Streaming Integer Extensions for Snitch

by

Chen Sun

to obtain the degree of
Master of Science in Computer Engineering
at the Delft University of Technology,
Faculty of Electrical Engineering, Mathematics and Computer Science,
to be defended publicly on Friday September 30, 2022 at 1:00 PM.

Student number:	5262763	
Project Duration:	February, 2022 - September, 2022	
Thesis committee:	Dr.ir, J.S.S.M. Wong	TU Delft, supervisor
	Dr.ir.T.G.R.M.van Leuken	TU Delft
	Sergio Mazzola	ETH Zurich, daily supervisor

An electronic version of this thesis is available at
https://repository.tudelft.nl/

Abstract

The prosperity of the Internet-of-Things (IoT) imposes increasing demand on endpoint microcontroller-based devices' performance and energy efficiency. The MCUs are demanded to process the raw data acquired from the sensors with the integer-based workload, such as digital signal processing (DSP) algorithms and quantized neural network (QNN) inference. Currently, the Snitch system built around the tiny RV32I Snitch core aims to achieve high performance in floating-point applications. Novel hardware extensions have been implemented in its floating-point subsystem to achieve high floating-point unit (FPU) utilization, such as stream semantic registers (SSRs) and floating-point repetition (FREP) hardware loop. However, it only has RV32IM instruction set support for integer computation, which does not satisfy the increasing demand from the integer workload we mentioned. In this work, we present a unified Snitch architecture with integer extensions targeting integer workload acceleration. Some existing custom extensions to address performance bottlenecks in DSP and QNN applications were proposed, which are Xpulpimg ISA and sub-byte single-instruction-multiple-data (SIMD) ISA, respectively. Both extensions are built on the outdated version of Snitch in another many-core system Mempool. In our work, we first integrated the DSP-oriented ISA extension Xpulpimg and the sub-byte SIMD ISA extension into the mainline Snitch. Then we extended the existing floating-point SSR to have integer support. To evaluate the proposed extensions, we benchmarked the Snitch core complex (CC) with integer matrix multiplication algorithms and compared the performance between the baseline RV32IM Snitch and our extensions. A speedup of 5.9×, 22.6×, and 77.4× in terms of MACs/cycle with respect to the baseline was measured for 32-bit, 8-bit and 4-bit data sizes, respectively. Post-synthesis figures have been obtained from GlobalFoundries 22 nm technology for area and timing evaluations. Our integer extensions only introduced 12% area overhead compared with the original FP-capable Snitch CC, and they led to no measurable impact in terms of the maximum effective frequency with FP extensions enabled.

Acknowledgments

This project marked the end of my student's life at TU Delft and ETH Zurich. My study life during the last two years is not an easy one, but definitely a memorable experience.

First of all, I would like to thank Professor Luca Benini for giving me this opportunity to work on this exciting and challenging project in Integrated Systems Laboratory at ETH Zurich.

Then I would like to thank my daily advisors, Sergio Mazzola, Paul Scheffler, Georg Rutishauser, and Samuel Riedel, for their relentless support and guidance throughout the project. I have learned a wide range of knowledge, from small details of digital design and computer architecture to working systematically as an engineer.

I also would like to thank my supervisor at TU Delft, Professor Stephan Wong, for his continuous help and guidance remotely in the Netherlands.

Last but not least, I am thankful to my parents and families, as they always support my decisions and give me the relief and power to study and live in another country. I want to thank my friends, no matter old or new, no matter living in the same city or even overseas. We had a lot of great times together during this long journey. In particular, I would like to thank my girlfriend Yue Zhang who always encourages me and brings joy into my life.

Contents

Chapter 1

Introduction

In recent years we are experiencing the exponential growth of the Internet-of-Things (IoT). Increasing demand for small and battery-powered IoT endpoint devices is imposed for various application domains, such as agriculture [1], health monitoring [2], and smart home [3]. IoT requires the devices to interact with the environment and communicate over a low-power wireless network after applying signal processing algorithms [4]. Current IoT endpoint devices integrate multiple sensors and are built around microcontroller units (MCUs). MCUs are efficient for controlling purposes and processing data sampled from the sensors. Machine learning algorithms, including deep learning algorithms, are widely deployed on IoT nodes as they empower the devices with smart capabilities and provide efficient processing or classification of the raw data acquired by sensors [5]. Modern deep neural network (DNN) applications are improved to overcome the limitations in memory footprint and computing resources with quantization methods. These techniques reduce the precision of the DNNs to 8-bit, 4-bit, or even lower, incurring a limited or negligible loss in accuracy [6] [7], making DNNs more suitable to run on endpoint platforms. With the ubiquitousness of these prominent integer-based workloads, such as digital signal processing (DSP) and low-precision machine learning algorithms, it is necessary to improve the performance of the hardware for lower response latency and higher energy efficiency.

An explosion in the number of hardware accelerators customized to solve one particular problem efficiently has been observed [8] [9] [10] [11] [12] [13]. The accelerators are great candidates to speed up certain tasks and enhance the performance of a general-purpose system as described in [14]. However, they lack the programmability for all kinds of flexible demand [15]. Single issue processor cores are very energy-efficient and flexible. However, they suffer from the von Neumann bottleneck as they must explicitly fetch and issue the load and store operations to feed their functional unit or accelerators. Techniques to mitigate this issue, such as caching [16] and super-scalar out-of-order pipelines [17], lead to either high hardware complexity or low energy efficiency. Snitch is the in-house solution striving to address these challenges [18]. The Snitch ecosystem is an in-house solution targeting energy-efficient, high-performance systems. It is built

around the minimal RISC-V Snitch integer core, only about 15 kilo-gate equivalents in size. It can optionally be coupled to accelerators such as a floating-point unit (FPU) or a direct memory access (DMA) engine. Snitch's floating-point subsystem is improved with lightweight extensions, achieving almost 100% FPU utilization in many data-oblivious problems with regular access patterns. However, there is still space to improve the integer performance in the Snitch system as it only has base RISC-V integer ISA support for integer computations.

1.1 Problem statement

Recently, some accelerator-based ISA extensions for Snitch have been explored: a subset of the custom Xpulpv2 extension for DSP applications, namely Xpulpimg [19], and a set of SIMD arithmetic instructions operating on packed sub-byte (4-bit) values [20]. Both these extensions aim to boost the performance and energy efficiency of integer-based workloads such as signal processing and low-/mixed-precision machine learning algorithms. Another interesting hardware extension is known as *Stream Semantic Registers* (SSRs) proposed in the Snitch system. It maps the ISA registers to memory streams, which reduces the pressure on the processor to handle data loading, storing, and iterations. This extension also helps the Snitch system to achieve high functional unit utilization in dense floating-point workloads [18].

However, there is no unified version of Snitch that consists of all of these useful extensions. The Xpulpimg and sub-byte SIMD ISA extensions were implemented in another Snitch-based many-core system called *Mempool* targeting image signal processing. In this system, the outdated version of the Snitch core was used as a general-purpose control processor and extended to support these ISA extensions for DSP algorithms. Meanwhile, an integer processing unit (IPU) was implemented to process most performance-oriented instructions introduced in these two ISA extensions as an accelerator. On the other hand, the SSRs only support FPU in the mainline Snitch.

The need for fast and efficient integer data processing and the existing problems lead to the following research questions:

- *How can we define a unified Snitch architecture with available and possible integer extensions for efficient processing of integer workloads?*
- *What are the performance benefits and cost of the potential integer extensions in Snitch?*

1.2 Goals and methodology

We aim to create a unified Snitch system targeting integer performance to tackle the above-mentioned problems with the following contributions:

- the integration of the existing Xpulpimg and sub-byte SIMD ISA extensions into the mainline Snitch system;

- the extension of the existing floating-point SSRs to support integer SSRs but maintain all the floating-point abilities, and the integration of the integer SSRs into the mainline Snitch system;
- the evaluation of these integer extensions in terms of performance, area, and timing. In particular, we measured the performance by software benchmarking and collected the area and timing results from the post-synthesis results.

1.3 Overview

The remainder of this report is organized as follows: in Chapter 2, we first introduce the background and the preliminary knowledge to follow our work's description. Chapter 3 introduces our main contribution to this project, especially the hardware architecture and its implementation. Chapter 4 describes our evaluation methodology and the results in terms of software benchmarking and synthesis. We draw the final conclusion of the project and identify the outlook for future work in Chapter 5.

Chapter 2

Background

This chapter will introduce the preliminary knowledge needed to understand the subsequent chapters of our work. We will first start to introduce the *RISC-V* instruction set architecture (ISA) in Section 2.1. Then *stream semantic registers* (SSRs) that we will extend for integer support, will be introduced in Section 2.2. Section 2.3 introduces the *Snitch* system, which is the main hardware platform we will work on in this project. Section 2.4 introduces a Snitch-based many-core image signal processor *Mempool*. It was enhanced with DSP-oriented ISA extensions that we will port to the mainline Snitch in our work.

2.1 RISC-V open ISA

Instruction set architecture (ISA) is an abstract model of a computer. It specifies the behavior that how machine code can run on the hardware, and it is independent of the characteristics of the hardware implementation, providing binary compatibility [21]. RISC-V is an instruction set architecture (ISA) conceived by developers at the University of California, Berkeley, in 2010 [22]. It is a unique, even revolutionary innovation since it is a common, free, and open-source ISA to which software can be ported, hardware can be developed, and processors can be built to support it. This is in contrast to most common ISAs, which were kept proprietary for either historical or business reasons, such as x86, ARMv8 and AMD64. Although they are still widely used in industry and academia, there are still disadvantages: they could preclude the creation and sharing of full RTL implementations of them as they require commercial licenses from their vendors, and they are complicated to fully implement in hardware [23]. Hence, the emergence of the RISC-V ISA offers a new possibility to define and create computer systems.

As the name suggests, RISC-V ISA makes use of a reduced instruction set computer (RISC) design. RISC-V is actually a family of related ISAs, and there are four base ISAs currently:

- RV32I: It is the base 32-bit integer ISA with 32 registers (`x0`-`x31`). Each register is

32-bit wide. Register `x0` is hardwired with all bits to 0, while the other 31 registers are general-purpose. The instruction encoding in RV32I is 32 bits fixed in length and must be aligned on a four-byte boundary in memory. It includes integer computation, control transfer, load and store, memory ordering, environment call and break points, and HINT instructions.

- RV32E: It is a subset variant of RV32I to support small microcontrollers with only 16 integer registers.
- RV64I: It is the 64-bit variant of RV32I.
- RV128I: It is the 128-bit variant of RV32I.

The RISC-V ISA has high modularity and flexibility to support various application scenarios, from power-efficient embedded processors to high-performance processors. It can be extended with instruction set extensions based on a base integer ISA to provide additional functionalities. Table 2.1 lists some standard extensions proposed in [22]. Apart from the standard extensions, RISC-V also supports independently developed extensions from developers of academia and industry. It allows users to develop highly specialized custom accelerators for important application domains and also build prototypes for pushing the boundaries of cutting-edge research.

Extension	**Purpose**
M	Integer multiplication and division
A	Atomic instructions
F	Single-precision floating-point
D	Double-precision floating-point
Q	Quad-precision floating-point
L	Decimal floating-point
C	Compressed instructions
B	Bit manipulation
J	Dynamically translated languages
T	Transactional memory
P	Packed-SIMD
V	Vector operations
Zicsr	Control and status register
Counters	performance counters and timers
Zifencei	Instruction-fetch fence
Zam	Misaligned atomics
Ztso	Total store ordering

Table 2.1: Standard RISC-V instruction set extensions [22]

A significant benefit of using RISC-V is that all cores, whether based on the basic open-source RISC-V ISA or in-house designs, will meet the exact RISC-V specification,

ensuring the ability to use a standard software toolchain. Hence, the RISC-V ecosystem is necessary to make this open-source ISA thrive. The ecosystem components are diverse, spreading across all layers from low-level firmware and compilers to a fully functional operating system kernel, applications, and design and verification tools. They are essential to ensure the success of RISC-V.

2.2 Stream semantic registers (SSRs)

Stream semantic registers (SSRs) are an ISA extension proposed in [24] to accelerate data-oblivious problems [25]. The fundamental idea of this extension is to map registers to memory streams, so that load and store instructions that follow affine access patterns are converted to register read/writes [24]. Their high-level architecture and integration into a RISC-V floating-point subsystem are demonstrated in Figure 2.1. In this example, three floating-point registers are mapped to three SSR lanes so that the register read and write requested can be sent to either the FPU register file or rerouted to the SSR streamer. The SSR streamer consists of a switch mapping accesses to the selected architectural registers to the targeted SSR lanes for memory streams. The address generators are used to emit affine address patterns for the memory streams in advance via a memory-mapped configuration interface between the processor core and the streamer. The FIFO inside each SSR lane can decouple the response data from memory and hide the memory access latency. Each SSR lane can be put into either read or write mode. In read mode, the address generator can be configured to generate an address pattern based on the loops in the program, and then data is fetched from memory and stored in the FIFO; in write mode, the address generator can tag each data written into the FIFO with an address and send it to the memory system.

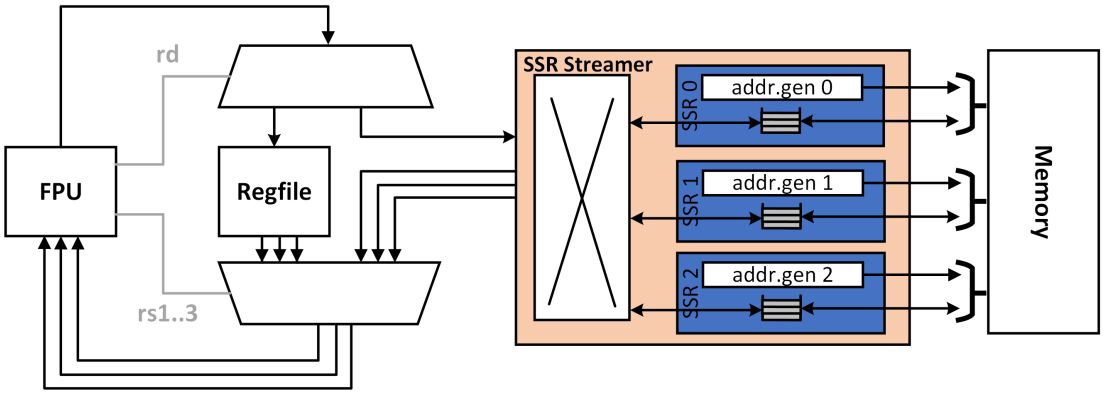

Figure 2.1: SSR architecture

The usage of SSRs follows a simple sequence:

1. Configuring the address pattern;
2. Enabling the SSRs by writing to a control and status register (CSR);

3. Computation;
4. Disabling the SSRs at the end of the program;

The configuration registers of the SSR streamer are memory mapped and can be accessed by the processor via load and store instructions, sitting inside each address generator. They can be set according to the nested loop dimensions, the increment strides of the address pointers, and the stream direction (read or write). There is a *repeat* configuration register that allows each data loaded from memory to be emitted to the core multiple times, which is useful if a value loaded from memory needs to be used as an operand multiple times. This programming model also allows the programmer to switch between conventional register file usage and stream semantics on-the-fly.

