TÉCNICO LISBOA

RISC-V Streaming Extension Support on the Spike Simulator

João Pedro Rosa Baptista

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Nuno Filipe Valentim Roma
Prof. Nuno Filipe Simões Santos Moraes da Silva Neves

Examination Committee

Chairperson: Prof. António Manuel Raminhos Cordeiro Grilo
Supervisor: Prof. Nuno Filipe Valentim Roma
Member of the Committee: Prof. João Carlos Viegas Martins Bispo

November 2023

Declaration

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

Acknowledgments

This document marks the end of a long journey that lasted 5 long exciting years. This journey would not be possible without the help of my parents who were always carrying me on their shoulders. I am eternally grateful to my parents for the constant deposited trust throughout these years. I want to also give a special thank you to my entire family for their support. A special thanks to my brother, my friends and my girlfriend for the immeasurable laughs and good moments.

Of course, this academic journey would not be the same without my dissertation supervisors. Thank you, Prof. Nuno Roma and Prof. Pedro Tomás for allowing me to learn so much with you. I want to also give a special thanks to Nuno Neves who was always present during this dissertation.

This thesis was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project 2022.06780.PTDC.

Abstract

Vectorial and Single-Instruction-Multiple-Data (SIMD) instruction-set extensions have gained added attention in the last decade, as a result of an increased prevalence of computational demanding application domains, pushing the need to exploit as much data-level parallelism as possible. The numerous Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) extensions from Intel/AMD or the NEON and Scalable Vector Extension (SVE) extensions from ARM are some well-known examples of these Instruction Set Architecture (ISA) extensions. Following these same steps, the well-known RISC-V ISA has recently established a comparable vectorial extension, known as the RISC-V Vector (RVV). Other ISA extensions have also been developed to enhance the performance and power/energy efficiency of computing systems. The extremely successful Unlimited Vector Extension (UVE), developed at the INESC-ID HPCAS lab, is one of those extensions. Its prime objective is to provide consolidated support for data-stream processing, alleviating the Central Processing Unit (CPU) from the memory indexing/addressing tasks, while also simplifying loop control. Spike is recognized as the golden reference functional RISC-V ISA software simulator. Making justice to its title, Spike already supports a large collection of extensions including the pertinent RVV extension. However, it still lacks accompany the recent arise of stream-based ISA extensions. Having what was stated in mind, this Thesis proposal aims to define a new RISC-V stream-based extension, integrate it with the rest of the RISC-V ISA and introduce support for the defined extension on the Spike functional simulator. As a result, users will be able to explore such extension on C/C++ applications.

Keywords

SIMD instruction-set extensions; Stream; RISC-V ISA ; RVV ; UVE; Spike;

Resumo

Extensões de instuções vetoriais e Single-Instruction-Multiple-Data (SIMD) têm ganho atenção adicional na última década, como resultado de uma prevalência crescente de domínios exigentes em termos de computação, aumentando assim a necessidade de explorar ao máximo a possibilidade de paralelismo ao nível dos dados. As extensões SSE e AVX da Intel/AMD ou as extensões NEON e SVE da ARM são dois exemplos conhecidos destas Instruction Set Architecture (ISA) extensões. Da mesma maneira, foi recentemente estabelecida uma extensão vetorial comparável para o ISA do RISC-V, conhecida como extensão RISC-V Vector (RVV). Outras extensões também foram desenvolvidas recentemente com o fim de melhorar o desempenho e a eficiência de energia dos sistemas de computação. A extensão de instruções Unlimited Vector Extension (UVE) desenvolvida no laboratório HPCAS do INESC-ID, cuja principal contribuição é fornecer suporte consolidado para processamento de streams de dados, aliviando o processador de tarefas de indexação/endereçamento de memória, além de simplificar o controle de ciclos, é um destes casos. O simulador Spike apesar de ser reconhecido como o simulador de referência para o ISA do RISC-V, ainda não tem acompanhado o recente crescimento de extensões de instruções baseadas no conceito de stream. Averiguando o relatado, propomos a definição de uma nova extensão RISC-V baseada no conceito de stream, integração da mesma com o resto do ISA da RISC-V e introdução de suporte para a extensão definida no simulador Spike. Deste modo, os usuários poderão simular e testar o desempenho desta nova promissora extensão quando usada em aplicações de C/C++.

Palavras Chave

Extensões de instuções vetoriais; Stream; RISC-V ISA; RVV; UVE; Spike;

Contents

1 Introduction 1

1.1 Motivation . . . 2

1.2 Objectives . . . 3

1.3 Contributions . . . 4

1.4 Outline . . . 5

2 Background and State-of-the-Art 7

2.1 Instruction Set Architecture . . . 8

2.1.1 RISC-V . . . 8

2.2 Vectorial and Single-Instruction-Multiple-Data (SIMD) extensions . . . 9

2.2.1 Multi-Media Extensions (MMX) . . . 10

2.2.2 Streaming SIMD Extensions (SSE) . . . 10

2.2.3 Advanced Vector Extensions (AVX) . . . 11

2.2.4 NEON Extension . . . 12

2.3 Vector-Length Agnostic SIMD extensions . . . 13

2.3.1 Scalable Vector Extension (SVE) . . . 13

2.3.2 RISC-V Vector (RVV) . . . 16

2.3.3 SVE and RVV wind-up . . . 17

2.4 Data-flow and Stream-based approaches . . . 18

2.4.1 Unlimited Vector Extension (UVE) . . . 20

2.4.2 Compilation Support . . . 21

2.5 Simulation tools . . . 22

2.6 Spike Simulator . . . 23

2.6.1 Overview . . . 23

2.6.2 Main Functional Modules . . . 24

2.7 Discussion . . . 25

2.8 Summary . . . 26

3 RISC-V stream-based extension 27

3.1 Overview 28

3.2 Memory Access Pattern Description 28

3.2.1 Multidimensional Access Encoding 28

3.2.2 Imperfect Loop Access Encoding 29

3.2.3 Indirect Memory Access Encoding 30

3.3 Architectural State 34

3.3.1 Microarchitecture Overview 34

3.3.2 Available registers 36

3.4 Instructions Design 36

3.4.1 Stream Configuration Instructions 36

3.4.2 Stream Control Instructions 40

3.4.3 Flow Control Instructions 40

3.5 Instruction Set Overview 42

3.6 Summary 44

4 Spike Simulator Extension 45

4.1 Overview 46

4.2 Stream Unit Implementation 46

4.2.1 Streams Implementation 49

4.3 Instruction Macros 51

4.3.1 Integration with RISC-V Paradigm 54

4.4 Summary 58

5 Results Overview and Discussion 59

5.1 Overview 60

5.2 Methodology 60

5.2.1 Spike Simulator 60

5.2.2 RISC-V Opcodes 60

5.2.3 RISC-V Proxy Kernel 60

5.2.4 Compiler 61

5.2.5 Benchmarks 61

5.3 Results 61

5.3.1 Implementation Results 62

5.3.2 Benchmark Results 64

6 Conclusions and Future Work **69**
6.1 Conclusions 70
6.2 Future Work 71
Bibliography **71**

List of Figures

2.1 SISD vs SIMD Operations. 9
2.2 NEON and floating-point register file. Image from [1]. 12
2.3 SVE registers. Image from [2]. 14
2.4 *whilelt* instruction syntax. 15
2.5 vector CSRs. Image from [3]. 16
2.6 ”*vsetvli*” instruction syntax. 17
2.7 Saxpy kernel implementation on ARM SVE and RISC-V V. Image from [4]. 18
2.8 Overview of Decouplcd-Stream Paradigm vs Conventional Out-of-Order. Image from [5]. . 20
2.9 Saxpy kernel implementation on UVE. Image from [4]. 21
2.10 Compilation flow for stream-based vector extensions. Image from [6]. 22
2.11 Spike main modules Overview. 24
2.12 Spike Overview with the addition of the vector unit. 25

3.1 Stream Representation 31
3.2 Stream Representation with Stride of 2 31
3.3 2-Dimensional Stream Representation 32
3.4 2-Dimensional Strided Stream Representation 32
3.5 Lower Triangular Memory Access Pattern 33
3.6 Lower Triangular Memory Access Pattern Representation. 33
3.7 Indirect Memory Access Pattern 34
3.8 Indirect Memory Access Pattern Representation. 34
3.9 Microarchitecture Overview with a Stream Unit 35
3.10 3-Dimensional Stream Representation. 38
3.11 inner loop processing with outer loop memory access Representation. 41
3.12 Normal Streaming versus Vector-coupled Streaming. 42

4.1 Spike Overview with the addition of the Stream Unit. 46

5.1 Testing Protocol used to validate the modified Spike with the defined Stream-based Extension. 62
5.2 RVV versus RVV + Stream. 66
5.3 Scalar versus Scalar + Stream. 68

List of Tables

3.1 Stream Control Instructions 40
3.2 Stream-based extension Instructions 43

5.1 Benchmarks Executed with the defined Stream-based Extension 63

Listings

2.1 SAXPY kernel implementation using ARM SVE extension. 15
2.2 SAXPY kernel implementation using RISC-V V extension. 16
3.1 Lower Triangular Memory Acces Pattern C code 29
3.2 Indirect Memory Acces Pattern C code 30
3.3 Simple Array access C Code 31
3.4 Simple Array access with Stride C Code 31
3.5 2-Dimensional Array access C code 32
3.6 2-Dimensional Strided Array access C code 32
3.7 inner loop processing with outer loop memory access C Code 41
3.8 SAXPY benchmark with Scalar RISC-V instructions 44
3.9 SAXPY benchmark with the stream-based extension and Scalar RISC-V instructions . . . 44
4.1 Stream Unit Class (Part 1) 47
4.2 Stream Unit Class (Part 2) 47
4.3 Stream Unit Init Function 47
4.4 Processor Function initiating the Stream Unit 48
4.5 Stream Structure Code 49
4.6 Dimension Structure Code 50
4.7 Modifier Structure Code 50
4.8 scrt.sta.ld.w instruction Macro code 52
4.9 STREAM_TYPE_SIZE Macro Code 52
4.10 STREAM_GET_PARAMS Macro Code 53
4.11 STREAM_ASSIGN_TYPE Macro Code 53
4.12 STREAM_UPDATE_DIMENSIONS Macro Code 54
4.13 *vadd.h* file Code 55
4.14 Vadd Macro Code 55
4.15 Vadd Modified Macro Code (Part 1) 55
4.16 Vadd Modified Macro Code (Part 2) 56

4.17 ADD_ITERATE_LOAD_STREAMS Macro Code 56
4.18 EXECUTE_ITERATE_LOAD_STREAM Macro Code 57
5.1 Memory Copy Kernel with RVV 65
5.2 Memory Copy Kernel with RVV and the stream-based extension 65
5.3 Memory Copy Kernel with scalar RISC-V instructions 67
5.4 Memory Copy Kernel with scalar RISC-V instructions and the stream-based extension . . 67

Acronyms

SIMD	Single-Instruction-Multiple-Data
SISD	Single-Instruction-Single-Data
ISA	Instruction Set Architecture
RVV	RISC-V Vector
UVE	Unlimited Vector Extension
MMX	Multi-Media Extensions
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions
SVE	Scalable Vector Extension
CPU	Central Processing Unit
VLA	Vector-Length Agnostic
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
ISS	Instruction Set Simulator
HPC	High-Performance-Computing
CSR	Control Status Register
SSR	Stream Semantic Register
SSA	Static Single Assignment
IR	Intermediate Representation
ILP	Instruction-Level Parallelism
DLP	Data-Level Parallelism
DSA	Domain-Specific Architectures

DSL Domain-Specific Languages

FIFO First-In First-Out

1

Introduction

Contents

1.1 Motivation 2
1.2 Objectives 3
1.3 Contributions 4
1.4 Outline 5

1.1 Motivation

Since the beginning of the computing area, there is a perpetual competition to maximise processor performance. There were times when the growth of the processing power would be predictable as the number of transistors increased, as stated in Moore's Law [7]. Helping Moore's Law was the Dennard Scaling [8] projection, which stated that as transistor density increased, power consumption per transistor would drop, making computers energy efficient. Although, this era has ended [9]. With the emergence of the *"Thermal Wall"* problem, the slow down of Dennard scaling and the end of Moore's Law, the procedure of shrinking transistors, ensuring even smaller circuits, is reaching its physical boundaries.

One consequence in the computing world was the stagnation of the single core frequency. In an effort to mitigate this issue, the main focus shifted from exploring Instruction-Level Parallelism (ILP) to finding ways of combining instruction, task and Data-Level Parallelism (DLP), which constitute the core of the architecture paradigm present in modern high-performance processors. Furthermore, the recent proliferation of certain application domains, such as deep learning, with increasing computational performance demands, has reinforced the need to further exploit DLP. In this scenario, the development of vectorial and Single-Instruction-Multiple-Data (SIMD) instruction-set extensions, which allow multiple data elements to be processed at the same time, has been gaining renewed attention.

