Xpulpimg version; the instructions introduced by Xpulpimg are highlighted in blue.

Benchmark	IPC	Cycles	Issues	Loads	Stores	Stalls
Baseline matmul	0.97	33788	32790	8446	326	204
Xpulpimg matmul	0.92	26300	23979	8333	291	350

Table 5.7.: Absolute benchmark figures averaged on the 16 simulated cores for the 32-bit matrix multiplication kernel.

Benchmark	Hot-loop size	MACs/iteration	MACs/cycle
Baseline matmul	30	8	0.259
Xpulpimg matmul	22	8	0.335

Table 5.8.: Benchmark results with respect to MAC operations for the 32-bit matrix multiplication kernel.

- we exploited compiler intrinsics for 8-bit SIMD operations, for which code generation is not supported; note that we also had to use some *shuffle* operations to transpose the chunk of the input matrix B to correctly organize the SIMD packed registers for the dot-products with the input chunk of matrix A;
- since we employed 8-bit SIMD, a further possible level of unrolling was exposed, computing a 2×4 chunk of the output matrix for each complete run of the hot-loop and 4 MACs for each output element during each hot-loop iteration;
- the further level of unrolling increased the registers occupation, which we reduced by switching from array indexing to pointer incrementing to avoid a massive increase in stack accesses.

With the described optimizations made possible by Xpulpimg, the Xpulpimg version of the *matmul* hot-loop loads 24 elements (a 2×4 chunk from matrix A and a 4×4 chunk from matrix B) with 6 load-word operations (since 4 sequential byte elements can be loaded with a single load-word), transpose the chunk of matrix B and performs 32 MACs with 8 dot-product instructions.

The results of the kernel benchmark averaged on the 16 cores are reported in Table 5.10; Table 5.11 reports the relative performance measured in terms of MAC operations.

With SIMD, four 8-bit elements can be loaded with a single load-word instruction, greatly reducing the number of memory accesses per useful operation; post-increment also reduces the number of instructions needed for pointers management. An additional overhead due to the shuffle operations is present for the need of transposing at runtime the chunk extracted from the second matrix operand. Still, 18 instructions (6 loads, 4 moves, 8 shuffles) are issued to load 24 elements, with respect to the 13 instructions needed to load 8 elements in the baseline kernel. Finally, dot-product operations perform 4 MACs per instruction, accounting for a total of 32 MACs per iteration. With the described improvements, the 8-bit *matmul* proved to be **4.6× faster** on the Xpulpimg version of the MemPool CC with respect to the RV32IMA baseline, as shown in Fig. 5.5.

The speed-up determined by Xpulpimg is far above the 8-bit SIMD 4× speed-up theoretical limit; this is caused by the better unrolling that the use of SIMD itself made possible. The reduction of the hot-loop size due to the absence of the explicit address increments also contributes to such a speed-up. A significant improvement of 5× can be also observed in the number of load memory accesses, mainly due to the possibility of loading multiple elements with a single instructions; this means that SIMD also relieves the memory hierarchy from some of the pressure caused by DSP algorithms, allowing better timing performance thanks to reduced congestions and memory stalls, but also better power consumption.

Baseline	Xpulpimg
`lb a0,-1(s1)`	`p.lw a3,a4(a1!)`
`lb a3,0(s3)`	`p.lw a5,s11(a1!)`
`lb t2,1(s3)`	`p.lw a2,s10(a0!)`
`lb a5,0(s0)`	`p.lw t4,s10(a0!)`
`lb t4,0(s1)`	`p.lw t3,s10(a0!)`
`lb a6,0(s2)`	`p.lw t1,s10(a0!)`
`lb a2,1(s2)`	`mv a6,a2`
`lb t1,1(s0)`	`pv.shuffle2.b a6,t4,s7`
`mul t0,a0,a3`	`mv a7,t3`
`mul a3,a3,a5`	`pv.shuffle2.b a7,t1,s7`
`mul a0,a0,t2`	`pv.shuffle2.b t3,t1,s6`
`mul a5,t2,a5`	`pv.shuffle2.b a2,t4,s6`
`add t5,t0,t5`	`mv t1,a6`
`add t6,a0,t6`	`pv.shuffle2.b t1,a7,s5`
`mul t0,t4,a6`	`pv.shuffle2.b a6,a7,s4`
`add t3,a3,t3`	`mv a7,a2`
`mul t4,t4,a2`	`pv.shuffle2.b a7,t3,s5`
`mul a6,a6,t1`	`pv.shuffle2.b a2,t3,s4`
`add a7,a5,a7`	`pv.sdotsp.b t5,a3,t1`
`mul a2,a2,t1`	`pv.sdotsp.b t6,a3,a6`
`addi s4,s4,2`	`pv.sdotsp.b t0,a3,a7`
`add t5,t0,t5`	`pv.sdotsp.b t2,a3,a2`
`add t6,t4,t6`	`pv.sdotsp.b s0,a5,t1`
`add t3,a6,t3`	`pv.sdotsp.b s1,a5,a6`
`add a7,a2,a7`	`pv.sdotsp.b s2,a5,a7`
`addi s0,s0,2`	`pv.sdotsp.b s3,a5,a2`
`add s3,s3,s5`	`bltu a1,s8,loop`
`add s2,s2,s5`	
`addi s1,s1,2`	
`bltu s4,a4,loop`	

Table 5.9.: Assembly code improvement of the 8-bit matrix multiplication hot-loop from the baseline to the Xpulpimg version; the instructions introduced by Xpulpimg are highlighted in blue.

Benchmark	IPC	Cycles	Issues	Loads	Stores	Stalls
Baseline matmul	0.93	67988	63459	16639	325	269
Xpulpimg matmul	0.97	15594	15070	3320	281	226

Table 5.10.: Absolute benchmark figures averaged on the 16 simulated cores for the 8-bit matrix multiplication kernel.

Benchmark	Hot-loop size	MACs/iteration	MACs/cycle
Baseline matmul	30	8	0.248
Xpulpimg matmul	27	32	1.15

Table 5.11.: Benchmark results with respect to MAC operations for the 8-bit matrix multiplication kernel.

5.4. Tile-level synthesis results

To characterize the impact of the Xpulpimg implementation from the broader perspective of the MemPool cluster, we analyzed its implementation cost in a MemPool tile. In this section, we illustrate the post-synthesis area figures of the MemPool CCs collected from the synthesis of a tile.

Targeting 500 MHz, worst-case conditions (SS, 0.72 V, 125 °C) and a configuration of 16-core for the MemPool system, we measured a post-synthesis area of the MemPool tile of 740.6 kGE after Xpulpimg implementation, with respect to the 673.1 kGE of the baseline version, accounting for an increase of 67.5 kGE (10%). With the MemPool CC replicated 4 times in each tile, the main drivers of such an increase are the Snitch core and its coprocessor, with their averaged area figures summarized in Table 5.12 and Fig. 5.6: while Snitch area increased of about 4 kGE on average (15.1%), the area of the coprocessor approximately doubled, with an increase of 13 kGE, implementing most of the Xpulpimg instruction set.