SSRs aim to improve functional unit (FU) utilization. In RISC architecture like RISC-V, the FU utilization is poor: integer ALUs and FPUs are rarely kept busy in every cycle. They only need to process one computation in a few cycles. The micro-architecture changes to tackle this issue do not come for free and are usually expensive, such as out-of-order superscalar [26], complex instruction set computer (CISC) machines [16], vector processors [27], and VLIW cores [28]. SSRs eliminate explicit load and store instructions by encoding them into a subset of the processor's registers, increasing the proportion of instructions for useful work, especially computations. It is also a lightweight extension that leaves the existing instructions in the ISA untouched but allows them to leverage data streams with highly flexible usage.

2.3 The Snitch ecosystem

The Snitch ecosystem was first proposed in [18]. It provides a RISC-V cluster as a base for multicore systems targeting high-performance computing and high energy efficiency. It is built around the minimal Snitch integer core, only 15 kGE in size and supports the entire integer base RV32I instruction set architecture. The core is a general purpose, single-stage, single-issue, in-order design, aiming to maximize energy efficiency and minimize area footprint [18].

The design principle of the Snitch core is to achieve high flexibility and compute-to-control ratio with a small area overhead. Therefore, a generic accelerator interface was implemented in the Snitch core based on an AXI-like handshake, offloading entire 32-bit performance-oriented RISC-V instructions into various accelerators. This interface has two independent channels: one request channel for forwarding an instruction with up to three operands, and one response channel to write back the results [18]. This accelerator interface was extended with an arbiter to support different hardware extensions for various application domains at the CC's hardware abstraction level. For example, a SIMD-capable 64-bit FPU was coupled to the core to form a core complex (CC) supporting single- and double-precision floating-point extensions. And a cluster DMA was added to the default Snitch cluster which connects to a 512-bit system bus for data transfer between the cluster tightly coupled data memory (TCDM).

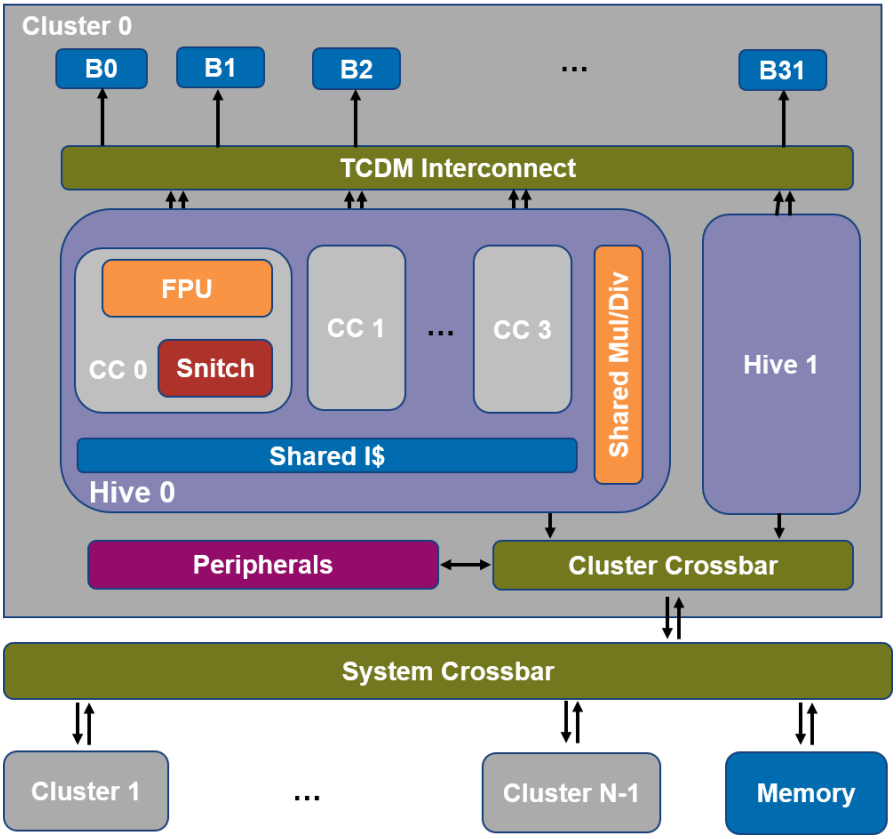

Figure 2.2: Hierarchy of the Snitch ecosystem

Figure 2.2 shows the hierarchical architecture of a Snitch cluster with default configurations. A Snitch integer core and a floating-point subsystem (FPSS) form a CC. The FPSS consists of a 64-bit FPU, a load-store unit (LSU), an SSR streamer with three SSR lanes, a register file and an instruction decoder. Also, an FPU instruction sequencer enabling the floating-point repetition (*FREP*) hardware loop is integrated between the core and the FP-SS. It can buffer new instructions offloaded by the Snitch core and allows us to loop over a given number of instructions a given number of times, by repeatedly emitting them to the FPSS. The core itself can continue working on the instruction stream with the FPSS working in parallel. The SSRs and FREP hardware extensions help the Snitch achieve a high compute-to-control ratio and FPU utilization, making the system a competitive candidate for high-performance floating-point applications. Table 2.2 gives a dot-product example of how SSRs and FREP improve performance: the left code solves the problem without, the right with SSRs and FREP extensions. The only essential operation for computation is `fmadd` in both cases. The left vanilla FPSS needs six instructions per hot loop due to loads, index incrementation and branching. While the right code configures the SSRs `ft0` and `ft1` to redirect the stream vectors with a one-time setup and runs only `fmadd` in the FREP loop.

Baseline		**SSRs and FREP**	
dotp: fld	ft0, 0(a0)	call	ssr_setup_ft0
fld	ft1, 0(a1)	call	ssr_setup_ft1
fmadd.d	ft2, ft0, ft1, ft2	frep	reps, 1, 0b1001, 4
addi	a0, a0, 8	fmadd.d	ft2, ft0, ft1, ft2
addi	a1, a1, 8		
bne	a0, t0, dotp		

Table 2.2: Assembly code improvement of a dot-product hot loop from the baseline to the SSRs and FREP extensions

By default, four CCs are grouped together to form a Snitch *hive*, an instruction cache, and a MULDIV are shared among them. Integer multiplication and division instructions can be offloaded to this shared MULDIV through the accelerator interface of the Snitch core. This design choice aims to share expensive and uncommonly used resources [29]. Two hives with a shared TCDM memory and some peripherals form a *cluster*. The system can be expanded further with multiple clusters and low-level system memory connected by a wide system crossbar for more intensive workloads. The cluster is highly parameterized. Hence different topologies are available for various purposes.

2.4 Mempool system

2.4.1 Mempool architecture

Mempool is a 32-bit many-core RISC-V system with 256 cores sharing a large pool of Scratchpad Memory [30]. Figure 2.3 presents an overview of the Mempool architecture. It consists of four local groups. Each group can communicate with other groups via butterfly network-on-chip. There are 16 tiles within each group as shown in the left part of Figure 2.3. The tiles are the base of Mempool's hierarchy. It consists of four Mempool core complex (CC), an L1 TCDM with 16 banks, and a 4-way L1 instruction cache as depicted in Figure 2.4. Each Mempool CC is built based on the vanilla Snitch core with RV32IMA ISA. And the core is coupled with an accelerator for integer multiplication and divisions. Similarly, tiles can also communicate with others within a local group via butterfly networks. Mempool can achieve high parallelism thanks to its architectural pattern, which makes it also a competent candidate for multimedia applications such as image processing.

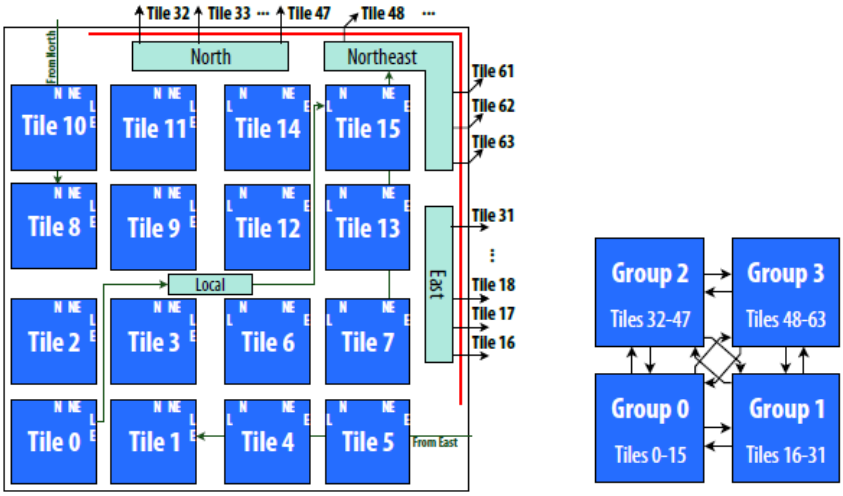

Figure 2.3: Architecture of the Mempool cluster [30]. The left shows a detailed view of the local group, the right presents the entire cluster formed by four local groups.

2.4.2 Xpulpimg ISA and sub-byte SIMD ISA extensions

Recently, Mempool is enhanced with the *Xpulpimg* ISA extension [19], which is a subset of the *Xpulp* custom RISC-V ISA for digital signal processing (DSP). The Xpulp instruction set was developed with the aim of helping the RISC-V processors proposed in [31] to achieve similar performance as the state-of-the-art proprietary-ISA-based MCUs for IoT applications.

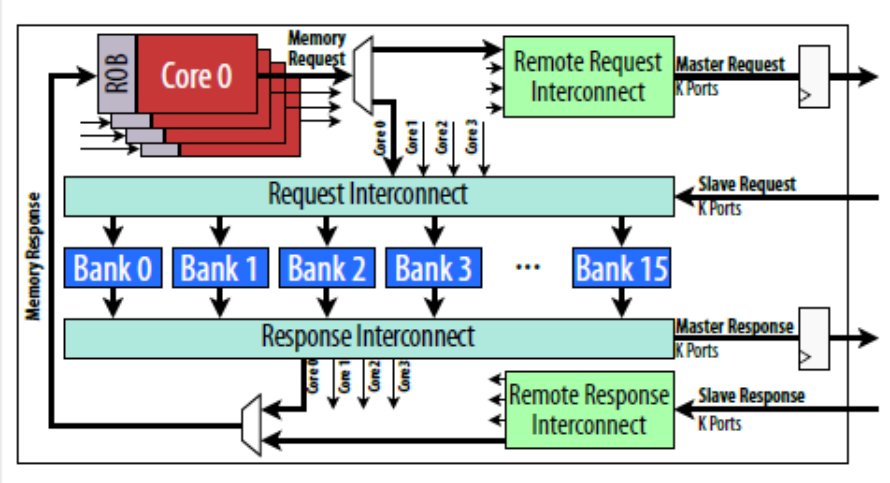

Figure 2.4: Architecture of Mempool tiles [30]

Xpulpimg ISA extension consists of the following parts [19]:

- Generic arithmetic and logic operations for DSP computations: such as absolute value, comparison, clip, sign- or zero-extension, immediate branching, etc.
- Extended addressing modes for load and store operations:
  - Post-increment mode: the actual address is computed at the same time as the memory accesses. It is updated by doing an addition between the base address and an offset stored in a register or the sign extension of an immediate, and the final address is written back to the registers.
  - Register-register mode: the actual address is computed as the sum of the offset and the base address. In this case, both the offset and the base address are stored in registers.
- Multiply-accumulate (MAC) operations;
- 16- or 8-bit packed SIMD operations that support three execution modes:
  - *vectorial* that uses two vectors from two registers;
  - *scalar* that uses one vector and one scalar from two registers;
  - *immediate scalar* that uses one vector and one scalar, the vector comes from the register while the scalar is in immediate format.

  In addition, there are different types of operations, such as generic arithmetic operations, sum-dot-product, bit manipulation, and shuffle operations.

This extension is selected and implemented based on the impact of the extended instructions on the software concerned with the image processing domain. For example,

dot-product operations on 16-bit or 8-bit input data are demonstrated to be suitable for media processing applications, which leads to high speedups [32] [33].

The Xpulpimg extension was implemented based on an outdated version of the Snitch core. In order to support these new instructions, the Snitch core was extended in a parameterized style to maintain its original extensibility without introducing extra area overhead on the baseline Snitch. The Xpulpimg extensions can be enabled or disabled to achieve an Xpulpimg-capable Snitch or a vanilla RV32IMA Snitch. The Xpulpimg ISA extension is an accelerator-based extension that most instructions are executed inside the accelerator instead of the Snitch core. This helps to maintain the extensibility of Snitch. Hence, the related hardware modifications mainly consist of two parts: the Snitch core and an Integer Processing Unit (IPU) as an accelerator. First, the Snitch core was extended to support new instructions and new addressing modes, which involve the decoding logic, the integer register file, the scoreboarding control logic, the write-back control logic, etc. Then, many arithmetic instructions in Xpulpimg were executed inside the IPU considering the modularity of the Snitch core. The IPU was connected to the Snitch core through the accelerator interface. It receives the instructions offloaded from the core, decodes these instructions, and sends the results back to the core.

There is another sub-byte SIMD ISA extension applied to Mempool. This ISA extension is a part of the *XpulpNN* ISA extension [20], which was proposed to accelerate quantized neural network (QNN) inference on MCUs. It extended the packed SIMD operations in Xpulpimg to support lower bit-width (i.e. 4-bit). As Xpulpimg ISA supports 16-bit and 8-bit SIMD instructions, a SIMD unit was implemented inside the IPU in Mempool. And the sub-byte SIMD ISA was integrated into the IPU in another branch of the Mempool system.

Besides the hardware required for these ISA extensions, the infrastructure of Mempool system was also extended to support these new features. In particular, the pulp-gcc compiler [34] was extended to have complete support for the programmability of these custom ISA extensions.

2.5 Conclusion

In this chapter, we explained the necessary background knowledge for this project. First, we introduced the RISC-V open-source ISA along with its extensions. Next, SSRs are presented as a custom RISC-V extension to improve FU utilization, which we will extend to enhance the integer FU utilization in our work. Then, we introduced the hardware platform we will work on in this project, the Snitch cluster, and its detailed architecture was explained. Finally, another RISC-V system Mempool was introduced. Mempool was improved for ISP purposes with the ISA extensions, namely Xpulpimg and sub-byte SIMD. We are interested in porting them to the mainline Snitch and exploring their impact.