Traditional concepts of these extensions, such as the numerous extensions from Intel [10–12] and the NEON [1] extension from ARM are based on apriori fixed-size registers, which despite fulfilling their purpose, pose questionable issues. First, having a fixed vector length leads to portability issues, since any modification of the length requires a new instruction set to be defined, and therefore new code needs to be written, compiled and deployed. Next, since the optimal vector length always depends on the workload, with a fixed vector length, it is often hard to choose the perfect vector length for a given application. This problem is addressed by Vector-Length Agnostic (VLA) SIMD extensions that do not rely on a fixed vector size. In [3], the recognized RISC-V organization denoted the development of a VLA SIMD extension named RISC-V Vector (RVV), which mainly adds the feature of tuning the vector length at runtime. In [2], ARM also denoted the development of a VLA SIMD extension named Scalable Vector Extension (SVE), which mainly relies on predication to drive vectorized loop control flow decisions. When compared with the typical SIMD extensions, the added overhead, resulting from the instructions needed to attain a variable vector length, can constrain the application throughput.

Despite parallelism being a solution in terms of performance, by itself, does not solve the challenge of energy-efficient computation that was exacerbated by the end of Dennard scaling. To tackle the lack of energy-efficient computation, Domain-Specific Architectures (DSA) and Domain-Specific Languages (DSL) emerged, which rely on designing architectures tailored to a specific problem domain, offering significant performance and efficiency. Attached to the growth of DSA, some techniques, such as memory decoupling, started to gain relevance [13, 14]. By decoupling the memory accesses from

the computation, it is possible to achieve more performance and efficiency in both domains. This is mainly attained by allowing data acquisition to occur in parallel with data manipulation. In fact, DSAs are in most cases related to the use of data-flow and stream-based techniques to improve memory transactions [15–18].

Based on the growth of DSA, general-purpose processors adopted data streaming techniques to mainly combat the VonNeumann architecture limitations [5, 19]. These limitations are clear by the fact that the performance of memory devices has not improved at the same rate as that of processors, leading to an increase in the gap between Central Processing Unit (CPU) and memory speeds.

Bonded with this adoption, several Instruction Set Architecture (ISA) extensions featuring streaming techniques were developed. One novel example is the Unlimited Vector Extension (UVE) extension [4], which by aggregating both VLA SIMD and data streaming approaches promises to reduce the CPU's workload associated with memory addressing/indexing in general-purpose RISC-V-based processors. Despite the considerable number of streaming extensions being developed, there is still not one associated with the official RISC-V extensions.

Despite the growth of stream-based approaches, there is still little hardware support for such approaches. By the beginning of this dissertation, there was no hardware support for the stream-based extension that is going to be proposed. Therefore, the only valid solution to test the proposed extension is through a functional simulator. Connected with this whole paradigm, Spike [20] is a RISC-V ISA simulator, which implements a functional model of one or more RISC-V harts. Spike, more known as the golden reference RISC-V ISA simulator, supports all standard RISC-V instructions as well as a vast number of extensions, in which the previously mentioned RVV SIMD extension. However, despite its title, Spike still lacks in providing support for the newly raised stream-based approaches.

1.2 Objectives

In general-purpose processors, the performance is majorly limited by the time spent with memory addressing/indexing actions. To solve this issue, new streaming techniques have been adopted into general-purpose processors [5,19], promising to reduce the existent gap between processing and memory access speed. Despite the considerable number of developed extensions based on streaming, the RISC-V paradigm still does not have an official stream-based extension. Based on what was stated, this represents an open chance to **propose a new RISC-V stream-based extension**.

Moreover, to exploit DLP at maximum in general-purpose processors, the development of SIMD extensions reached its peak [1, 10–12]. Furthermore, by making use of a variable vector length, VLA SIMD extensions further enhance the performance and energy efficiency [2, 3]. This opens a chance to **combine the performance enhancement delivered by RVV, with the addition of decreasing the**

memory access latency by using streaming approaches.

Additionally, Spike, recognized as the golden reference ISA simulator by RISC-V, implements a functional model of one or more RISC-V harts. However, **Spike does not support any extension that denotes streaming techniques**.

Following the apriori statements, the main objective of this Thesis is to introduce support for a new RISC-V stream-based extension, fully compatible with the RISC-V paradigm, despite being more focused in RVV, on the Spike functional simulator. Moreover, this Thesis goal can be divided into three major objectives:

- Assess the main requirements to introduce a new RISC-V stream-based extension.
- Investigate the possibility of interaction and interoperability between a stream-based extension and standard RISC-V extensions, more specifically the RVV extension.
- Validate the defined extension through ISA simulation.

1.3 Contributions

This Thesis work is grounded on previous works [4, 5, 19] that demonstrated that the use of streaming mechanisms is the go-to in order to improve the overall performance in the memory transactions present in High-Performance Computing applications. Accordingly, this work first specified the necessary requirements to define a new RISC-V stream-based extension. Furthermore, it discussed the possibility of integrating a full stream-based extension in the RISC-V paradigm. Finally, to give evident proof of the enhancement delivered by the stream-based extension, this work aimed to provide support for the defined extension on the Spike Simulator. Therefore, during the development of this thesis, the following contributions were achieved:

- Definition of a novel stream-based extension, whose main focus is to increase performance in memory access.
- Integration of the new stream-based extension with the subsets of the RISC-V ISA, more specifically the RVV extension.
- Deployment of the proposed extension in the Spike Simulator and its functional validation and interoperability verification with other standard RISC-V extensions.

1.4 Outline

Apart from this first Introduction chapter, this document is constituted by Chapter 2, which focuses on covering the evolution of SIMD extensions, beginning with their emergence to enhance performance for multi-media applications and finishing with the recent adoption of stream-based extensions to fight the limitations of the Von Neumann architecture on general-purpose processors. It aims to introduce the necessary concepts and backgrounds in ISA that are necessary to follow the work that was developed in this thesis. Next, Chapter 3 focuses on defining the stream-based extension being proposed. First, delves into the core concepts of data streaming and stream representations used in the base of the proposed extension. Next, it aims to actually provide the definition and behaviour of the actual instructions present in the stream-based extension. Following, Chapter 4 focuses on denoting what was implemented to provide full support for the defined extension in the Spike functional Simulator. First, it aims to represent the actual streaming components simulated that are the base for the execution of the instructions. Next, it aims to actually represent the execution behaviour of the instructions. Moreover, Chapter 5 focuses on providing an overview of the methods used to evaluate the reliability and performance of the proposed extension. Finally, Chapter 6 focuses on providing conclusions on the written document and discussing possible future developments that can expand what was implemented in this Thesis work.

2

Background and State-of-the-Art

Contents

2.1 Instruction Set Architecture 8
2.2 Vectorial and SIMD extensions 9
2.3 Vector-Length Agnostic SIMD extensions 13
2.4 Data-flow and Stream-based approaches 18
2.5 Simulation tools 22
2.6 Spike Simulator 23
2.7 Discussion 25
2.8 Summary 26

To better understand the underlying work of this Thesis, this Chapter discusses general concepts and background of the RISC-V paradigm. Additionally, an analysis of Single-Instruction-Multiple-Data (SIMD) technologies is provided, more focused on detailing the evolution of the development of SIMD extensions. Moreover, Vector-Length Agnostic (VLA) SIMD extensions, such as the RISC-V Vector (RVV) extension [3], and the provided trade-offs in regard to earlier fixed vector-length extensions are tackled. After, the emergence and evolution of data-flow and stream-based approaches into general-purpose processors will be addressed, culminating in the analysis of a novel Instruction Set Architecture (ISA) extension called Unlimited Vector Extension (UVE) [4], which combines both SIMD techniques and streaming approaches. Furthermore, a brief discussion of the lack of compilation support for the recent streaming approaches will be denoted. Finally, the whole paradigm of the Spike simulator will be described.

2.1 Instruction Set Architecture

An ISA is an abstract model of a computer that dictates how the Central Processing Unit (CPU) is administered by the software. To the programmer or compiler developer, the ISA can be defined as a manual, since it is the part of the processor that is visible, supplying the only interface via which a user can communicate with the hardware. The ISA acts as a bridge between the hardware and the software, establishing both what the processor is capable of doing and how it gets done [21].

In general, an ISA stipulates the supported data types, instructions, registers, main memory management methods used by the hardware, important features (such as virtual memory and memory consistency), and the input/output model used by several ISA implementations. Regarding what was mentioned above, from a developer's perspective, it is undeniable the importance of understanding the instruction set's capabilities and how the compiler uses them in order to write more efficient code.

An ISA may be classified regarding the architectural complexity, where the two main ISA classes are Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC). RISC architecture makes use of more generalized and simple instructions that in the majority of cases are executed in a single clock cycle. In contrast, CISC architecture focuses on including more specialized and complex instructions that take more clock cycles to execute.

2.1.1 RISC-V

RISC-V [22] is an open standard ISA developed by the RISC-V International organization. Concerning computing architectures, RISC-V processors are based on RISC concepts. Contrary to other ISAs available, like the ones designed by Intel, AMD and ARM, RISC-V is available under open-source licences that do not require fees. In fact, since its creation in 2015, RISC-V was designed with the intention of supporting not only the base ISA and optional standard extensions but also providing support for future

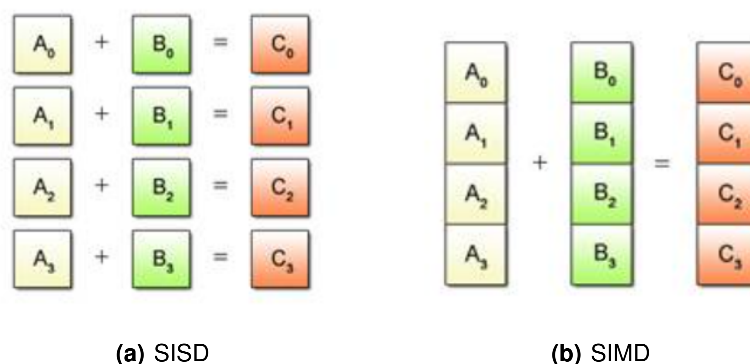
custom ISA extensions. This flexibility to use extensions to the standard ISA can offer support for specific applications, which resulted in RISC-V becoming the ISA of choice for innovations. One example arising from this characteristic is the recently RVV extension.

2.2 Vectorial and SIMD extensions

SIMD systems [1–3, 10–12] enhance, as the name indicates, the execution of the same operation on multiple elements present in a dataset. This SIMD parallelism is commonly accomplished by resorting to vectorization, where successive instances of a scalar operation, which operates on a set of single operands, are transformed into a vector instruction that operates on multiple operands at once.

In the last two decades, an increasing number of processors that adopt SIMD operations have been developed. This adoption was majority handled by adding extensions to the existing ISA. These extensions have been established as the essential go-to method to potentiate the exploitation of Data-Level Parallelism (DLP) in High-Performance-Computing (HPC) workloads. Multimedia application's vast field is one domain where these extensions are of high importance, mainly because these applications tend to operate on narrower data types than the native word size. For example, graphics applications use 3 x 8 bits for colours and one 8-bit for transparency, audio applications use 8, 16 or 24-bit samples. To accommodate narrower data types, carry chains have to be disconnected. For example, a 256-bit adder can be divided to perform simultaneously 32, 16, 8 or 4 additions on 8, 16, 32 or 64-bit, respectively. These examples highlight the possible advantage of using SIMD instructions.

To better explain this mechanism, Figure 2.1, denotes the difference between using Single-Instruction-Single-Data (SISD) or SIMD operations on performing a simple sum between two sets of four elements each.



(a) SISD **(b)** SIMD

Figure 2.1: SISD vs SIMD Operations.

When processing large sets of data, one of the major factors limiting performance is the amount of CPU time that is taken to perform data processing instructions. This CPU time hardly depends on the

number of instructions it takes to deal with the entire data set. The small example provided in 2.1 shows, on a small scale, the impact of using SIMD over SISD, where instead of needing four instructions with SISD, one was enough when using SIMD. This impact significantly grows with the increase of elements in the data set.

Moreover, in this Chapter, the evolution of the development of SIMD extensions is retracted, specifically denoting the Multi-Media Extensions (MMX) [10], the Streaming SIMD Extensions (SSE) [11], the Advanced Vector Extensions (AVX) [12], the NEON [1], the Scalable Vector Extension (SVE) [2], the RVV [3] and the UVE [4] extensions.

2.2.1 Multi-Media Extensions (MMX)

The first recognised milestone in the development of SIMD extensions was in 1996 with the arrival of the Multi-Media Extensions (MMX) [10]. Intel's MMX extensions to the x86 ISA defined four new data types. The packed byte (8 bytes packed into one 64-bit quantity), the packed word (4 words packed into one 64-bit quantity), the packed doubleword (2 doublewords packed into one 64-bit quantity) and the quadword packed (one 64-bit quantity). To support these new data types MMX provided eight 64-bit general-purpose registers. These registers, named MM0 through MM7, were designed to be random-access registers and to hold only MMX data. Therefore, one instruction could either be applied to two 32-bit integers, four 16-bit integers, or eight 8-bit integers at once, enhancing the capability to perform parallel operations on multiple data elements packed into 64 bits.

