KSE | Kyiv School of Economics

School of Engineering

Micro- and Nanoelectronics Programme

Master's Thesis

Parametric ISA Extension Support in Snitch SoC

Ihor Sachok

Supervisor
KSE, Dr. I. V. Korotkyi, ykorotkyi@kse.org.ua

Expert
ETH, M. Bertuletti, mbertuletti@iis.ee.ethz.ch
ETH, L. Colagrande, colluca@iis.ee.ethz.ch

Submission date
15 December 2025

Academic Integrity Statement

I, undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures and text from other authors have been clearly cited and listed in the bibliography.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.
- Any assistance received (e.g. from peers, tutors, or online forums) is acknowledged in the acknowledgements section.

I understand that failure to comply with these declarations constitutes academic misconduct and may lead to disciplinary action.

Place, date: Kyiv, December 2025

Signature 

Contents

List of Acronyms v

Acknowledgements vii

Abstract viii

1. Introduction 1

2. Background 4
   2.1. Overview of the Snitch cluster 4
   2.2. Snitch toolchain and simulation 7
   2.3. Overview of instructions encoding 10
   2.4. Overview of Xpulpv2 ISA and its usage 11

3. Theoretical part 15
   3.1. Determination of overlaps between instruction opcodes 15
   3.2. Decoder modification 17

4. Hardware Architecture 19
   4.1. Xpulppostmod subextension 19
   4.2. Xpulpbr subextension 22
   4.3. Instructions offloaded to the Integer Processing Unit (IPU) . . 23

5. Verification 28
   5.1. Xpulpv2 extension testing 28
   5.2. LLVM compiler modification 29
   5.3. Continuous Integration (CI) flow adaptation 31
   5.4. Open-source synthesis flow 32

6. Results 36
   6.1. Performance for different configurations 36

6.2. Area evaluation 38

7. Conclusion and Future Work 40

Bibliography 42

A. Opcodes analysis table 44

List of Acronyms

ALU . . . . . Arithmetic Logic Unit

API . . . . . Application Programming Interface

ASIC . . . . . Application-Specific Integrated Circuit

CC . . . . . Core Complex

CI . . . . . Continuous Integration

COPIFT . . . . Co-Operative Parallel Integer and Floating-point Threads

CPU . . . . . Central Processing Unit

DMA . . . . . Direct Memory Access

DSP . . . . . Digital Signal Processing

EDA . . . . . Electronic Design Automation

FPU . . . . . Floating-Point Unit

FREP . . . . . Floating-Point Repetition

HDL . . . . . Hardware Description Language

IC . . . . . Integrated Circuit

IIS . . . . . Integrated Systems Laboratory

IP . . . . . Intellectual Properties

IPU . . . . . Integer Processing Unit

ISA . . . . . Instruction Set Architecture

LSU . . . . . Load-Store Unit

MAC . . . . . Multiply-Accumulate

ML . . . . . Machine Learning

MulDiv . . . . . Multiply-Divide

PC . . . . . Program Counter

PDK . . . . . Process Design Kit

PPA . . . . . Power, Performance, and Area

RTL . . . . . Register Transfer Level

SIMD . . . . . Single Instruction-Multiple Data

SOA . . . . . State Of Art

SSR . . . . . Stream Semantic Register

TCDM . . . . . Tightly Coupled Data Memory

Acknowledgements

I would like to thank my advisors, Marco Bertuletti and Luca Colagrande, for their guidance and help throughout this work. I greatly appreciate the opportunity to work under your supervision. I am thankful to my supervisor, Professor Ievgen Korotkyi, IIS group and Professor Luca Benini for making it possible to take part in their project. Also, I would like to thank ETH and KSE universities' staff who made this cooperation possible.

Abstract

In this thesis, the configurable RISC-V compute cluster Snitch is investigated. The work focuses on extending the baseline Snitch cluster SoC with support for Xpulpv2 custom instruction set extension, while enabling configurability between two variants of the architecture. To achieve this, an overlap between custom Snitch instruction sets and Xpulpv2 instructions is analyzed. A solution for the integer core decoder is proposed, along with an MCPU model for the compiler targeting the Xpulpv2 configuration. The software testing for the implemented extension was provided, including the development of the CI flow. The second part of the thesis presents an open-source synthesis flow utilizing open-source tools and libraries. Performance and area data for 3 different configurations are evaluated. The results of the post-synthesis area of the modules were compared to the literature.

Index Terms: RISC-V, Manicore, Snitch SoC, Open-Source EDA, custom instruction set extensions

Chapter 1

Introduction

Nowadays, integrated circuit design mostly remains a field with limited accessibility for a wide community of engineers and researchers. Absence of access to technology Process Design Kit (PDK)s, Electronic Design Automation (EDA) tools, and design Intellectual Properties (IP)s, which are required assets for chip design, slows down education of new specialists [1] as well as prevents inter-instituion cooperation. Open-source EDA tools and PDKs can address the stated problem [2][1] [2]. While removing this barrier, they reduce the cost of end-to-end chip development and enable cooperation on open-source projects.[3]

These days, parallel computing has become a foundation for modern applications like machine learning[2], data analytics, scientific simulations, image and signal processing[3], and different artificial intelligent applications. Development of hardware systems that are designed for parallel computations are trying to find a balance between energy efficiency, like specific hardware accelerators for a desired algorithm, and flexibility and programmability, as GPUs and CPUs.

An example of a flexible open-source hardware system for parallel computing is the Snitch cluster, developed in ETH Zurich and the University of Bologna [3]. Snitch is an open-source, lightweight RV32IMAFD core. It is small (10kGE in its basic configuration) and is particularly suitable to build parallel [3] and massively parallel [4] systems. The cluster can be configured and extended based on the desired application. Since the project's hardware, software, and toolchain are completely open, a number of projects were build based on the Snitch integer core.

[1] https://theopenroadproject.org/

[2] https://www.yosyshq.com/

[3] https://smart-silicon.net/open-source-eda-can-it-lead-a-new-era-of-hardware-design/

Some projects focused on extending the Instruction Set Architecture (ISA) of Snitch with standard and custom extensions, a feature intrinsically provided by the RISC-V ISA. For example, besides the `frep`, `ssr`, `smallfloat`, and `copift` custom extensions, supported by the upstream Snitch core[4], in the Spatz cluster[5], the Spatz [5] vector processor is used as a co-processor to Snitch for the implementation of the standard RISC-V vector instruction set, in MemPool [4], the Snitch is used for massive replication and the custom xpulpv2, zfinx and zhinx extensions are added[6].

In other projects, the Snitch cluster is also used as a benchmark to test the integration of accelerators in a complete system. For example, the RedMulE tensor accelerator [6] includes the Snitch cluster as a benchmarking environment[7]. The Snax cluster [7], developed by MICAS group in KU Leuven, focuses on providing an efficient way of accelerator integration[8]. It includes Data Streamer and Data Reshuffler for managing data access patterns, as well as a number of different accelerators.

In the context of an open-source project, the development of multiple forks of the project is a natural and even desired phenomenon. The possibility to test new features, explore alternative design choices, and adapt existing architecture to specific needs reflects the core principle of an open-source ecosystem. At the same time, contribution to the upstream project is highly valuable. This ensures that the improvements can be shared with a broad community and used as a reference solution of dealing with overlaps of instruction opcodes for future developments. The global aim of this work is to make the Snitch public repository a benchmark for new instruction set development and testing, and provide an environment suitable for these needs.

However, contribution to the upstream branch is not always straightforward. In particular, focusing on the project that extended Snitch ISA, we point out that the encoding of the new custom instructions may overlap with each other. Such overlaps pose a challenge in instruction decoding, as well as in the compilation of software, since the compiler backend relies on a non-conflicting opcode space.

This thesis focuses on improving the quality of the Snich Cluster as an open-source project in two main directions:

- The first objective is to find a way to include different extensions in the core decoder, suggesting a strategy to customize its ISA for different

[4]https://github.com/pulp-platform/snitch$_cluster$

[5]https://github.com/pulp-platform/spatz

[6]https://github.com/pulp-platform/mempool

[7]https://github.com/yvantor/snitch$_cluster/tree/yt/redmule-test$

[8]https://github.com/kuleuven-micas/snax$_cluster$

application domains. Streamlining the Snitch decoder should make it more reusable by external contributors, allowing them to use the Snitch Cluster repo as a public benchmark for their own custom extensions.

- The second goal is to provide an open-source flow for area evaluation and give an example of its application. In fact, the benchmarking of the included ISA extensions would not be complete without the evaluation of PPA, but the PPA evaluation in the Snitch cluster is currently based on proprietary tools and cannot be open-sourced as part of its public repository.

In particular, the Mempool fork of the Snitch cluster was considered. In the work I show how the custom extensions developed for image processing can be included in Snitch decoder, obviating the problem of opcodes overlapping. Furthermore, the flow of the open-source Snitch repository was extended by the synthesis flow on open-source tools using ihp130 library, which can be used for area evaluation. In the thesis I provide an example of comparison between configurations regarding their area and performance.

The main part of the thesis consists of the following chapters:

1. Background - overview of the mainstream Snitch cluster and its toolchain, instruction encodings, and related works.
2. Hardware Architecture - implementation of Xpulpv2 for Snitch cluster.
3. Verification - Xpulpv2 tests, CI, and open-source synthesis
4. Results - performance and area for different configurations
5. Conclusions

Chapter 2

Background

This chapter provides an overview of existing work, the architecture of the Snitch cluster, and its toolchain. The Snitch cluster custom extensions and instructions encoding are considered.

2.1. Overview of the Snitch cluster

The topic of the presented work is to provide configurable ISA extensions for the Snitch cluster[3], so the Snitch Cluster design should be considered first. While the current work mostly focuses on extending the decoder in a Snitch core, it is crucial to understand how this computational system was built and what its main features are.

The Snitch cluster from the main git branch[1] was chosen as a system to extend. To understand the specifics of the computational cluster, its current architecture and toolchain should be reviewed. This overview is based on the article[3] and github page[2]

To begin with, the computational system is configurable and consists of one or more Hives. Hives are connected to the Tightly Coupled Data Memory (TCDM) crossbar, which attaches to a banked shared memory. TCDM is a memory used in the system; it also supports atomic instructions via a special unit placed in front of each memory. TCDM interconnect is purely a combination, and memories consist of macros. The Hive can be a unit of repetition because all signals that cross the boundary are fully decoupled. Thats why the pipeline registers can be inserted there.