Chapter 3

Hardware Architecture

The hardware implementation of the integer extensions for Snitch is our major contribution in this project. Our main hardware target is the Snitch core complex (CC) in the mainline repository, which is the smallest unit of repetition in the mainline Snitch system. At CC's abstraction level, the Snitch core can be enhanced with additional hardware extensions to support different application domains. Our hardware implementation mainly consists of two parts:

- Xpulp ISA integration: includes Xpulpimg sub-byte SIMD ISA extension
- Integer SSRs support

As described in Section 2.4.2, Xpulpimg ISA is a partial implementation of the Xpulp ISA. It is a DSP-oriented ISA extension developed in the mainline Mempool system. An integer processing unit (IPU) was introduced as an accelerator for all the performance-oriented instructions from Xpulpimg. The sub-byte SIMD ISA is a partial implementation of the XpulpNN ISA to accelerate computation on 4-bit data with the SIMD approach.

In order to integrate these ISA extensions into the mainline Snitch, we followed the methodology proposed in [19] by extending the baseline Snitch core in terms of the decoding logic, the integer register file, and some control logic. In addition, we integrated the IPU into the Snitch CC as an accelerator. Figure 3.1 gives an overview of the modifications involved for the Xpulp ISA extensions.

The integer SSRs support is an extension of the existing floating-point SSR. We kept all the existing floating-point data features while adding integer data features in the SSR streamer, and the SSR streamer was shared between the integer and floating-point datapath. The general architectural changes we brought for the integer SSR extension are shown in Figure 3.2. We first extended the Snitch core to have a stream interface for data exchange using SSRs. Then a control and status register was added in the core for the activation of the SSR. Some necessary control logic changes were also implemented. Finally, we extended the modules inside the SSR streamer: a switch to direct streams from either the FPSS or the Snitch core to the targeted SSR lanes; SSR lane to generate address patterns and fetch data from memory. The detailed hardware architecture of the

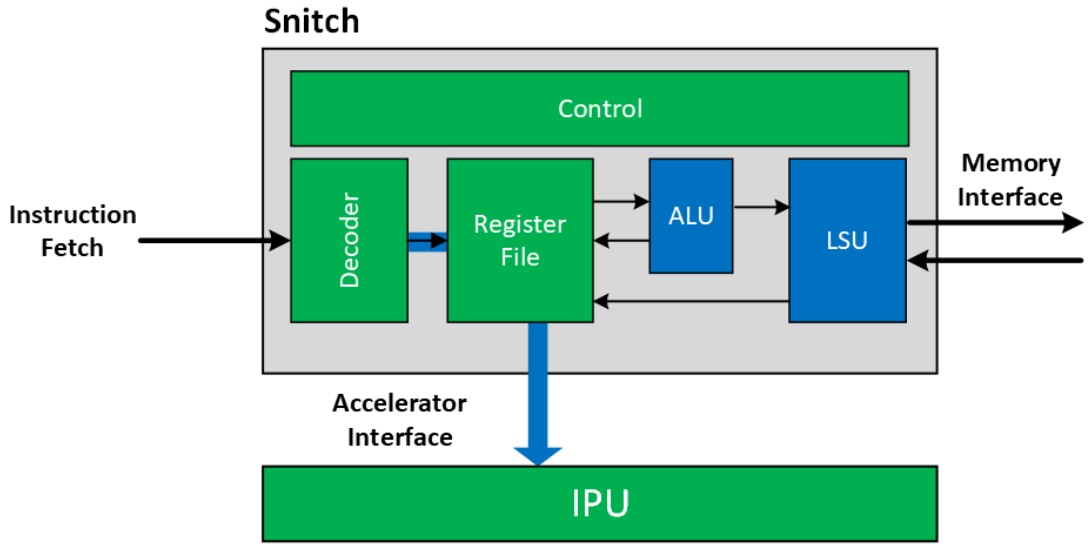

Figure 3.1: Block diagram of the modifications for Xpulp ISA extensions. We introduced architectural or logic changes in blocks shown in **green**. While the existing modules we did not touch are marked in **blue**.

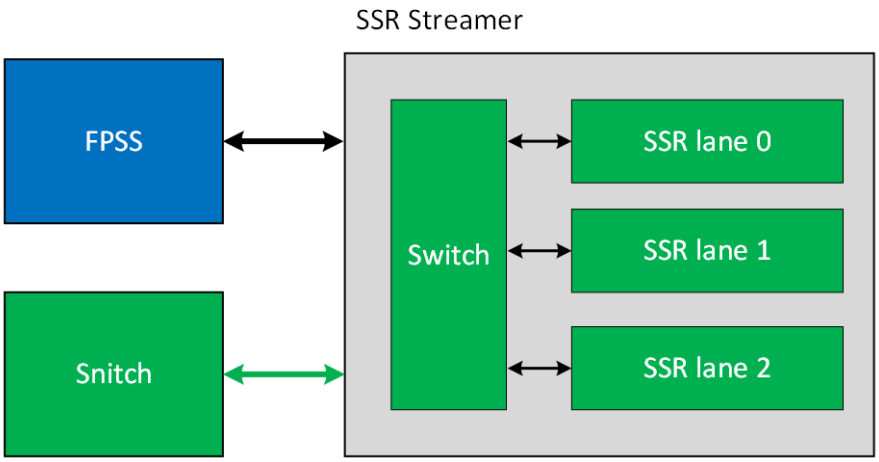

Figure 3.2: Block diagram of the modifications for the integer SSR extension. We introduced architectural or logic changes in blocks shown in **green**. While the existing modules we did not touch are marked in **blue**.

Xpulpimg and sub-byte SIMD ISA extensions and the architectural implementation for integer SSRs in the Snitch CC are introduced in the following sections.

3.1 Xpulpimg ISA integration

The first part of our hardware implementation is integrating the Xpulpimg ISA into the mainline Snitch. Xpulpimg ISA extension was developed in the mainline Mempool system to improve the integer workloads' performance in digital signal processing and exploit general-purpose parallelism in the ISP domain.

Mempool system is built based on the outdated version of Snitch core. Thus, the methodology to port Xpulpimg to the mainline Snitch follows a similar approach as described in [19]. We took the mainline IPU module as an IP and connected it to the accelerator interface of the Snitch core, and some modifications were added to the Snitch core to support this new ISA extension.

We ported the Xpulpimg into the Snitch core in a parameterized style according to [19]. This method keeps the extensibility of the Snitch core and does not introduce extra area overhead when the extension is not necessary for the specific application domain.

To support Xpulpimg ISA, we modified the following logic of the Snitch core:

- the decoding logic;
- the integer register file;
- the scoreboarding and write-back control logic;
- the logic connected to the load-store Unit(LSU).

First, the decoding logic was extended to support all the instructions introduced in Xpulpimg ISA. Also, the accelerator interface is prepared at this stage to exchange data via an AXI-like handshake between the core and the IPU: the IPU should be selected as the accelerator target to offload instructions through arbitration. Then the *valid* from the core is asserted if the instruction is valid and the core is not stalled by others. Finally, the operands and the instruction encoding are sent to their corresponding ports on the accelerator interface. In addition, we added the decoding of a new 5-bit immediate field for immediate branching instructions in Xpulpimg. The original branch mechanism in RISC-V compares the content of two registers [35]. However, immediate branching instructions compare the content of a register with this new 5-bit immediate [19]. This immediate field shares the same encoding space as `rs2`, and it is sign-extended before being sent to the ALU [19]. The output of the ALU is used to determine whether to take the branch.

An additional reading port was added to the integer register file, as many arithmetic instructions in Xpulpimg need to read from register `rd` as a third source operand, such as MAC operation, SIMD sum-dot-product operations, SIMD shuffles, and bit insert operations. The scoreboading logic was extended at this point to consider `rd` as a third source register. This modification can detect the potential read-after-write (RAW)

hazards of `rd`. In the floating-point datapath, the third operand port of the accelerator interface connects the address bus of the LSU inside the Snitch core and the LSU inside the FPU for load/store operations. However, since we need to send three operands from the integer register file to the IPU for Xpulpimg instructions, we muxed the port for the third operand of the accelerator interface: it selects `rd` when the instructions need the content of `rd` only, otherwise, it is wired to the LSU as the original. On the other hand, this extra read port is also necessary for the register-register store instructions. For example, the instruction `p.sb rs2, rd(rs1!)` uses `rd` as a third source operand to store the register offset and also a destination register. Also, since `rd` is used as the second source operand, while `rs2` is used as the data source for the store operation, we extended the scoreboarding to also detect the hazards of `rs2`.

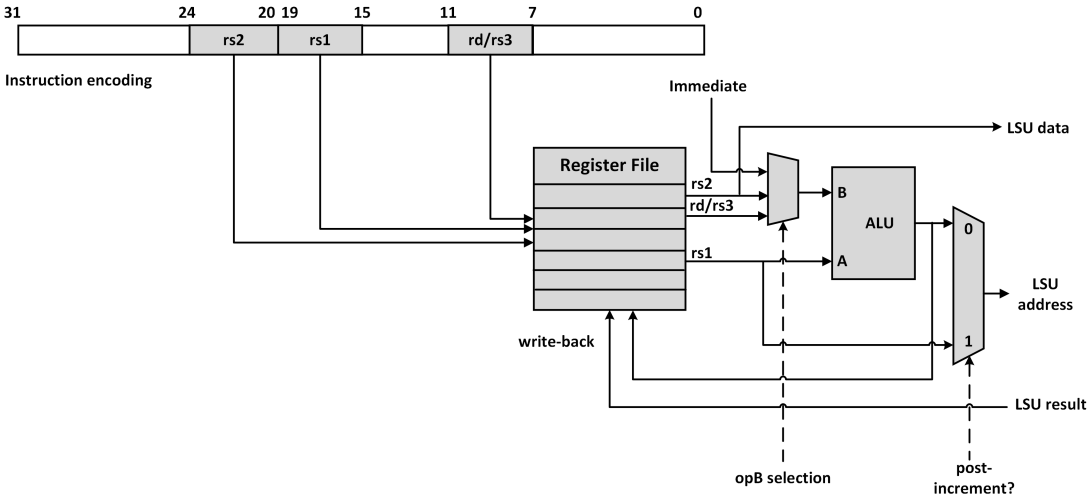

Figure 3.3: Block diagram of the modifications for extended load/store instructions, the third operand, and the post-increment mechanism [19]

Figure 3.3 shows the hardware modifications required for the extended load and store instructions. In post-incrementing load/store instructions, `rs1` is utilized for two purposes: it stores the base address as a source register; it stores the address after the post-incrementing update as a destination register. Therefore, we extended the scoreboarding logic and the write-back control logic to consider `rs1` as a destination register for post-incrementing instructions. This helps to detect the RAW hazards of `rs1`. We also muxed the register write-back port of `rd` with `rs1` to avoid introducing a large area overhead, since the retirement of a post-increment instruction on `rs1` and the retirement of a basic RV32I instruction are mutually exclusive [19].

We extended the logic that feeds the address to the LSU. Originally, the actual address is hardwired to the output of the ALU since the load/store instructions in RV32I use the ALU to compute the actual address for memory accesses. However, in the register-register addressing mode, the actual address is computed at the same time as the issuing, and the results are written back to `rs1`. Hence, we muxed the actual address sent to the

LSU between the ALU output and the register file output for `rs1`.

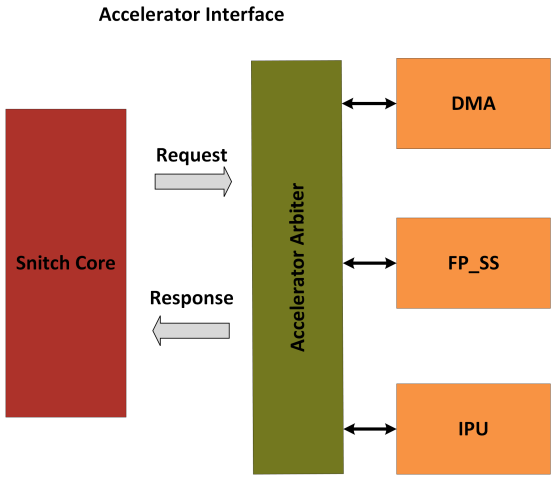

Figure 3.4: Block diagram of the Snitch CC accelerator offloading datapath extended with Xpulpimg

Apart from the logic required by the Snitch core, the IPU is integrated into the mainline Snitch by connecting it to the accelerator interface of the Snitch core. As the accelerator interface was extended with demux and arbiter to support FPU and DMA, we added new ports for demuxing and arbitration for the IPU. Then the IPU module can be instantiated and directly connected to these ports in Snitch CC as shown in Figure 3.4.

3.2 Sub-byte SIMD ISA integration

Xpulpimg ISA has the support of SIMD computing paradigm on 16- and 8-bit vector operands, while sub-byte SIMD ISA is part of the XpulpNN ISA extension, which extends the SIMD operations for 4-bit operands, namely *nibble*. It aims to boost the computation of low-bit-width data types. This ISA was originally also implemented in a diverging branch of Mempool system.

To port this ISA extension into the mainline Snitch, it requires us to extend the decoding logic of the Snitch core at the beginning, and then these SIMD instructions are executed by the SIMD unit inside the IPU as same as in Xpulpimg ISA. Figure 3.5 depicts the extended SIMD unit inside the IPU. Since we need to vectorize the operands on the granularity of nibbles as the smallest data size, but also bytes and half-words as larger sub-word data sizes, the data packing logic was first extended to have a new mechanism to adjust three input operands into vectors on the granularity of 4-bit, 8-bit, or 16-bit. Then an extra 4-bit domain was added to perform the computation through simple parameterized instantiation.