In [10], the performance of MMX was tested by carrying out a vector dot product and a 16x16 matrix-vector multiplication with MMX technology, achieving speedups of $6\times$ and $5.8\times$, respectively, over regular operations. The work team in [23] evaluated the performance of MMX for several digital signal processing algorithms and measured a speedup in the range from $0.49\times$ to $5.5\times$ for various applications. In [24], it is tested the performance of MMX for neural network applications, concluding that on MMX all operations required for a neural classification with an already trained large network could be accelerated by a factor of up to $9.8\times$.

Despite the results, MMX still presents a major weakness. In addition to MMX providing only integer operations, the MMX registers are aliases for the existing floating-point registers. As a result, operations involving the floating-point stack might also affect the MMX registers and vice versa, making it almost impossible to work with floating-point and SIMD operations in the same program.

2.2.2 Streaming SIMD Extensions (SSE)

To solve the bottleneck that existed with MMX, Intel added the Streaming SIMD Extensions (SSE) [11] to its x86 ISA. Since its emergence in 1999, SSE was followed by multiple generations that ended with

SSE4 in 2007 [11, 25–27].

Contrary to MMX, SSE [11] mainly provided support for floating-point instructions and operated on a new independent register set. This register set was composed of eight 128-bit registers, named XMM0 through XMM7, that only used four 32-bit single-precision floating-point numbers as data type. In addition, SSE also added a few integer instructions that worked on the MMX registers. SSE2 [25], later introduced, added two major features to the previous generation. SSE2 provided support for 64-bit double-precision floating-point numbers and allowed the MMX integer operations to be performed on the XMM registers. Therefore, SSE2 users could benefit from SIMD operations on any data type (from 8-bit integer to 64-bit float) entirely by using the XMM registers.

SSE3's [26] main addition was the capability of performing operations between two elements that were stored in the same register. SSE4's [27] main purpose was to add instructions not mainly focused on multimedia applications, for example, the *popcnt* instruction, which counts the number of bits set to 1, was widely employed in cryptography.

In [28], by using SSE to implement an image processing application, a speedup of $2.49\times$ and $1.74\times$, over using regular operations and MMX, respectively, was obtained.

2.2.3 Advanced Vector Extensions (AVX)

The numerous generations of the SSE instructions were a significant improvement over the previous MMX instructions, helping to accelerate a wide range of applications. Although, as processor speeds continued to increase and the demand for more powerful vector processing grew, it became clear that SSE was not sufficient to meet the needs of some applications.

To satisfy these needs, the next step was again accomplished by Intel, which extended its x86 ISA with the Advanced Vector Extensions (AVX) [12] in 2011. AVX's core improvement is the support for 256-bit vector operations. AVX defined sixteen 256-bit registers, named YMM0 through YMM15 and a 32-bit control/status register called MXCSR. The YMM registers alias over the already mentioned 128-bit XMM registers, treating the XMM registers as the lower half of the corresponding YMM register. Therefore, in addition to allowing any multiple of 32-bit floating-point type that adds to 128 bits (single-precision), it also allows any multiple of 64-bit floating-point type that adds to 256 bits (double-precision). It also allows multiples of any integer type that adds up to 128 bits. Furthermore, AVX also featured three-operand, non-destructive operations.

AVX2 [29], later released in 2013, focused on the expansion of most vector integer instructions to 256 bits and on the introduction of new instructions for packing and unpacking integers and floating-point values. AVX-512 [30] released in 2016 provided support for 512-bit vector operations, defining thirty-two 512-bit registers, named ZMM0 through ZMM31, that were also alias over the already mentioned YMM and XMM registers. On top of that, it added eight new opmask registers, named k0 through k7, for

masking most of the AVX-512 instructions.

In [12], the AVX instructions were tested on computing Mandelbrot set images achieving an average speedup of $7.14\times$ over expanding the complex types with floats and an average speedup of $2.01\times$ over using an intrinsic-based SSE version. In [31], the AVX instructions were used to perform N-body simulation for self-gravitating collisional systems, achieving an overall speedup of $2\times$ and $5\times$ over using SSE instructions and without using any SIMD instructions, respectively.

2.2.4 NEON Extension

Following the steps of Intel, ARM also introduced its own SIMD extension in 2007 called NEON [1], which was designed to be fully compatible with the ARMv7 and ARMv8 architectures. Contrarily to the regular use of the 32-bit ARM register file, NEON instructions and floating-point instructions share a different register file, referenced in Figure 2.2, which is a collection of registers that can be accessed as 32-bit, 64-bit or 128-bit. The selection of the registers depends on whether the instruction to be performed is a NEON instruction or a vector floating-point instruction.

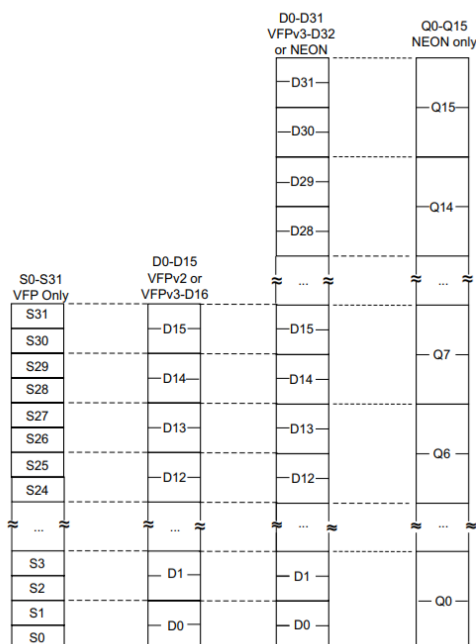


Figure 2.2: NEON and floating-point register file. Image from [1].

As seen by Figure 2.2, NEON registers present a valuable difference from the ones presented by Intel since they can be represented with two different views, covering a greater number of cases. NEON registers can be represented as thirty-two 64-bit registers, named D0 through D31, or sixteen 128-bit registers, named Q0 through Q15. NEON instructions work on specific data types varying from 8-bit,

16-bit, 32-bit or 64-bit signed/unsigned integers to 16-bit or 32-bit floating-point numbers. The contents of the NEON registers are vectors of elements of the same data type. A vector is divided into lanes and each lane contains a data value called an element.

Therefore, the number of lanes in a NEON vector depends on the size of the vector and the size of the data elements in the vector. Summing up, while D registers can keep vectors containing either eight 8-bit, four 16-bit, two 32-bit or one 64-bit element/s, Q registers can keep vectors containing sixteen 8-bit, eight 16-bit, four 32-bit or two 64-bit elements. Nonetheless, NEON instructions always operate on 64-bit or 128-bit vectors.

The numerous SIMD extensions ascribed so far achieved innumerable performance enhancements in multiple domains. However, all of them were developed to operate with fixed-size registers. Despite simplifying the implementation and limiting the hardware requirements, since the ideal vector length often depends on the task being attained, this approach could never reap all the possible performance. Additionally, any change to the register length normally entails the use of newer instruction-set extensions, leaving earlier implementations of code outdated. In [32], the authors denoted the referenced problems, exploring dynamically-sized vector operations on SIMD architectures and the potential for variable-length vector registers to overcome the apriori limitations.

2.3 Vector-Length Agnostic SIMD extensions

To overcome the problems mentioned, a different approach of SIMD extensions denominated Vector-Length Agnostic (VLA) SIMD extensions started to appear. VLA SIMD extensions are a set of instructions and data types that allow a processor to perform SIMD operations on vectors of varying lengths. This way, these extensions allow a programmer to write code that can operate on vectors of any length, rather than being limited to a fixed set of vector lengths. This can be useful in situations where the length of the vector is not known at compile time, or when the vector length may vary based on the input data. Furthermore, these extensions grant more flexibility in terms of the architectures that can perform SIMD instructions. For example, HPC processors can make use of large vectors to attain high throughput, while low-power processors can adopt smaller vectors to fulfil power and resource constraints. Moreover, these extensions grant portability, as the same code can be re-utilized in processors with different vectorial architectures without needing to recompile it. Two dominant examples of these type of extensions are the SVE [2] and the RVV [3].

2.3.1 Scalable Vector Extension (SVE)

The Scalable Vector Extension (SVE) [2] is a VLA SIMD extension to the A64 instruction set of the ARMv8-A architecture, first introduced in the ARM Cortex-A73 processor and released in 2017. SVE,

as a VLA SIMD extension, is intended to provide a flexible, scalable solution for vector processing. Therefore, its primary feature is supporting vector lengths from 128 bits to 2048 bits in 128-bit increments. SVE introduces a new set of registers, represented in Figure 2.3. In a nutshell, SVE establishes thirty-two vector registers, named Z0 through Z31, sixteen predicate registers, named P0 through P15, one special-purpose first-fault register and a set of control registers, named ZCR EL1 through ZCR EL3.

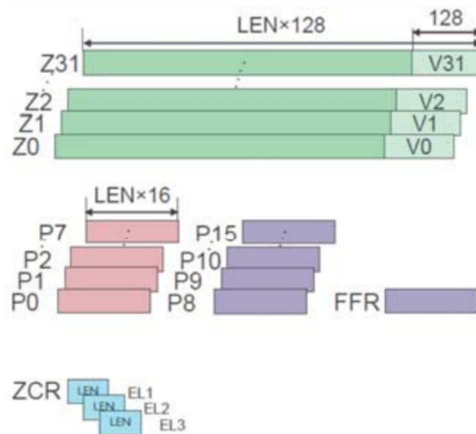


Figure 2.3: SVE registers. Image from [2].

The size of each of the thirty-two Z registers is implementation-dependent (*LEN* denotes the vector length to be supported) within the apriori mentioned range, where the low 128 bits overlap with the already mentioned NEON registers. Therefore, these registers provide scalable containers for 64-bit, 32-bit, 16-bit and 8-bit data elements. The P registers are based on predicated instructions that determine which vector elements to process. To do so, P registers hold one bit for each 8-bit data element in the Z registers, where the value of each individual bit in the P registers represents if the corresponding data element is or is not active. The first-fault register is a dedicated predicate register that captures the cumulative fault status of a sequence of SVE vector load instructions. SVE provides a first-fault option for some SVE vector load instructions. This option suppresses memory access faults if they do not occur as a result of the first active element of the vector. Instead, the first-fault register is updated to indicate which of the active vector elements were not successfully loaded. The control registers allow the virtualization of the effective vector width.

Overall, SVE's design relies heavily on predication, which is used to drive vectorized loop control flow decisions. To do so, SVE counts with a family of *while* instructions that work with scalar count and limit registers to populate a predicate with the loop iteration controls that would have been calculated by the corresponding sequential execution of the loop. To better understand this behaviour, Listing 2.1 denotes an example kernel of the SAXPY algorithm implemented using the SVE extension.

Listing 2.1: SAXPY kernel implementation using ARM SVE extension.

```
1
2  mov x4, #0 ;Initialize iteration counter (i) in r4
3  whilelt p0.d, x4, x3 ;Set predicate p0 for each r4 element that is lower
       than
4
5  ld1rd z0.d, p0/z, [x2] ;Broadcast value A to v0, with predicate zeroing
6  .loop:
7  ld1d z1.d, p0/z, [x8, x4, lsl #2] ;Loads to v1 the values pointed by X[i]
       with a stride of 32-bits elements
8  ld1d z2.d, p0/z, [x9, x4, lsl #2] ;v2:=Y[i]
9  fmla z2.d, p0/m, z1.d, z0.d ;Fused multiply-accumulate Y[i]=A*X[i]+Y[i],
       with predicate merging
10 st1d z2.d, p0, [x1, x4, lsl #2] ;Store v2 to memory pointed by Y[i]
11 incs x4 ;Increment i based on the vector size and SP (VL/32)
12 .latch:
13 whilelt p0.d, r4, r3
14 b.any.loop ;Loop again if all of the predicate elements are 0
```

As mentioned, the main feature of the SVE execution is performed through the "*whilelt*" instruction, which syntax is represented in Figure 2.4.

whilelt "predicate", "start value", "comparison value"

Figure 2.4: *whilelt* instruction syntax.

In simple terms, by executing this instruction, the predicate is filled with ones while the incremented start value is less than the comparison value. Furthermore, the "*incs*" instruction iterates the start value by the vector size, simplifying the memory addressing calculations and removing more unnecessary iteration counters. This process is repeated each iteration of the loop. At last, the branch instruction controls the flow of the loop code, by performing a branch to the *.loop* tag if the iterator variable does not meet the finish condition. The condition codes needed for the branch instruction to work correctly are set by the "*whilelt*" instruction each time it is called.