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

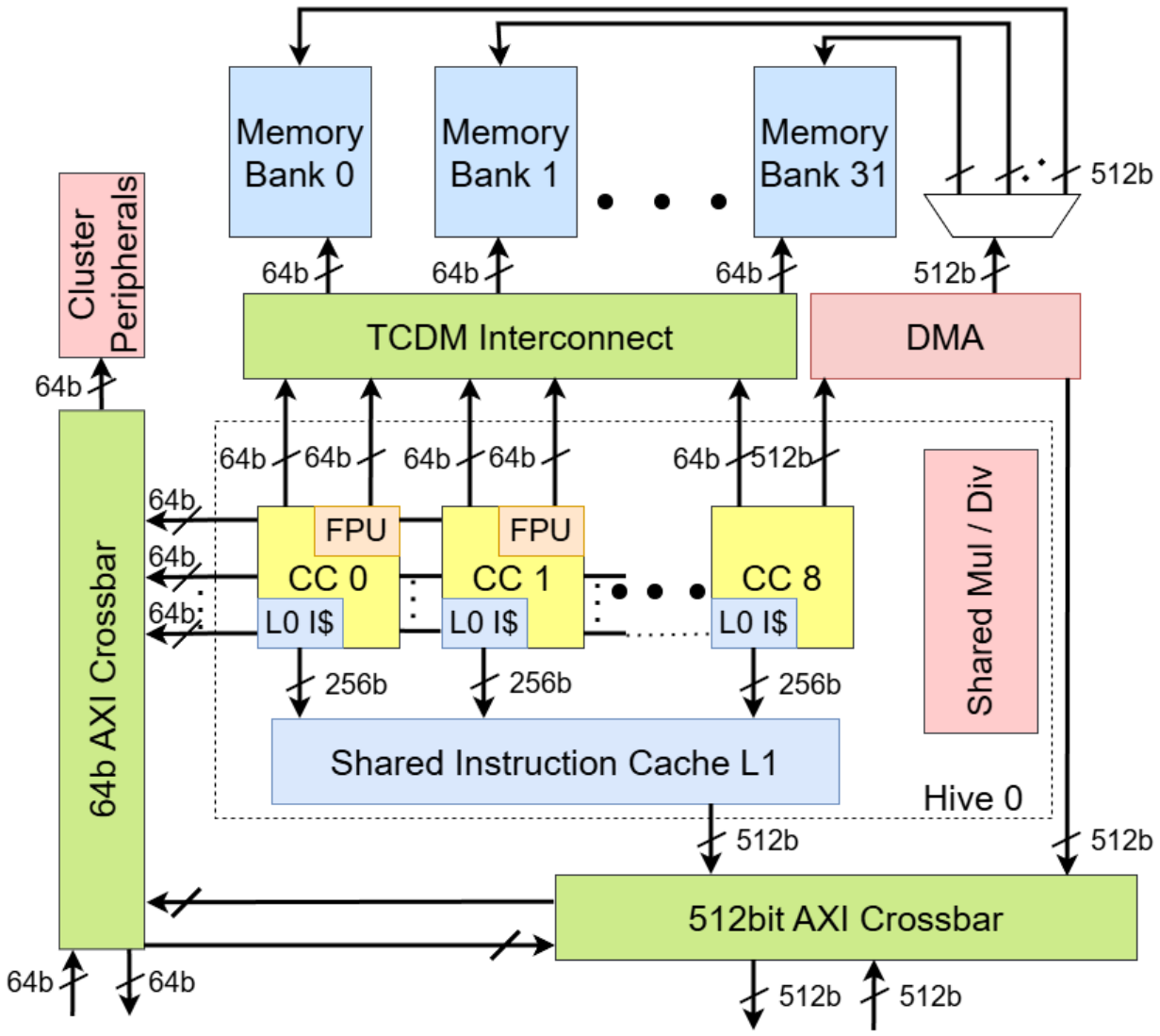


Figure 2.1.: High-level Snitch cluster diagram. It contains the smallest unit of repetition Snitch Core Complex. N Core complexes form a Hive, and M Hives form a Cluster. They are connected with Axi Cluster Crossbar. Each Hive is connected with TCDM bank via TCDM Interconnect.

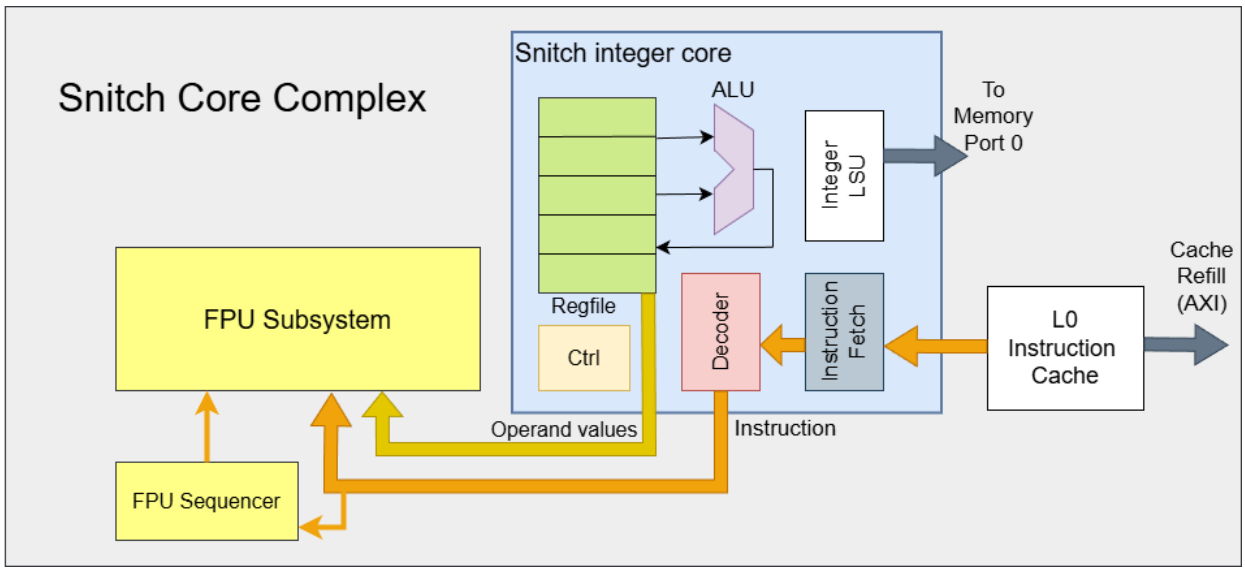


Figure 2.2.: Snitch Core Complex diagram. It depicts the Snitch integer core, FPU Sequencer, and FPU Subsystem

Each Hive consists of several Core Complex (CC) that, in the default configuration, share a Multiplication/Division unit and an instruction cache. The number of Core Complexes in a Hive is also configurable. The presence of a Multiply divide unit that the cores in a Hive can use makes the RV32M extension available in that system.

Now, let's consider Core Complex. The Core Complex is the smallest unit of repetition. It contains the RV32IMAFD Core (Snitch) and various accelerators that provide a hardware solution for ISA instructions that the integer core currently supports. For instance, in the older version, a custom Integer Processing Unit was included. In the default configuration, Core Complex includes Floating-Point Unit (FPU) Subsystem and FPU Sequencer. The Floating Point Unit supports single-, half-, and quarter-precision formats, which are useful for Machine Learning (ML) algorithms. Since floating-point multiply-add requires long combinational paths, the floating-point unit is pipelined to achieve a reasonable frequency. The FPU includes custom ISA extensions. The first is Stream Semantic Register (SSR). The extension eliminated the need for explicit load and store instructions if the data has an access pattern. Semantically, the instructions are similar to vector operations, because they also operate on vectors. The second ISA extension is the Floating-Point Repetition (FREP) instruction, which is used in FPU sequence Buffer. By using this instruction, floating-point instructions can be sequenced in a configurable manner, and the integer core is available for other tasks. This extension makes a Snitch Cluster pseudo-dual-issue system by enabling parallel operations between the Snitch integer control core and the FPU. The proposed extensions were tested on widely used algorithms, such as matrix multiplication, 2D convolution, and kNN distance calculation. While

providing minimal overhead in terms of area, SSR and FREP reduce the number of memory operations and the branching overhead. Over 4x speedup was achieved for the examples mentioned.

The integer core is called Snitch. It is tuned for area and energy efficiency. This ultrasmall system consists only of 9 to 20 kGE. It contains the Control unit, Register file, Integer Load-Store Unit, Decoder, and Instruction Fetch unit. It is not pipelined, and the Core Complex also includes L0 Instruction Cache from which the instructions are fetched. The current configuration also includes the Direct Memory Access (DMA) Engine. It is needed to achieve independent data transfer that hides latency and provides high throughput. However, accelerators like FPU can be disabled in a custom config. The default configuration includes one hive with 8 compute cores and 1 DMA core. The detailed description of the currently supported RISC-V and ISA instruction sets will be considered later in the next part, since they are among the work's focuses.

2.2. Snitch toolchain and simulation

To simulate the system, either open-source Verilator or proprietary QuestaSim can be used. Using a Python script, the specific configuration for the cluster top file is generated. By default, it represents the system described above; however, the main repository includes different experimental configurations such as frep_xl.json or omega.json. For instance, omega.json represents a cluster with omega TCDM. The testbench includes a simulated cluster and environment to load the software program into memory and verify the number of cycles and the program's correctness. The make utility is used to assemble a testbench for Questasim or Verilator.

The Snitch cluster repository includes a number of software programs to test its features and to demonstrate algorithms that can be run on the computational system. By default, the software runtime includes headers for memory allocation, cluster interrupts, dma, ssr, synchronization, and others.

Runtime headers and software are compiled with LLVM, which includes a compiler snitch model that lists supported extensions.

Snitch core is RV32IMAFD core, which means that 32-bit instructions include:

1. I Integer instruction set basic arithmetic, logic, and memory. This is a basic instruction set that is implemented on the RISC-V core.

The set includes arithmetic and logical operations, comparisons, shifts, load/store (memory access), branches, and jumps.

2. M Integer Multiplication and Division set. It implies multiplication and division operations. In the default Snitch configuration, it is supported via a separate Multiplication-Division Unit
3. A Atomic instructions. This is a standard RISC-V extension that provides instructions for atomic read-modify-write of memory to support synchronization between multiple cores running in the same memory space.
4. F Single-Precision Floating Point extension. It adds instructions for single-precision floating-point operations. (32-bit floating point that corresponds to the IEEE 754 standard). It includes floating-point arithmetic, comparisons, and conversions between integers and floating-point numbers.
5. D Double-Precision Floating Point. Adds support for double-precision operations (64-bit values, conforming to the IEEE 754 standard). In addition to F extension, it includes arithmetic, comparison, and conversion between integers and floating-point numbers.
6. Zfh Half-Precision Floating Point. This extension supports 16-bit floating-point values (half-precision), according to the IEEE 754-2008 standard. It includes instructions: FP16 load/write: FLH, FSH (analogous to load/store for half-precision); FP computation operations on half-precision: similar to FP32 operations, but over FP16 and operations for conversion between half and single and half and double precision floating point.