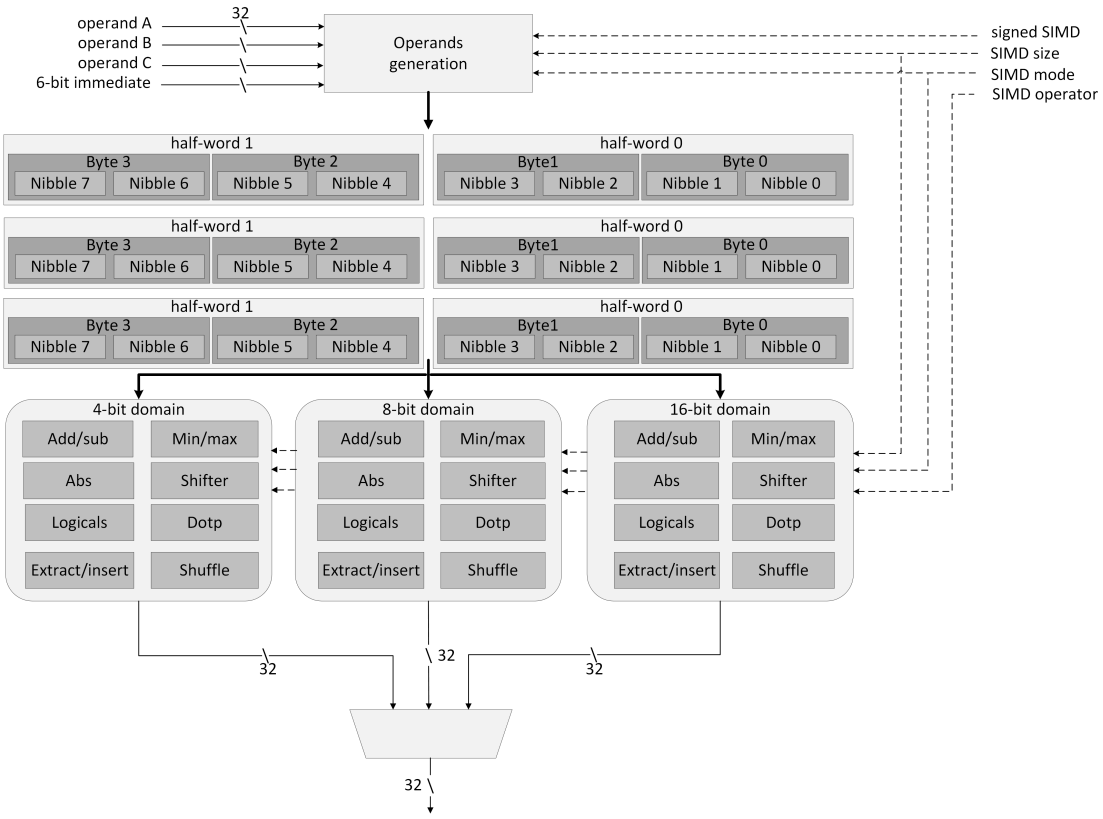

Figure 3.5: Block diagram of the extended SIMD unit in the IPU

3.3 Snitch integer SSR extension

The integer SSR extension requires the hardware modifications of both the Snitch core and the SSR streamer. The Snitch core was extended to be able to control the usage of the integer SSRs, and the arbitration of the SSR streamer between the integer and the floating-point datapath. Also the integer register file was extended to have an SSR interface that redirects the register reads and writes to the memory streams from the SSR streamer. The detailed hardware extension of the Snitch core is described in Section 3.3.1. The extended SSR streamer supports providing 64-bit double-precision floating-point data streams for the Snitch FPU and 32-bit integer data streams for the Snitch core. Its architecture is detailed in Section 3.3.2.

3.3.1 Snitch architecture extension

The Snitch core extension follows a similar implementation of SSR support in FPU with an invented arbitration mechanism, which involves the following necessary modifications:

- A control and status register (CSR) to enable or disable stream semantics;
- The indices of the registers assigned with stream semantics must be changed;
- The register file must be extended to intercept and re-reroute accesses to a subset of registers;
- The control logic of the core must be extended to support the additional stall conditions introduced by the stream interface;
- A newly invented arbitration mechanism is necessary to achieve the mutual exclusion of the SSR streamer between the integer and the floating-point datapath.

Control and status registers

According to the usage of SSRs in the floating-point subsystem of Snitch cluster, the first requirement of the integer SSR extension is to have a control and status register (CSR) to enable or disable stream semantics. We added the `CSR_INT_SSR` CSR with address `0x7C2` to the Snitch core. It is a standard read/write CSR in the machine CSRs address range. It contains a single bit (the LSB) to enable or disable integer stream semantics. And the subset of registers with stream semantics is fixed in hardware and can only be enabled or disabled all at once.

Register index mapped to SSR

The register calling convention in the floating-point subsystem assigns temporary registers `ft0`, `ft1` and `ft2` with stream semantics. When the SSR is enabled and the instruction uses this subset of registers, the register reads and writes are forwarded to the external

streams from the SSR streamer instead of accessing the registers. This subset is mapped to the physical floating-point register file with register indices from zero to two. However, the register `x0` (the physical register of index 0) in the Snitch core is hardwired to logic 0 according to the basic RISC-V calling convention [22]. Therefore, we shifted the indices of the integer registers that mapped with stream semantics to five, six and seven, which corresponds to `t0`, `t1` and `t2` respectively. In this way, we kept using temporary registers in the integer register file for stream semantics.

Register file

A vanilla integer register file supplies the operands for the subsequent pipeline by the read ports, and write ports are used to store back the result of an instruction [24]. To support SSR we need to intercept access to a certain set of registers, and redirect the read or write transaction on the external stream interface instead of accessing the physical register file [24]. Hence, additional hardware is required to wrap the register file as the interface for streams. Figure 3.3.1 and 3.3.1 depict the hardware change per read and write port of the integer register file.

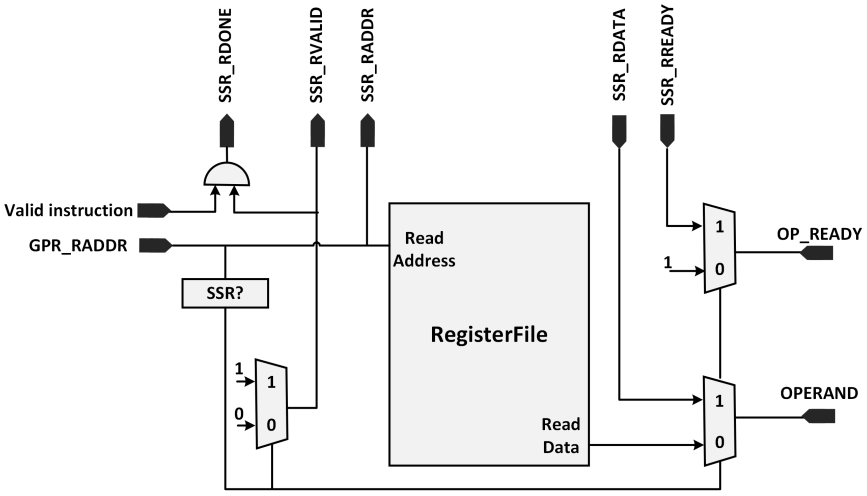

Figure 3.6: Additional logic required per register file read port for SSRs

The data exchange on the SSR interface is achieved via a two-phase handshake: the Snitch core asserts its read *ssr_rvalid* or write *ssr_wvalid* signal if an access has stream semantics ("SSR?"). This is determined by checking two conditions:

1. The register address (read address or write address) must be one of the registers with stream semantics(`t0`, `t1` and `t2` in our implementation);

2. The CSR for stream semantics must be enabled.

If both conditions are met, the access is routed to the SSR streamer via the stream interface. In the read mode, the operands can be directly fetched from memory by the

SSR streamer instead of accessing the register file. The *ssr_rready* signal is pulled to high as long as the data in the respective SSR lane's FIFO is available. Finally, if the core can consume this coming data, it asserts the *ssr_rdone* signal. In the write mode, the *ssr_wdone* signal is always asserted, and the writing data is redirected to the SSR streamer if the above-mentioned SSR checking conditions are met, and the SSR lanes are ready to consume the data by asserting *ssr_wready*. This interface allows the SSR streamer to assert back pressure to the Snitch core to stall the data exchange if a request cannot be handled at once.

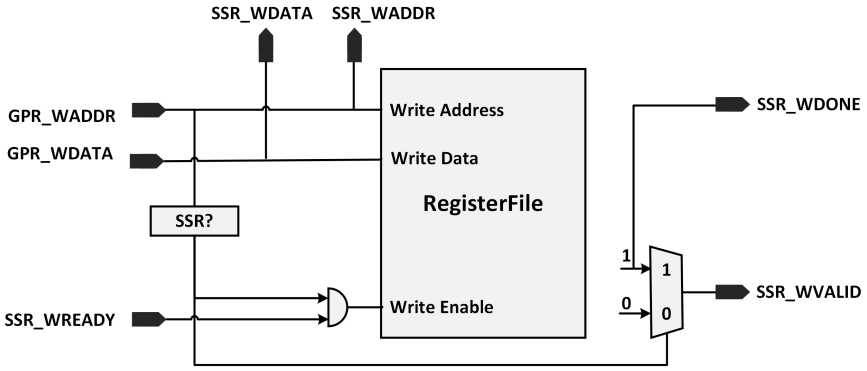

Figure 3.7: Additional logic required per register file write port for SSRs

Control logic

We extended the scoreboarding logic and the write-back control logic for the SSR support. The dependencies of the destination register should not be tracked when the SSR is enabled since there is no physical register file access. Also, the operands should be ready if it is used by SSR and they do not need to be marked in the scoreboard in this case. The write-back results are rerouted to the SSR streamer at the retirement phase of the instruction. It needs to check whether the destination register is mapped to stream semantics and if the respective SSR lane is ready to take the data.

Arbitration mechanism for SSR sharing

We would like to mux the SSR streamer for both the integer datapath and the floating-point datapath considering the area-performance trade-off. Meanwhile, this design choice is also based on the fact that one stream can only be used by one master as the integer and floating-point datapath are mutually exclusive. Thus the SSR streamer cannot be active for both datapaths at the same time. We proposed an arbitration mechanism for SSR sharing: it uses a 2-bit register inside the Snitch core to track the usage status of the SSR streamer.

Since the execution of the FPU is decoupled from the Snitch core, floating-point instructions could be committed at an arbitrary time later than its issuing from the Snitch

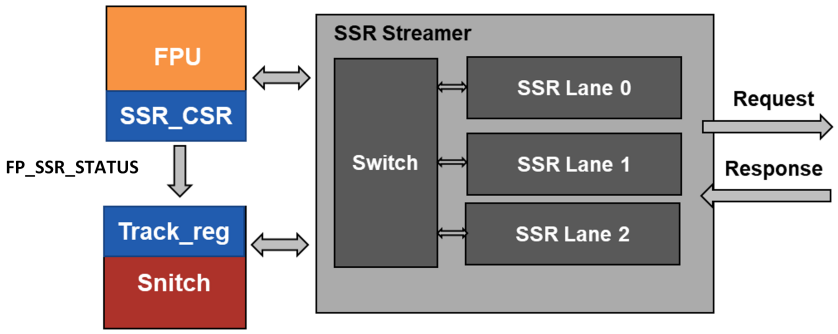

Figure 3.8: SSR streamer sharing and arbitration mechanism support

core. Also, the CSR for enabling or disabling the floating-point SSRs locates in the FPU instead of the Snitch core. Table 3.1 presents a pseudo-code example using both the floating-point and integer SSRs, the left code is the program order, and the right is the potential execution order. The program starts with enabling the floating-point SSRs and then does the required computation. Afterward, it disables the floating-point SSRs, and immediately enables the integer SSRs for the next integer kernels. In this case, if the FPU is busy when the disabling is issued, the disabling could be delayed with arbitrary cycles in the FPU, while the integer SSRs has already been enabled by the Snitch core, which breaks up the entire system.

Program order		**Potential execution order**	
set	fp_ssr_en	set	fp_ssr_en
...		...	
clear	fp_ssr_en	set	int_ssr_en
set	int_ssr_en	clear	fp_ssr_en

Table 3.1: Pseudo code example using floating-point and integer SSRs

This issue can be solved by wiring a feedback signal from the FPU to the Snitch core to inform the core of the usage status of the SSR streamer in the floating-point datapath, as depicted in Figure 3.3.1. Finally, we can achieve a 4-state finite state machine (FSM) with the 2-bit register and this feedback signal as presented in Figure 3.3.1. The default state is `IDLE` as no one is using the SSR streamer. It can switch between `IDLE` and `INT_ACTIVE` when the CSR for integer SSRs is enabled or disabled, as the responsible CSR was added in the Snitch core. As for the floating-point datapath, this FSM jumps from `IDLE` to `FP_ISSUE` when the CSR for floating-point SSRs is enabled. And it only jumps to `FP_ACTIVE` when the Snitch core receives the SSR status signal from the FPU and the signal is pulled to high, which means the SSR streamer is in use by the floating-point datapath. Finally, if the floating-point SSRs are disabled and the Snitch core receives logic zero brought by the SSR status signal, the machine returns back to `IDLE`. This mechanism guarantees the robustness of the system in case of the SSR extension is used

by both the floating-point and the integer datapath. It ensures that the SSR streamer is only used by one master at a time.

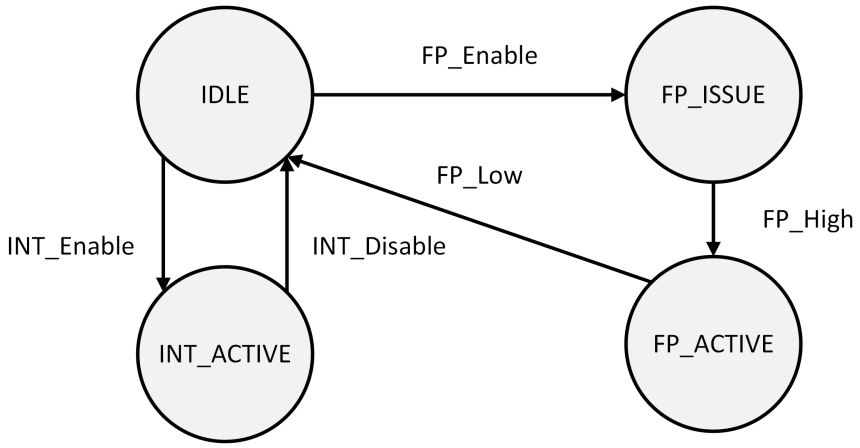

Figure 3.9: Finite state machine (FSM) for mutual exclusion of SSRs

3.3.2 SSR streamer

The SSR streamer is the hardware extension to support SSRs. It wraps around the FPU's register file originally, and it is configurable by the Snitch core to configure memory streams with an affine address pattern according to the available loops in the program. We extended the SSR streamer to support integer streams by connecting it to the Snitch core, and its resource sharing between the floating-point and the integer datapath. The extensions of the SSR streamer include two major modules:

- SSR switch: it can map the streams of register accesses from the master to the SSR lanes based on our requirements. We extended it to support both integer and floating-point memory stream mapping.
- SSR lane: it can generate addresses for memory accesses, fetch data from memory and store it in the FIFO in the read mode. We extended it to support feeding 32-bit integers to the Snitch core.

SSR switch

The read and write ports of the register file is exposed as separate streams at the boundary of the core. The SSR switch uses the register address to map each access on these streams to the targeted SSR lane [24]. The original implementation contains three such lanes for floating-point datapath, one for the `ft0`, one for the `ft1` and one for the `ft2` registers. We extended the switch to support the following additional functionalities:

- it can map the integer register access on the streams to the SSR lanes;

- it can mux the streams from the SSR lane to the corresponding master that sends the request.