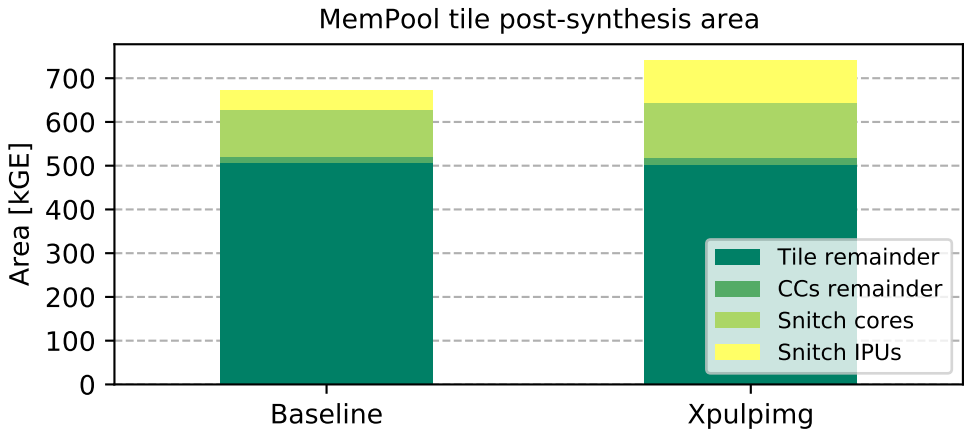


Figure 5.6.: Breakdown of MemPool tile post-synthesis area figures in its baseline and Xpulpimg versions.

	MemPool tile	MemPool CC	Snitch core	Snitch IPU
Baseline	673.1 kGE	41.7 kGE	27.2 kGE	11.2 kGE
Xpulpimg	740.6 kGE	59.7 kGE	31.3 kGE	24.2 kGE

Table 5.12.: MemPool tile post-synthesis area figures at 500 MHz in the worst-case corner; the areas regarding the MemPool CC and its sub-modules are averaged over its 4 replicas.

5.5. Power analysis

In this section, we analyze the power consumption of a 16-core MemPool cluster from the point of view of a tile synthesized in the worst-case corner (SS, 0.72 V, 125 °C), comparing its baseline and Xpulpimg implementations. We executed the power analysis running the 8-bit *matmul* kernel described in Section 5.3.2 at 500 MHz, in typical conditions (TT, 0.80 V, 25 °C).

Each tile consumes, on average, 20.9 mW, about 2.2 mW more than the baseline (18.7 mW), with an increase of 11.8%. The power consumption of the four Snitch cores in the tile does not suffer from the Xpulpimg introduction, staying around 6.8 mW (32.5% of tile consumption), as well as their integer register files, consuming overall about 3.5 mW (51.5% of the cores consumption).

The four Snitch IPU coprocessors in the tile, on the other hand, consume 2.82 mW (13.5% of tile consumption), 3× more than the baseline (0.93 mW), for which they only accounted for the 5% of the consumption. In particular, the DSP units of the coprocessors account for 2.52 mW in total, the 89.4% of the coprocessor power consumption.

The post-synthesis power figures at tile level are summed up in Table 5.13, with a graphical power breakdown shown in Fig. 5.7.

In terms of energy efficiency measured with respect to the overall number of multipli-

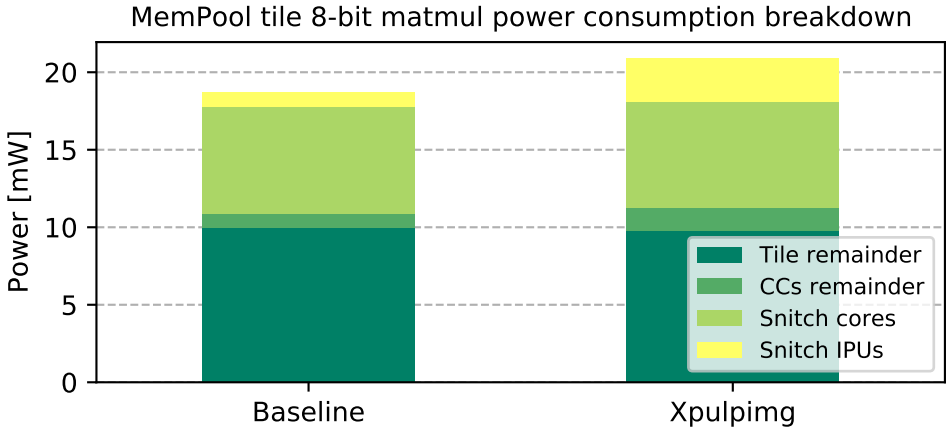


Figure 5.7.: Breakdown of MemPool tile post-synthesis power figures in its baseline and Xpulpimg versions.

	MemPool tile	Snitch cores	Snitch IPUs
Baseline	18.7 mW	6.86 mW (36.7%)	0.93 mW (5%)
Xpulpimg	20.9 mW	6.79 mW (32.5%)	2.82 mW (13.5%)

Table 5.13.: MemPool tile post-synthesis power analysis running *matmul* at 500 MHz in the typical corner.

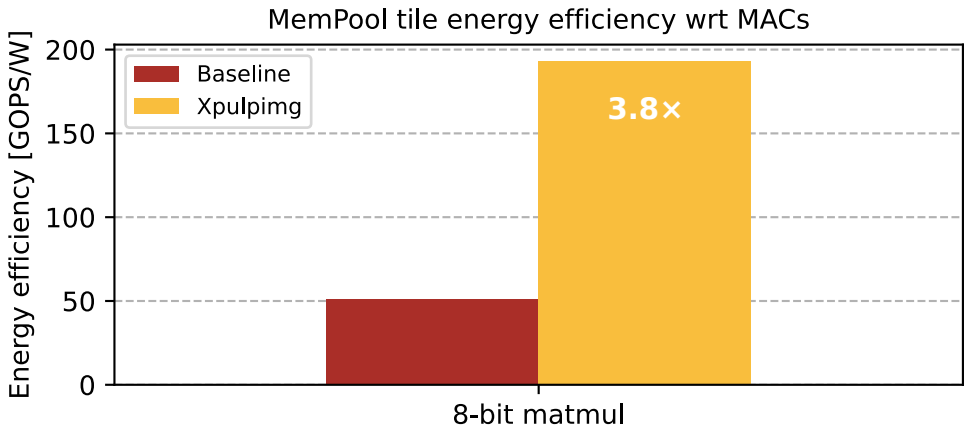


Figure 5.8.: Energy efficiency of the MemPool tile measured with respect the multiplications and additions operations performed in the 8-bit *matmul* kernel.

cation and addition operations (fused under single MACs in the Xpulpimg version) in the *matmul* kernel, we estimated a figure of 193.3 GOPS/W for the Xpulpimg version, with a baseline of 51.2 GOPS/W, as shown in Fig. 5.8. The overall effect of the Xpulpimg extension is a substantial increase in the throughput of DSP algorithms. Hence, at the cost of a slightly higher power consumption, the benchmarked matrix multiplication resulted in a much shorter execution time. We can thus conclude an increase in the energy efficiency of the MemPool tile of $\mathbf{3.8}\times$ with respect to the RV32IMA baseline.

Chapter 6

Conclusion and Future Work

We presented Xpulpimg, a custom subset of the RISC-V open ISA including domain-specific DSP instructions selected for image processing purposes. We implemented Xpulpimg in the MemPool CC, extending the Snitch processor core and its accelerator, the Snitch IPU, with post-increment and register-register addressing modes for load and store instructions, multiply-accumulate instructions, generic arithmetic operations for DSP and packed-SIMD operations with 8 and 16 bits, particularly targeting dot-product instructions.

We also established an open and modular environment for a full Xpulpimg support, providing tools for instructions encoding management, instruction set simulation, software compilation and unit test verification. With such tools, we also defined a framework for a convenient and standardized approach for further extensions of Xpulpimg.