In [33], SVE performance was compared with the NEON extension, concluding that by using SVE it is achieved a higher vector utilization and speedups of up to 3 times. In [34], the overall number of auto-vectorized loops and the speedups achieved were compared by using NEON and SVE VLA code, concluding that, despite SVE achieving better performance for the majority of cases, the overhead of the predication instructions on SVE resulted in worse performance for some cases.

2.3.2 RISC-V Vector (RVV)

More recently, another VLA SIMD extension was developed by the recognised RISC-V organization. RISC-V Vector (RVV) [3] is a vector processing extension for the RISC-V ISA, initially proposed in 2015 and later refined in 2021 as part of the RISC-V Vector 1.0 specification [3]. As a matter of fact of being a recent extension and RISC-V presenting an open-source nature, more and more RVV is being used for multiple purposes. In [35], a microarchitectural design of a vector unit compilant with the RVV extension is introduced. In [36], the use of RVV is being studied under the topic of communications signal processing. From this point, the focus passes only on describing RVV for the 1.0 specification.

RVV adds thirty-two vector registers, named v0 through v31 and seven unprivileged Control Status Registers (CSRs) (*vstart, vxsat, vxrm, vcsr, vtype, vl, vlenb*), which description is referenced in figure 2.5, to a base scalar RISC-V ISA.

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start position
0x009	URW	vxsat	Fixed-Point Saturate Flag
0x00A	URW	vxrm	Fixed-Point Rounding Mode
0x00F	URW	vcsr	Vector control and status register
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register
0xC22	URO	vlenb	VLEN/8 (vector register length in bytes)

Figure 2.5: vector CSRs. Image from [3].

Different from SVE's predication approach, RVV's approach to attain VLA is rooted in the dynamic configuration of a specific vector length (*VLEN*). *VLEN* is constrained by the constant *ELEN*. While *ELEN* denotes the maximum size in bits of a vector element that any operation can produce or consume (must be ≥ 8 and a power of 2), *VLEN* represents the number of bits in a single vector register (must be $\geq$ *ELEN*, a power of 2 and no greater than 2^{16}). RVV's instructions encoding is monomorphic, so there are specific instructions for each different data type.

Furthermore, RVV uses mask bits to support instruction predication. There is one mask bit per element in a vector, which, depending on its value, determines if the instruction being executed is/is not performed in the correspondent element. Different from SVE which had specific registers to hold this information (P registers), every of the RVV's thirty-two registers can work as mask registers, in which the mask bits are sequentially packed one bit after each other, starting from the least significant bit of the vector register file.

To better understand this whole RVV extension paradigm, Listing 2.2 denotes an example kernel of the SAXPY algorithm implemented using the RVV extension.

Listing 2.2: SAXPY kernel implementation using RISC-V V extension.

```

saxpy:
vsetvli a4, a0, e32, m8 ; Ask for n elements of size 32b and group 8 vector
    registers
vlw.v v0, (a1) ; Load data from X to v{0-7}
sub a0, a0, a4 ; Decrement n with the number of processed elements
slli a4, a4, 2
add a1, a1, a4
vlw.v v8, (a2) ; Load data from Y to v{8-15}
vfmacc.vf v8, fa0, v0 ; Floating-point multiply and acumulate
vsw.v v8, (a2) ; Store data
add a2, a2, a4
bnez a0, saxpy
ret
```

As mentioned, the RVV extension is dependent on the dynamic configuration of a specific vector length. This configuration is executed through the use of the "*vsetvli*" instruction, which syntax is represented in Figure 2.6.

vsetvli "configured size", "size", #element size, [#grouping factor]

Figure 2.6: "*vsetvli*" instruction syntax.

The configuration starts with requesting a vector size (in elements) and an element size. Then, the total requested size (element size * vector size) is compared with the implemented vector length, the minimum of both is used to configure all the vectors and is written to the destination operand. Moreover, the operand "*grouping factor*" can be used to group a series of consecutive registers, being particularly useful when the total requested size is larger than the implemented vector size. In detail, a grouping factor of 4 will group registers 0 to 3 into register 0 and so on for the remaining vector registers.

Moreover, regarding the SAXPY RVV example, by executing the "*vsetvli*" instruction, the information of the total remaining elements of the loop is predisposed. Then the actual number of processable elements in the iteration is subtracted to the total remaining elements, ending the loop when there are no remaining elements.

2.3.3 SVE and RVV wind-up

Despite the crucial improvements delivered by the referenced VLA SIMD extensions, these extensions also come with issues that limit their performance and range of uses. in [4], the authors were able

to identify these extensions bottlenecks. While in SVE, its large instruction overhead comes from the dependence on predication, in RVV this overhead mainly falls on the vector control instructions needed. Figure 2.7 illustrates the referenced overhead instructions on the saxpy kernel implementations on SVE and RVV.

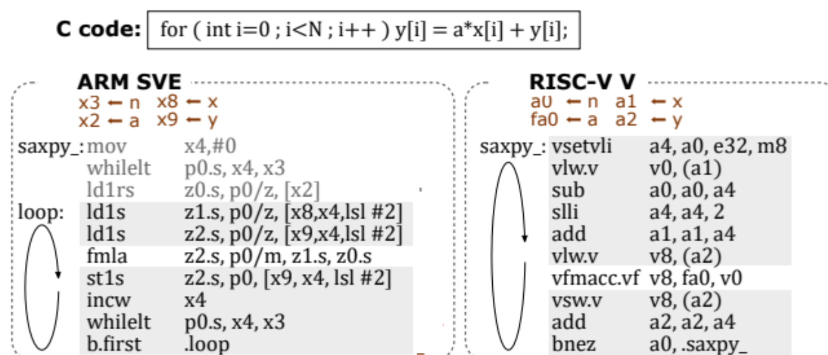


Figure 2.7: Saxpy kernel implementation on ARM SVE and RISC-V V. Image from [4].

Overall, both SVE and RVV can impose a significant amount of instruction overhead (shaded instructions in both figures), which is caused by memory indexing, loop control, and even memory access, none of which directly increase the throughput of data processing. The majority of the loop code frequently consists of these overhead instructions, wasting CPU resources and adversely affecting performance.

2.4 Data-flow and Stream-based approaches

From a different point of view, more recently, in general-purpose processors, the speed at which a processor can process data is outscaling the speed at which data can be accessed from memory. The existent gap, known as the "memory wall" [37], is a major bottleneck in the performance of multiple systems. For example, the performance of data-parallel applications is no exception to the rule and is often constrained by this memory hierarchy problem.

Coupled with the emergence of domain-specific architectures, data-flow and stream-based approaches have been gaining ground. Early concepts of these approaches [15, 16] were often based on accurate representations of data access patterns to substantially accelerate data acquisition. More recently, in [17], the authors proposed a stream-dataflow model and architecture, mainly providing abstractions that balanced the tradeoffs of vector and spatial architectures. These demonstrate it is possible to attain the specialization capabilities of both on an important class of data-processing workloads. In [38], the authors proposed a domain-specific ISA for Neural Networks, called Cambricon, based on a load-store architecture that integrates scalar, vector, matrix, data transfer and control instructions. Overall,

these new approaches lead the way for programmers to investigate various complementary features to boost throughput, such as memory access decoupling and specialization, data prefetching, and efficient parallel computation. Owing to these features, data streaming has lately been adopted beyond domain-specific computing into general-purpose processors to overcome the Von Neumann architecture's limitations.

In [5], the authors realized a vast and relevant study to find and tune the best configuration to specialize memory primitives. The first conclusion was that the best way of exposing rich semantic information about memory operations at fine grain at the ISA level was done by using streams as the structure for memory accesses. Streams denote repeated patterns of memory access, occurring due to loops and nested loops. Thus, given the premise of specialization for streams, the authors enumerated the ways of using such feature.

Stream-based prefetching: Subject to knowing the access patterns and the relationship to the core's control flow, an effective stream-based programmable prefetcher can deep prefetch for regular and irregular memory accesses, allowing stream requests to be decoupled from the core's instruction window. Decoupling has the main advantage of minimising the latency of memory accesses.

Stream-decoupling: The main idea of stream decoupling is to create a direct and fast interface between the data that is prefetched, and the core instructions. This interface can be obtained using a specialized memory access engine, which could generate stream requests, and pseudo-registers, which could keep the stream data. The principal advantages of stream-decoupling are the removal of address generation instructions from the general core pipeline and the capability of vectorization of memory on traditionally non-vectorizable code.

Cache-Awareness: Since streams are precise definitions of an access pattern, cache policies could take advantage of that knowledge. A possible stream engine, which has information about the streams and the access pattern, could make requests to the cache in a way that is aware of stream behaviours. Furthermore, cache could bypass streams based on the expected footprint of the stream, avoiding cache pollution.

Figure 2.8 denotes a stream paradigm developed by the authors making use of the referenced designs in comparison to a conventional out-of-order paradigm.

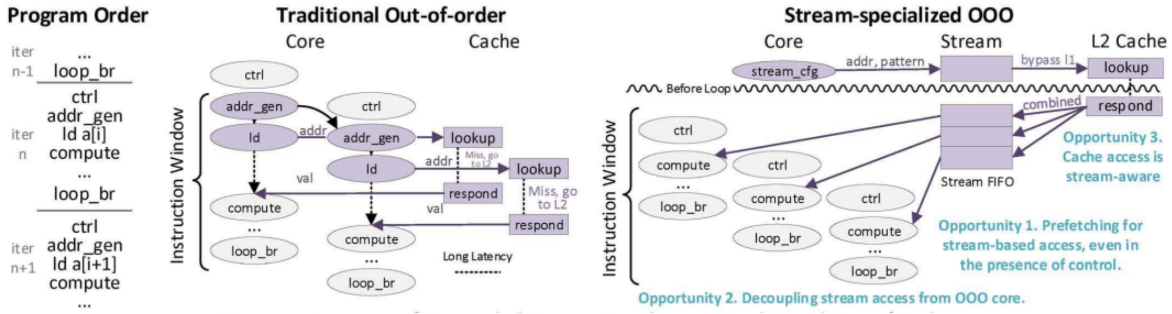


Figure 2.8: Overview of Decouplcd-Stream Paradigm vs Conventional Out-of-Order. Image from [5].

Moreover, complementary to its design, the authors described a new decoupled-stream ISA with Stream-ISA extensions.

In [19] a new stream-based extension to boost utilization and increase energy efficiency was also proposed. The proposed Stream Semantic Register (SSR) is a lightweight, non-invasive RISC-V ISA extension, which implicitly encodes memory accesses as register read/writes, eliminating a large number of loads/stores. SSR extension is possible based on the evolution of the previously mentioned stream approach. This evolution is grounded by the full extension of stream specializations to the processor pipeline, configuring all memory access patterns in the loop preamble and automatically streaming data directly to the processor's registers.

2.4.1 Unlimited Vector Extension (UVE)

Up to this point, VLA SIMD and stream-based extensions were tackled apart from each other, both proving to enhance the performance of general-purpose processors. In this circumstance, Unlimited Vector Extension (UVE) [4] combines VLA processing with data streaming in RISC-V-based modern general-purpose processors.

In addition, UVE counts with a streaming interface that enables an effective prefetch of data, while facilitating the vectorization by linearizing non-coalesced memory accesses. UVE associates each data stream to a vector register, empowering instructions to directly use the corresponding stream. By doing so, the progression/iteration of streams is always guaranteed, happening after each interaction with the vector. Therefore, loop control is performed with a reduced and basic set of stream-conditional branches. Thus, UVE stream model is described by using a hierarchical descriptor-based representation.

UVE supports all common data types, from byte to double-word and the standard set of RISC-V operations. UVE includes 32 vector registers, named through u0 to u31, each one with a length that only has a minimum value defined corresponding to the width of the supported data types: byte (8-bits), half-word (16-bits), word (32-bits), and double-word (64-bits). To allow VLA processing, UVE counts with 32 predicate registers, named p0 through p31, which similarly to the ones presented in SVE allow

per-lane execution control. As such, boundary conditions of vector processing are automatically solved by disabling the out-of-bounds elements.

The presented characteristics allow UVE to significantly mitigate the instruction overhead presented in the SVE and RVV extensions. Figure 2.9 represents the saxpy kernel implementation on UVE. In comparison to the saxpy kernel implementations on SVE and RVV presented in Figure 2.7, the reduction of instructions overhead with UVE is evident, needing only a single branch instruction for control.