First, we added an extra SSR interface to connect the SSR streamer to the Snitch core. Then we mapped the streams from the integer register file to the SSR lanes. Here these lanes were mapped to `t0`, `t1` and `t2` registers in the Snitch core. The stream multiplexing is achieved by a 1-bit control signal from the Snitch core: it will select the corresponding datapath according to the current state of the FSM we introduced in Figure 3.3.1 for the correct arbitration. For example, the integer datapath is selected at state `INT_ACTIVE` as the control signal is 0, then the streams from the integer register file are mapped to the SSR lanes. On the other hand, the floating-point datapath is selected at state `FP_ACTIVE` as the control signal is asserted, and the streams from the floating-point register file are mapped to the SSR lanes in this case.

SSR lane

The SSR lane is the central module inside the SSR streamer. It is employed to generate addresses and send requests to the TCDM, and feed the response data to the master (e.g. FPU). Originally, the address generators inside each lane were configured to generate 64-bit aligned addresses as the last three bits of the address were masked to 0. And each time 64-bit double-word data was fetched from the TCDM for floating-point datapath. We proposed an address filter approach using FIFO to store metadata and split 32-bit words out of each 64-bit double word. The extended SSR lane's architecture is shown in Figure 3.10.

In this approach, we first configured the address generator to generate 32-bit aligned addresses by masking only the last two bits of the address. This is useful to track the address sequence as we can get the address information of each 32-bit data when the address was emitted. We added an intermediary unit (address filter) between the address generator and the remaining modules inside the SSR that communicates with three different components based on valid-ready handshakes:

- It consumes the address from the address generator when its downstream modules (metadata FIFO and memory request hardware) are ready;
- It produces valid addresses and the memory request signal to the existing request mechanism, which happens when a new double word address is detected and there is valid data at the address generator interface;
- It produces the metadata and sends it to the downstream FIFO when there is a valid address from the address generator.

The handshake mechanism can stall and continue the exchange of data whenever the producer or the consumer dessert their respective control signals in any cycle, which frees the designer from using fine-grained data flow control in terms of the hardware.

Then we introduced the concept of metadata, which consists of 3 bits as shown in Figure 3.11:

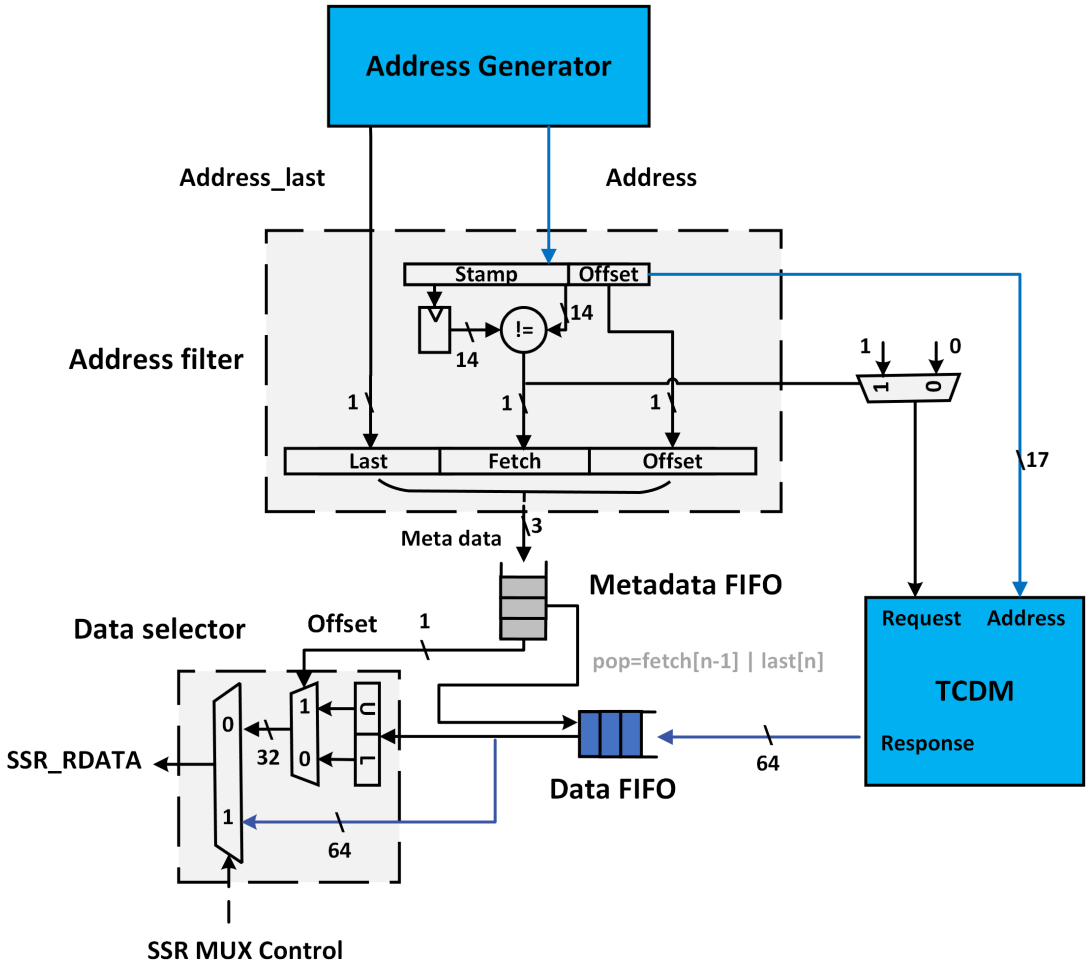

Figure 3.10: Block diagram of the extended SSR lane. The existing modules and datapath are marked in blue, while the added modules and datapath are marked in grey and black respectively.

- **Offset**: it is the 3rd LSB of the address. It is used to split two 32-bit data out of one 64-bit data. For example, a 64-bit double word is stored at address 0x00000, and it contains two 32-bit integers stored at address 0x00000 and 0x00004, respectively. Then we can distinguish them as their 3rd LSBs are different. The lower 32-bit uses 0 while the upper 32-bit uses 1.
- **Fetch**: it is obtained by comparing the upper 14 bits of the two consecutively generated addresses. The address emitted earlier is registered for one cycle and used to compare with the later one. If the upper bits are different, it means two double words are needed by these two addresses, and the memory request will be sent to the TCDM with the later address. On the contrary, if the upper bits are the same, it means these two addresses are pointing to a single double-word data, then only one memory access will happen and the second memory access will be filtered to avoid repeating data fetching and expensive latency.
- **Last**: it is a signal coming from the address generator indicating the end of the address stream. It is necessary to ensure the last data stored in the data FIFO can also be read out.

Figure 3.11: The organization of metadata

We used a lookahead FIFO to store the metadata, and the metadata is used to control the data FIFO that stores the data we fetch from memory and split the 32-bit word at the output stage. Each time the address filter module generates 3-bit metadata when it receives a valid address from the address generator, thus each metadata contains all the required information of its corresponding address, which is useful for downstream processing. `Fetch` and `Last` are used for reading control of the data FIFO. The head element of the data FIFO is popped if `Fetch` of the second element in the metadata FIFO is 1, or `Last` of the first element in the metadata FIFO is 1. `Fetch` = 1 means the consecutive addresses use different 64-bit double word data in TCDM instead of sharing the same double word and vice versa. In this case, we need to pop the head element from the data FIFO, otherwise the subsequent address can reuse the same double-word data and we need to make sure this data is not popped. The lookahead mechanism ensures that two elements can be read in one cycle: the element at the head of the FIFO and the element behind it. In this way, we can check two consecutive addresses in a single cycle, which is consistent with our requirements. While `Last` = 1 means, it is the last required data of the stream. Therefore it also needs to be popped and sent to the master. This lookahead FIFO is achieved by adding a register at the head of the FIFO. The register and the FIFO are both implemented with valid-ready handshaking interfaces so that it is easy to fit this module in our SSR lane. The size of this metadata

FIFO is eight, which is able to store all the addresses in the flight safely: the size of the data FIFO is four and the latency of TCDM is two cycles.

Finally, the data selection is achieved inside the SSR switch as we need to select 32-bit words for integer datapath but bypass this data splitting for floating-point datapath. This is controlled by the SSR datapath selection signal from the Snitch core: when this signal is 0, the integer datapath is selected, and the 64-bit double words are split into 32-bit words based on the `Offset`. The upper 32-bit word is selected if the `Offset` is 1, otherwise the lower 32-bit word is selected. On the other hand, when the SSR mux signal is 1, the floating-point datapath is selected, and the 64-bit data can be directly fed to the FPU instead of passing through the data-splitting logic. In addition, the data selection module is implemented with valid-ready handshaking interfaces as the consumer of the `Offset` metadata and the 64-bit double words from the data FIFO, and the producer of the data to the FPU/Snitch core. It only generates a valid output when it finishes handshakes with both upstream inputs successfully. This mechanism also makes sure it can select the correct 32-bit words required for the integer datapath.

3.4 Verification

The verification during our hardware implementation is based on the existing test bench for the Snitch cluster. We can write bare-metal C programs and use pulp-riscv-gnu toolchain [34] to compile them to binaries. The test bench then takes the binaries as the input stimuli for the RTL simulation on the Snitch cluster.

We verified our ISA extensions by integrating the existing `riscv-tests` [36] verification framework into the mainline Snitch. The methodology we adopted in our design iteration is shown in Figure 3.12. `riscv-tests` environment provides two phases of verification:

- Behavioral simulations of the instructions with Spike ISA simulators [37]
- RTL simulations by generating test binaries

We used the custom pulp-gcc toolchain extended with Xpulpimg and sub-byte SIMD instruction set to compile the tests, and then we ran the RTL simulation on a single core of the Snitch cluster. We also automated the testing process for efficiency and also ensuring every new instruction introduced by the ISA extension was tested.

For our SSR extension, we verified our implementation step by step corresponding to our progress. First, we verified the integer reading and writing using SSRs with a simple array-addition kernel after we added the SSR interface to the integer register file of Snitch, and the SSR switch was extended to have two interfaces as a slave module. Next, we verified our SSR arbitration mechanism with some edge cases that could break up the system. It also guarantees the robustness of the mutual exclusion strategy we proposed. Finally, we verified all the extensions in the streamer and the Snitch core in the system with a software test suite. This test suite targets both floating-point SSRs and integer SSRs, to ensure that the floating-point datapath is not broken by our extension and functionality of our extension is correct. It mainly contains two parts:

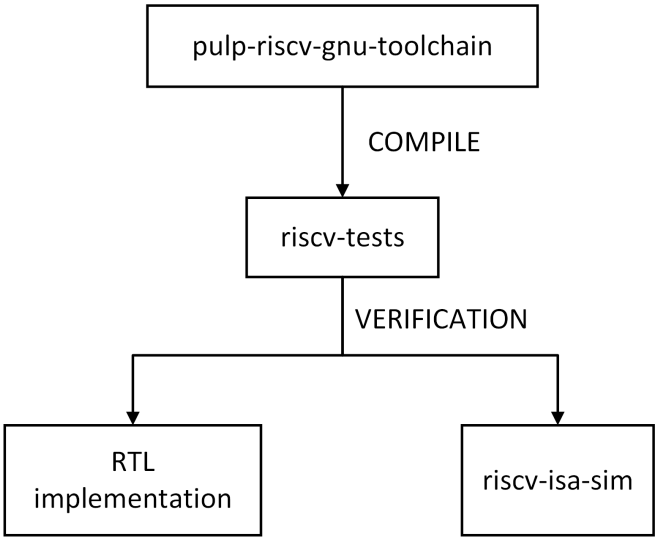

Figure 3.12: ISA extension verification methodology

- Functional tests: this aims to verify the functionalities of our design. We tested the system with element-wise operations of two arrays, such as addition and MAC operation. The types of arrays include both the 64-bit double-precision floating-point and the 32-bit integer;
- Stress tests: this aims to verify the system with some edge cases. We tested the system with element-wise operations of two arrays with large array sizes, i.e., hundreds of elements per array. This makes sure both FP SSRs and integer SSRs can work correctly when reading a large amount of data is necessary.

3.5 Conclusion

In this chapter, we defined a unified Snitch system with integer extensions for integer workloads acceleration. The hardware implementation of these integer extensions in Snitch was described, which consists of the following three parts:

- Xpulpimg ISA integration;
- Sub-byte SIMD ISA integration;
- Integer support for SSRs and the datapath sharing of SSR streamer.

For the Xpulpimg and sub-byte SIMD ISA integration, we modified the logic inside the Snitch core in a parameterized style to support these new instructions without harming the extensibility and modularity of the Snitch core. Also, we integrated the Snitch IPU as an integer accelerator into the Snitch CC through the generic accelerator interface of

the Snitch core. For the integer SSR extension, we extended the Snitch core to be able to redirect the register read or write transaction to the external streams and arbitrate the usage of the SSR streamer for floating-point and integer datapath. Also, the SSR streamer is extended to have 32-bit integer support maintaining the original floating-point abilities. We followed a top-down approach to introduce our hardware extensions from high level to details. Finally, we introduced our verification methodology during our implementation.

Chapter 4

Results

We integrated the Xpulpimg and sub-byte SIMD ISA extensions into the mainline Snitch, and we added integer SSR support as described in Chapter 3. We aimed to evaluate the impact of these different integer extensions for Snitch in terms of performance, area, and timing. The main object of our evaluation is the Snitch core complex (CC). This level of abstraction only contains the vanilla Snitch core and different hardware extensions applied to the core, such as FPU, SSR streamer, etc.

In this chapter, we present the performance, area, and timing results of our integer extensions in comparison with the baseline RV32IM Snitch CC. First, we describe our evaluation setup in terms of benchmarking and synthesis in Section 4.1. Then we analyze the performance of the Snitch CC with different extensions according to the benchmark results in Section 4.2. Finally, Section 4.3 presents the synthesis results showing the cost in terms of the area and the timing.

4.1 Evaluation setup

4.1.1 Benchmarking methodology

We used cycle-accurate RTL simulations in QuestaSim 10.7b to measure the performance of the Snitch CC. The traces for the Snitch core were recorded during the simulations and parsed to a readable format by a Python script. The program execution sequence can be found in the results of the traces, and the execution cycles of the benchmark kernels are tracked by the CSR counter in the Snitch core. The evaluation flow is shown in Figure 4.1.