Snitch is a tiny integer core developed with the idea to optimize its area occupation while targeting an high IPC by employing a simple scoreboarding mechanism for outstanding transactions and out-of-order write-back; the MemPool system, which massively replicates Snitch hundreds of times, benefits from such a small control area overhead. We aimed to exploit this concept extending it to the DSP domain, to gather the most out of MemPool in the domain of image processing. For the same reason, we synthesized the MemPool CC extended with Xpulpimg for the modern GlobalFoundries' 22FDX FD-SOI technology, looking for the Pareto-optimal curve in terms of area and timing.

We compared post-synthesis MemPool CC figures with a baseline version implementing the RV32IMA ISA. With an average area increase of 40.4% for the MemPool CC synthesized in typical conditions in a sweep from 500 MHz to 2 GHz, the maximum achievable operating frequency of the design worsened of just 3.6%, decreasing from 1.85 GHz to 1.75 GHz. At the 500 MHz target frequency of MemPool, we measured an area of 740.6 kGE for the tile synthesized in worst-case conditions, with an increase of 10% with respect to the baseline.

In terms of benchmarks, the cluster extended with Xpulpimg showed a 2.7× speed-up for the 8-bit 2D convolution kernel, extremely common in the image processing

domain, and a speed-up of up to 4.6× for the matrix multiplication. We also measured a substantial increase of up to 3.8× in terms of energy efficiency with respect to the baseline, with an overall increase in the power consumption of a tile of 2.2 mW running at 500 MHz in typical conditions.

Xpulpimg only represent the basic instruction set to run image processing and DSP algorithms with higher efficiency. Further work concerning Xpulpimg may include the introduction of additional instructions useful for image processing, leveraging the already developed framework. As a matter of fact, Xpulpimg currently includes only instructions selected from the Xpulp custom extension. While several other Xpulp instructions may be evaluated (hardware loops, fixed-point instructions, additional SIMD support instructions), custom instructions may help to find even better trade-offs between performance and implementation cost. Domain-specific image processing algorithms, as well as other ISPs, may be taken as sources of inspiration for additional extensions.

Further work in a more generic scope may also go towards further benchmarking of the MemPool CC extended with Xpulpimg DSP functionalities, in comparison with the CV32E40P core. Such an analysis may be useful to study the use of Snitch in PULP architectures, along with its effects on area, timing performance and energy efficiency.

Appendix A

Xpulpimg Instruction Set

This appendix presents the full Xpulpimg instruction set extension. In the following, the mnemonics of the instructions are reported, along with their behavior and the specification of the employed operands fields from the instruction encoding.

This documentation is inspired from the CV32E40P core ISA specification [20] and from its reference RTL implementation [22]; for further details about the instructions and their encoding, refer to the mentioned sources.

The operands from the standard RISC-V ISA, used throughout the Xpulpimg extension, are specified in the following:

- `rs1` – first source register operand, whose index is encoded in the bits `19:15`;
- `rs2` – second source register operand, whose index is encoded in the bits `24:20`;
- `rd` – destination register operand, whose index is encoded in the bits `11:7`;
- `bimm12s` – 12-bit branch offset, encoded in the bits `31`, `7`, `30:25`, `11:8` and sign-extended;
- `iimm12s` – 12-bit I-type immediate, encoded in the bits `31:20` and sign-extended;
- `simm12s` – 12-bit S-type immediate, encoded in the bits `31:25`, `11:7` and sign-extended;

A.1. Generic arithmetic operations

This extension includes advanced arithmetic and logic operations to increase the efficiency of the ISA by reducing the number of instructions needed for generally useful DSP computation, such as minimum, maximum, absolute value, sign- or zero-extension. Such operations performed in one single instructions also reduce the need of employing control flow instructions, leading to higher IPC. A partial support for fixed-point operations is also included by supporting clip operations. Immediate branching instructions, allowing a comparison between a register and an immediate operands, are also supported.

General ALU instructions

`p.abs rd, rs1`	`rd` = *abs*(`rs1`)
`p.slet rd, rs1, rs2`	`rd` = `rs1` $\leq$ `rs2` ? 1 : 0 Comparison is signed
`p.sletu rd, rs1, rs2`	`rd` = `rs1` $\leq$ `rs2` ? 1 : 0 Comparison is unsigned
`p.min rd, rs1, rs2`	`rd` = `rs1` $\leq$ `rs2` ? `rs1` : `rs2` Comparison is signed
`p.minu rd, rs1, rs2`	`rd` = `rs1` $\leq$ `rs2` ? `rs1` : `rs2` Comparison is unsigned
`p.max rd, rs1, rs2`	`rd` = `rs1` $>$ `rs2` ? `rs1` : `rs2` Comparison is signed
`p.maxu rd, rs1, rs2`	`rd` = `rs1` $>$ `rs2` ? `rs1` : `rs2` Comparison is unsigned
`p.exths rd, rs1`	`rd` = *Sext*(`rs1[15:0]`)
`p.exthz rd, rs1`	`rd` = *Zext*(`rs1[15:0]`)
`p.extbs rd, rs1`	`rd` = *Sext*(`rs1[7:0]`)
`p.extbz rd, rs1`	`rd` = *Zext*(`rs1[7:0]`)

Clip instructions

New operand fields:

- `pimm5u` – Xpulp 5-bit immediate encoded in the bits `24:20` and zero-extended.

`p.clip rd, rs1, pimm5u`	If `rs1` $\leq -2^{\texttt{pimm5u}-1}$, `rd` = $-2^{\texttt{pimm5u}-1}$ else if `rs1` $\geq 2^{\texttt{pimm5u-1}} - 1$, `rd` = $2^{\texttt{pimm5u-1}} - 1$ else `rd` = `rs1` If `pimm5u` == 0, consider $-2^{\texttt{pimm5u}-1}$=-1, $2^{\texttt{pimm5u-1}} - 1$=0
`p.clipu rd, rs1, pimm5u`	If `rs1` ≤ 0, `rd` = 0 else if `rs1` $\geq 2^{\texttt{pimm5u-1}} - 1$, `rd` = $2^{\texttt{pimm5u-1}} - 1$ else `rd` = `rs1` If `pimm5u` == 0, consider $2^{\texttt{pimm5u-1}} - 1$=0
`p.clipr rd, rs1, rs2`	If `rs1` $\leq -$`rs2` $- 1$, `rd` = $-$`rs2` $- 1$ else if `rs1` $\geq$ `rs2`, `rd` = `rs2` else `rd` = `rs1`
`p.clipur rd, rs1, rs2`	If `rs1` ≤ 0, `rd` = 0 else if `rs1` $\geq$ `rs2`, `rd` = `rs2` else `rd` = `rs1`

Immediate branching

New operand fields:

- `pimm5s` – Xpulp 5-bit immediate encoded in the bits `24:20` and sign-extended.

The addition between the program counter and the (`bimm12s` << 1) offset is always signed.

`p.beqimm rs1, pimm5s, bimm12s`	If `rs1` == `pbimm5s`, PC = PC + (`bimm12s` << 1)
`p.bneimm rs1, pimm5s, bimm12s`	If `rs1` != `pbimm5s`, PC = PC + (`bimm12s` << 1)

A.2. Extended L/S addressing modes

This extension includes new addressing modes for the load and store operations of the standard RISC-V RV32I ISA. In particular, two new addressing modes are introduced:

- *post-increment* – the memory access is performed at the address specified by the base address only; the register containing the base address is then incremented of the specified offset and written back to the register file;
- *register-register* – the offset for the memory address is sourced from a register rather than an immediate; this addressing mode can be also coupled with the post-increment mode.

Load and store instructions with these new addressing modes comes in all the widths (byte, half-word, word) and signedness (signed and unsigned sub-word loads).

Load instructions

The addition between the `rs1` base address and the immediate or register offset is always signed.