These are the RISC-V official extensions included in the standards. However, the RISC-V approach allocates some opcodes to custom instructions, providing an opportunity to develop custom instruction sets that may be more efficient for the given problem. Snitch cluster core supports the following custom instruction sets:

1. RV32b_Custom: bit manipulation. This instruction set provides logical operations with operand inversion, bit/byte permutation instructions, bitfield masking, rotation and shift, bit counting, and comparison instructions.
2. Frep_custom: includes custom commands for floating-point repetition. This extension was described previously. Currently, it includes only one instruction frep.o

3. Ssr_custom: instruction set that provides custom instructions for Stream Semantic Registers extension. It is necessary for correct SSR performance in the Snitch Cluster.[8]
4. copift_custom: this instruction set includes 3 instructions that are used for dual-issue execution of mixed integer and floating-point instruction sequences in Co-Operative Parallel Integer and Floating-point Threads (COPIFT) methodology.[9]
5. flt_occamy: this instruction set was developed in the Occamy project.[10] It represents an instruction set for small float extension. It includes:
    - Smallfloat scalar sub-extension. It has the same format as F extension and represents instructions for half-precision floats (RV32Xfhalf), alternate half-precision floats (RV32Xfalthalf), quarter-precision floats (RV32Xfquarter), and alternate quarter-precision floats (RV32Xfaltquarter).
    - Smallfloat vectorial sub-extension. It includes vectorial single-precision floats (Xfvecsingle), vectorial half-precision floats (Xfvechalf), vectorial alternate half-precision floats (Xfvecalthalf), vectorial quarter-precision floats (Xfvecquarter), and vectorial alternative quarter-precision floats (Xfvecaltquarter).
    - Auxiliary smallfloat operations which have the same sub-sets. The extension provides load/store, arithmetic, comparison, conversion, and signed operations for floating-point data of different sizes. From a hardware perspective, they are used in the FP unit.
6. dma: this extension provides a full set of instructions for Asynchronous Data Movement in the Snitch cluster. It includes initialization, address setup, transfer start, and others.

The LLVM compiler supports Snitchs extensions, which are listed here. By compiling the program, a binary is obtained. The binary afterwards can be loaded to the testbench and run in rtl simulation.

After the simulation, a Python script can generate and annotate traces. The annotated traces show the assembly instructions executed on the cores, the number of cycles, when each instruction was executed, and the C code line corresponding to each assembly instruction. This makes debugging software programs much easier

To sum up, the open-source repository provides a fully functional environment for software and hardware simulation. The Snitch cluster GitHub repository includes a Docker preconfigured environment set up in advance, and the

Register Transfer Level (RTL) simulation can be run using the open-source Verilator.

2.3. Overview of instructions encoding

Before determining the overlaps, it is worth considering the instruction encodings currently used in RISC-V and, specifically, in the PULP project. [3]

In the past, all popular commercial ISAs were proprietary. However, nowadays, a tendency to use open source ISA for the projects among organizations and research groups increases. In particular, the RISC-V ISA is quite popular since it provides a completely free and open instruction set architecture.

The RISC-V ISA standards give a description of a RISC architecture with three basic ISAs: RV32I, RV32E, and RV64I. The difference is the width of the registers and the number of registers. RV32I includes 1 specific (x0) and 31 general-purpose registers. (x1-31). The width of the data used in the ISA is 32 bits. RV64 operates on data of different lengths, basically on 64-bit data. And RV32E specifies fewer registers. The described registers are used for integer data. Also, the RISC-V specification describes 32 registers f0-f31, which are used for single- or double-floating point values. This is crucial for Snitch, since it is designed to perform both integer and floating-point computations. Additionally, the RISC-V standard defines two special user-visible registers: pc and fst. The Program Counter (PC) is used to hold the address of the instruction that is executed at the moment. And the second special register fsr the floating point status register stores the operating mode and exception status for the FPU.

The instruction's encoding depends on the Instruction Type. The base ISA includes four main instruction formats (R/I/S/U). These instructions have a fixed length of 32 bits.

1. R-type Register to Register instruction type. These instructions require 2 registers to perform their operation. The output is also stored in the register. An example of such instructions is arithmetic and logical instructions: ADD, SLT, SUB, AND, OR, XOR

2. I-type Immediate instructions. These instructions have an immediate as one of the operands. The second operand is a register. The output is written to another register. ADDI, SLTI, ANDI, and ORI are examples

[3] https://github.com/pulp-platform

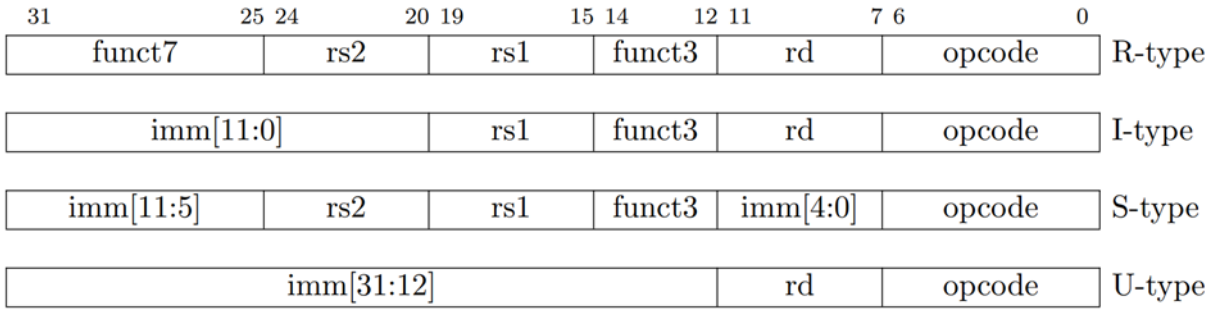


Figure 2.3.: RISC-V base instruction formats [11]

of such instructions. Also, the immediate should be short, the field for it is 12 bits.

3. U-type Upper Immediate (Long immediate). It is used to efficiently load large (20-bit) values and is used in instructions like LUI (Load Upper Immediate) and AUIPC (Add Upper Immediate to PC)
4. S-type Store type.

The RISC-V specification specifies the source registers (rs1 and rs2) and the destination register (rd) at the same position in the opcode to simplify decoding. Immediates are placed at the beginning of opcodes to reduce the hardware complexity required for decoding. Particularly, the sign value for immediates is always in 31 bits.

The fields funct3 and opcode fully specify the instruction, while other fields can hold different values. On the one hand, the ability to freely encode custom user-defined instructions is a significant advantage for the RISC-V architecture. On the other hand, user-defined instructions with different encodings from different projects can overlap. Without special treatment, the first met in the case statement instruction will be executed. This can lead to misbehavior of an algorithm and incorrect results. This issue leads to the conclusion that two instruction sets with overlapping encodings cannot be used in the same system. The possibility to configure an instruction set and, consequently, the system's hardware can be very useful for users.

2.4. Overview of Xpulpv2 ISA and its usage

A great feature of the Snitch Cluster is scalability and expandability. Its low energy consumption and small area occupation made its use in different systems possible. To better understand the connection between ISA instruction usage and target applications, it is worth reviewing the existing utilisation of the Snitch cluster and its core.

One of the applications of the Snitch core presented in the literature is MemPool[4], an energy-efficient multicore general-purpose RISC-V system designed for extremely parallel, compute-intensive workloads. The work examines the challenge of scaling a single-cluster architecture to hundreds of cores. In the presented work, the number of Snitch cores was 256. While the article focuses on cluster interconnection, interaction with shared L1 memory, and programming model, in the current work, it is more important to understand which extensions and features of the Snitch core were used. First, the Snitch cluster was chosen for its small footprint and low power consumption. The second reason is that the core can issue multiple load/store requests when the read-after-write dependency is absent, whereas the pipeline would not be blocked. The third is that it is possible to add complex pipelined computational units that can offload an instruction without stalling the core. In the article, the Snitch cluster architecture was extended for usage in Mempool:

1. The reorder buffer for Load-Store Unit (LSU) that retires loads in order was presented
2. A pipelined IPU was connected to Snitchs accelerator port with the aim of supporting Xpulpimg instruction set.
3. The register file was extended to include three ports to support Multiply-Accumulate (MAC) operations

These extensions result in significant speedup for DSP kernels. However, it is important to understand which problems and algorithms Mempool was designed to solve. In the article[4] examples kernels were presented:

- matrix-matrix multiplication with each core multiplying a 4x4 output tile
- 2D convolution with 3x3 kernels
- 2D discrete cosine transform on 8x8 block
- A key BLAS routing computing
- The dot product (the scalar product of two vectors)

MAC extension significantly reduces the number of instructions per kernel. MemPool shows good results regarding scalability. This enables parallelisation across a large number of cores with minimal synchronisation overhead. While the current work also focuses on Snitch cluster development to support Xpulpv2 extension configuration, the MemPool project remains a good reference for the current development.

The Xpulpimg for the Snitch core in Mempool was proposed in [12]. In this work, image-processing algorithms for image signal processors were considered, specifically high-dynamic-range algorithms.

For Xpulpimg 173 instructions from Xpulpv2 extensions were selected. They were selected based on relevance for image processing algorithms. The instructions were grouped into sections: generic arithmetic operations that reduce control flow instructions and increase code density; Extended load/store addressing modes: register-register mode and post-increment mode; Multiply-accumulate instructions: custom instructions from Xpulpv2 that multiply the content of two registers and perform accumulation or subtraction; Packed-SIMD instructions for 16-bit and 8-bit sub-words dot-product operations, addition, subtraction, comparison, shifts, logical operations, and instructions for packing and unpacking SIMD data. To support custom RISC-V instructions from the Xpulpimg instruction set in the thesis, Snitch IPU and Snitch integer core were extended. The existing integer coprocessor was extended with a Digital Signal Processing (DSP) Unit that performs the selected instructions. The suggested extension was verified using two algorithms commonly used in DSP and image processing: 2D integer convolution of a 32x32 matrix with a 3x3 kernel, and matrix multiplication of 64x64 matrices. The simulation was performed on a 16-core Mempool system with 64 memory banks (64 KiB of shared L1 TCDM). The algorithms were optimized using compiler intrinsics and assembly-level optimization. Verilator 4.024 2019-12-08 was used to perform RTL simulations. The simulation showed 2.7x speed-up for the 8-bit 2D convolution program and 4.6x speed-up for the matrix multiplication. In addition, the area increase was evaluated in the work. The calculation showed a 40.4% average increase in area.

In addition to official ISA extensions, the RISC-V standards provide a possibility to extend the instruction set for the desired target and create user-defined ISA extensions. One such open-source instruction set is Xpulp, which is oriented toward DSP workloads, which was discussed previously. The extension was developed to achieve high performance and code density comparable to those of proprietary MCUs used in IoT applications, where efficient sensor data processing and energy efficiency are crucial. The current version of the Xpulp extension includes the following functional blocks:

- Extended addressing modes for load and store operations with post-increment writing (register offset and automatic post increment of the base address, which is stored in the register);
- Hardware loops that eliminate the need for explicit loops and calculations for counter increment);
- Bit manipulation operations;

- Extends general arithmetic logic unit instructions;
- MAC instructions;
- Fixed-point operations for addition, subtraction, and multiplication;
- Packed Single Instruction-Multiple Data (SIMD) for 8- and 16-bit data with three addressing modes: vector-to-vector, vector-to scalar(register), and vector-to-scalar(immediate). The packed-SIMD extension includes generic arithmetic operations, comparison and manipulation instructions, dot product operations, and complex-number operations.

There are C/C++ compilers which are available for compilation of code with XPULP extension: PULP GCC is based on the standard GCC for RISC-V and was extended to support additional instructions, including Xpulpv2.[4]

[4]https://github.com/pulp-platform/pulp-riscv-gcc

Chapter 3

Theoretical part

This section discusses the problem of overlapping instruction opcodes in the Snitch integer core and analyzes possible approaches to resolving these conflicts at the decoder level.