We benchmarked the Snitch CC with the matrix multiplication (*matmul*) algorithm. It is a typical integer workload widely used in DSP and machine learning applications. We also developed a set of kernels in C language based on different integer extensions for the evaluation:

- The vanilla Snitch CC with RV32IM ISA;

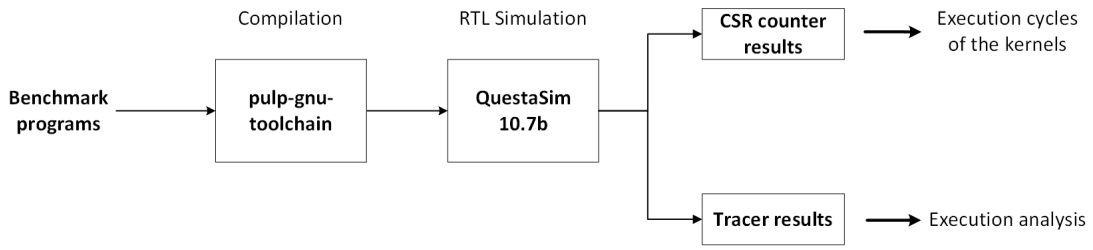

Figure 4.1: Performance evaluation flow

- The Snitch CC with Xpulp ISA (Xpulpimg and sub-byte SIMD) extension;
- The Snitch CC with integer SSR extension;
- The Snitch CC with Xpulp ISA and integer SSR extensions.

We compiled these kernels with the pulp-gcc-toolchain extended with Xpulpimg and sub-byte SIMD instruction set. To exploit the highest performance of these extensions, we optimized these kernels by employing available compiler intrinsics and manual inline assembly. Loop unrolling is also applied to these kernels. In addition, these *matmul* kernels have been simulated with different data widths such as 32-bit, 8-bit and 4-bit, aiming to explore the performance impact on the typical lower data widths.

We utilized the 8-core Snitch cluster configuration for simulation, but ran the programs on a single core for the evaluation of CC. In this cluster, a 128 KiB TCDM with 32 memory banks is shared between eight cores. Since we also benchmarked with different dimensions of the matrices, we ensured that the input and output matrices could fit in the TCDM for the simulation reasons. This not only prevents us from long stalls due to potential lower-level memory accesses, allowing us to focus on the performance of our extensions but also ensures the correct data can be fetched by the SSR streamer for the integer SSR extension.

4.1.2 Synthesis methodology

We synthesized the Snitch CC with GlobalFoundries 22FDX FD-SOI technology in typical conditions (TT, 0.80V, 25°C) using Synopsys Design Compiler 2022.03. The aims of the synthesis consist of two parts:

- We would like to understand the area and timing impact of these integer extensions on the Snitch CC, targeting the default operating frequency of 1 GHz;
- We would like to explore the highest effective frequency of the Snitch CC with different extensions. We changed the clock constraint varying from 2.8 ns to the potentially lowest effective clock period. And we drew the Area-Timing (AT) plots with a set of Pareto points collected from the synthesis results. In our AT plots, the x-axis represents the clock period, the y-axis stands for the area in terms of GE.

4.2 Benchmarking Results

4.2.1 32-bit matrix multiplication

We first evaluated our extensions with 32-bit matrix multiplication kernels. We varies the dimension of the input matrices to have four sets: 16×16, 32×32, 64×32, 64×64. Note that we did not enlarge the dimension further than 64×64 since we need to ensure the input and output matrices can be safely stored in TCDM as explained in Section 4.1.1. We compared the results of four different versions of *matmul* kernels based on different extensions within each matrix dimension. Each version was optimized by loop unrolling in the following way:

- The baseline RV32IM *matmul* kernel: we unrolled the outer loop to compute a small 2 × 2 square of the output matrix per iteration. The inner loop is also unrolled, performing 2 MAC operations for each element inside the 2×2 square per iteration. In summary, each hot-loop loads four elements from each input matrix, and then eight MACs are performed.

- The Xpulp *matmul* kernel: it follows the same structure as the baseline kernel. However, it is compiled for Xpulpimg architecture which uses the post-increment load and store operations as well as the fused-MAC operation.

- The SSRs *matmul* kernel: All the load operations are encoded into the register read operations for the SSR kernel, and we only need to do a one-time setup for the address generator according to the loops. As the existing multiplier needs four cycles to finish each computation, if we do the addition using the destination register of the multiplication as the source subsequently, a false dependency will occur, which hampers achieving the highest throughput. Hence, we unrolled the inner loop by a factor of 8 at the assembly level to hide the latency of the multiplier and solve the false dependencies issue. The loop unrolling of the SSR kernel is based on the repeating functionality of the SSR extension. By configuring the repeat times N, each data can be reused by N time. In this case, each iteration of the hot-loop computes one element from matrix A with eight elements from matrix B (i.e., 8 MACs are performed), and additions are performed after all the multiplications are finished instead of the normal serialized execution of multiplication and addition. This unrolling factor is bounded by the number of available registers, as we already need 18 registers for the unrolling by eight (16 destination registers for multiplication and accumulation, two registers for stream semantics as input).

- The Xpulp+SSRs *matmul* kernel: the Xpulp+SSRs kernel is unrolled by 16 with the similar approach as the SSRs kernel. However, we use the fused-mac operation introduced by the Xpulpimg ISA instead of the decoupled multiplication and accumulation in RV32IM. Hence, we can unroll the inner loop further from 8 to 16 to achieve higher throughput. Also, this loop unrolling mechanism already

eliminates the false dependencies as we compute 16 elements back-to-back with 16 different accumulation registers.

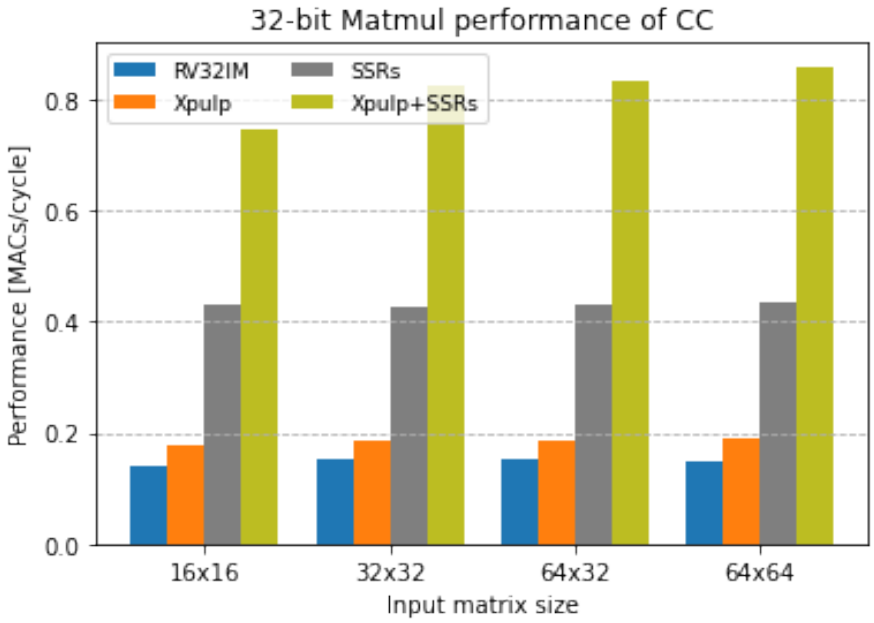

Figure 4.2: Performance of the Snitch CC with different integer extensions on 32-bit matrix multiplication kernels

Table 4.1 depicts the assembly code of the hot loop with different extensions. We can observe that Xpulp kernel eliminates the index incrementation with post-increment load instructions `p.lw`, and combines the multiplication and addition into a single fused instruction `p.mac`. SSRs kernel is unrolled by eight and optimized to solve the RAW dependency manually, but it still needs two individual instructions per MAC operation. Note that the load operations have been encoded into reading registers `t0` and `t1`. Xpulp+SSRs kernel leverages the benefits of both the SSRs and Xpulp ISA to achieve high FU utilization.

We proposed a metric to evaluate the performance: *hardware efficiency*. It can be measured in terms of MACs per cycle. Figure 4.2 shows the benchmarking results of the Snitch CC on 32-bit matrix multiplication kernels. Compared with the baseline, Xpulp ISA extension leads to the maximum speedup of **1.3**× in terms of MACs/cycle; SSR extension achieves **3.5**× speedup, while Xpulp+SSR extensions reaches **5.9**× speedup.

Ideally, the Xpulp+SSR extensions can achieve 1 MAC/cycle at maximum. However, we only reach 0.86 MACs/cycle when the input matrix size is 64×64. We analyzed the results of the traces to understand this performance gap. First, when the matrix size is small (e.g., 16× 16), we observed that the performance is slightly lower than the larger matrices. And we found that the setup overhead for the SSR is significant, which takes around 16% of the execution time of the kernel. Secondly, as we unrolled the inner loop

Baseline		Xpulp		SSRs		Xpulp+SSRs	
lw	s10, 0(a5)	p.lw	t6,8(t2!)	mul	t5,t0,t1	p.mac	s5,t0,t1
lw	a1, 0(t1)	p.lw	t4,8(s0!)	mul	t6,t0,t1	p.mac	a5,t0,t1
lw	s8, 0(t3)	p.lw	t5,512(a1!)	mul	s0,t0,t1	p.mac	a4,t0,t1
lw	s9, 4(t3)	p.lw	t3,512(s4!)	mul	s4,t0,t1	p.mac	a3,t0,t1
lw	a2, 4(a5)	p.lw	t1,8(s1!)	mul	s5,t0,t1	p.mac	a2,t0,t1
mul	s11, s10, s8	p.mac	a5,t6,t5	mul	s6,t0,t1	p.mac	a1,t0,t1
lw	a4, 4(t1)	p.lw	a6,8(s2!)	mul	s7,t0,t1	p.mac	a0,t0,t1
lw	a3, 0(t4)	p.lw	a7,512(s5!)	mul	s8,t0,t1	p.mac	a6,t0,t1
lw	s6, 4(t4)	p.lw	a0,512(s3!)	add	a5,t5,a5	p.mac	s2,t0,t1
addi	a5, a5, 8	p.mac	a4,t6,t3	add	a4,t6,a4	p.mac	s1,t0,t1
addi	t1, t1, 8	p.mac	a3,t4,t5	add	a3,s0,a3	p.mac	s0,t0,t1
addi	t3, t3, 512	p.mac	a2,t4,t3	add	a2,s4,a2	p.mac	t6,t0,t1
addi	t4, t4, 512	p.mac	a5,t1,a7	add	a1,s5,a1	p.mac	t5,t0,t1
mul	s8, a1, s8	p.mac	a4,t1,a0	add	a0,s6,a0	p.mac	t4,t0,t1
add	a0, s11, a0	p.mac	a3,a6,a7	add	a6,s7,a6	p.mac	t3,t0,t1
mul	s10, s10, s9	p.mac	a2,a6,a0	add	a7,s8,a7	p.mac	a7,t0,t1
add	a7, s8, a7	bne	t0,a1, pc-64	addi	t3,t3,-1	addi	s4,s4,-1
mul	a1, a1, s9			bnez	t3, pc - 68	bnez	s4, pc-68
add	a6, s10, a6						
mul	s8, a2, a3						
add	a1, a1, s7						
mul	a3, a4, a3						
add	a0, s8, a0						
mul	a2, a2, s6						
add	a7, a3, a7						
mul	a4, a4, s6						
add	a6, a2, a6						
add	s7, a4, a1						
bne	t5, a5, pc-112						

Table 4.1: Assembly code improvements with different extensions of the 32-bit matrix multiplication hot-loop

by 16, the hot loop consequently consists of 16 MAC operations for computation, one instruction for index incrementation, and one instruction for branching. Hence, there is an 11% loss caused by bookkeeping. Additionally, we use some instructions to store the results of the output matrix and reset the accumulating registers.

4.2.2 8-bit matrix multiplication

We used matrix multiplication algorithms for the 8-bit data type following a similar approach as its 32-bit counterpart. There are still four sets of input matrices dimensions: 16×16, 32×32, 64×64, 128×128. Note that the largest dimension is increased to 128 since the data size of 8-bit shrinks by four times compared with 32-bit data. We developed three versions of kernels, namely baseline RV32IM, Xpulp and Xpulp+SSRs. The baseline and Xpulp+SSRs versions follow the similar optimization as their 32-bit implementation explained in Section 4.2.1, except that the Xpulp+SSRs kernel exploits the 8-bit SIMD sum-dot-product instruction introduced by the Xpulpimg ISA, which also requires the matrix B to be stored in the transposed format. As for the Xpulp kernel, the following optimizations were applied:

- the compiler intrinsics were used for 8-bit SIMD operations. Some *shuffle* operations are also used to transpose the chunk of the input matrix B for the correct data packing of the SIMD registers for the dot products.
- we also added loop unrolling to compute a 2×4 square of the output matrix per iteration. In each hot loop, 4 MACS are performed for each output element.

The results of the kernel benchmarked on the Snitch CC are presented in Figure 4.3. Xpulp extension achieves a maximum speedup of **5.3**×, while Xpulp+SSR extensions are observed to be **22.6**× faster with respect to the RV32IM baseline. Note that with 64×64 input matrices, the Xpulp+SSR kernel even has a better performance compared with the larger matrices. Since we unrolled the inner loop by 16, the compiler also did extra optimizations so that the most inner loop was fully unrolled and book-keeping instructions for the hot loop were eliminated.

4.2.3 4-bit matrix multiplication

The 4-bit matrix multiplication also shares a similar setup as 8-bit, with the same four sets of input matrices dimensions and three versions of kernels. However, since the smallest granularity of the processor is a byte (8-bit), we have to at least pack two 4-bit data into one 8-bit data for storage purposes. Then in the baseline RV32IM kernels, it needs to first unpack a byte into two 4-bit sub-bytes, then sign-extended them both to 8-bit. Finally the computations with full bit-width (i.e., 32-bit level) are performed by the ALU and the multiplier. In contrast, the Xpulp extension supports 4-bit SIMD operations, which handles this small data type more efficiently. The 32-bit word is loaded to the SIMD registers and 8 MACs are performed within a single instruction. The performance results of 4-bit matrix multiplication are shown in Figure 4.4.