`p.lb rd, iimm12s(rs1!)`	`rd` = *Sext*(Mem8[`rs1`]) `rs1` = `rs1` + `iimm12s`
`p.lbu rd, iimm12s(rs1!)`	`rd` = *Zext*(Mem8[`rs1`]) `rs1` = `rs1` + `iimm12s`
`p.lh rd, iimm12s(rs1!)`	`rd` = *Sext*(Mem16[`rs1`]) `rs1` = `rs1` + `iimm12s`
`p.lhu rd, iimm12s(rs1!)`	`rd` = *Zext*(Mem16[`rs1`]) `rs1` = `rs1` + `iimm12s`
`p.lw rd, iimm12s(rs1!)`	`rd` = Mem32[`rs1`] `rs1` = `rs1` + `iimm12s`
`p.lb rd, rs2(rs1!)`	`rd` = *Sext*(Mem8[`rs1`]) `rs1` = `rs1` + `rs2`
`p.lbu rd, rs2(rs1!)`	`rd` = *Zext*(Mem8[`rs1`]) `rs1` = `rs1` + `rs2`
`p.lh rd, rs2(rs1!)`	`rd` = *Sext*(Mem16[`rs1`]) `rs1` = `rs1` + `rs2`
`p.lhu rd, rs2(rs1!)`	`rd` = *Zext*(Mem16[`rs1`]) `rs1` = `rs1` + `rs2`

`p.lw rd, rs2(rs1!)`	`rd` = Mem32[`rs1`] `rs1` = `rs1` + `rs2`
`p.lb rd, rs2(rs1)`	`rd` = *Sext*(Mem8[`rs1` + `rs2`])
`p.lbu rd, rs2(rs1)`	`rd` = *Zext*(Mem8[`rs1` + `rs2`])
`p.lh rd, rs2(rs1)`	`rd` = *Sext*(Mem16[`rs1` + `rs2`])
`p.lhu rd, rs2(rs1)`	`rd` = *Zext*(Mem16[`rs1` + `rs2`])
`p.lw rd, rs2(rs1)`	`rd` = Mem32[`rs1` + `rs2`]

Store instructions

New operand fields:

- `rs3` – third source register operand, whose index is encoded in the bits `11:7`.

The addition between the `rs1` base address and the immediate or register offset is always signed.

`p.sb rs2, simm12s(rs1!)`	Mem8[`rs1`] = `rs2` `rs1` = `rs1` + `simm12s`
`p.sh rs2, simm12s(rs1!)`	Mem16[`rs1`] = `rs2` `rs1` = `rs1` + `simm12s`
`p.sw rs2, simm12s(rs1!)`	Mem32[`rs1`] = `rs2` `rs1` = `rs1` + `simm12s`
`p.sb rs2, rs3(rs1!)`	Mem8[`rs1`] = `rs2` `rs1` = `rs1` + `rs3`
`p.sh rs2, rs3(rs1!)`	Mem16[`rs1`] = `rs2` `rs1` = `rs1` + `rs3`
`p.sw rs2, rs3(rs1!)`	Mem32[`rs1`] = `rs2` `rs1` = `rs1` + `rs3`
`p.sb rs2, rs3(rs1)`	Mem8[`rs1` + `rs3`] = `rs2`
`p.sh rs2, rs3(rs1)`	Mem16[`rs1` + `rs3`] = `rs2`
`p.sw rs2, rs3(rs1)`	Mem32[`rs1` + `rs3`] = `rs2`

A.3. MAC operations

This extension includes 32-bit multiply-accumulate operations. Not only do these instruction use the destination register to write-back the result, but they also use it as a source operand to perform the accumulation operation. This operation can be an addition of the multiplication result to the content of the destination register, or a subtraction from it.

Multiply-accumulate instruction

`p.mac rd, rs1, rs2`	$rd = rd + rs1 \cdot rs2$
`p.msu rd, rs1, rs2`	$rd = rd - rs1 \cdot rs2$

A.4. Packed-SIMD extension

This extension introduces packed-SIMD operations for sub-words of:

- 8 bits (`.b` mode) – to perform 4 operations on the 4 bytes of a 32-bit word at the same time;
- 16 bits (`.h` mode) – to perform 2 operations on the 2 half-words of a 32-bit word at the same time.

SIMD instructions comes with three execution modes affecting the second operand:

- *vectorial* (default mode) – the two input registers `rs1` and `rs2` are considered vectors of two 16-bit or four 8-bit elements, and an element-wise operation is performed;
- *scalar replication* (`.sc` mode) – the lowest half-word or byte of `rs2` is considered a scalar and used for the operation with the array of half-words or bytes in `rs1`;
- *immediate scalar replication* (`.sci` mode) – this variation has the same behavior of the scalar replication mode but uses the 6-bit signed or unsigned immediate as scalar operand.

Note that SIMD is not supported by the compiler toolchain; the compiler only provides SIMD intrinsics.

New operand fields:

- `pimm6s` – Xpulp 6-bit immediate encoded in the bits `25:20` and sign-extended to 8 or 16 bits;
- `pimm6u` – Xpulp 6-bit immediate encoded in the bits `25:20` and zero-extended to 8 or 16 bits.

In the following tables, the second operand is referred to as `op2`, and it varies based on the mentioned execution modes. Also, the indices of the packed-SIMD arrays `rd`, `rs1` and `op2` range from `0` to `1` fro 16-bit operations and from `0` to `3` for 8-bit operations:

- index = `0` – stands for bits `15:0` for 16-bit operations, or bits `7:0` for 8-bit operations;
- index = `1` – stands for bits `31:16` for 16-bit operations, or bits `15:8` for 8-bit operations;
- index = `2` – stands for bits `23:16` for 8-bit operations;
- index = `3` – stands for bits `31:24` for 8-bit operations;

SIMD ALU instructions

`pv.add[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` + `op2[i]`
`pv.sub[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` − `op2[i]`
`pv.avg[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = (`rs1[i]` + `op2[i]`) >> 1 Shift is arithmetic
`pv.avgu[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd[i]` = (`rs1[i]` + `op2[i]`) >> 1 Shift is logical
`pv.min[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` ≤ `op2[i]` ? `rs1[i]` : `op2[i]` Comparison is signed
`pv.minu[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd[i]` = `rs1[i]` ≤ `op2[i]` ? `rs1[i]` : `op2[i]` Comparison is unsigned
`pv.max[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` > `op2[i]` ? `rs1[i]` : `op2[i]` Comparison is signed
`pv.maxu[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd[i]` = `rs1[i]` > `op2[i]` ? `rs1[i]` : `op2[i]` Comparison is unsigned
`pv.srl[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` >> `op2[i]` Shift is logical
`pv.sra[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` >> `op2[i]` Shift is arithmetic
`pv.sll[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` << `op2[i]` Shift is logical
`pv.or[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` | `op2[i]`
`pv.xor[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` ^`op2[i]`
`pv.and[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` & `op2[i]`
`pv.abs.{h,b} rd, rs1`	`rd[i]` = `rs1[i]` ≤ 0 ? −`rs1[i]` : `rs1[i]` Comparison is signed

SIMD Dot-product instructions

`pv.dotup[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd` = $\sum_i$ `rs1[i]` · `op2[i]` All operations are unsigned
`pv.dotusp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd` = $\sum_i$ `rs1[i]` · `op2[i]` `rs1[i]` are treated as unsigned, `op2[i]` are treated as signed
`pv.dotsp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd` = $\sum_i$ `rs1[i]` · `op2[i]` All operations are signed
`pv.sdotup[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd = rd +` $\sum_i$ `rs1[i]` · `op2[i]` All operations are unsigned
`pv.sdotusp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd = rd +` $\sum_i$ `rs1[i]` · `op2[i]` `rs1[i]` are treated as unsigned, `op2[i]` are treated as signed
`pv.sdotsp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd = rd +` $\sum_i$ `rs1[i]` · `op2[i]` All operations are signed