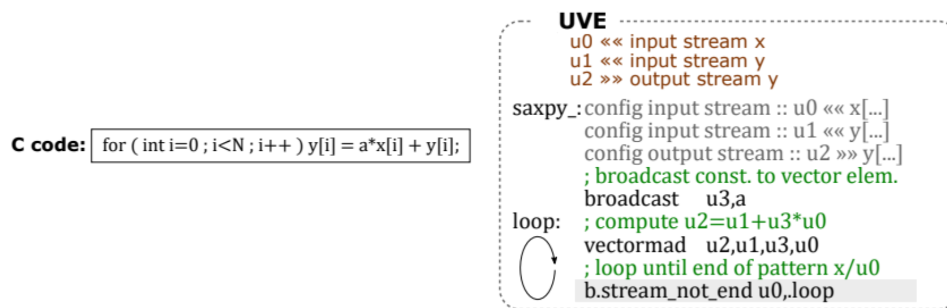


Figure 2.9: Saxpy kernel implementation on UVE. Image from [4].

In terms of microarchitecture, UVE principally adds a dedicated Streaming Engine, which is responsible for managing the state of the streams and issuing memory requests to the processor's pipeline. In [4] it was concluded that for vectorized benchmarks the UVE extension provides an average performance advantage of $2.4\times$ over SVE. The major features that led to a better performance from UVE were the significant code reductions and the streaming infrastructure, which successfully reduced the load-to-use latency and increased the effective memory hierarchy utilization.

2.4.2 Compilation Support

As previously mentioned, a new age of computing acceleration was achieved by the recent return of data streaming and data-flow paradigms in general-purpose processors. However, as a consequence of being recent approaches, their execution model is typically incompatible with the internal Static Single Assignment (SSA) form employed by contemporary compilers, turning the propagation of steam paradigms constrained by the absence of compilation support.

To get over this restriction, [6] suggested a new alternative compilation flow that is fully implemented as an LLVM Intermediate Representation (IR) analysis and transformation pass. In short, the compilation flow works by first making use of the LLVM IR to examine data-flow graphs and memory access patterns in loops. Following, using the gathered information, a high-level data streaming representation is created. At last, the respective created representation can be directly compiled for stream-based vector extensions and linked to conventional machine code. Figure 2.10 represents this compilation flow, in

specific to compile UVE code.

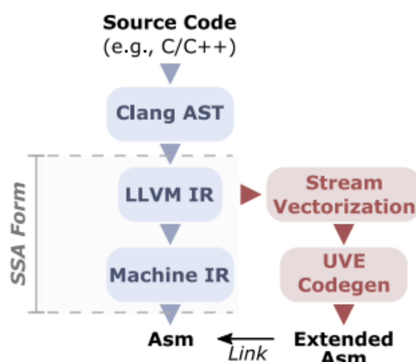


Figure 2.10: Compilation flow for stream-based vector extensions. Image from [6].

2.5 Simulation tools

In this Section, the going-through that culminated in the choice of the Spike [20] simulator as the principal and required tool for the development of this Thesis work will be explained. It will tackle Instruction Set Simulators (ISSs), architectural simulators and a brief overview of some of the most salient tools and frameworks that can be used for RISC-V simulation.

First, it is important to separate ISSs from architectural simulators. While ISSs are focused on emulating the execution of individual instructions for software development, architectural simulators provide a broader view of the entire computer system and are used for architectural exploration and performance evaluation during hardware design. One of the principal milestones of this Thesis is to propose a new ISA extension. However, there is no available hardware support. As such, the only option is to rely on validation with ISSs. RISC-V counts with a worldwide thriving community that has gone through strong efforts to create top-notch simulation tools. As a result, since the initial versions of the RISC-V ISA were released, numerous simulators have been released. These simulators employ many methodologies and tools to fulfil a wide range of different purposes. Some of these are high performance, transparent processor visualisation and architectural exploration.

Gem5 [39], despite being developed by ARM and not by the RISC-V community, is a well-known standard open-source tool for execution-driven simulation in academic matters. Although it has excellent modelling capabilities, it still lacks simulation throughput, adding to the fact that it is a tool hard to extend. Imperas offers riscvOVPsim [40] which despite being a simulator that allows the development and debugging of code for the target RISC-V processor on an x86 host computer, is not yet fully open source. Coyote [41] is a recent open-source, execution-driven simulation tool, based on the canonical RISC-V ISA, that has a focus on the movement of data throughout the memory hierarchy. QEMU [42]

from RISC-V international, is a generic and open source machine emulator and virtualizer, which offers an excellent simulation throughput. RV8 [43] is a high-speed simulator that primarily focuses on CPU simulations and uses just-in-time compilation techniques to enhance execution performance. R2VM [44] is also a simulator that targets CPU simulations by making use of binary translation techniques. It can vary between fast and accurate simulations to handle a variety of use scenarios. Renode [45] is a simulation system that supports multi-node networks of embedded systems in a distributed simulation. RISC-V VP [46], RISC-V-TLM [47], HIFIVE1-VP [48] are examples of recent RISC-V ISSs based in SystemC TLM.

Despite the widespread of RISC-V simulation environments, the Spike simulator still holds the title of being the golden reference functional RISC-V ISA software simulator. Adding to that, Spike is a decent choice for CPU simulations, providing a significant simulation throughput. Furthermore, Spike is a completely open-source and extendable simulator that provides support for the standard RISC-V ISA and a vast collection of extensions. More relevant to the proposed work, Spike is one of the few simulators that already provide support for the RVV extension. The combining of these advantages led to Spike being the principal key to validate this Thesis work.

2.6 Spike Simulator

As mentioned, part of this Thesis work relies on the usage of the Spike Simulator tool [20]. Due to the lack of documentation related to Spike, a vast study was made to acknowledge the whole behaviour of this simulator. As so, in this Section, the whole paradigm of the Spike Simulator will be described, beginning with a brief overview of the tool, together with a summary of the various modules present in its constitution.

2.6.1 Overview

Spike [20], the golden reference RISC-V ISA Simulator is an open-source simulator whose principal feature is to implement a functional model of one or more RISC-V processor cores. Spike in addition to using a combination of C++ language features and design patterns to implement its functionality, also uses object-oriented programming techniques to represent the various components of a simulated processor, such as the registers, memory, and pipeline. Spike provides support for all standard base integer and floating point RISC-V instructions as well as a vast number of extensions.

Spike presents a modular design which eases its modification, its extendability, and the capability of performing ISA testing. Moreover, Spike provides a rich interactive debug mode, allowing one to visualize run-time contents of integers and floating point registers, contents of a physical/virtual address memory position and slide-through instructions. Furthermore, Spike can also be used with other tools

in the RISC-V toolchain, such as the GCC compiler, the GDB debugger and the RISC-V Proxy Kernel. Spike is also widely used by developers working with RISC-V processors.

2.6.2 Main Functional Modules

Figure 2.11 portrays a straightforward view of the Spike Simulator, depicting its most relevant simulated components.

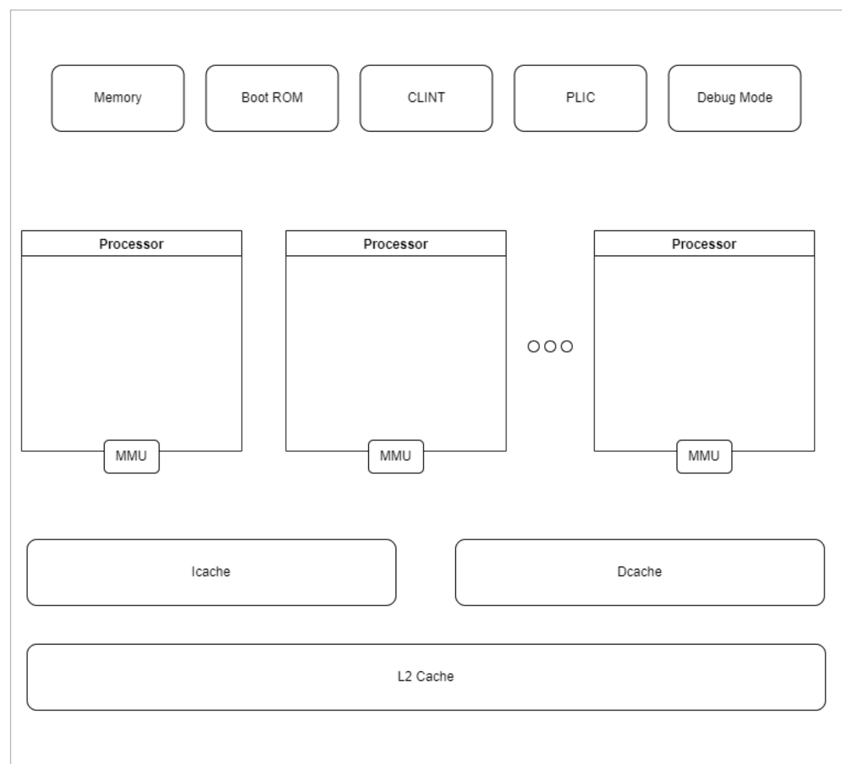


Figure 2.11: Spike main modules Overview.

For the purpose of the following description, every directory mentioned has per base the *riscv-isa-sim* root directory.

The processor-dependent code is located in the *riscv/processor.h* and *riscv/processor.cc* directories. The whole processor simulation is realized under the *processor_t* class, in which the connections with the other simulated components are also established. Inside the *processor_t* class, the *state_t* structure denotes the whole state of the processor cores, defining all the registers (*reg_t* structures). Moreover, the definition of the instructions is located in the *riscv/insns* folder. Related to the instructions, all their *MATCH* and *MASK* attributions are under the *riscv/encoding.h* directory. In addition, Spike holds a sizeable amount of Macros related to the execution of the instructions under the *riscv/decode.h* and *riscv/decode_macros.h* directories.

Furthermore, the memory-management code is under the *riscv/mmu.h* and *riscv/mmu.cc* directories. The whole Memory Management Unit mechanism simulation is realized under the *mmu_t* class. Finally, other important components, such as the Data Bus (*bus_t* class), the Read Only Memory (*rom_device_t* class), the Main Memory (*mem_t* class) and the Interrupt Controller (*clint_t* class) code is under the *riscv/devices.h* and *riscv/devices.cc* directories.

Apart from the main simulated components, Spike also owns a vast collection of modules and instruction Macros connected to the whole set of instruction extensions that it supports. In particular, it already provides support for the RVV extension. This support is obtained through the code located in the *riscv/vector_unit.h* and *riscv/vector_unit.cc* directories, which mainly simulate all modules present in the vector unit needed to execute RVV instructions. Figure 2.12 denotes the addition of the vector unit in the constitution of Spike's main functional modules. Furthermore, Spike keeps all the Macros related to the execution of RVV instructions under the *riscv/v_ext_macros.h* directory.

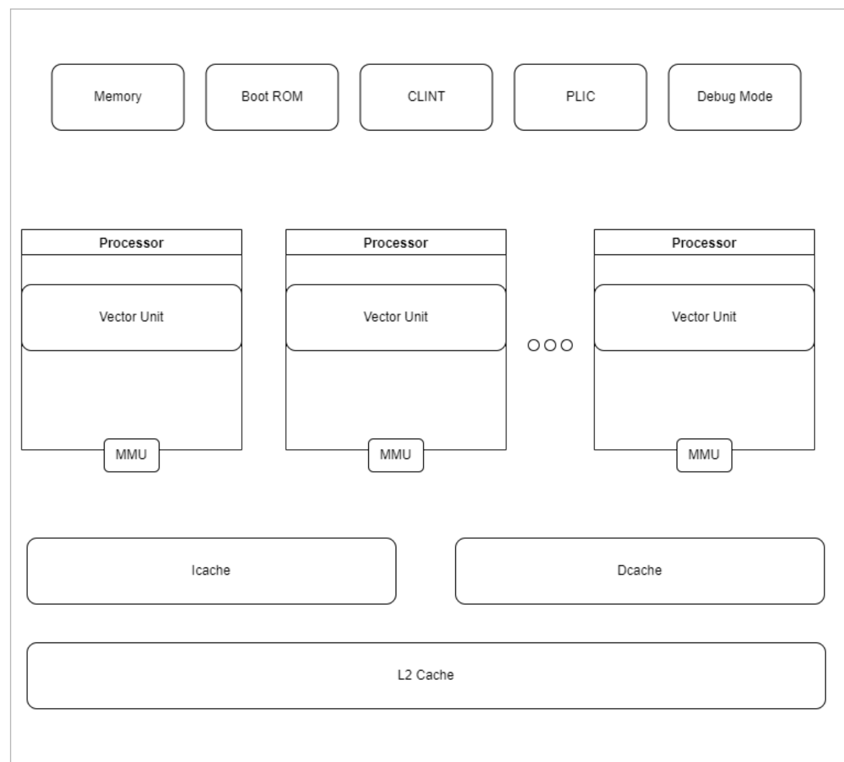


Figure 2.12: Spike Overview with the addition of the vector unit.

2.7 Discussion

To tackle the performance limitation and portability issues caused by regular SIMD extensions that make use of apriori fixed-size registers, VLA SIMD extensions like the mentioned RVV began to rise. However,

such extensions present major bottlenecks, mostly because of the added instruction overhead needed to attain the runtime variation of the vector lengths. Besides, stream-based extensions started to appear to mitigate the existing gap between the processor computation speed and the data movement latency.