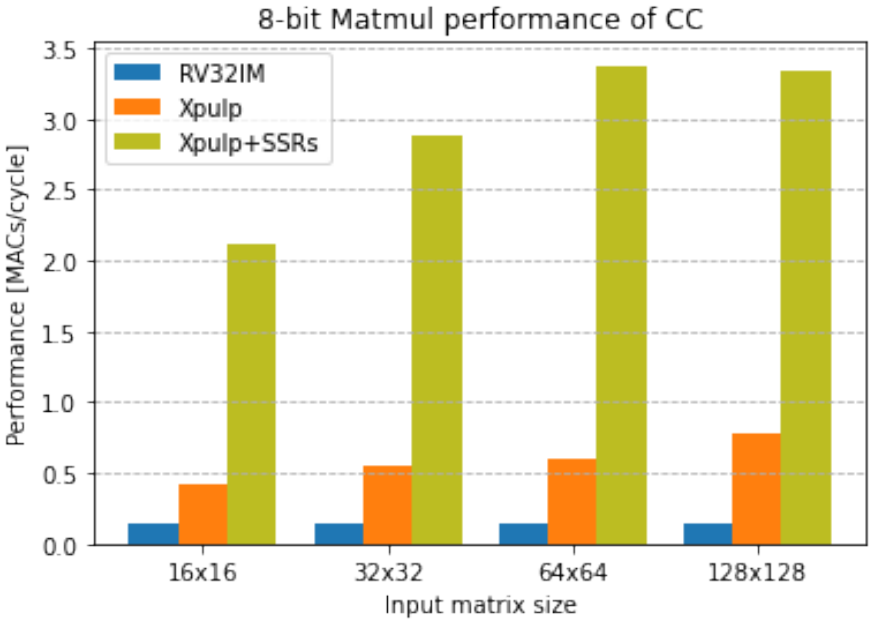

Figure 4.3: Performance of the Snitch CC with different integer extensions on 8-bit matrix multiplication kernels

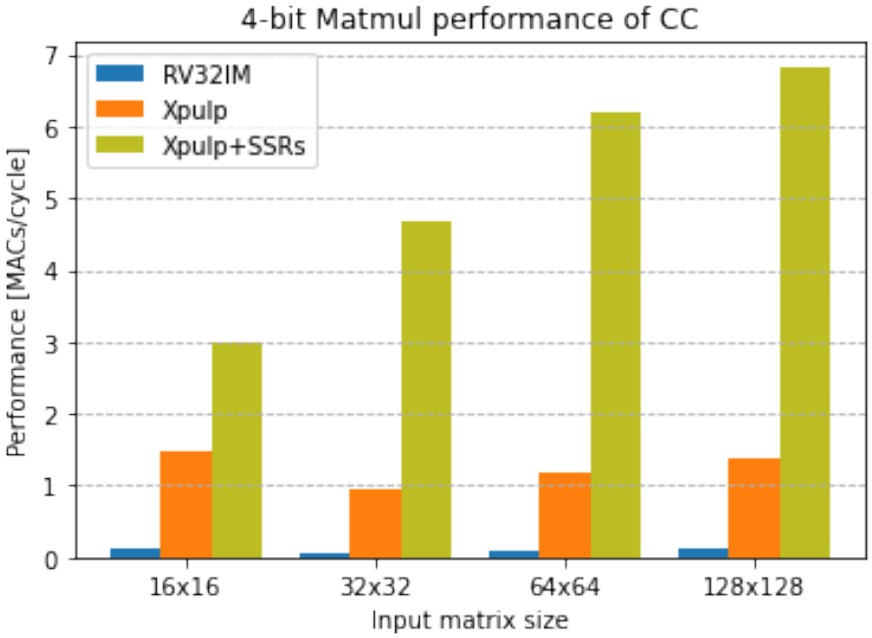

Figure 4.4: Performance of the Snitch CC with different integer extensions on 4-bit matrix multiplication kernels

The baseline has a poor performance on 4-bit data type as the Snitch core has the disadvantage of processing it under the support of 32-bit datapath modules. The speedup of **15.3**× is observed from the Xpulp kernel with respect to the baseline. Note that the performance of the Xpulp kernel reaches its peak with the 16×16 input matrices, thanks to the fewer load operations required for SIMD registers. The Xpulp+SSRs kernel exhibits significant speedup, which also scales linearly as the matrix dimension increases. For input matrices with 128×128 elements, It achieves a maximum speedup of **77.4**×.

4.3 Synthesis Results

The area cost of our extensions under certain timing conditions is evaluated with logic synthesis on the Snitch CC. First, we targeted a 1 GHz clock with appropriate IO delays. Figure 4.5 and 4.6 shows the area of the Snitch CC with different extensions. Here we use the gate equivalent (GE) as a measure of circuit area. From these figures, we observe that the Xpulp ISA extension, including the Xpulpimg and sub-byte SIMD ISA introduced an area overhead of 32.7 kGE. In particular, the IPU dominates with an area of 27.6 kGE. The critical path passes through the accelerator offloading datapath instead of the register file in the baseline. The SSR extension brought an area overhead of 30.8 kGE, which consists of the logic extension we implemented inside the Snitch core and the extended SSR streamer. The original SSR streamer has already consumed an area of 25.8 kGE. Thus our extension to support integer SSR only introduced an extra overhead of approximately 5 kGE. With the integer SSR extension, the critical path of the Snitch CC passes through the Snitch register file and the SSR streamer. When the floating-point extension is enabled, the FPU coprocessor is presented in the Snitch CC. This hardware needs around 255 kGE, and the retiming optimization is applied to improve the operating frequency of the Snitch CC. Besides, we found that our extensions on the integer datapath did not exhibit a measurable impact on the critical path of the floating-point datapath. Compared with the FP-capable CC, our Xpulp ISA and integer SSRs extensions increased the area by only 12%. For the Xpulp-capable CC, our integer SSRs extension leads to an area overhead of 42%.

We also explored the highest effective frequency of the Snitch CC when different extensions are applied with the same setup configuration for the synthesis. At this step, we set the clock period constraint to sweep from 2.8 ns to the potentially lowest values. The highest effective frequency can be calculated by the sum of the clock period and the worst negative slack when the timing constraint is not met and the slack is a negative value. Figure 4.7 and 4.8 plots the Snitch CC area against effective longest path for different combinations of extensions. The results of the highest effective frequency and the critical path are summarized in Table 4.2. Note that the baseline vanilla Snitch of RV32IMA instruction set achieves a maximum frequency of 2 GHz, and the longest path passes through the Snitch register file. The Xpulp extension decreases the maximum frequency to 1.9 GHz, which also changes the longest path, passing through the accelerator offloading datapath. The integer SSRs extension does not degrade the frequency, but the critical path passes through the Snitch register file

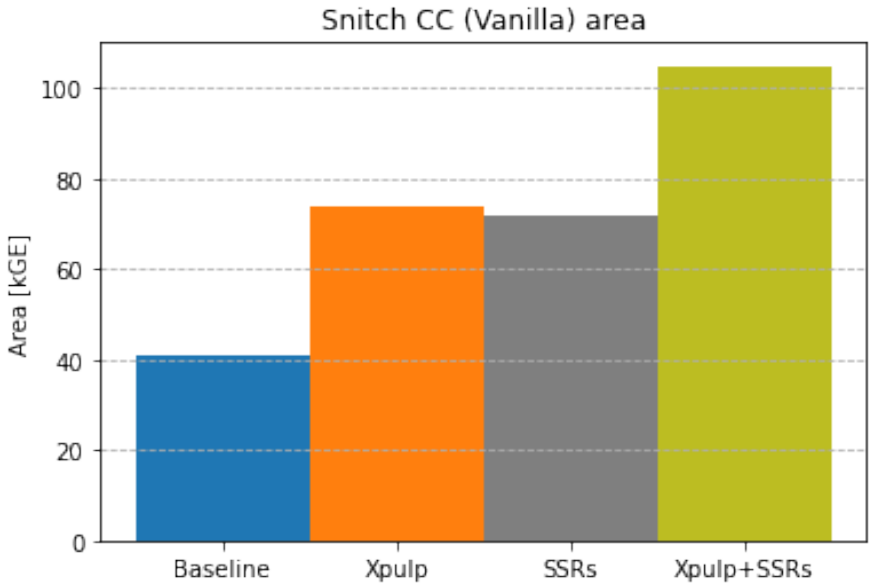

Figure 4.5: Snitch CC Area at 1 GHz with different integer extensions. Note that the baseline is the RV32IMA Snitch

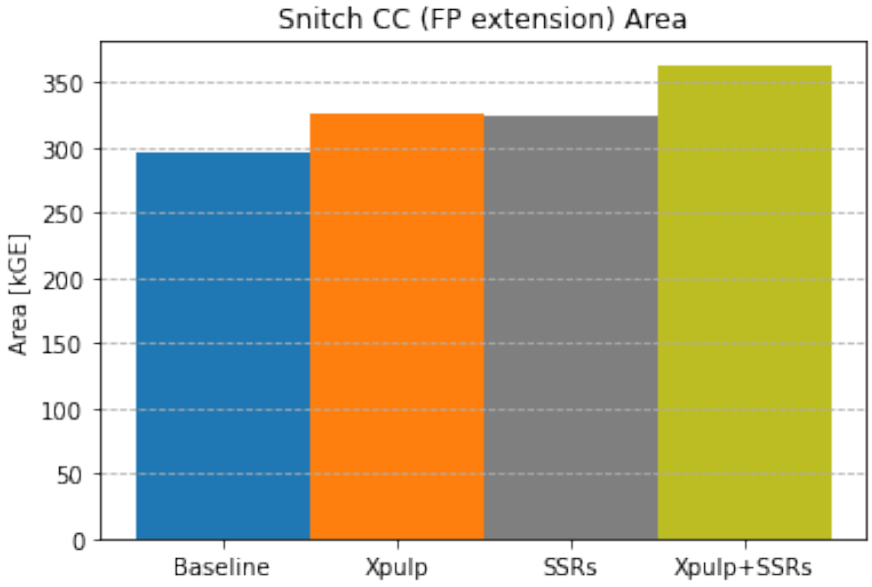

Figure 4.6: Snitch CC Area at 1 GHz with different integer extensions. Note that the baseline is the RV32IMA Snitch with FP extension

and the SSR streamer in this case. On the other hand, we can notice that the maximum frequency is worsened by the floating-point subsystem to 1.5 GHz. However, these results also prove that our integer extensions do not influence the frequency when the floating-point extension is presented.

	Frequency	**Critical path**
Snitch RV32IMA	2.0 GHz	Snitch register file
Snitch with Xpulp	1.9 GHz	Accelerator offloading datapath
Snitch with integer SSRs	2.0 GHz	Snitch register file and SSR streamer
Snitch with Xpulp+SSRs	1.9 GHz	Accelerator offloading datapath
Snitch with FP	1.5 GHz	Floating-point subsystem
Snitch with FP+Xpulp	1.5 GHz	Floating-point subsystem
Snitch with FP+integer SSRs	1.5 GHz	Floating-point subsystem
Snitch with FP+Xpulp+SSRs	1.5 GHz	Floating-point subsystem

Table 4.2: Highest effective frequency and corresponding critical paths under this frequency of Snitch CC with different extensions

4.4 Conclusion

In this chapter, the benchmark results and the synthesis results on the Snitch CC have been detailed to analyze the impact of our design. We used the integer matrix multiplication algorithm as our benchmark, varying the input matrices' size, data widths, and kernels belonging to different extensions for the performance evaluation. When both the Xpulp ISA extension and the integer SSR extension are enabled, the Snitch CC achieves a maximum speedup of 5.9×, 22.6× and 77.4× in terms of MACs/cycle for 32-bit, 8-bit and 4-bit *matmul* respectively. For 32-bit *matmul*, we analyzed the underlying reasons why the Xpulp+SSR kernel does not achieve 1 MAC/cycle theoretically. We observed the following: the book-keeping instructions for each hot-loop result in an 11% loss in performance, and the setup overhead for SSR is not negligible when the matrix size is small; also we consumed some instructions to store the result matrices and reset the accumulation register. For lower bit-width data such as 8-bit and 4-bit, we noticed that their computation benefits greatly from our extensions thanks to the SIMD nature. We synthesized the Snitch CC with different extensions in 22 nm technology to evaluate the area and timing impact. The Xpulp ISA extension increased the Snitch CC area by 32.7 kGE at 1 GHz. While the integer SSR extension only contributes 4.9 kGE with respect to the existing floating-point SSR. In general, our Xpulp ISA and integer SSRs extensions increased the area by only 12% compared with the FP-capable CC at 1 GHz. Similarly for the Xpulp-capable CC, our integer SSRs extension leads to an area overhead of 42%. In addition, our hardware extension only decreases the highest effective frequency from 2 GHz to 1.9 GHz, and the longest path in the floating-point datapath is not affected. In

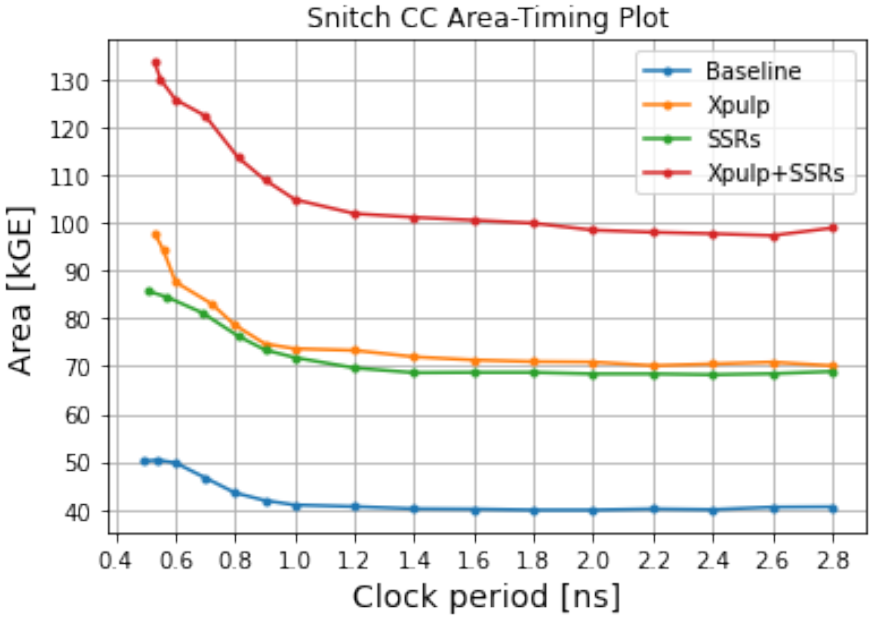

Figure 4.7: Area-Timing plot of the Snitch CC. Note that the baseline is the RV32IM vanilla Snitch CC

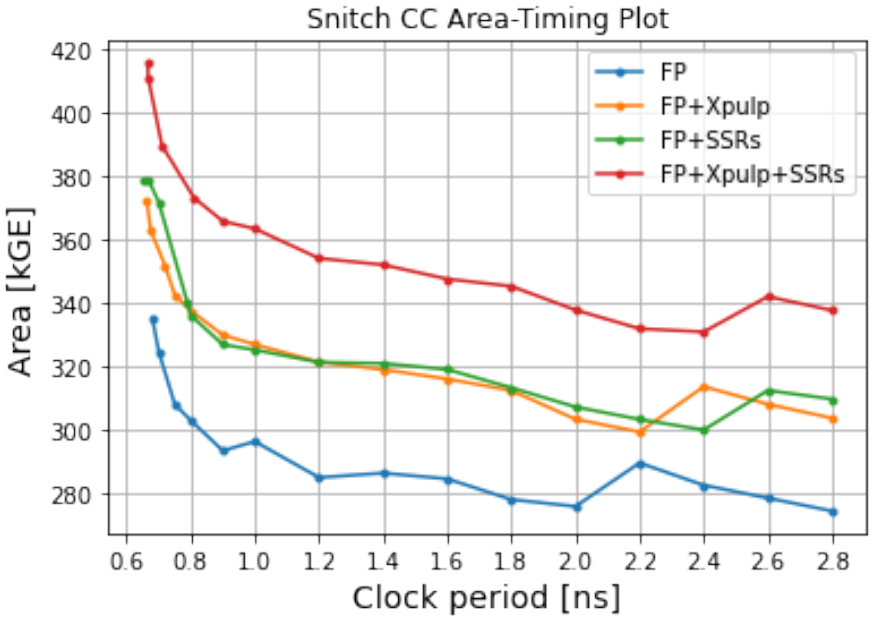

Figure 4.8: Area-Timing plot of the Snitch CC. Note that the baseline is the RV32IM Snitch CC with FP extensions

summary, our extensions improve the performance remarkably, only introducing the reasonable area overhead and timing influence.