SIMD support instructions

`pv.extract.h rd, rs1, pimm6s`	`rd` = *Sext*(`rs1[i]`) With `i` = `pimm6u[0]`
`pv.extract.b rd, rs1, pimm6s`	`rd` = *Sext*(`rs1[i]`) With `i` = `pimm6u[1:0]`
`pv.extractu.h rd, rs1, pimm6s`	`rd` = *Zext*(`rs1[i]`) With `i` = `pimm6u[0]`
`pv.extractu.b rd, rs1, pimm6s`	`rd` = *Zext*(`rs1[i]`) With `i` = `pimm6u[1:0]`
`pv.insert.h rd, rs1, pimm6s`	`rd[i]` = `rs1[15:0]` With `i` = `pimm6u[0]`; the rest of the bits of `rd` are untouched

`pv.insert.b rd, rs1, pimm6s`	`rd[i]` = `rs1[7:0]` With `i` = `pimm6u[1:0]`; the rest of the bits of `rd` are untouched
`pv.shuffle2.h rd, rs1, rs2`	`rd[i]` = `src[j]` Where `src` = `rs2[i][1]` == 1 ? `rs1` : `rd`, while `j` = rs2[i][0]
`pv.shuffle2.b rd, rs1, rs2`	`rd[i]` = `src[j]` Where `src` = `rs2[i][2]` == 1 ? `rs1` : `rd`, while `j` = rs2[i][1:0]

Appendix B

2D Convolution Algorithm

In the following, the developed C algorithm for the 8-bit 3×3 2D convolution is presented in its two flavors: for the RV32IMA basic instruction set and optimized for Xpulpimg.

The input image is `in`, an 8-bit signed `in_y` $\times$ `in_x` matrix. It gets convolved with an input 3×3 kernel `k` of 8-bit unsigned elements, to give as output `out`, a 32-bit signed `in_y` $\times$ `in_x` matrix.

Listing B.1: Single-core 2D convolution kernel optimized for the baseline version of MemPool. This algorithm initially loads a full 3×3 window of the input matrix; the inner loop convolves the current window with the kernel, subsequently loading the following 3-element row and shifting down the window. The inner loop is iterated until the column is fully convolved, moving then to the following 3-element column.

```
void conv2d_3x3_unrolled_i8_rv32ima(
    int8_t const volatile *__restrict__ in,
    uint32_t in_x, uint32_t in_y,
    uint8_t const volatile *__restrict__ k,
    int32_t volatile *__restrict__ out
) {
  int32_t sum;
  uint32_t weight = 0;
  uint32_t i, j; // loop counters

  for (int i = 0; i < 9; ++i)
    weight += k[i];

  // for every column but first and last
  for (i = 1; i < in_x - 1; ++i) {
    // first row of the window
```

```
    int8_t elem_00 = in[i - 1];
    int8_t elem_01 = in[i + 0];
    int8_t elem_02 = in[i + 1];
    // second row of the window
    int8_t elem_10 = in[in_x + (i - 1)];
    int8_t elem_11 = in[in_x + (i + 0)];
    int8_t elem_12 = in[in_x + (i + 1)];
    // third row of the window
    int8_t elem_20 = in[2 * in_x + (i - 1)];
    int8_t elem_21 = in[2 * in_x + (i + 0)];
    int8_t elem_22 = in[2 * in_x + (i + 1)];

    // for every row but first and last
    for (j = 1; j < in_y - 1; j++) {
      // element-wise mul with acc between input window and kernel
      sum = 0;
      sum += elem_00*k[0]; sum += elem_01*k[1]; sum += elem_02*k[2];
      sum += elem_10*k[3]; sum += elem_11*k[4]; sum += elem_12*k[5];
      sum += elem_20 * k[6]; sum += elem_21*k[7]; sum += elem_22*k[8];

      // move window down of one row
      elem_00 = elem_10; elem_01 = elem_11; elem_02 = elem_12;
      elem_10 = elem_20; elem_11 = elem_21; elem_12 = elem_22;
      // load the new third row of the window
      elem_20 = in[(j + 2) * in_x + (i - 1)];
      elem_21 = in[(j + 2) * in_x + (i + 0)];
      elem_22 = in[(j + 2) * in_x + (i + 1)];

      out[j * in_x + i] = sum / weight; // store output
    }
  }
}
```

Listing B.2: Single-core 2D convolution kernel from Listing B.1 optimized for Xpulpimg by means of SIMD data structures and compiler intrinsics. `v4s` and `v4u` are the vectorized 32-bit data types to hold 4 independent signed or unsigned bytes and perform 8-bit packed-SIMD operations by means of SIMD compiler intrinsics, such as `__builtin_pulp_dotusp4` and `__builtin_pulp_sdotusp4`.

```
typedef signed char v4s __attribute__((vector_size (4)));
typedef unsigned char v4u __attribute__((vector_size (4)));

void conv2d_3x3_unrolled_i8_xpulpimg(
```

```
    int8_t const volatile *__restrict__ in,
    int32_t volatile *__restrict__ out,
    uint32_t in_y, uint32_t in_x,
    uint8_t const volatile *__restrict__ k
){
  int32_t sum;
  uint32_t weight = 0;
  uint32_t j, i, t;

  for (int i = 0; i < 9; ++i)
    weight += k[i];

  v4u coeff_0 = (v4u){k[0], k[1], k[2], 0};
  v4u coeff_1 = (v4u){k[3], k[4], k[5], 0};
  v4u coeff_2 = (v4u){k[6], k[7], k[8], 0};

  // for every column but first and last
  for (i = 1; i < in_x - 1; i++) {
    // load window
    v4s elem_0 = (v4s){in[i-1], in[i], in[i+1], 0};
    v4s elem_1 = (v4s){in[i-1+in_y], in[i+in_y], in[i+1+in_y], 0};
    v4s elem_2 = (v4s){
      in[i-1+in_y*2], in[i+in_y*2], in[i+1+in_y*2], 0
    };

    // for every row but first and last
    for (j = 1; j < in_y - 1; j++) {
      t = j * in_y + i; // output matrix index
      // element-wise multiply with accumulation using intrinsics
      sum = __builtin_pulp_dotusp4(coeff_0, elem_0);
      sum = __builtin_pulp_sdotusp4(coeff_1, elem_1, sum);
      sum = __builtin_pulp_sdotusp4(coeff_2, elem_2, sum);
      out[t] = sum/weight; // store output

      // load a new row
      v4s new_data = (v4s){
        in[(j+2)*in_y+i-1], in[(j+2)*in_y+i], in[(j+2)*in_y+i+1], 0
      };
      // move window down of one row
      elem_0 = elem_1; elem_1 = elem_2; elem_2 = new_data;
    }
  }
}
```

Appendix C

Matrix Multiplication Algorithm

In the following, the developed C algorithms for the 32-bit and 8-bit flavors of the matrix multiplication kernel are presented, both in the baseline RV32IMA versions and optimized for Xpulpimg.

The algorithm performs the matrix multiplication between the two 32-bit or 8-bit input matrices `A`, with dimensions `M`×`N`, and `B`, with dimensions `N`×`P`. The output is stored in the 32-bit `M`×`P` matrix `C`. The core executing the function has an ID identified by the input `id`, with the computation equally distributed on `numThreads` cores.