Based on that, this work aims to introduce a RISC-V stream extension fully compatible with RVV. This extension has the vision of keeping the already performance enhancement delivered by the RVV extension, adding a way to mitigate the RVV's instruction overhead and decrease the memory access latency with the help of stream-based approaches. Since UVE is a successful stream-based SIMD RISC-V extension that tackled this same goal, the proposed solution follows some of the main ideas present in the UVE model.

2.8 Summary

This Chapter aimed to discuss general concepts and background related to the proposed work. The denoted concepts needed to be apriori-defined to understand the work that will be presented next. In summary, first, the whole RISC-V paradigm was tackled. Furthermore, the evolution of the development of SIMD extensions was discussed. Moreover, the concept of VLA SIMD extensions was presented, referencing some of the top-notch examples. Next, the emergence and evolution of stream-based and data-flow approaches into general-purpose processors were presented. The novel UVE extension, which combines both SIMD and streaming approaches, was examined. Finally, the main RISC-V simulation tools were discussed, culminating in the analysis of the Spike functional simulator.

3

RISC-V stream-based extension

Contents

3.1 Overview 28

3.2 Memory Access Pattern Description 28

3.3 Architectural State 34

3.4 Instructions Design 36

3.5 Instruction Set Overview 42

3.6 Summary 44

3.1 Overview

In this Chapter it is presented the proposed RISC-V stream-based extension. To do so, first, it is described the underlying memory access descriptor representation. Following, it is discussed the architectural state by defining the necessary architecture features and mechanisms to support the proposed extension. Finally, it is presented the set of proposed instructions for stream configuration, manipulation and stream-based control flow.

3.2 Memory Access Pattern Description

In the context of this Thesis, a **stream** is defined as a continuous flow of data that often exhibits a predictable or constant pattern as soon as it starts executing. Because of this, even though the pattern, length, or data type may not be known at compile time, they must be defined through a collection of variables whose values may be determined when the stream is generated. Furthermore, sequential and orderly processing (loading and storing) of the streams must be guaranteed.

To represent a stream of data, the concept of a descriptor is herein used. An access pattern to memory must be described by a descriptor, which in most cases specifies a starting address, word length (in bits), and size. To express more complex patterns, such as multidimensional strides, extra fields might be added. Descriptors may also be made dependent or hierarchical to reflect advanced memory access patterns (such as hexagonal and diagonal patterns). However, there is a natural equilibrium between a descriptor's complexity and its ability to simply represent various patterns.

3.2.1 Multidimensional Access Encoding

Based on what was stated, in the extension being defined, a similar solution to the one successfully demonstrated at the Unlimited Vector Extension (UVE) extension [4] is proposed. The authors stated that by using a descriptor composed of a three-element tuple {offset, size, stride}, all the straightforward single-dimension patterns can be described. Furthermore, multidimensional access patterns can be described by chaining these descriptors. The formal function for calculating each memory address is defined as seen in the following equation:

$$y(X) = \text{offset} + x_0 + \sum_{k=n}^{n} x_k \times \text{stride}_k, \quad x_k \in \{0, \ldots, \text{size}_k\} \tag{3.1}$$

To describe straightforward patterns, a single three-element tuple is needed, constituted namely by an offset, a size and a stride. In a single dimension, the offset represents the starting memory location of the pattern that is being described. However, for multidimensional descriptions, the starting memory

location is dependent on the offsets and strides of every dimension. The size indicates how many components to produce in a particular dimension. The stride denotes the location of the following element inside the same dimension. These parameters are what makes a descriptor, the minimal representation for a stream. Based on that, to represent a simple stream, a descriptor constituted with an **offset**, a **size** and a **stride** is defined. This introduced descriptor is called **dimension**. Moreover, since the data type of the stream only affects the stream's calculated addresses and not the pattern itself, the element size of the stream is included in the instruction names and not in the instruction arguments. This means that the parameter data type is also present in the first dimension of any description.

3.2.2 Imperfect Loop Access Encoding

Although it is possible to describe a decent amount of memory access patterns relying only on the previously introduced descriptor, a vast amount of examples are still not possible to be represented. Listing 3.1 denotes the C code of one memory access pattern that is impossible to be represented by simply using the dimension descriptor explained before.

Listing 3.1: Lower Triangular Memory Acces Pattern C code

```
1 int i = 0, j = 0, k = 0;
2
3 for (; i < Nr; i++) {
4     k++;
5     for (j = 0; j < k; j++) {
6         A[i*Nc + j];
7     }
8 }
```

By analyzing the Listing 3.1 C code, it is observed that for each outer loop iteration, the inner loop parameters are modified. To provide the described functionality, a new descriptor type called **modifier** is introduced. This new descriptor has the ability to change a dimension descriptor's parameter. This modifier is represented by a tuple comprised of four parameters. The first parameter is the **target**, which serves as an identifier of the descriptor parameter to edit (offset, size or stride). The second parameter is the **behaviour**, which specifies the operation to be carried out (increment or decrement). The third is the **displacement**, which is the constant value utilised by the operation to update the parameter value. The final value of the tuple is the **size** of the modifier and it specifies the total number of iterations for which the modification should occur.

3.2.3 Indirect Memory Access Encoding

In some applications, it is common to find the use of indirect memory accesses. Listing 3.2 denotes the C code of one example of an indirect memory access pattern. The denoted memory access pattern cannot be represented by the use of the previously introduced descriptors.

Listing 3.2: Indirect Memory Acces Pattern C code

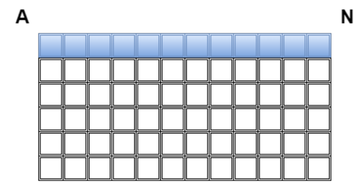
```
int i = 0;

for (; i < Nc; i++) {
    B[A[i]];
}
```

Despite by coupling both memory accesses together, it is simple to describe the indirect pattern (the access $B[A[i]]$ can be described as $*(B + A[i])$), the description needs to use the values of $A[i]$ to modify the parameter offset of the pattern B dynamically. To be able to represent such a pattern, it can be re-utilised the idea of the already explained modifier with the slight chance that the source value (displacement) needs now to be dynamic. In addition, the incrementation and decrementation behaviours would restrict the description's potential given that the displacement value is now taken via a stream. Therefore, a modified descriptor called **dynamic modifier** is introduced. This modifier is still represented by a tuple of four parameters. Two of them, the **target** and the **size**, still hold the same behaviour as in the already explained modifier descriptor. The last two parameters are the **behaviour** and the **source**. The behaviour still has the same functionality with the addition that it has a total of five possible etiquettes: Add, Sub, Inc, Dec and Set. The source acts like the displacement but in this case, is dynamic.

Linear Patterns Description Examples

Listing 3.3 denotes a C code of a simple array access pattern and figure 3.1 denotes the correspondent stream representation. To represent this simple array access, it is only needed the starting memory location of the pattern (offset set to the base address of the A array), the number of elements to generate (size of N) and how to proceed to go to the next element within the same dimension (stride of 1).



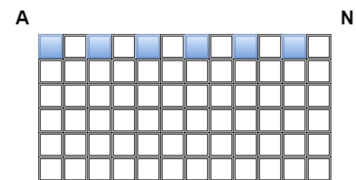
Stream Representation
Dimension 0:
offset = A
stride = 1
size = N

Figure 3.1: Stream Representation

Listing 3.3: Simple Array access C Code

```
1 float A[N * N];
2 for (int i = 0; i < N; i++) {
3     A[i];
4 }
```

Listing 3.4 denote another scenario of a simple array access pattern and the correspondent stream representation in Figure 3.2. In this case, the starting memory location remains the same, the number of elements to generate is reduced by half and some of the elements present in the A array are meant to be bypassed. For that, in the stream representation, the size (N/2) and the stride (2) values were changed.



Stream Representation
Dimension 0:
offset = A
size = N/2
stride = 2

Figure 3.2: Stream Representation with Stride of 2

Listing 3.4: Simple Array access with Stride C Code

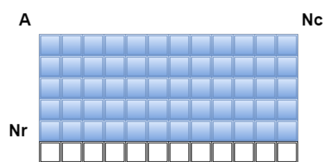
```
1 float A[N * N];
2 for (int i = 0; i < N/2; i+=2) {
3     A[i];
4 }
```

By the apriori examples, it is clear that by using the **dimension** descriptor, such simple single-dimension patterns are represented with ease. However, there are obviously memory access patterns that are multidimensional such as 2-D or 3-D patterns. Representing such patterns is acquired by adding more dimensions to the stream representation. In detail, each additional dimension can be configured by adding one descriptor on top, composed of the three aforementioned regular parameters (offset, size and stride). Listing 3.5 denotes a C code of a 2-dimensional Array access pattern and figure 3.3 denotes

the correspondent stream representation.

Listing 3.5: 2-Dimensional Array access C code

```
1  float A[Nr][NC];
2
3  for (int j = 0; j < Nr; j++) {
4      for (int i = 0; i < Nc; i++) {
5          A[j][i];
6      }
7  }
```



Stream Representation

Dimension 0:	Dimension 1:
offset = A	offset = 0
size = Nc	size = Nr
stride = 1	stride = Nc

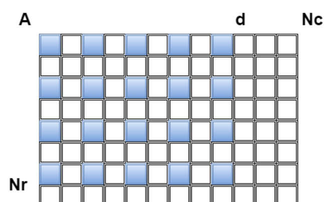
Figure 3.3: 2-Dimensional Stream Representation

In this case, the starting memory location of the second dimension is dependent on the base address of the previous dimension. As so, the offset of the second dimension is set to 0 because the access is well-bounded within the array's limits. If this value was set to 1, all horizontal accesses (based on figure 3.3) would be offset by one position to the right. The size of the second dimension corresponds to the elements intended to be processed in this transformation. In the denoted example, this translates to value Nr as this is the total iterations the outermost loop performs. The stride of the second dimension is the distance between two sequential elements in the same dimension. It is observed that the distance between the elements, in the outermost dimension, corresponds to the full width of the array or, in our example, value Nc.

Listing 3.6 denotes another scenario of 2-dimensional array access and the correspondent stream representation in figure 3.4. It is observed that by messing with the stride value of the descriptor, 2-dimensional streams can also represent more twisted patterns.

Listing 3.6: 2-Dimensional Strided Array access C code

```
1  float A[Nr][NC];
2
3  for (int j = 0; j < Nr; j+=2) {
4      for (int i = 0; i < d; i+=2) {
5          A[j][i];
6      }
7  }
```



Stream Representation

Dimension 0:	Dimension 1:
offset = A	offset = 0
size = d/2	size = Nr/2
stride = 2	stride = 2*Nc

Figure 3.4: 2-Dimensional Strided Stream Representation

In summary, to achieve more intricate memory patterns, it is possible to cascade up **dimension** descriptors. In the proposed solution, the maximum number of dimensions that can be described is 8.

Complex Patterns Description Examples

Listing 3.1, previously referenced, denotes the complex lower triangular memory access pattern. This memory access pattern can only be described by the use of the **modifier** descriptor. Relying on this new descriptor type the example denoted in figure 3.5 is now possible to be represented. Figure 3.6 denotes the full representation of the Lower Triangular memory access pattern, making use of the dimension and the modifier descriptor types.

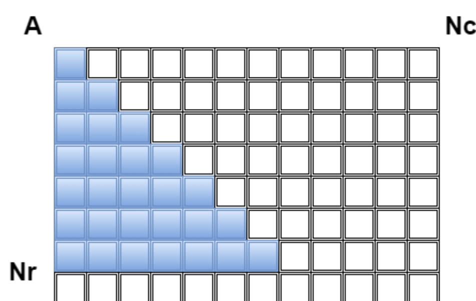


Figure 3.5: Lower Triangular Memory Access Pattern

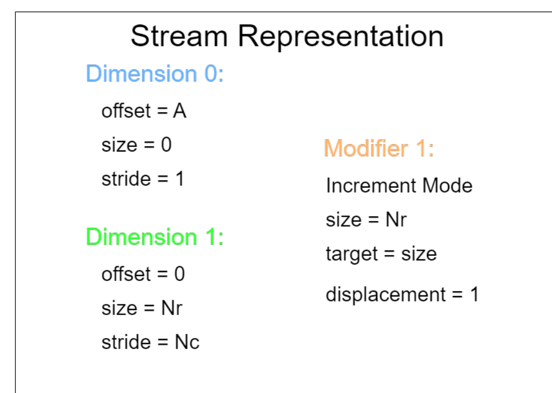


Figure 3.6: Lower Triangular Memory Access Pattern Representation.