3.1. Determination of overlaps between instruction opcodes

As a result of the literature review and GitHub forks of the Snitch cluster repository, it was observed that the Xpulpv2 instruction set is useful for various applications and is often present in projects that use the Snitch integer core. Currently, the Mempool project supports the following Xpulpv2 instruction subsets: Xpulpabs, Xpulpbitop, Xpulpbr, Xpulpclip, Xpulpmacsi, Xpulpminmax, Xpulppostmod, Xpulpslet, Xpulpvect, Xpulpvectshufflepack. The implementation of the instructions from these subsets will be discussed later in this work.

However, it is not yet supported in the main branch. The key problem here is the overlap of custom instruction opcodes. Thats why it requires a detailed investigation and modification of the decoder in the Snitch core. Also, the solution should be configurable to provide flexibility.

During the development of RISC-V-based hardware platforms with custom user-defined extensions, a need arises to provide a consistent, formally correct representation of instruction encodings, control registers, and exception causes. The approach should be universal enough to be used for all design subsystems: decoder, compilers, and simulators. To meet these requirements,

PULP developed a specialized encoding generator. It converts RISC-V instruction descriptions into a set of different files in various programming and hardware description languages. The script receives an instruction description as input. Then the MATCH and MASK pairs are calculated. They define fixed and changing bits of the instructions opcode. It automatically determines the format of each instruction and extracts its arguments (rd, rs1, rs2, etc) and correctly forms the immediates. This eliminates the need for manual encoding analysis and prevents consistency errors while adding new instructions or instruction sets. The script is a key automation tool for the design on the PULP platform.[1]

In the Snitch cluster instruction opcodes system, a Verilog file is generated by the shell script generate-opcodes. There, all the instruction sets are specified, the script parse_opcodes from the repository riscv_opcodes is executed, and the opcode description files are passed to it. As a result of the execution, a System Verilog file is received. In the file, a local parameter is defined with a 32-bit string representing each instruction. Constant bits are described by zeros and ones, and variable bits are indicated by the ? symbol. This data is directly used by the Snitch decoder in casez statements. The method ensures correct, unambiguous instruction definitions at the hardware level.

In the current state, if two instructions overlap, the script stops generating with an error. Since it is usually an issue if two instructions can be decoded by the same opcode. However, the purpose of this work is to operate in such cases and resolve these collisions in the decoder, without changing the instruction encoding itself. Changing the encoding of the custom instructions may seem like a more efficient way to solve this issue. That would allow the usage of all of the extensions simultaneously. However, this solution does not address the existing problem, since the extensions that are considered are used in different projects and are already defined in the compiler. That is why the encodings cannot be changed, and the issue should be solved in a different way. In this work, it is suggested to make hardware and toolchain changes to parametrically enable various instruction sets in the config. To address this issue, the parse_opcodes script was modified. A flag –allow-overlap that allows overlap between instructions was provided. The new version of parse_opcodes was pushed to the snitch branch of the riscv-opcodes repository.[2]. Future projects can rely on the modified version of the script to generate instructions in the presence of overlaps. In the modified version, only a warning is sent if there are instructions with possible overlap. So, the final riscv_opcodes file will include all supported instructions, while the selection of overlapping instructions is based on a flag passed from the config in the hardware decoder.

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

[2] https://github.com/pulp-platform/riscv-opcodes/tree/snitch

From the perspective of consistent implementation of the modified Snitch decoder, it would be quite useful to first determine the overlapping instructions. For this purpose, the script analyze_opcodes from the riscv folder in the PULP platform git repository was used. The script enables an automatic analysis of the opcodes of the listed RISC-V extensions. Analyze_opcodes first calls the parse_opcodes utility that was described earlier for every extension description file. Then the exact bitmask for each instruction is collected: the funct8, rs2, rs1, funct3, rd, and opcode fields. After extracting the fields, the script does a pairwise comparison of all instructions and identifies clashes. The clash means that two different opcodes overlap in the values of the bit fields, considering dont care (?) masks. In addition to conflict analysis, the tool can generate a list of free instruction opcodes, i.e., bit patterns that do not overlap with any existing instruction in the set.

The script supports several visualization methods. The flag report clash_matrix enables a mode that generates a LaTeX table of conflicts between extensions. The instruction sets are listed horizontally and vertically, with cells indicating the number of overlapping instructions. The report inst_table mode outputs a detailed table of all instructions, including conflicting ones. Finally, there exists a flag to report free opcodes. In this work, the first two modes of the script were used: first, to visualize and analyze the overlapping extensions, and second, to identify the exact instructions that overlap. The generated table is provided in Appendix A.

3.2. Decoder modification

From the RTL point of view in Snitch, the keyword casez is used in the decoder. Based on the semantics of System Verilog, this structure allows ignoring z (or ?) bits of the expression during the comparison and comparing only non-z values. The analogue of the casez expression for an if statement is ==?. It allows to do the same comparison but for two operands only.

As it was found, the Xpulpv2 instructions cannot be used with DMA, SSR, copift, and frep extensions. In the table in Appendix A, the overlapping instructions can be found. It is worth mentioning here how the overlap of instruction opcodes in the decoder was avoided. There are two ways to solve this issue in the System Verilog code of the decoder.

1. The first possible solution is to write a second separate decoder, which will be enabled via a flag. This solution is clear from an architectural point of view: there are two decoders that can be selected during cluster compilation in config. However, this will increase code complexity

and require a separate decoder for every overlapping project, further overcomplicating the RTL module.

2. The second solution is to provide configurability for case branches with overlapping instructions: based on a config flag, the correct branch is generated. This enables the reuse of a single decoder implementation and avoids duplicating RTL code while implementing the desired functionality.

Chapter 4

Hardware Architecture

This chapter describes the hardware architecture modifications introduced to support the Xpulpv2 instruction set extensions in the Snitch cluster.

4.1. Xpulppostmod subextension

In the standard RISC-V instruction set architecture, memory access relies on a single addressing mode. The effective address is calculated by adding a base address, stored in a register, to a sign-extended immediate value. The simple register-plus-offset mechanism is the only way to form a memory address in the base RISC-V architecture. The Xpulppostmod sub-extension from Xpulpv2 introduces two additional addressing modes for load and store instructions. First variant: the effective memory address is computed as the sum of an offset and a base address, both taken from registers. Second variant: the effective address may correspond to the base address, which is stored in a register. After accessing the register's value, it can be incremented by an offset. While the offset can be store either in another register or be an Immediate. Such addressing models are supported for word (32-bit), half-word (16-bit), and byte (8-bit) memory access. Table 4.1 summarizes all the extended post-increment load and store instructions.

To support the described instructions, the following changes to the Snitch integer core were provided:

1. The decoder was updated to parametrically support the introduced instructions. As was found, instructions from this subset overlap with instructions from DMA, SSR, copift, and frep. In the decoder, an if branch was added. Based on the enabled flag, the Postmod instruction

Instruction	Description
`p.lb[u],h[u],w rd, iimm12s,rs2(rs1)`	Load a value from `rs1 + iimm12s,rs2` into `rd`
`p.lb[u],h[u],w rd, iimm12s,rs2(rs1!)`	Load a value from address `rs1` into `rd`, then increment `rs1` by `iimm12s,rs2`
`p.sb,h,w rs2, simm12s,rd(rs1)`	Store the value from `rs2` at address `rs1 + simm12s,rd`
`p.sb,h,w rs2, simm12s,rd(rs1!)`	Store the value from `rs2` at address `rs1`, then increment `rs1` by `simm12s,rd`

Table 4.1.: Xpulppostmod load and store instructions

will be decoded. If the flag is not enabled, the decoder will continue to decode the previously present instructions.

2. Register file for integer registers was modified: an additional reading port was added to support instructions with three operands (if rs3 is used)
3. The load-store unit is modified to support post-increment instructions: after a load or store operation, the base address is automatically updated.
4. The immediate logic was expanded by adding a new immediate format variant and correct sign/zero extensibility for them.
5. The hazard detection and stall logic was adapted to new instructions: dependency on the third operand (rs3) and features of post-increment addressing were considered.

The inserted changes are presented in a diagram in Figure 4.1.

The extended load and store instructions are beneficial for virtually all data processing algorithms. Especially, when the memory access pattern is regular, yielding speed-up of up to 20% [ ]. Using these instructions, two operations are issued in the same cycle, thus fewer cycles are needed per useful computation. This increases overall throughput, while also reducing code size. Moreover, the register-register addressing feature enables post-increment even if element offsets are not known at the time of compilation.

In post-increment load and store instructions, regardless of the offset being given as a register or an immediate, the register rs1 is used as the base address as well as an additional destination register for storing the result of

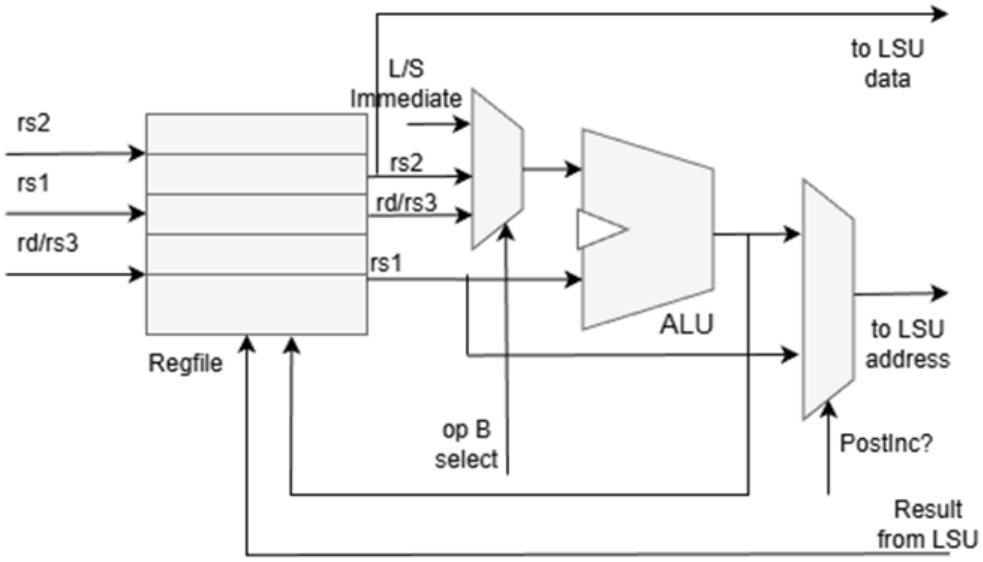


Figure 4.1.: Diagram of the load-store logic with insertions to support the third operand and post-increment mechanism from xpulppostmod subextension

base increment. To support such behavior, the scoreboarding mechanism was extended. The entry of rs1 is marked busy when a post-increment instruction is issued. This ensures that the stall will acquire whenever the next instruction will attempt to read the same register. The write-back logic was also adapted to treat rs1 correctly as a destination register. The update of rs1 in a post-increment instruction happens in the same cycle when a post-increment instruction is issued, while usual load instruction writes back after the memory access completes. This mechanism ensures that a post-increment instruction write-back to rs1 and the standard instruction retirement can not occur simultaneously. So, the write-back port can be normally used for integer instructions on rd and handle the post-increment change of rs1.