Listing C.1: Multi-core 32-bit matrix multiplication kernel optimized for the baseline version of MemPool. The computation of the output matrix is equally distributed over all the available cores, basing on their IDs. The outer loop is unrolled to compute a 2×2 chunk of the `C` matrix at each iteration; each element of `C` corresponds to the dot-product of a row from `A` and a column from `B`: the inner loop is unrolled to compute and accumulate, at each iteration, two terms of the dot-product sum for each element of the 2×2 chunk of `C`.

```
void matmul_unrolled_2x2_parallel_i32_rv32ima(
    int32_t const *__restrict__ A,
    int32_t const *__restrict__ B,
    int32_t *__restrict__ C,
    uint32_t M, uint32_t N, uint32_t P,
    uint32_t id, uint32_t numThreads
) {
  // parallelize by assigning each core one row
  uint32_t const c = 8; // how many columns to split the matrix into
  uint32_t const c_start = (P / c) * (id % c);
  uint32_t const c_end = (P / c) * ((id % c) + 1);

  // for every couple of rows assigned to this core
```

```
  for (uint32_t i = 2 * (id / c); i < M; i += 2 * (numThreads / c)) {
    // for every couple of columns belonging to this chunk
    for (uint32_t j = c_start; j < c_end; j += 2) {
      // initialize accumulators
      int32_t c00 = 0;
      int32_t c01 = 0;
      int32_t c10 = 0;
      int32_t c11 = 0;

      // traverse the full rows from A and columns from B
      for (uint32_t k = 0; k < N; k += 2) {
        // explicitly load the values first to help with scheduling
        int32_t val_a00 = A[(i + 0) * N + k + 0];
        int32_t val_a01 = A[(i + 0) * N + k + 1];
        int32_t val_a10 = A[(i + 1) * N + k + 0];
        int32_t val_a11 = A[(i + 1) * N + k + 1];
        int32_t val_b00 = B[(k + 0) * P + j + 0];
        int32_t val_b01 = B[(k + 0) * P + j + 1];
        int32_t val_b10 = B[(k + 1) * P + j + 0];
        int32_t val_b11 = B[(k + 1) * P + j + 1];

        // perform 2 MACs for each element of the 2x2 output chunk
        c00 += val_a00 * val_b00;
        c00 += val_a01 * val_b10;
        c01 += val_a00 * val_b01;
        c01 += val_a01 * val_b11;
        c10 += val_a10 * val_b00;
        c10 += val_a11 * val_b10;
        c11 += val_a10 * val_b01;
        c11 += val_a11 * val_b11;
      }
      // store outputs of the computed 2x2 chunk
      C[(i + 0) * P + j + 0] = c00;
      C[(i + 0) * P + j + 1] = c01;
      C[(i + 1) * P + j + 0] = c10;
      C[(i + 1) * P + j + 1] = c11;
    }
  }
}
```

Listing C.2: Multi-core 32-bit matrix multiplication kernel from Listing C.1 optimized for the Xpulpimg-extended ISA. The level of unrolling is the same of the baseline version, but the load and store memory accesses are manually optimized at assembly-level for an optimal register allocation.

```
void matmul_unrolled_2x2_parallel_i32_xpulpimg(
    int32_t const *__restrict__ A,
    int32_t const *__restrict__ B,
    int32_t *__restrict__ C,
    uint32_t M, uint32_t N, uint32_t P,
    uint32_t id, uint32_t numThreads
) {
  // parallelize by assigning each core one row
  uint32_t const c = 8; // how many columns to split the matrix into
  uint32_t const c_start = (P / c) * (id % c);
  uint32_t const c_end = (P / c) * ((id % c) + 1);

  // A and B matrix memory addresses increments
  uint32_t const A_incr = (N * sizeof(int32_t)) - sizeof(int32_t);
  uint32_t const B_incr = (P * sizeof(int32_t)) - sizeof(int32_t);

  // for every couple of rows assigned to this core
  for (uint32_t i = 2 * (id / c); i < M; i += 2 * (numThreads / c)) {
    // for every couple of columns belonging to this chunk
    for (uint32_t j = c_start; j < c_end; j += 2) {
      // initialize accumulators
      int32_t c00 = 0;
      int32_t c01 = 0;
      int32_t c10 = 0;
      int32_t c11 = 0;

      // traverse the full rows from A and columns from B
      for (uint32_t k = 0; k < N; k += 2) {
        int32_t *idx_a = &A[i * N + k];
        int32_t *idx_b = &B[k * P + j];
        int32_t val_a00, val_a01, val_a10, val_a11
        int32_t val_b00, val_b01, val_b10, val_b11;
        // explicit asm loads of A and B elements
        asm volatile(
          "p.lw␣%[a00],␣4(%[addr_a]!)␣\n\t"
          "p.lw␣%[a01],␣%[a_incr](%[addr_a]!)␣\n\t"
          "p.lw␣%[a10],␣4(%[addr_a]!)␣\n\t"
          "p.lw␣%[a11],␣0(%[addr_a])␣\n\t"
          "p.lw␣%[b00],␣4(%[addr_b]!)␣\n\t"
```

```
          "p.lw␣%[b01],␣%[b_incr](%[addr_b]!)␣\n\t"
          "p.lw␣%[b10],␣4(%[addr_b]!)␣\n\t"
          "p.lw␣%[b11],␣0(%[addr_b])␣\n\t"
          : [ a00 ] "=&r"(val_a00), [ a01 ] "=&r"(val_a01),
            [ a10 ] "=&r"(val_a10), [ a11 ] "=&r"(val_a11),
            [ b00 ] "=&r"(val_b00), [ b01 ] "=&r"(val_b01),
            [ b10 ] "=&r"(val_b10), [ b11 ] "=&r"(val_b11),
            [ addr_a ] "+&r"(idx_a), [ addr_b ] "+&r"(idx_b)
          : [ a_incr ] "r"(A_incr), [ b_incr ] "r"(B_incr)
          : "memory");

        // perform 2 MACs for each element of the 2x2 output chunk
        c00 += val_a00 * val_b00;
        c00 += val_a01 * val_b10;
        c01 += val_a00 * val_b01;
        c01 += val_a01 * val_b11;
        c10 += val_a10 * val_b00;
        c10 += val_a11 * val_b10;
        c11 += val_a10 * val_b01;
        c11 += val_a11 * val_b11;
      }
      int32_t *idx_c = &C[i * P + j];
      // explicit asm stores of C output
      asm volatile(
        "p.sw␣%[s00],␣4(%[addr_c]!)␣\n\t"
        "p.sw␣%[s01],␣%[c_incr](%[addr_c]!)␣\n\t"
        "p.sw␣%[s10],␣4(%[addr_c]!)␣\n\t"
        "p.sw␣%[s11],␣0(%[addr_c])␣\n\t"
        : [ addr_c ] "+&r"(idx_c)
        : [ s00 ] "r"(c00), [ s01 ] "r"(c01),
          [ s10 ] "r"(c10), [ s11 ] "r"(c11),
          [ c_incr ] "r"(B_incr)
        : "memory");
    }
  }
}
```