In detail, in this case, the modifier for any outer loop iteration will modify the size parameter of the inner loop. Since the behaviour is in increment mode and the displacement is one, in each iteration of it (In this case the modifier will act a total of Nr times), the size of the first dimension will be incremented by one. The modifier will be applied every time dimension 1 (outer) is iterated, and the first time it is applied right before the inner dimension iteration. This is in agreement with a for-loop mechanism. Take note that there is a first configuration pass on the outer dimensions before the start of the inner dimension. Each modifier is tied to a single dimension, so it is iterated along with that dimension. The modifier will, however, change the dimension that is immediately below it.

In the solution proposed, there can only be one modifier associated with each dimension, so the maximum-sized description can have 8 dimensions and 7 modifiers. Due to the lack of a lower dimension, the first dimension cannot have a modifier.

Listing 3.2, previously referenced, denotes an indirect memory access pattern. This memory access pattern can only be described by the use of the **dynamic modifier** descriptor. Relying on this new

descriptor type the example denoted in figure 3.7 is now possible to be represented. Figure 3.8 denotes the full representation of the Indirect memory access pattern, making use of the dynamic modifier descriptor.

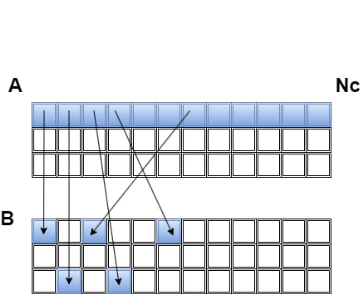


Figure 3.7: Indirect Memory Access Pattern

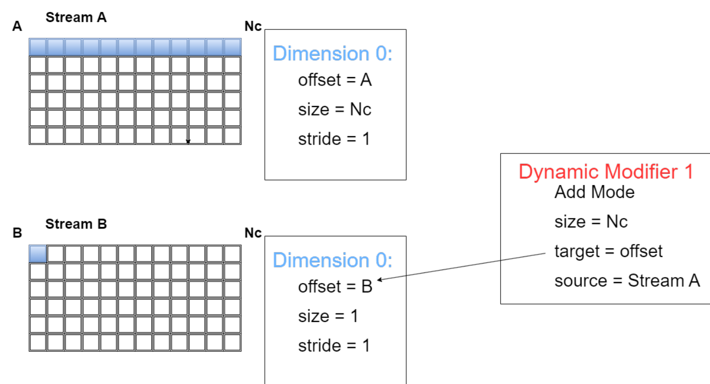


Figure 3.8: Indirect Memory Access Pattern Representation.

3.3 Architectural State

The process of designing the Instruction Set Architecture (ISA) extension is the most crucial part of the work, as the supporting microarchitectures will be affected by any mistake in the ISA definition. Therefore, before going to the actual definition of the proposed extension, it is important to understand a brief overview of the architectures that will support this extension. As so, this Section tackles the architectural components that will support the behaviour specified by the stream-based extension.

3.3.1 Microarchitecture Overview

Streaming is a complex process that traditional Central Processing Unit (CPU) pipelines are not ready to support. Consequently, the CPU pipeline needs to be embedded with a streaming mechanism that handles all actions denoted by the stream paradigm. Figure 3.9 denotes a Microarchitecture overview of a CPU pipeline extended with a *Stream Unit*.

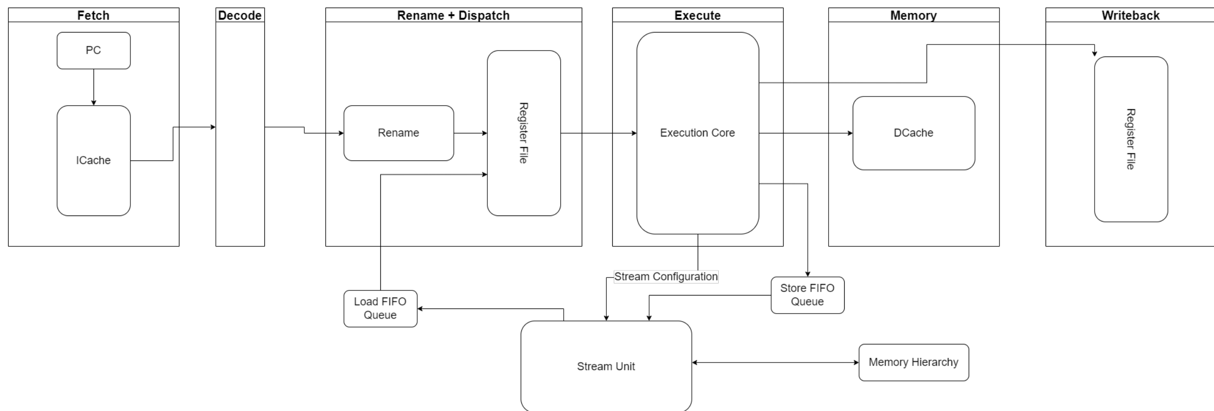


Figure 3.9: Microarchitecture Overview with a Stream Unit

To understand the behaviour of the *Stream Unit* denoted in Figure 3.9, the main actions in the operation of streams will be, in resume, described:

Stream Configuration: Whenever a new stream configuration instruction reaches rename, it is registered on a stream configuration reordering structure, which is part of the *Stream Unit*. This engine, similar to a re-order buffer, processes the configuration instructions in order, and as soon as the corresponding operands are available. Upon completion of the configuration, the *Stream Unit* immediately starts processing the stream, either by pre-loading data (for input streams), or by calculating store addresses (on output streams), and waiting for the commit of store data.

Stream Iteration: A stream iteration process is logically performed by reading/writing from/to an input/output stream. This is performed at rename, by signalling the *Stream Unit* to iterate the speculative stream state. For an output stream, this also implies reserving space in the Stream Store First-In First-Out (FIFO) buffers and then waiting for both data and commit signals to arrive, to complete the operation. On input streams, the devised solution attempts to minimize the load-to-use latency, by allocating the head of input streams to physical registers. As a consequence, when a stream-consuming instruction reaches rename, the operand is immediately read and a new data element is pre-loaded to a different physical register.

Stream Termination: The termination of a Stream is achieved at commit, either through an explicit termination instruction or by committing an instruction that signals the completion of the streaming pattern. When such an event occurs, all the structures in the *Stream Unit* are released, allowing the resources to be allocated to a new stream configuration.

Memory Coherence: On the core, stream and non-stream operations are kept coherent by matching the stream load/store state with the core load/store queues and by solving possible stream load/store dependencies through typical request delay, replay, or squash mechanisms. Hence, data written by the conventional pipeline can be immediately read by a newly configured input stream, and data written by an output stream can be loaded using a conventional load instruction. This ensures a reliable transition

between sequential code and stream loops.

3.3.2 Available registers

The stream-based extension being defined is integrated with the rest of the subsets of the RISC-V ISA. As so, in the set of instructions that will be presented, there are no arithmetic or data-transfer instructions. The only instructions are associated with stream configuration, manipulation and control flow. The vision of this extension is to cooperate the configured streams with the arithmetic and data transfer instructions present in the RISC-V subsets. Therefore, a set of new registers is not needed. The only addition needed is to integrate the rest of the RISC-V registers with mechanisms to identify streams.

When cooperating with the Standard RISC-V ISA, the proposed extension will make use of the scalar and floating point registers defined. As so, these registers will be integrated with mechanisms to identify whether a load/store stream is associated with them.

When cooperating with the RISC-V Vector (RVV) extension, the proposed extension will make use of the RVV vector registers. As so, these registers will also be integrated with mechanisms to identify whether a load/store stream is associated with them. Furthermore, in the specific case of the vector registers, these will also be integrated with mechanisms that will identify stream loop tails.

3.4 Instructions Design

This Section will describe in detail all the instructions present in the stream-based extension being defined. It is important to note that despite this extension being focused on the streaming paradigm, it also contains other mechanisms that were adopted to make it an updated and reliable extension. One example is the possible integration with Single-Instruction-Multiple-Data (SIMD) behaviour, so deeply exploited in Chapter 2. Furthermore, before going to the definition of the actual instructions, it is important to reference that since this Thesis goal is to create an extension based on RISC-V, the names of the instructions pretend to follow this same label. Therefore, inspired by the instructions present on the RVV extension, since this extension is mainly based on streaming mechanisms, all the instructions present on this extension have as a prefix the letter ”s” (Stream). Following the concept present in the RISC-V instructions, the rest of the instruction’s name reflects the nature of the instruction.

3.4.1 Stream Configuration Instructions

In this subsection, the subset of stream configuration instructions is examined. After defining the memory pattern descriptors in Section 3.2, it is now important to create the instructions that can efficiently yet

simply represent those patterns. To create the instructions for stream configuration, this subsection makes reference to the previously established pattern descriptors.

First, let's start by describing single-dimension streams. Remembering image 3.1, to describe a simple pattern with one dimension, only a dimension descriptor denoting the offset, the size and the stride is needed. Therefore, to describe a one-dimension stream, the related configuration instruction must provide the offset, the size and the stride parameters. Additionally, this configuration instruction must also set the data width. Furthermore, the instruction also needs to set the transaction direction of the stream, differentiating whether it is a continuous flow of data meant to be loaded from memory or a continuous flow of data meant to be stored in memory. Hence, the proposed stream configuration instruction has the following format:

```
scrt.<dir>.<width>    Vd, Rs1, Rs2, Rs3
```

The mnemonic name "scrt" references "Stream Create". The "dir" parameter sets the transaction direction of the stream, where "ld" is for a continuous flow of data meant to be loaded from memory and "st" is for a continuous flow of data meant to be stored in memory. Furthermore, the "width" parameter sets the data width, whether possibilities are "d" (8 bytes), "w" (4 bytes), "s" (2 bytes) and "b" (1 byte). Finally the actual instruction parameters "Vd", "Rs1", "Rs2" and "Rs3" denote respectively the destination register, the offset, the size and the stride.

One dimension can completely describe a stream, the previous stream configuration instruction starts and ends the stream description. However, as already explained, there are examples like 3-dimensional memory accesses that require a chain of descriptors to represent the whole of the 3 dimensions. Therefore, the starting stream configuration instruction can have the "sta" field to denote that other dimension descriptors will be chained up.

Thus, let's now describe multi-dimension streams. Remembering image 3.3, it is known that to describe this 2-dimensional access pattern, 2 dimension descriptors are needed. To describe the first dimension of the stream, the previously explained configuration instruction is used. In this case, since the stream is multi-dimensional, the first stream configuration instruction will have the "sta" field.

With the first dimension configuration done, to add a higher level configuration, it is necessary to describe the remaining configuration. Notice that, a dimension that is defined after another, is implicitly an outer dimension. With that stated, to define the second dimension, it is simply needed another instruction that targets the same destination register. Additionally, since it is a 2-dimensional stream, the complete description of the stream is supposed to terminate with the next stream configuration instruction. With this, the proposed configuration instruction that intends to add a dimension representation and terminate the stream description has the following format:

send Vd, Rs1, Rs2, Rs3

In this case, the source registers contain the configuration data of the second dimension. It is important to note that when adding a configuration, the direction of the transaction and the element width are not needed anymore, this information is only necessary for the first configuration.

Up to this point, with the two stream configuration instructions presented, the description of a 2-dimensional stream can be denoted. However, in other cases, it might be necessary to represent up to 8-dimensional streams. To provide the description of another dimension, another instruction that targets the same destination register needs to be provided. Consequently, to represent the middle dimensions of the stream, another configuration instruction that at the same time is supposed to add a dimension description and not terminate the full stream description is needed. Therefore, the proposed stream configuration instruction that follows this behaviour has the following format:

sapp Vd, Rs1, Rs2, Rs3

In this case, the source registers contain the configuration data of the dimension that is being appended to the full stream description.

To better understand the instructions described so far, let us analyze the 3-dimensional stream representation denoted in figure 3.10.

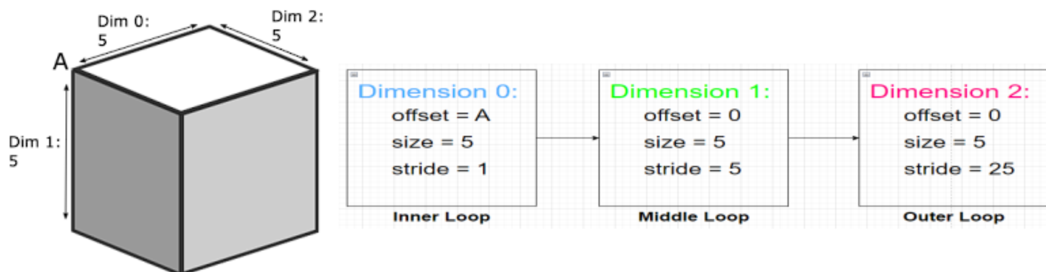


Figure 3.10: 3-Dimensional Stream Representation.