The added register-register store instructions require three source-operands: rs1 provides the base address, rs2 provides the data, that will be written to the memory, rs3 contains the register-based offset. To support three inputs, the integer register file was extended with a third read port. The first two inputs did not change their purpose. The third port is indexed using the rd field, which functionally corresponds to rs3.

Using rd as a source registers introduces potential RAW hazards. To correctly solve such cases, the scoreboard logic was extended to detect and handle hazards in rd before the register is accessed. In the baseline ISA in load and store instructions, the base address is held in rs1 and is forwarded to the first operand of the Arithmetic Logic Unit (ALU), while the second operand receives the immediate that is offset. The ALU computes the addition of two operands, and the output is used as the effective memory address.

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.: Immediate branching instructions from Xpulpbr sub-extension

The introduction of the register-register addressing mode extends the logic of the mechanism, allowing the second operand to be taken from a general-purpose integer register rather than an immediate. In the postmod instructions, rs2 provides the offset for register-register load operations, and rs3 provides an offset for register-register store operations. While in both cases the value is added to the value stored in rs1 to generate the target effective address.

To support post-increment functionality, the addressing path between the Snitch core and the LSU was redesigned. Instead of connecting the LSU address port to the result output of ALU, a multiplexer was added that selects between the ALU output and the unmodified value of rs1. This enables the operation of post-increment instructions in two phases within the same cycle. The LSU receives the original vase address to use it for the memory access, and at the same time, the ALU computes the incremented address, which is then written back to rs1 as a post-increment. When the basic memory access operations are used, the output of the ALU is selected as effective address. This flexible addressing scheme preserves compatibility with standard RISC-V ALU load and store instructions while enabling hardware support for extended addressing modes.

4.2. Xpulpbr subextension

In the standard RISC-V ALU, there are only branch instructions that compare the values of two registers and, based on the result, go to the next instruction. In Xpulpv2, an immediate branching was suggested. These instructions compare a value in the register with an immediate operand to determine whether a branch should be taken or not. The 12-bit B-type immediate is the branch target, representing a signed PC-relative offset in steps of two bytes. This enables a reachable range of ś 4KiB.

The value stored in register rs1 is compared with pimm5s 5-bit-signed-extended immediate. The program counter offset is given by bimm12s, a 12-bit-signed B-type immediate.

The hardware changes to the Snitch integer core are minimal, since the standard RISC-V ISA already implements branching. The same mechanism is used for immediate branching. The only difference is the comparison between rs1 and pimm5s. To support this operation, a dedicated 5-bit immediate field was added for decoding. The field reuses the encoding format of rs2 and should be sign-extended before it is forwarded to the ALU. In ALU, it is compared against the value stored in rs1. The ALU produces a Boolean result indicating whether the branch condition is met. Based on the result the following program from the program counter is fetched or specified in the immediate filed instruction.

Immediate branching is particularly advantageous if the second operand of a comparison is known at compilation time and can be used directly as an immediate value. Typical examples are loops with a determined iteration step (such as convolution with fixed kernel size or matrixes multiplication), counters with statically defined terminal values, and conditional statements that compare a register against a constant. In these cases the usage of immediate branching instructions eliminates the need to store the second operand. This reduces the pressure on the register file and can be especially beneficial in compute-intenstive kernels with high register usage.

4.3. Instructions offloaded to the IPU

The Snitch processor is designed as a lightweight, small integer core that can be extended with functional units for specific operations or computations accelerators. Instead of integrating complex units in the core directly, Snitch uses a decoupled offloading mechanism where the selected instructions are sent to specific hardware accelerator modules, such as floating-point unit or an integer processing unit. This approach keeps the integer core extremely compact, while allowing complex instructions be executed.

Snitch system provides an efficient interface for uploading instructions to the accelerators. It is implemented with a dedicated request-response interface defined in the system. The interface includes two independent unidirectional channels:

- Request channel. It includes signals acc_qreq_o, acc_qvalid_o, and acc_qready_i. The core sends a structured accelerator request that contains the decoded instruction, operand values, and control field. The handshake follows a ready/valid protocol. The integer core asserts valid_o when a request is available, and the accelerator asserts ready_i_ when the request can be accepted.

- Response channel. It includes acc_prsp_i, acc_pvalid_i, acc_pready_o. This interface is designed to receive responses from the accelerator side. After processing the offloaded instruction, the accelerator returns a response to the integer core containing the result and status bits. The core acknowledges receipt of the result data via the acc_pready_o signal.

These two channels operate independently of each other, allowing the accelerator to do computations with arbitrary latency, which is necessary for complicated computations. While an accelerator instruction is executed, the Snitch core stalls the pipeline, which is waiting for the result to ensure in-order execution.

When the decoder in Snitch integer core identifies the instruction that should be offloaded to an accelerator, the core does not send the operands to ALU. Instead, an accelerator request is constructed. It encapsulates the source register operands, destination register index, instruction opcode, and funct fields, optional immediate data, and control bits, which are required by the accelerator. The request is sent to the accelerator request channel. After sending the request, the integer core waits for a matching response from the accelerator, which provides the value written back to the integer register file, destination register index, and status or exception flags in case of invalid operations or special cases.

In the Snitch microarchitecture, the communication between the integer core and accelerators relies on two strongly typed data structures: acc_req_t and acc_resp_t. These types are defined in the Snitch package. They specify the information fields that are exchanged between the integer core and an accelerator over the accelerator interface. The request type acc_req_t consists of the signals that are needed for an accelerator to execute the offloaded instruction. It includes source register operands, destination register index, instruction opcode, and funct fields, immediate values, and additional accelerator-specific control bits. And vice versa, the response type, acc_respt_t, encapsulates the information produced as a result of execution on accelerator, destination register index, and statusexception flags.

Because of this structure, the core does not track any accelerator-specific state. Information about offloaded instructions and the operation of the accelerator is included in the response itself. This typed interface provides a clean abstraction boundary between the integer core and accelerators. It allows easy integration of new accelerators in the system.

Currently, Snitch SoC includes five hardware accelerator units that extend the functionality of the lightweight integer core. First accelerator is the FPU, which implements single and double precision floating point arithmetic as well

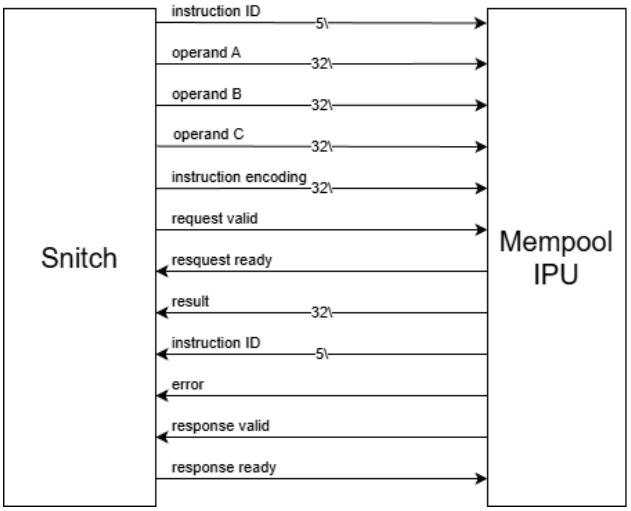


Figure 4.2.: Interface between Snitch and IPU accelerator

as performs operations of custom smallfloat formats. While including its own register file and decoder, the floating point unit acts as an accelerator, being connected to the integer core with an accelerator interface. Another accelerator that the mainstream version includes is the SSR engine and FREP unit, which are used for specialized instructions that reduce loop overhead and eliminate redundant instruction fetches. The fourth accelerator that is currently used is a hardware DMA engine for offloading large memory transfers, enabling high-throughput data movement.[13] Also, the early versions of Snitch included a dedicated Integer Processing Unit that implemented rv32_b extensions as well as custom instructions. However, this module is not used by current projects. That is why it is suggested to eliminate it from the new version of the Snitch cluster.

In principle, the Snitch cluster defines the acc_addr_e type for enumerating accelerators used in the design. Adding a new accelerator requires adding a new address for it. In my work, a custom IPU engine was replaced by an Xpulpv2 Integer Processing Unit; the only required change was the address name. Also, in Snitch Core Complex, the dedicated signals for the new accelerator should be connected to the stream demultiplexer and stream arbiter ports. This integration is required to ensure that the request generated by the core is correctly dispatched to the corresponding accelerator, and that responses from multiple hardware units are arbitrated consistently before being forwarded back to the core.

Many instructions from Xpulpv2, such as MAC, insert, and some SIMD operations, require the current value of rd as one of the operands in addition to conventional source operands and immediates. To support this functionality, operands fed to the offloading accelerator interface were modified to include three operands.

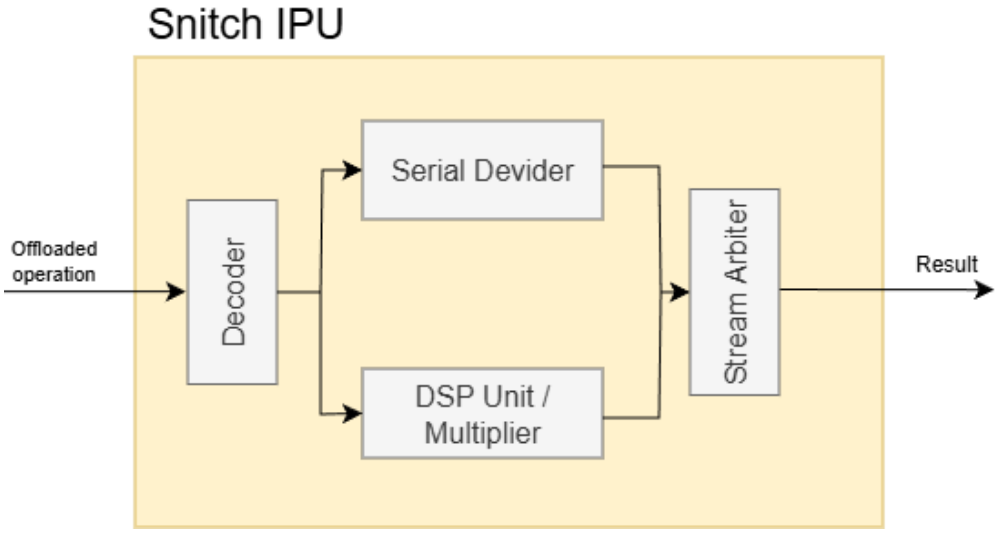


Figure 4.3.: Integer Processing Unit

The Integer Processing Unit for Mempool was designed and presented by Sergio Mazzola in his Master's thesis[12]. For the current work, IPU was taken as a finished functional unit from the Mempool project without making modifications to it. However, it is worth considering its main components and operations that it implements.

Firstly, the Integer Processing Unit includes its own decoder for the instructions that were offloaded to it. From the RTL point of view, the decoder is built in the same manner as the decoder in the Snitch Integer core. The only difference is that it includes only the instructions that are executed on this accelerator. Currently, the Snitch IPU includes two configurations: a custom DSP Unit or a Multiplier. The Multiplier here is identical to the one used in Shared Muldiv in the Snitch cluster. So, after decoding, the operation is executed on the Serial Divider, DSP Unit, or Multiplier, depending on the configuration and instruction. After the execution, the Stream Arbiter regulates the path to the control core. The DSP engine is implemented as a single-stage execution unit. It includes an internal decoder, which interprets the opcode of the offloaded instruction and produces an intermediate control representation to drive the internal units. The overall architecture of the DSPU is presented in Figure 4.3.