Listing C.3: Multi-core 8-bit matrix multiplication kernel optimized for the baseline version of MemPool. It has the same structure of Listing C.1, but accepts 8-bit input matrices.

```
void matmul_unrolled_2x2_parallel_i8_rv32ima(
    int8_t const *__restrict__ A,
    int8_t const *__restrict__ B,
    int32_t *__restrict__ C,
    uint32_t M, uint32_t N, uint32_t P,
    uint32_t id, uint32_t numThreads
) {
  // parallelize by assigning each core one row
  uint32_t const c = 8; // how many columns to split the matrix into
  uint32_t const c_start = (P / c) * (id % c);
  uint32_t const c_end = (P / c) * ((id % c) + 1);

  // for every couple of rows assigned to this core
  for (uint32_t i = 2 * (id / c); i < M; i += 2 * (numThreads / c)) {
    // for every couple of columns belonging to this chunk
    for (uint32_t j = c_start; j < c_end; j += 2) {
      // initialize accumulators
      int32_t c00 = 0;
      int32_t c01 = 0;
      int32_t c10 = 0;
      int32_t c11 = 0;

      // traverse the full rows from A and columns from B
      for (uint32_t k = 0; k < N; k += 2) {
        // explicitly load the values first to help with scheduling
        int8_t val_a00 = A[(i + 0) * N + k + 0];
        int8_t val_a01 = A[(i + 0) * N + k + 1];
        int8_t val_a10 = A[(i + 1) * N + k + 0];
        int8_t val_a11 = A[(i + 1) * N + k + 1];
        int8_t val_b00 = B[(k + 0) * P + j + 0];
        int8_t val_b01 = B[(k + 0) * P + j + 1];
        int8_t val_b10 = B[(k + 1) * P + j + 0];
        int8_t val_b11 = B[(k + 1) * P + j + 1];

        // perform 2 MACs for each element of the 2x2 output chunk
        c00 += val_a00 * val_b00;
        c00 += val_a01 * val_b10;
        c01 += val_a00 * val_b01;
        c01 += val_a01 * val_b11;
        c10 += val_a10 * val_b00;
```

```
        c10 += val_a11 * val_b10;
        c11 += val_a10 * val_b01;
        c11 += val_a11 * val_b11;
      }
      // store outputs of the computed 2x2 chunk
      C[(i + 0) * P + j + 0] = c00;
      C[(i + 0) * P + j + 1] = c01;
      C[(i + 1) * P + j + 0] = c10;
      C[(i + 1) * P + j + 1] = c11;
    }
  }
}
```

Listing C.4: Multi-core 8-bit matrix multiplication kernel from Listing C.3 optimized for the Xpulpimg-extended ISA. The main structure of the algorithm is inspired from the matrix multiplication from the PULP DSP library [36]: compiler intrinsics are used to perform 8-bit SIMD operations, which makes possible a further level of unrolling. Hence, the outer loop now computes a 2×4 chunk of the output matrix. The inner loop firstly performs a runtime transposition of the chunk from the B matrix, to correctly use the SIMD packed data, then it computes and accumulates four terms of the dot-product for each one of the element of the 2×4 output chunk. Additionally, we substituted array indexing with pointer incrementing and explicitly listed the load and store sequences in assembly, to obtain an optimal register utilization.

```
typedef signed char v4s __attribute__((vector_size (4)));

void matmul_unrolled_2x4_parallel_i8_xpulpimg(
    int8_t const *__restrict__ A,
    int8_t const *__restrict__ B,
    int32_t *__restrict__ C,
    uint32_t M, uint32_t N, uint32_t P,
    uint32_t id, uint32_t numThreads
) {
  // masks for shuffles
  static v4s mask0 = {0, 1, 4, 5};
  static v4s mask1 = {2, 3, 6, 7};
  static v4s mask2 = {0, 2, 4, 6};
  static v4s mask3 = {1, 3, 5, 7};

  uint32_t k = 0; // loop counter for P
  int32_t const N_decr = -N + 4; // row decrement for A matrix
  uint32_t const P_incr = (P * 4) - 12; // row increment for C matrix
```

```
// for every group of 4 columns assigned to this core
for (k = id; k < P / 4; k += numThreads) {
  int8_t *idx_a = &A[0];            // start_a
  int32_t *idx_c = &C[k * 4];       // start_c
  int32_t const *end_c = &C[P * M]; // actually (P * M) + (k * 4)
  // until the end of the 2x4 chunk assigned to this core is reached
  while (idx_c < end_c) {
    // initialize accumulators
    int32_t c00 = 0;
    int32_t c01 = 0;
    int32_t c02 = 0;
    int32_t c03 = 0;
    int32_t c10 = 0;
    int32_t c11 = 0;
    int32_t c12 = 0;
    int32_t c13 = 0;

    int8_t const *end_a = idx_a + N;
    int8_t *idx_b = &B[k * 4]; // start_b

    // until all the terms of the output dot-products are computed
    while (idx_a < end_a) {
      v4s aVec0, aVec1;
      v4s t0, t1, t2, t3;
      // explicit asm loads of A and B elements
      asm volatile(
        "p.lw␣%[a0],␣%[a_incr](%[addr_a]!)␣\n\t" // go to next row, \
            same column
        "p.lw␣%[a1],␣%[a_decr](%[addr_a]!)␣\n\t" // go to previous \
            row, one column forward
        "p.lw␣%[t0],␣%[b_incr](%[addr_b]!)␣\n\t"
        "p.lw␣%[t1],␣%[b_incr](%[addr_b]!)␣\n\t"
        "p.lw␣%[t2],␣%[b_incr](%[addr_b]!)␣\n\t"
        "p.lw␣%[t3],␣%[b_incr](%[addr_b]!)␣\n\t"
        : [ a0 ] "=&r"(aVec0), [ a1 ] "=&r"(aVec1), [ t0 ] "=&r"(t0),
          [ t1 ] "=&r"(t1), [ t2 ] "=&r"(t2), [ t3 ] "=&r"(t3),
          [ addr_a ] "+&r"(idx_a), [ addr_b ] "+&r"(idx_b)
        : [ a_incr ] "r"(N), [ a_decr ] "r"(N_decr), [ b_incr ] "r"(P\
            )
        : "memory");

      // transpose B chunk before multiplying with A chunk
```

```
        v4s t4 = __builtin_shuffle(t0, t1, mask0); // 0,1,4,5
        v4s t5 = __builtin_shuffle(t2, t3, mask0); // 8,9,12,13
        v4s t6 = __builtin_shuffle(t0, t1, mask1); // 2,3,6,7
        v4s t7 = __builtin_shuffle(t2, t3, mask1); // 3,7,11,15
        v4s bVec0 = __builtin_shuffle(t4, t5, mask2); // 0,4,8,12
        v4s bVec1 = __builtin_shuffle(t4, t5, mask3); // 1,5,9,13
        v4s bVec2 = __builtin_shuffle(t6, t7, mask2); // 2,6,10,14
        v4s bVec3 = __builtin_shuffle(t6, t7, mask3); // 3,7,11,15

        // perform 4 MACs for each element of the 2x4 output chunk
        c00 = __builtin_pulp_sdotsp4(aVec0, bVec0, c00);
        c01 = __builtin_pulp_sdotsp4(aVec0, bVec1, c01);
        c02 = __builtin_pulp_sdotsp4(aVec0, bVec2, c02);
        c03 = __builtin_pulp_sdotsp4(aVec0, bVec3, c03);
        c10 = __builtin_pulp_sdotsp4(aVec1, bVec0, c10);
        c11 = __builtin_pulp_sdotsp4(aVec1, bVec1, c11);
        c12 = __builtin_pulp_sdotsp4(aVec1, bVec2, c12);
        c13 = __builtin_pulp_sdotsp4(aVec1, bVec3, c13);
      }
      // explicit asm stores of C output
      asm volatile(
        "p.sw␣%[s00],␣4(%[addr_c]!)␣\n\t"
        "p.sw␣%[s01],␣4(%[addr_c]!)␣\n\t"
        "p.sw␣%[s02],␣4(%[addr_c]!)␣\n\t"
        "p.sw␣%[s03],␣%[c_incr](%[addr_c]!)␣\n\t"
        "p.sw␣%[s10],␣4(%[addr_c]!)␣\n\t"
        "p.sw␣%[s11],␣4(%[addr_c]!)␣\n\t"
        "p.sw␣%[s12],␣4(%[addr_c]!)␣\n\t"
        "p.sw␣%[s13],␣%[c_incr](%[addr_c]!)␣\n\t"
        : [ addr_c ] "+&r"(idx_c)
        : [ s00 ] "r"(c00), [ s01 ] "r"(c01), [ s02 ] "r"(c02),
          [ s03 ] "r"(c03), [ s10 ] "r"(c10), [ s11 ] "r"(c11),
          [ s12 ] "r"(c12), [ s13 ] "r"(c13), [ c_incr ] "r"(P_incr)
        : "memory");

      idx_a += N; // adjust A matrix pointer
    }
  }
}
```