Chapter 5

Conclusion and Future Work

5.1 Summary

In Chapter 2, we explained the necessary background knowledge for this project. First, we introduced the RISC-V open-source ISA along with its extensions. Next, SSRs are presented as a custom RISC-V extension to improve FU utilization, which we will extend to enhance the integer FU utilization in our work. Then, we introduced the hardware platform we will work on in this project, the Snitch cluster, and its detailed architecture was explained. Finally, another RISC-V system Mempool was introduced. Mempool was improved for ISP purposes with the ISA extensions, namely Xpulpimg and sub-byte SIMD. We are interested in porting them to the mainline Snitch and exploring their impact.

In Chapter 3, we defined a unified Snitch system with integer extensions for integer workloads acceleration. The hardware implementation of the integer extensions in Snitch was described, which consists of the following three parts:

- Xpulpimg ISA integration;
- Sub-byte SIMD ISA integration;
- Integer support for SSRs and the datapath sharing of SSR streamer.

For the Xpulpimg and sub-byte SIMD ISA integration, we modified the logic inside the Snitch core in a parameterized style to support these new instructions without harming the extensibility and modularity of the Snitch core. Also, we integrated the Snitch IPU as an integer accelerator into the Snitch CC through the generic accelerator interface of the Snitch core. For the integer SSR extension, we extended the Snitch core to be able to redirect the register read or write transaction to the external streams and arbitrate the usage of the SSR streamer for floating-point and integer datapath. Also, the SSR streamer is extended to have 32-bit integer support maintaining the original floating-point abilities. We followed a top-down approach to introduce our hardware extensions

from high level to details. Finally, we introduced our verification methodology during our implementation.

In Chapter 4, the benchmark results and the synthesis results on the Snitch CC have been detailed to analyze the impact of our design. We used the integer matrix multiplication algorithm as our benchmark, varying the input matrices' size, data widths, and kernels belonging to different extensions for the performance evaluation. When both the Xpulp ISA extension and the integer SSR extension are enabled, the Snitch CC achieves a maximum speedup of 5.9×, 22.6×, and 77.4× in terms of MACs/cycle for 32-bit, 8-bit and 4-bit *matmul* respectively. For 32-bit *matmul*, we analyzed the underlying reasons why the Xpulp+SSR kernel does not achieve 1 MAC/cycle theoretically. We observed the following: the book-keeping instructions for each hot-loop result in an 11% loss in performance, and the setup overhead for SSR is not negligible when the matrix size is small; also we consumed some instructions to store the result matrices and reset the accumulation register. For lower bit-width data such as 8-bit and 4-bit, we noticed that their computation benefits greatly from our extensions thanks to the SIMD nature. We synthesized the Snitch CC with different extensions in 22 nm technology to evaluate the area and timing impact. The Xpulp ISA extension increased the Snitch CC area by 32.7 kGE at 1 GHz. While the integer SSR extension only contributes 4.9 kGE with respect to the existing floating-point SSR. In general, our Xpulp ISA and integer SSRs extensions increased the area by only 12% compared with the FP-capable CC at 1 GHz. Similarly for the Xpulp-capable CC, our integer SSRs extension leads to an area overhead of 42%. In addition, our hardware extension only decreases the highest effective frequency from 2 GHz to 1.9 GHz, and the longest path in the floating-point datapath is not affected. In summary, our extensions improve the performance remarkably, only introducing the reasonable area overhead and timing influence.

5.2 Main contributions

The major contributions of this work consist of the following:

- We integrated the existing Xpulpimg and sub-byte SIMD ISA extensions into the mainline Snitch system;
- We extended the existing floating-point SSRs to support integer SSRs but maintained all the floating-point abilities and integrated the integer SSRs into the mainline Snitch system;
- We evaluated these integer extensions in terms of performance, area, and timing.

These contributions solve the research questions we proposed in Chapter 1:

- *How can we define a unified Snitch architecture with available and possible integer extensions for efficient processing of integer workloads?*
- *What are the performance benefits and cost of the potential integer extensions in Snitch?*

The answers can be formulated as follows: The unified Snitch architecture is based on the vanilla Snitch integer core in the mainly Snitch system. We extended it by integrating the Xpulp ISA extensions and adding integer SSRs support at the CC level to boost its performance on integer-based workloads. The final architecture achieves remarkable speedup on integer matrix multiplication benchmarks: a maximum speedup of 5.9×, 22.6× and 77.4× in terms of MACs/cycle for 32-bit, 8-bit and 4-bit *matmul* respectively. The maximum effective frequency decreases by 5% when both the Xpulp ISA and integer SSRs extensions are enabled. And these two extensions increase the area of Snitch CC by 32.7 kGE and 4.9 kGE. In summary, our integer extensions empowers the Snitch system to achieve high performance in integer workloads, such as DSP and QNN, with reasonable area overhead.

5.3 Future work

Due to the time limit and the new ideas found during this project, there are many opportunities for future improvements. We list some potential working directions as follows:

- We could investigate the impact of the integer hardware loop proposed in [38]. This ISA extension provides the hardware setup for the nested loops before entering the loop body. Thus it eliminates the need for index incrementation and branching, which could potentially amend the performance gap in Xpulp+SSR extensions and approach the theoretical throughput.
- We could update the RISC-V toolchain in Snitch repository [39] to support Xpulpimg and sub-byte SIMD ISA. Since the mainline Snitch system uses PULP-LLVM toolchain [40] as the main compiler. The up-to-date PULP-LLVM does not support the Xpulpimg and the sub-byte SIMD ISA extensions we ported in the mainline Snitch. In our project, we still used the PULP-GNU toolchain in Mempool, where these instructions were originally implemented. Hence, an integrated and unified compiler can be exploited to form a complete ecosystem to facilitate future work.
- We could use additional representative integer applications for the performance evaluation. In our work, we only explored the impact of our integer extensions under different bit widths with matrix multiplication algorithms. However, there are lots of typical integer-based workloads, such as graph processing, convolutions, FFT, etc. More extensive performance evaluation could help us understand our extensions' benefits and bottlenecks, leading to further improvements.
- We could analyze the power consumption of our extensions. Power consumption is also one of the main concerns for current computer systems. We can also measure the energy efficiency of our system based on the power data to have a more exhaustive evaluation.

List of Acronyms

ALU . . . . . .arithmetic logic unit

CC . . . . . .core complex

CISC . . . . . .complex instruction set computer

CSR . . . . . .control and status register

DMA . . . . . .direct memory access

DNN . . . . . .deep neural network

DSP . . . . . .digital signal processing

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

FIFO . . . . . .first in, first out

FP . . . . . .floating-point

FPSS . . . . . .floating-point subsystem

FPU . . . . . .floating-point unit

FREP . . . . . .floating-point repetition

FSM . . . . . .finite state machine

FU . . . . . .functional unit

GE . . . . . .gate equivalent

IoT . . . . . .internet of things

IPU . . . . . .integer processing unit

ISA . . . . . instruction set architecture

ISP . . . . . image signal processing

LSB . . . . . least significant bit

LSU . . . . . load-store unit

MAC . . . . . multiply-accumulate

MCU . . . . . microcontroller unit

MULDIV . . . . integer multiply-divide unit

PC . . . . . program counter

PULP . . . . . Parallel Ultra-Low Power

QNN . . . . . quantized neural network

RAW . . . . . read-after-write

RISC . . . . . reduced instruction set computer

RTL . . . . . register transfer level

SIMD . . . . . single instruction multiple data

SSR . . . . . stream semantic register

TCDM . . . . . tightly coupled data memory

VLIW . . . . . very long instruction word

List of Figures

List of Tables

Bibliography

[1] J. chun Zhao, J. feng Zhang, Y. Feng, and J. xin Guo, "The study and application of the iot technology in agriculture," in *2010 3rd International Conference on Computer Science and Information Technology*, vol. 2, 2010, pp. 462–465.

[2] M. Hassanalieragh, A. Page, T. Soyata, G. Sharma, M. Aktas, G. Mateos, B. Kantarci, and S. Andreescu, "Health monitoring and management using internet-of-things (iot) sensing with cloud-based processing: Opportunities and challenges," in *2015 IEEE International Conference on Services Computing*, 2015, pp. 285–292.

[3] A. S. Crandall, N. C. Krishnan, B. L. Thomas, and D. J. Cook, "Casas: A smart home in a box," *Computer*, vol. 46, no. 07, pp. 62–69, jul 2013.

[4] S. Li, L. Xu, and S. Zhao, "5g internet of things: A survey," *Journal of Industrial Information Integration*, vol. 10, 02 2018.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[6] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[7] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.

[8] S. R. Sridhara, M. DiRenzo, S. Lingam, S.-J. Lee, R. Blazquez, J. Maxey, S. Ghanem, Y.-H. Lee, R. Abdallah, P. Singh *et al.*, "Microwatt embedded processor platform for medical system-on-chip applications," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 4, pp. 721–730, 2011.

[9] M. Konijnenburg, S. Stanzione, L. Yan, D.-W. Jee, J. Pettine, R. Van Wegberg, H. Kim, C. van Liempd, R. Fish, J. Schluessler *et al.*, "28.4 a battery-powered efficient multi-sensor acquisition system with simultaneous ecg, bio-z, gsr, and ppg," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 480–481.

[10] N. Ickes, D. Finchelstein, and A. P. Chandrakasan, "A 10-pj/instruction, 4-mips micropower dsp for sensor applications," in *2008 IEEE Asian Solid-State Circuits Conference*. IEEE, 2008, pp. 289–292.

[11] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 24–35.

[12] L. Cavigelli and L. Benini, "Origami: A 803-gop/s/w convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, 2016.

[13] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators," in *2019 IEEE high performance extreme computing conference (HPEC)*. IEEE, 2019, pp. 1–9.

[14] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 683–688.

[15] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 27–39.

[16] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[17] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 26–36. [Online]. Available: https://doi.org/10.1145/1815961.1815967

[18] 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, nov 2021.

[19] S. Mazzola, "ISA Extensions in the Snitch Processor for Signal Processing," *Master Thesis*, 2021.

[20] A. Garofalo, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "Xpulpnn: Enabling energy efficient and flexible inference of quantized neural network on RISC-V based iot end nodes," *CoRR*, vol. abs/2011.14325, 2020.

[21] Wikipedia. (2022) Instruction Set Architecture. Instruction Set Architecture. [Online]. Available: https://en.wikipedia.org/wiki/Instruction_set_architecture

[22] A. Waterman, L. Yunsup, P. David, and A. Krste, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.0*, May 2014. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf

[23] A. Waterman, "Design of the risc-v instruction set architecture," *PhD thesis*, 2016.

[24] 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.

[25] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[26] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 26–36. [Online]. Available: https://doi.org/10.1145/1815961.1815967

[27] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 02, pp. 530–543, feb 2020.

[28] R. Wilson, E. Beigne, P. Flatresse, A. Valentian, F. Abouzeid, T. Benoist, C. Bernard, S. Bernard, O. Billoint, S. Clerc, B. Giraud, A. Grover, J. Le Coz, I. Miro Panades, J.-P. Noel, B. Pelloux-Prayer, P. Roche, O. Thomas, Y. Thonnart, D. Turgis, F. Clermidy, and P. Magarshack, "A 460mhz at 397mv, 2.6ghz at 1.3v, 32b vliw dsp, embedding fmax tracking," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 452–453.

[29] A. Gonzalez, F. Latorre, and G. Magklis, "Processor microarchitecture: An implementation perspective," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–116, 2010.

[30] M. A. Cavalcante, S. Riedel, A. Pullini, and L. Benini, "Mempool: A shared-l1 memory many-core cluster with a low-latency interconnect," *CoRR*, vol. abs/2012.02973, 2020. [Online]. Available: https://arxiv.org/abs/2012.02973

[31] 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.

[32] A. Faroqui and V. G. Oklobdzija, "Impact of architecture extensions for media signal processing on data-path organization," in *Conference Record of the Thirty-Fourth Asilomar Conference on Signals, Systems and Computers (Cat. No. 00CH37154)*, vol. 2. IEEE, 2000, pp. 1679–1683.

[33] X. Zhang, Z. Li, and Q. Zheng, "Design of a configurable fixed-point multiplier for digital signal processor," in *2009 Asia Pacific Conference on Postgraduate Research in Microelectronics & Electronics (PrimeAsia)*. IEEE, 2009, pp. 217–220.

[34] PULP Platform. (2022) PULP RISC-V GNU toolchain. Accessed September 12, 2022. [Online]. Available: https://github.com/pulp-platform/pulp-riscv-gnu-toolchain

[35] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA version 20191213*, Dec. 2019. [Online]. Available: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

[36] RISC-V. (2022) RISC-V tests. GitHub. Accessed September 12, 2022. [Online]. Available: https://github.com/riscv-software-src/riscv-tests

[37] ——. (2022) RISC-V ISA simulator. GitHub. Accessed September 12, 2022. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

[38] OpenHW Group. (2022) CV32E40P User Manual. Accessed September 12, 2022. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40p-user-manual

[39] PULP Platform. (2022) Snitch. Accessed September 12, 2022. [Online]. Available: https://github.com/pulp-platform/snitch

[40] ——. (2022) PULP RISC-V LLVM toolchain. Accessed September 12, 2022. [Online]. Available: https://github.com/pulp-platform/llvm-toolchain-cd