The DSPU contains the following functional units:

1. Shared comparator, which is used by arithmetic and clipping operations;
2. Arithmetic unit for basic arithmetic functions, which are included in the Xpulpv2 extension;
3. Extension unit that manages sign- and zero-extensions;

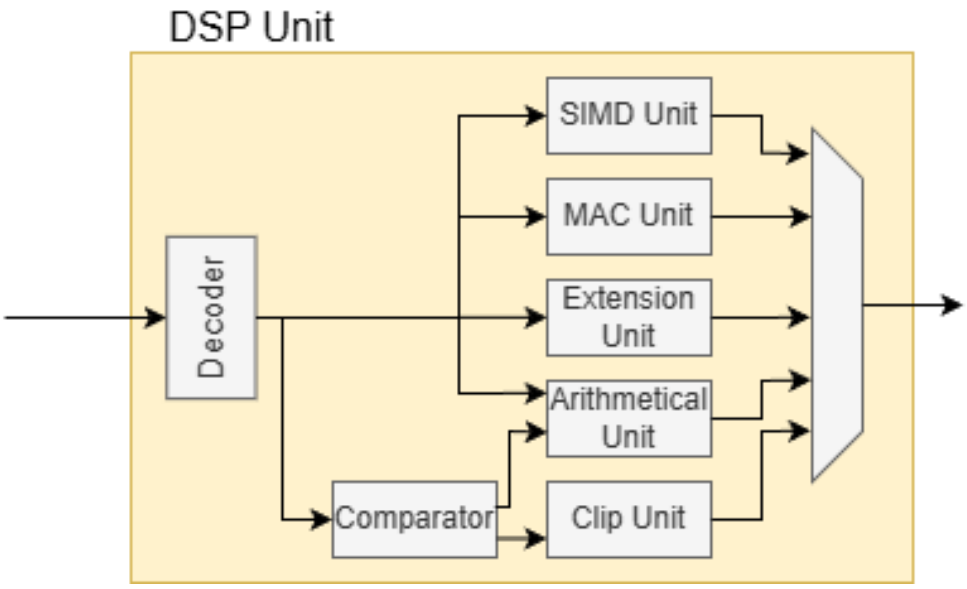


Figure 4.4.: DSP Unit diagram

4. Multiplier, which executes multiplication instructions and MAC operations;
5. SIMD unit, which enables support of all packed-SIMD instructions;

The DSPU functional blocks execute different subsets of the Xpulpv2 extension depending on the type of operation they implement. The arithmetic unit, paired with a shared comparator, executes generic arithmetic instructions, including absolute value, signed and unsigned comparisons, and minimum/maximum selection (Xpulpabs, Xpulpslet, Xpulpminmax subextensions). Clip operations (Xpulpclip) are handled by the dedicated clip unit. The MAC unit is integrated with the multiplier that executes all multiply-accumulate instructions (Xpulpmacsi subextension). The extension unit is responsible for width-adaptation operations proposed in Xpulpv2, namely p.exth and p.extb. And finally, packed-SIMD arithmetic, logical, and shuffle/pack instructions are executed by the SIMD unit. This modular organization ensures the usage of units best suited for the desired computation, as well as maximizes resource reuse.

Chapter 5

Verification

This chapter describes the verification methodology applied to validate the correctness and robustness of the implemented extensions to the Snitch cluster. It includes simple software tests and CI flow for Xpulpv2 extension development, and new target for LLVM compiler. The second part of the chapter describes open-source synthesis flow.

5.1. Xpulpv2 extension testing

To comply with the current upstream version of the Snitch cluster and to verify the correctness of the implemented extensions, simple software tests should be designed and included in the GitHub CI flow.

Since the operation organization in some way is similar to floating point operations, the tests for floating point sub-extensions can serve as a good example of the software testing construction. Most Xpulpv2 operations are offloaded to the accelerator, perform some arithmetical operation, and return the result, so the approach of tests can be the same as for floating point operations. [1] The organization of the tests in the Snitch repository for floating-point operations is the following:

The test is structured as a deterministic self-checking validation that executes exclusively on Snitch compute cores, using Snitch runtime Application Programming Interface (API). It has a modular organization: first, the program initializes a global error counter. It stores the number of tests and is decremented for each successful check. Then, the program proceeds through the separate test blocks for each instruction. Before instruction execution,

[1] https://github.com/pulp-platform/snitch_cluster/blob/main/sw/tests

input data is prepared in integer registers and transferred into floating-point registers to ensure deterministic, bit-accurate initialization. Each instruction under test is issued in inline assembly, ensuring that the exact RISC-V instruction is executed and preventing the compiler from replacing or transforming it. After execution, the software test compares the result with a preloaded reference value. Based on the comparison results, the global error counter is updated accordingly. After execution of all test blocks, the compute core returns the final value of the error counter. This enables an automated test framework to detect pass/fail conditions in simulation. The results of all software tests are checked in CI to verify proper performance across all hardware units.

Based on this example simple software tests for Xpulpv2 instructions were designed. At the beginning of the work, the snitch branch of llvm compiler on GitHub did not include mempool model to use for compilation. So, firstly, some instructions were tested using the following asm volatile semantics (an example for a postmod instruction):

```
asm volatile(                              //incr +1 to rs1
            ".word (0b000000000001 << 20) | \
             (     (11) << 15) | \
             (    0b000 << 12) | \
             (    (10) << 7)   | \
             (0b0001011 <<  0)   \n"
             :"+r"(rs1)
             :"=r"(rd)
             );
```

Although this representation clearly shows the location of each field of instruction opcode, and it may be helpful from a debugging point of view, it is entirely inconvenient to manipulate and is not suitable for regular practical use of instructions in software programs.

5.2. LLVM compiler modification

However, the issue with different compilation targets cannot be resolved by adding new sub-extensions to the target, because instruction opcodes overlap. For this purpose, a dedicated snitch-mempool Central Processing Unit (CPU) target in llvm for the snitch branch was created.[2] This change adds a new "snitch-mempool" model target for the LLVM compiler that includes

[2]https://github.com/pulp-platform/llvm-project/tree/snitch

the xpulpv2 extension and other extensions from snitch, excluding the overlapping ones. The dedicated flag "SNITCH_MEMPOOL" was added to the snitch make files to mark if the snitch-mempool model in the LLVM compiler should be used. The flag should be set to "ON" if the snitch mempool version is compiled.

With a dedicated compiler target and test examples in the upstream Snitch repository, a dedicated test was written for each added instruction. Here is a simple example for the test of p.abs instruction from Xpulpabs sub-extension. Firstly, to simplify debugging, it was decided to run tests on a single core with a fixed number. Thus, the generated traces can be easily checked. For post-increment load and store instruction usage of one core only is also practical since it eliminates possible memory conflicts and as a result, simplifies debugging. The values of integer registers are also predefined, as in other Snitch SoC software tests. Also, specific registers are used here. This also simplifies the debugging of instruction execution in traces. The result of the operation is compared with a predefined value; based on the comparison, the error count is incremented or not. The program will return zero upon successful execution.

```
#include <snrt.h>

int main() {
    uint32_t i = snrt_global_core_idx();
    snrt_cluster_hw_barrier();
    if (i == 2) {
        int errs = 0;
        int32_t result_rd;
        int32_t result_rs1;
        register int32_t rd asm("a3") = 0;
        register int32_t rs1 asm("a4") = -42;
        asm volatile(
            "p.abs a3, a4\n"
            : "=r"(rd)
            : "r"(rs1)
            : "a3", "a4"
        );
        result_rd = rd;
        result_rs1 = rs1;
        if (!((result_rd == 42) && (result_rs1 == -42))) {
            errs = errs + 1;
        }
        return errs;
```

```
    } else
        return 0;
    snrt_cluster_hw_barrier();
    return 0;
}
```

Software tests for all the instructions that were added in the current work can be found on the GitHub page of the project [3]

5.3. CI flow adaptation

Continuous Integration is a development practice in which changes to a shared repository are committed frequently and are automatically validated. Committing code often enables earlier error detection and reduces the amount of code a developer must review when debugging. Updating code frequently makes it easier to merge changes from different team members. In RTL design, CI plays a crucial role in ensuring functional correctness and maintaining design consistency. Whenever new code is committed to the repository, a CI system can automatically trigger a sequence of checks. Building and testing code or hardware requires a server. A CI workflow using GitHub Actions offers two options for that: workflows can run on GitHub-hosted virtual machines, or on machines that the user hosts[14]. In the context of an open-source project the use of GitHub virtual machines is preferable. After the commit is pushed to the remote repository, the CI workflow runs tests and reports the results, so a developer can see whether the change introduced an error. GitHub Actions suggests the CI workflow template based on the repository's language and framework. However, hardware design often requires specific tests. A great feature of GitHub CI is the possibility to create your own custom workflow file to run CI tests. A custom GitHub CI flow was created in the Snitch repository. The CI flow is based on GitHub Actions and utilizes a Docker-based infrastructure. Docker[4] is the container platform that allows implementing secure, independent packages called containers. Usage of Docker infrastructure guarantees a deterministic environment for all verification steps. In Snitch CI, after the push to GitHub the workflow automatically creates two container images: first for hardware simulation and the second for software compilation and simulation.

The next CI jobs build and deploy project documentation based on the hardware container image. The job pytest runs tests on Python scripts used in

[3]https://github.com/pulp-platform/snitch_cluster/tree/develop
[4]https://www.docker.com

the project. The following jobs simulate the Snitch Cluster itself using Verilator. First, the hardware is compiled for the Verilator simulator.[5]. The open-source tool Verilator is used for this project to translate SystemVerilog RTL code into a cycle-accurate simulation model without requiring proprietary EDA simulators. The special configuration is used for hardware; in this configuration, a cluster includes only 4 compute cores and 1 DMA core. If the hardware compiles successfully, the CI reports a positive test result. The next job is the compilation and execution of the software programs built for the Snitch cluster. This CI step runs software tests using a Python script that executes the programs on the Verilator model, verifies the results, and returns a summary table. Using this table, the developer can easily understand which hardware modules are malfunctioning. After the traces are annotated, the simulation is run for a simple program. And finally, the last job does the comparison between automatically generated files and the current RTL files in the directories. This ensures the consistency between files and the repository, and automatically generated files.

For the Mempool configuration of the Snitch SoC a separate job was added during my work. First, in a hardware container, the hardware testbench for the Verilator is built in the hardware contatiner. After that, the software is compiled in the same container for the snitch-mempool compiler target. In the same way as for the default configuration, the tests for the mempool version of the cluster, including xpulpv2 and other common tests, are simulated.

This part of the design is necessary for new configurations of the Snitch cluster to ensure the correctness of their implementation and to guarantee that new architectural configurations are compatible with the toolchain.