List of Acronyms

ALU . . . . . .arithmetic logic unit

AT . . . . . .area-timing

CC . . . . . .core complex

CRF . . . . . .camera response function

CSR . . . . . .control and status register

DSP . . . . . .digital signal processing

DSPU . . . . . .digital signal processing unit

EOC . . . . . .end-of-computation

FD-SOI . . . . .Fully-Depleted Silicon-Over-Insulator

HDL . . . . . .hardware description language

HDR . . . . . .high dynamic range

IoT . . . . . .Internet of things

IPC . . . . . .instruction per cycle

IPU . . . . . .integer processing unit

ISA . . . . . .instruction set architecture

ISP . . . . . .image signal processor

LSU . . . . . .load-store unit

MAC . . . . . multiply-accumulate

MCU . . . . . microcontroller unit

MSB . . . . . most significant bit

MSU . . . . . multiply-subtract

NT . . . . . near-threshold

PC . . . . . program counter

PULP . . . . . Parallel Ultra-Low Power

RAW . . . . . read-after-write

RISC . . . . . reduced instruction set computer

RTL . . . . . register-transfer level

SIMD . . . . . single-instruction-multiple-data

SIMD . . . . . single instruction, multiple data

SPM . . . . . scratchpad memory

TCDM . . . . . tightly-coupled-data-memory

TVM . . . . . test virtual machine

WAW . . . . . write-after-write

List of Figures

List of Tables

Bibliography

[1] R. Thabet, R. Mahmoudi, and M. H. Bedoui, "Image processing on mobile devices: An overview," in *International Image Processing, Applications and Systems Conference*. IEEE, 2014, pp. 1–8.

[2] H. Ayed, J. Ermont, J.-l. Scharbarg, and C. Fraboul, "Towards a unified approach for worst-case analysis of Tilera-like and Kalray-like NoC architectures," in *2016 IEEE World Conference on Factory Communication Systems (WFCS)*. IEEE, 2016, pp. 1–4.

[3] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 Tensor Core GPU: Performance and Innovation," *IEEE Micro*, no. 01, pp. 1–1, 2021.

[4] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the Kalray MPPA-256 integrated manycore processor," *Procedia Computer Science*, vol. 18, pp. 1654–1663, 2013.

[5] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, 2009.

[6] J. Redgrave, A. Meixner, N. Goulding-Hotta, A. Vasilyev, and O. Shacham, "Pixel visual core: Google's fully programmable image vision and AI processor for mobile devices," in *Proc. IEEE Hot Chips Symp.(HCS)*, 2018, pp. 1–18.

[7] M. Cavalcante, S. Riedel, A. Pullini, and L. Benini, "MemPool: A shared-L1 memory many-core cluster with a low-latency Interconnect," *arXiv preprint arXiv:2012.02973*, 2020.

[8] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Transactions on Computers*, 2020.

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

[10] A. Waterman and K. Asanovic, "The RISC-V instruction set manual, volume I: Unprivileged ISA," *EECS Department, UC Berkeley*, 2019.

[11] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[12] A. S. Waterman, "Design of the RISC-V instruction set architecture," Ph.D. dissertation, UC Berkeley, 2016.

[13] A. Waterman and K. Asanovic, "The RISC-V instruction set manual, volume II: Privileged ISA," *EECS Department, UC Berkeley*, 2019.

[14] RISC-V. (2021) RISC-V opcodes. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/riscv/riscv-opcodes

[15] ——. (2021) RISC-V GNU toolchain. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain

[16] ——. (2021) RISC-V ISA simulator. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/riscv/riscv-isa-sim

[17] ——. (2021) RISC-V tests. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/riscv/riscv-tests

[18] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "A comparison between processor architectures for multimedia applications," in *Proc. 15th Annual Workshop on Circuits, System and Signal Processing (ProRISC 2004), the Netherlands*, 2004, pp. 138–152.

[19] RISC-V. (2021) RISC-V P specification. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/riscv/riscv-p-spec

[20] OpenHW Group. (2021) CV32E40P User Manual. Accessed March 2, 2021. [Online]. Available: https://cv32e40p.readthedocs.io/en/latest

[21] PULP Platform. (2020) PULP RISC-V GNU toolchain. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/pulp-platform/pulp-riscv-gnu-toolchain

[22] OpenHW Group. (2021) CV32E40P. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/openhwgroup/cv32e40p

[23] PULP Platform. (2021) PULPissimo. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/pulp-platform/pulpissimo

[24] ——. (2021) Snitch. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/pulp-platform/snitch

[25] R. Raskar and J. Tumblin, "Computational photography," in *ACM SIGGRAPH 2005 Courses*, 2005, pp. 1–es.

[26] M. Levoy. (2014) HDR+: Low Light and High Dynamic Range photography in the Google Camera App. Google AI Blog. Accessed March 2, 2021. [Online]. Available: https://ai.googleblog.com/2014/10/hdr-low-light-and-high-dynamic-range.html

[27] P. E. Debevec and J. Malik, "Recovering high dynamic range radiance maps from photographs," in *ACM SIGGRAPH 2008 classes*, 2008, pp. 1–10.

[28] S. W. Hasinoff, D. Sharlet, R. Geiss, A. Adams, J. T. Barron, F. Kainz, J. Chen, and M. Levoy, "Burst photography for high dynamic range and low-light imaging on mobile cameras," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 6, pp. 1–12, 2016.

[29] P. Sen, N. K. Kalantari, M. Yaesoubi, S. Darabi, D. B. Goldman, and E. Shechtman, "Robust patch-based HDR reconstruction of dynamic scenes," *ACM Trans. Graph.*, vol. 31, no. 6, pp. 203–1, 2012.

[30] OpenCV. (2021) OpenCV. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/opencv/opencv

[31] M. A. Robertson, S. Borman, and R. L. Stevenson, "Dynamic range improvement through multiple exposures," in *Proceedings 1999 International Conference on Image Processing (Cat. 99CH36348)*, vol. 3. IEEE, 1999, pp. 159–163.

[32] T. Mertens, J. Kautz, and F. Van Reeth, "Exposure fusion," in *15th Pacific Conference on Computer Graphics and Applications (PG'07)*. IEEE, 2007, pp. 382–390.

[33] OpenCV. High Dynamic Range (HDR). Accessed March 2, 2021. [Online]. Available: https://docs.opencv.org/master/d2/df0/tutorial_py_hdr.html

[34] D. Schiavone. (2019) RISC-V PULPissimo test. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/davideschiavone/riscv_pulpissimo_test

[35] S. Riedel. (2020) MemPool power breakdown. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/pulp-platform/mempool/blob/183098b34c68b11ff8c590f666f90feba485ce7c/doc/Power_Breakdown.pdf

[36] PULP Platform. (2021) PULP DSP Library. GitHub. Accessed March 2, 2021. [Online]. Available: https://github.com/pulp-platform/pulp-dsp