5.4. Open-source synthesis flow

The Snitch SoC is an open-source project that enables a broad audience to explore its features, extend it, and build new projects on top of it. However, an essential part of its State Of Art (SOA): the backend flow, was still missing, preventing developers from getting full Power, Performance, and Area (PPA) information. Currently, only researchers from Integrated Systems Laboratory (IIS) group were able to run backend flow and post-synthesis or post-layout simulations. Nevertheless, even the synthesis flow can yield helpful information, including the configuration area, design correctness, and a possible

[5]https://www.veripool.org/verilator/

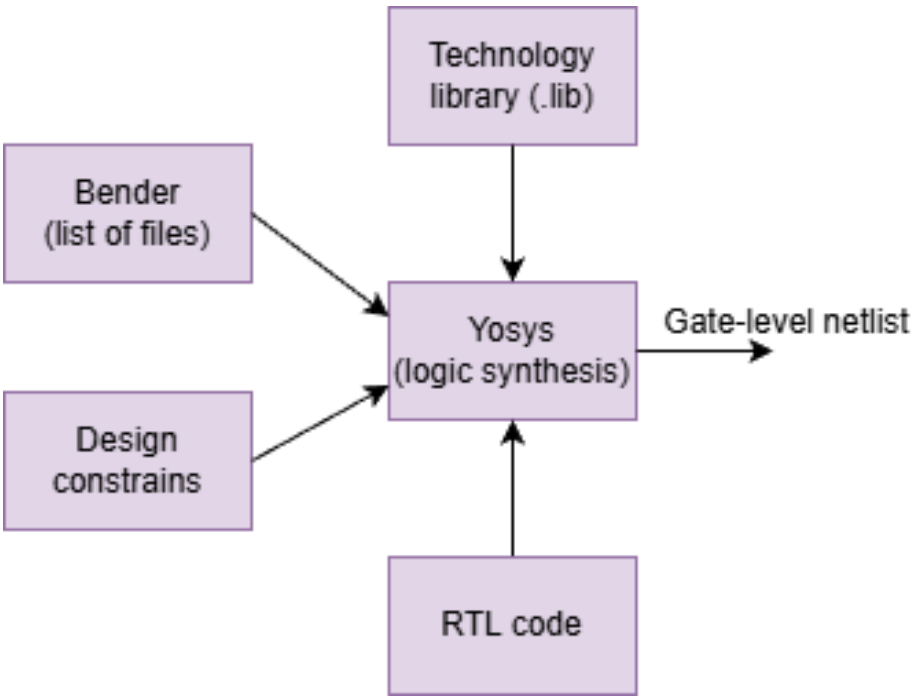


Figure 5.1.: Synthesis diagram

target frequency. This part of the research aimed to provide an open-source flow, suitable for these needs.

The Croc open-source backend flow[1] was selected as the reference for this work. It includes all scripts necessary to produce a nearly finished chip in IHP 130nm open-source technology[6].

Specifically for synthesis, the open-source Yosys 0.54 tool[7] was used. The EDA synthesis tool takes as input the RTL code described by Hardware Description Language (HDL), a list of files used in the project, a set of constraints, Yosys specifically includes only timing constraints, and design library (Liberty .lib) files from the desired technology. The Yosys output is a gate-level netlist, which consists only of gates from the provided technology logic library and wires connecting them. The diagram illustrates the process

Synthesis in the Yosys flow for Application-Specific Integrated Circuit (ASIC) consists of the following steps, which transform RTL code into a gate-level netlist using a standard-cell library.

- Front-end Preprocessing: Yosys first reads and parses the RTL code and builds an internal abstract syntax tree (AST). This structure represents the design in a form that the tool can use during the next steps

[6]https://github.com/IHP-GmbH/IHP-Open-PDK/tree/main
[7]https://www.yosyshq.com/

- Elaboration: during this step, Yosys expands all module instantiations, resolves parameters, and flattens the hierarchy. After this step, the design is fully expanded and expressed as a technology-independent netlist.
- Coarse Synthesis: Yosys converts the design into a network of generic logic elements, such as basic gates, multiplexers, and adders. And prepares the design for technology mapping
- The final step is technology mapping, in which generic logic elements are replaced with standard cells from the target library. The .lib timing models are used to choose proper gates. The Croc flow utilizes the ABC tool to perform standard cell mapping for combinational circuits and perform optimizations.

The considered synthesis flow uses the Bender tool[8] to generate a list of RTL files. For the Snitch cluster, this operation is performed the same way as for Croc. By listing all required targets, Bender automatically generates the desired file list.

The features of the synthesis flow for Snitch SoC are the following: first, the keep_hierarchy attribute was applied to the main components of the SoC, preventing hierarchy flattening during synthesis. This preserves the structural boundaries between major blocks such as the core, register file, LSU, FPU, and memory blocks. In particular, this will be useful for the area analysis of modules. The blackbox attribute is applied to tc_sram_blackbox since Snitch cluster includes TCDM cache, which will be generated as a macro.

The next peculiarity of the flow is the need to map latches to the technology. The Snitch cluster includes L1 memory: data and tag. Since this is a relatively small memory, latches can be utilized for this purpose. Moreover, the Pulp project introduced a flag that specifies whether the cache should be latch-based or a macro. However, the question of which type of memory is more prefered to use in the Snitch was not investigated and remains a question for the future works. For this purpose, a tehchmap file was written. During this process Yosys compares internal Yosys cells, which represent latches in the netlist, and if a specific type of latch is found, the tool replaces it with the technology-specific cell.[15] In particular, in Snitch synthesis flow sg13g2_dlhq_1 latches from IHP 130nm technology were used.

Another feature of the Snitch cluster project is that the distributed configuration of the FPU was used. The Yosys tool does not currently support register

[8]https://github.com/pulp-platform/bender

retiming, while the FPU pipeline was based on this function of proprietary tools. Without the distributed configuration, the FPU violates timings.

The target frequency was specified at 100 MHz, the same as Croc uses. Frequency is technology-dependent value. However, the maximum possible frequency utilizing this technology for this project can be investigated in future projects.

To validate the synthesized netlist, a QuestaSim post-synthesis simulation was performed. The synthesized netlist was set as top module, and behavioral models of gates were loaded. The resulting simulation performs gate-precise behavioral simulation. Using this simulation, the synthesis results were verified.

To verify that the design meets the required frequency, static timing analysis was performed. And specifically, the setup time should be checked. For these purposes, the OpenSTA tool, which is a part of OpenRoad, was used. In the current work, Openroad v2.0-16235-gedf00dff9 was used. Using the synthesized netlist and the corresponding standard-cell library, OpenSTA performs a full static timing analysis by propagating delays through combinational logic and sequential elements, while accounting for clock frequency and timing constraints. As a result of the STA, the design was found to achieve 80 MHz.

Chapter 6

Results

In the chapter, the performance and area of different Snitch SoC configurations are compared.

6.1. Performance for different configurations

The Snitch runtime software enables measurement of the number of cycles required to execute a given portion of code.

In the current work, the performance of a common algorithm was evaluated. Integer matrix multiplication was chosen as an example of a parallel and widely used operation. The main code for the default and Xpulpv2 versions was taken from the Mempool repository [1].

The Snitch configuration with one hive and 8 compute cores was used for the test.

The main modification to the Snitch cluster compared with Mempool is the allocation of data to L1 memory before multiplication execution.

The results are presented on Figure 6.1. Currently, the default configuration uses a shared MulDiv for a Hive, which means that 8 cores share a single Multiplication unit. The MulDiv unit is crucial for the matrix multiplication program, since it performs multiplication of the register values. The result is 37822 instructions for 32x32 matrix multiplication. The next configuration includes a private Multiply-Divide Unit for every Core Complex. A Multiply-Divide Unit is connected as an accelerator for each integer core in the cluster. In the private MulDiv configuration, the contention on the

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

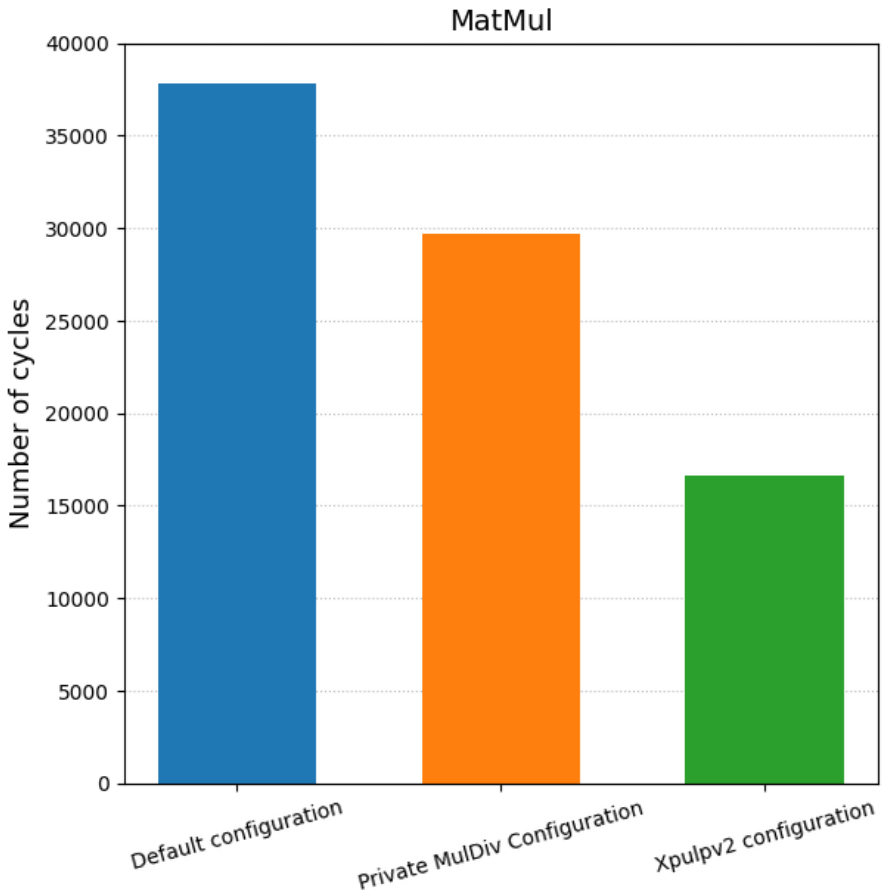


Figure 6.1.: Performance of Snitch Cluster with different configurations

shared multiplication unit is eliminated, since each core can issue multiplication and division operations independently on its own accelerator - private MulDiv. This reduces structural hazards and pipeline stalls caused by arbitration for the shared unit. As a result, the overall instruction throughput increases, especially for compute-intensive kernels such as matrix multiplication. In this configuration the number of instructions is 29725. Hence, the speedup is about 21% in comparison with the default configuration. The result is expected, since in this configuration, 8 MulDiv units are used for multiplication. For the Xpulpv2 configuration of the Snitch cluster the number of cycles is 16586, which means a speedup of 44% in comparison with a configuration with Private Muldivs. In the Xpulpv2 configuration, each Core Complex includes Snitch IPU as an accelerator instead of MulDiv. The custom instructions in the Xpulpv2 extension, designed for algorithms such as the Multiply-Accumulate instruction, enable a significant speedup.

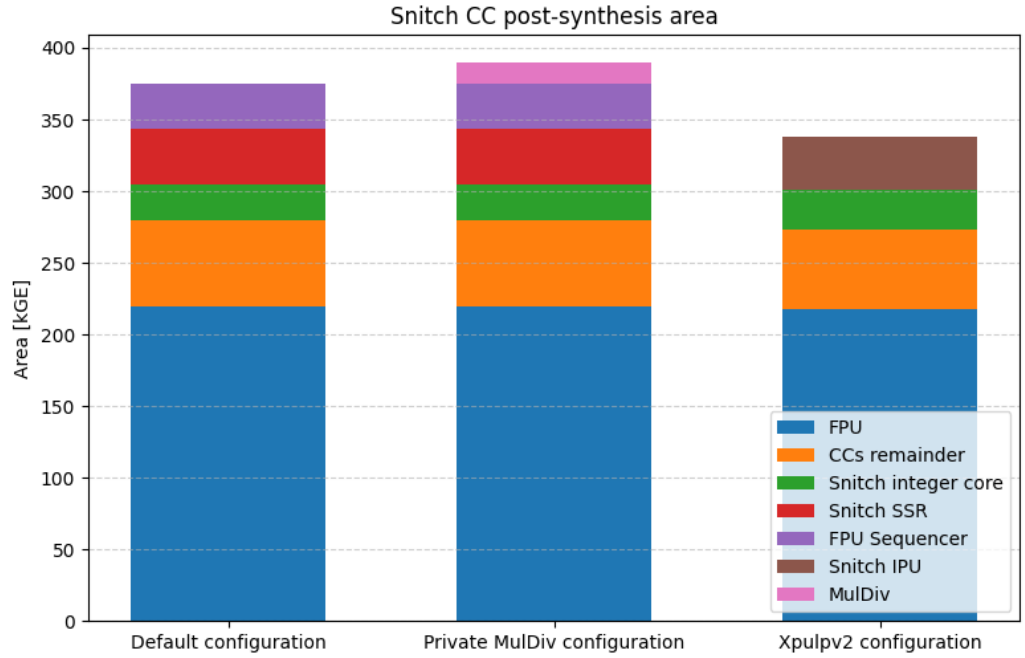


Figure 6.2.: Area of Snitch CC for different configurations

6.2. Area evaluation

The second contribution of the master's thesis is the open-source Synthesis flow, which was discussed earlier. It enables evaluation of the Snitch cluster area.

Here, we present the evaluation of Snitch Core Complex across three configurations used in the performance evaluation. The area of the Core Complex is interesting in this case, since the Xpulpv2 extension requires only changes to Core Complex, not other architecture. The area distribution between modules of Snitch Core Complex (CC) is presented in Figure 6.2.

The main contribution to the area is the Floating Point Unit, which area is 219 kGE. The Snitch Integer core has only 30 kGE, approximately 7 times smaller than the FPU. For the Xpulpv2 configuration, the integer core is 2.5kGE larger than in the default configuration because post-increment operations and immediate branches require additional hardware.

The Integer Processing Unit, which implements the biggest part of Xpulpv2 instructions is 36.6 kGE large. The MulDiv unit is 15.3 kGE, which is 2x smaller. In the second configuration, it is included in the Core Complex; in the first, it is shared. The are overhead for private MulDiv is 15.3kGE, which is only 4% of the area. However, due to custom extensions (FREP, SSR), the default configuration area is larger than in the Xpulpv2 version.

	Proprietary*	Open-source
Snitch core default	27.2 kGE	30.0 kGE
Snitch core Xpulpv2	31.3 kGE	32.4 kGE
MulDiv Unit	11.2 kGE	15.3 kGE
IPU	24.2 kGE	36.6 kGE

Table 6.1.: The area comparison

Additionally, to confirm the synthesis results, the module areas are compared. The kGE is a technology-independent value, so in principle, the comparison between the results of synthesis should give similar values. Table 6.1 presents the area of selected Snitch modules obtained in this work and compares them with the results reported in[12], in which the cluster was synthesised using proprietary tools*. In this work, a synthesis was performed using GlobalFoundries 22FDX FD-SOI technology using Synopsys Design Compiler 2019.12. The selected modules are the Snitch core in default and modified for Xpulpv2 configurations, Multiplication-Division Unit, and Integer Processing Unit. Only the IPU area differs from that obtained in[12]. However, the referenced work demonstrates that the area of the IPU depends on the target operating frequency. The values reported in Table 6.1 correspond to a target frequency of 500 MHz, whereas achieving higher target frequencies requires additional area due to more aggressive timing optimizations. Therefore, the observed differences in area are expected.

Here, the Snitch core for the default configuration and for Xpulpv configuration is presented. The table also includes Multiply-Divide (MulDiv) Units and the IPU. The results of Open-source synthesis are close enough to those achieved by proprietary tools. However, the synthesis results depend on the tool's logic optimization, and the results may vary slightly. To sum up, the open-source synthesis workflow provides useful information on module areas, which can be used by a broad audience. By making minor modifications, the area of other new custom accelerators can be expanded. However, making the open-source synthesis an essential part of the open-source benchmarking repository.

Chapter 7

Conclusion and Future Work

To conclude, in the current work, different configurations of the open-source project Snitch cluster were considered, and the problem of overlap of custom instruction opcodes was faced. Here on the example of the Xpulpv2 extension, a solution for configurable design in the presence of instruction overlap, proposed. It includes overlapping opcodes identification, modification of the Decoder in the integer core, and providing configuration flags. The solution also incorporates a separate CPU model in LLVM compiler for code compilation with desired extensions and simple software tests for the custom instructions. The resulting project was committed to the public Snitch repository.[1] The contribution to an open-source repository is valuable in the research community, since it can become a benchmark for the development of new custom extensions of the Snitch core for different problems and workloads.

The second contribution of the current work is the development of an open-source synthesis flow. Evaluation of Power, Performance and Area is a significant part of Integrated Circuit (IC) design. In the current work we propose a synthesis flow that allows getting the area of the cluster with the desired configuration as well as checking the timing constraints using open-source EDA tools and an open-source library. For reference, performance and area data were received for the different configurations of the Snitch cluster: the default configuration, the private MulDiv configuration and the Xpulpv2 configuration.

The post-synthesis area of the modules of the Snitch cluster was compared with the values from the literature. The results are close to those obtained by Proprietary tools. This indicates that the proposed open-source synthesis

[1] https://github.com/pulp-platform/snitch$_cluster/commit/6e2df136b80aa6e3ef9320d1368140f66a80446b$

flow provides a reliable estimation of the area characteristics of the Snitch cluster.

The current research opens an opportunity to the further development of configurable versions of the Snitch cluster targeting different application domains. The methodology applied in this work is not limited to the presented Xpulpv2 extension and can be reused for the integration of other custom ISA extensions, which can be implemented in the future work. The Snitch cluster open-source repository can become a benchmark for testing and evaluating custom user extensions, designed for different applications.

The open-source synthesis flow already provides valuable insights into the Snitch cluster area and timing. Nevertheless, it remains an open field for further investigation and improvement, since different logic optimization strategies, synthesis constraints, and technology mapping choices can significantly impact the synthesis result.

Also, a full back-end flow from RTL to GDS may be an aim of future research, as it would enable more realistic evaluations of area and timings by capturing physical design effects of placement, routing, and clock distribution, which are not visible at the synthesis level.

Bibliography

[1] P. Sauter, T. Benz, P. Scheffler, H. Pochert, L. Wüthrich, M. Povier, B. Muheim, F. K. Gürkaynak, and L. Benini, "Croc: An end-to-end open-source extensible risc-v mcu platform to democratize silicon," 2025. [Online]. Available: https://arxiv.org/abs/2502.05090

[2] K. Herman, R. Scholz, and S. Andreev, "Reflections on the first european open source pdk by ihp - experiences after one year and future activities," in 2024 31st International Conference on Mixed Design of Integrated Circuits and System (MIXDES), 2024, pp. 19–22.

[3] 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, vol. 70, no. 11, pp. 1845–1860, 2021.

[4] S. Riedel, M. Cavalcante, R. Andri, and L. Benini, "Mempool: A scalable manycore architecture with a low-latency shared l1 memory," IEEE Transactions on Computers, vol. 72, no. 12, p. 35613575, Dec. 2023. [Online]. Available: http://dx.doi.org/10.1109/TC.2023.3307796

[5] M. Perotti, S. Riedel, M. Cavalcante, and L. Benini, "Spatz: Clustering compact risc-v-based vector units to maximize computing efficiency," 2025. [Online]. Available: https://arxiv.org/abs/2309.10137

[6] Y. Tortorella, L. Bertaccini, L. Benini, D. Rossi, and F. Conti, "Redmule: A mixed-precision matrix-matrix operation engine for flexible and energy-efficient on-chip linear algebra and tinyml training acceleration," 2023. [Online]. Available: https://arxiv.org/abs/2301.03904

[7] R. A. Antonio, J. Dumoulin, X. Yi, J. V. Delm, Y. Deng, G. Paim, and M. Verhelst, "An open-source hw-sw co-development framework enabling efficient multi-accelerator systems," 2025. [Online]. Available: https://arxiv.org/abs/2508.14582

[8] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores," IEEE Transactions on Computers, vol. 70, no. 2, pp. 212–227, 2021.

[9] L. Colagrande and L. Benini, "Dual-issue execution of mixed integer and floating-point workloads on energy-efficient in-order RISC-V cores," arXiv preprint arXiv:250x.xxxxx, 2025, presented at Design Automation Conference (DAC 2025). [Online]. Available: https://doi.org/10.1109/DAC63849.2025.11132520

[10] G. Paulin, P. Scheffler, T. Benz, M. Cavalcante, T. Fischer, M. Eggimann, Y. Zhang et al., "Occamy: A 432-core 28.1 dp-gflop/s/w 83% fpu utilization dual-chiplet, dual-hbm2e risc-v-based accelerator for stencil and sparse linear algebra computations with 8-to-64-bit floating-point support in 12nm finfet," arXiv preprint arXiv:2406.15068, 2024. [Online]. Available: https://arxiv.org/abs/2406.15068

[11] A. Waterman, Y. Lee, D. Patterson, and K. Asanović, The RISC–V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1, CS Division, EECS Department, University of California, Berkeley, May 2016.

[12] S. Mazzola, "Isa extensions in the snitch processor for signal processing," Master's thesis, Politecnico di Torino, Torino, Italy, 2021, supervisors: Alberto Macii, Luca Benini, Samuel Riedel, Matheus De Araujo Cavalcante.

[13] T. Benz, M. Rogenmoser, P. Scheffler, S. Riedel, A. Ottaviano, A. Kurth, T. Hoefler, and L. Benini, "A high-performance, energy-efficient modular dma engine architecture," IEEE Transactions on Computers, vol. 73, no. 1, pp. 263–277, 2024.

[14] GitHub, Inc., "Github actions documentation," https://docs.github.com/en/actions, 2025, accessed: 2025-02-12.

[15] Yosys Developers, Yosys Open SYnthesis Suite Documentation, YosysHQ GmbH, 2025, accessed: December 16, 2025. [Online]. Available: https://yosyshq.readthedocs.io/projects/yosys/en/latest/

Appendix A

Opcodes analysis table