As described, to make the full description of this 3-dimensional stream, three configuration instructions, each describing one dimension are needed. Therefore, the full description of this stream is done by sequentially executing the following instructions:

scrt.sta.ld.w Vd, a1, a2, a3

sapp Vd, a4, a5, a6

send Vd, a7, a8, a9

With the first stream configuration instruction the stream description starts by configuring the first dimension. In this case, it is assumed that is intended to load a continuous flow of data from memory and that the data is word size. It is also assumed that the registers a1-3 contain the configuration data of the first dimension (A,5,1). With the second instruction, another dimension is being appended to the stream description, also assuming that the registers a4-6 contain the configuration data of the second dimension (0,5,5). Finally, with the third instruction, the final dimension description is added ending the full stream description, also assuming that the registers a7-9 contain the configuration data of the third and last dimension (0,5,25).

Summing up, with the stream configuration instructions denoted so far, the descriptors for all the linear patterns presented in the previous section can be represented. However, as represented in the previous section, two more types of descriptors were developed to represent more complex patterns, namely the modifier descriptor and the dynamic modifier descriptor. Therefore, it is also needed instructions that can configure these descriptions into a stream description.

Starting with the normal modifier. Remembering the lower triangular pattern denoted in figure 3.5 and the correspondent stream representation in figure 3.6, the two dimensions are represented with the stream configuration instructions presented. However, to constrain the stream to only fetch the lower triangle, an instruction denoting the modifier behaviour must be introduced before ending the stream description. Thus, the configuration instruction must provide the mode, the size, the target and the displacement parameters. Additionally, this instruction also needs to specify if it ends or not the full stream description. Hence, the configuration instruction for the modifier has the following format:

```
smod.<type>.<target>.<mode>    Vd, Rs1, Rs2
```

The mnemonic name ”smod” references ”Stream Modifier”. The ”type” parameter represents if the modifier ends the stream description (”end”) or not (”app”). Furthermore, The ”target” as already explained sets the parameter that the modifier will edit (”size”, ”offset” or ”stride”) and the ”mode” sets the operation to be carried out (”inc” or ”dec”). Finally, the actual instruction parameters ”Vd”, ”Rs1” and ”Rs2” denote respectively the destination register, the size and the displacement value.

Having the instruction that configures the normal modifier behaviour, it is easy to describe an instruction for the dynamic modifier based on the apriori. The only chances are that, in this case, the displacement value is traded with the source and there are more options for the mode selection. Therefore, the proposed configuration instruction for the dynamic modifer has the following format:

```
sdmod.<type>.<target>.<mode>    Vd, Rs1, Vs2
```

The mnemonic name ”sdmod” references ”Stream Dynamic Modifier”. The ”type” and ”target” parameters set the same as in the normal modifier instruction. The ”mode” parameter also sets the operation to be carried out, but in this case there are more options (”add”, ”sub”, ”inc”, ”dec” or ”set”). Once again the ”Vd” and ”Rs1” parameters denote respectively the destination and the size value. Finally, the ”Vs2” parameter denotes the register associated with the source stream.

3.4.2 Stream Control Instructions

During execution, the processor may need to switch context (for example, exception handling), which might limit the use of streams in those circumstances. To allow the momentary freeing/restoring of streams, authorizing concurrent execution of multiple processes without interfering with the stream configuration, a set of control instructions is provided. These instructions enable the configuration of each stream's state at any point in the code. Table 3.1 denote the available instructions and the correspondent behaviour.

Instructions	Behaviour
s.suspend	Momentarily disable the automatic iteration of a stream until the resume instruction is executed. During the suspended period, the streaming register will work as if no stream was configured.
s.resume	Enables again the automatic iteration of a stream.
s.terminate	Cease completely the stream configuration. The streaming register will work as if no stream was configured.

Table 3.1: Stream Control Instructions

3.4.3 Flow Control Instructions

In this subsection, the subset of the instructions that somehow change the flow of the execution is examined. The first instructions that without a doubt change the complete execution, are the branch instructions. Regarding the extension being defined, the only branch instructions that make sense are the ones that provide the programmer a way to conditionally jump regarding a correspondent stream state. In this case, knowing if a stream is already terminated or not is the only condition needed to know when to jump or not out of a looping situation. Taking what is stated into account, this extension defines two branch instructions. One instruction jumps when a correspondent-referenced stream is terminated and the other jumps when the stream is still not yet terminated. Both of the instructions format is the following:

sb.<condition >   Vd, <jump label >

The ”condition” parameter can be either ”nc” jumping to the label if the stream is not completed yet, or ”c” jumping to the label if the stream is completed. The ”Vd” parameter specifies the stream on which the conditions are going to be checked.

Moreover, it is usual for a code to process data in an inner for loop and then only write to memory at the end of that loop. Listing 3.7 denotes one of these examples and figure 3.11 denotes the correspondent behaviour being executed.

Listing 3.7: inner loop processing with outer loop memory access C Code

```
register float aux = 0;
for(int i=0; i<N; i++){
    for(int j=0; j<M; j++){
        aux += X[i][j];
    }
    Y[i] = aux;
    aux = 0;
}
```

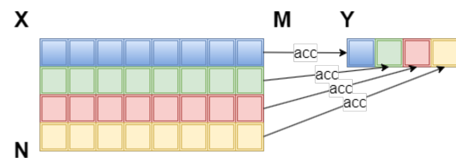


Figure 3.11: inner loop processing with outer loop memory access Representation.

In this case, despite being able to describe both the X and Y memory access patterns with the stream configuration instructions, it is not possible to distinguish the terminations of the loop ”j”. To describe such behaviour, it requires Y to be configured as a single element store stream, where the accumulation instruction output would be written to. Additionally, for this example to work as expected, the vector length must be a multiple of the innermost loop access size (M), otherwise, data from iteration ”i+1” would be loaded and processed in iteration ”i”, leading to incorrect behaviour and data loss. To get around this, a vector-coupled behaviour that can be enabled for any dimension of any stream was introduced. This vector-coupled behaviour allows a multi-dimensional stream to be logically divided into multiple streams with lower dimensionality, for example, it automatically converts a 2-dimensional stream (with dimensions ”i” and ”j”) into multiple ”i” linear dimensions ”j” while keeping the stream processing unchanged. Figure 3.12 shows the difference between using or not this behaviour.

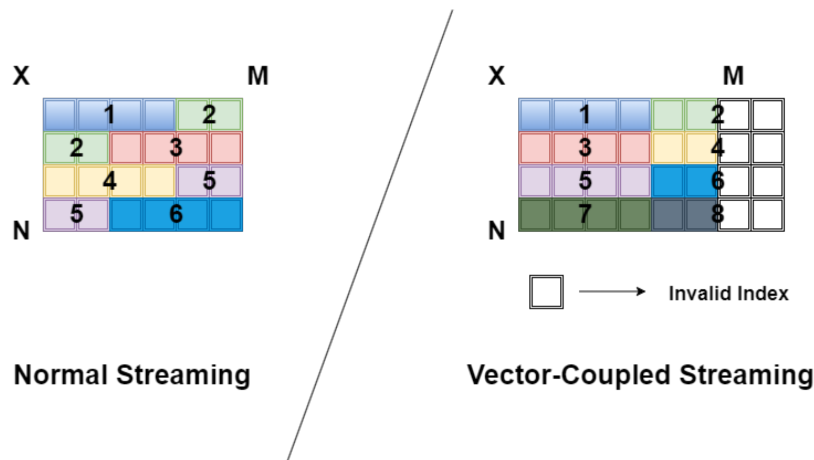


Figure 3.12: Normal Streaming versus Vector-coupled Streaming.

To activate this behaviour, the correspondent instruction has the following format:

```
scfgvec    Vd
```

This instruction needs to be put after the target dimension configuration. The fundamental principle of the vector coupling instruction is that the last vector of the vector coupled dimension will be flagged to represent its dimension termination. To capture this same flag, one new branch instruction with two possible behaviours was defined. Despite both behaviours being able to capture the dimension termination flag, one instruction will jump if the dimension ends and the other when the dimension does not end. This proposed instruction format is the following:

```
sb.<dimension condition >. <dimension number >    Vd, <jump label >
```

The "dimension condition" parameter can be either "ndc" jumping to the label if the stream dimension is finished, or "dc" jumping to the label if the stream dimension is completed. The "dimension number" parameter specifies the dimension on which the flag termination is set.

3.5 Instruction Set Overview

Table 3.2 denotes all the instructions presented in the proposed RISC-V stream-based extension.

Table 3.2: Stream-based extension Instructions

Instruction	Description
scrt.ld.[width]	Simple Load Stream
scrt.st.[width]	Simple Store Stream
scrt.sta.ld.[width]	Start load stream configuration
scrt.sta.st.[width]	Start store stream configuration
sapp	Append dimension descriptor to stream in configuration
send	Add dimension descriptor and end configuration of stream
smod.[type].[target].[mode]	Append modifier descriptor to stream in configuration
sdmod.[type].[target].[mode]	Append dynamic modifier descriptor to stream in configuration
s.suspend	Momentarily disable the automatic iteration of a stream
s.resume	Enables again the automatic iteration of a stream
s.terminate	Cease completely the stream configuration
sb.nc	Branch if stream not finished
sb.c	Branch if stream finished
sb.ndc.x	Branch if dimension x not finished
sb.dc.x	Branch if dimension x finished
scfgvec	Configure vector-coupled Streaming

To understand the defined stream-based extension and its integration with the rest of the RISC-V ISA, Listing 3.8 and Listing 3.9 denote the C code of the famous SAXPY benchmark, the first using only scalar RISC-V instructions, the second cooperating the defined RISC-V stream based extension with the scalar RISC-V instructions.

Listing 3.8: SAXPY benchmark with Scalar RISC-V instructions

```
asm volatile (
    "mv x5, %[n]\n"
    "fsgnj.s f1, f1, f1\n"
    "1:\n"
    "fld f2, 0(%[src1])\n"
    "fld f3, 0(%[src2])\n"
    "fmul.s f2, f2, %[a]\n"
    "fadd.s f1, f1, f2\n"
    "fadd.s f3, f3, f1\n"
    "fsw f3, 0(%[src3])\n"
    "addi %[src1], %[src1], 4\n"
    "addi %[src2], %[src2], 4\n"
    "addi %[n], %[n], -1\n"
    "bnez %[n], 1b\n"
    :
    : [src1] "r" (src1), [src2]
    "r" (src2), [a] "f"
    (a), [n] "r" (n)
);
```

Listing 3.9: SAXPY benchmark with the stream-based extension and Scalar RISC-V instructions

```
asm volatile(
    "scrt.ld.w  f1, %[src1], %[y], %[z] \t\n"
    "scrt.ld.w  f2, %[src2], %[y], %[z] \t\n"
    "scrt.st.w  f3, %[src2], %[y], %[z] \t\n"
     :
     : [src1] "r"(src1),
     [src2] "r"(src2), [y] "r"(size),
     [z] "r" (1), [A] "r" (A)
);
asm volatile(
    "fLoop1: \t\n"
    "fmul.s f2, f2, %[a] \n\t"
    "fadd.s f1, f1, f2 \n\t"
    "fadd.s f3, f3, f1 \n\t"
    "sb.nc  f1, fLoop1 \n\t"
    :: [a] "f" (a)
);
```

Comparing both kernels, when using the proposed stream-based extension, since the memory accesses are pre-configured with the stream configuration instructions, the principal processing loop does not need the load and store instructions. Moreover, the indexing instructions also disappear.

Referencing Listing 3.9, first, the streams needed are configured with the stream configuration instructions. Next, in the principal processing loop, by making use of the floating point RISC-V scalar instructions, the streams are iterated until termination. As previously mentioned, the floating point registers have information on whether a load/store stream is associated with them.

3.6 Summary

This Chapter presented the proposed RISC-V stream-based extension. First, the memory access descriptor representation was described. Following, the architectural state, defining the necessary architecture features to support the proposed extension, was discussed. Next, the set of all the proposed extension instructions was presented. Finally, a brief overview of the ISA extension was done.

4

Spike Simulator Extension

Contents

4.1 Overview 46
4.2 Stream Unit Implementation 46
4.3 Instruction Macros 51
4.4 Summary 58

4.1 Overview

In this chapter, it is described the implementation of the RISC-V stream-based extension, detailed in Chapter 3, on the Spike functional Simulator. Since the Spike simulator does not inherently support a streaming paradigm, to denote the stream-based extension's behaviour, it is necessary to first add simulated streaming mechanisms into the Spike architecture. To do so, first, the simulated components added to Spike will be described, that support the streaming mechanisms of the defined extension. Next, it is detailed the inclusion of the proposed set of instructions and their integration with the remaining RISC-V paradigm.

4.2 Stream Unit Implementation

As mentioned, Spike does not inherently support stream-based mechanisms. Therefore, the first task was to introduce a streaming mechanism into the Spike architecture. As such, a new simulated component was created, namely the **Stream Unit**, whose function is to provide the streaming mechanisms to support the defined extension. Figure 4.1 denotes how the Stream Unit is integrated into the actual Spike architectural overview.

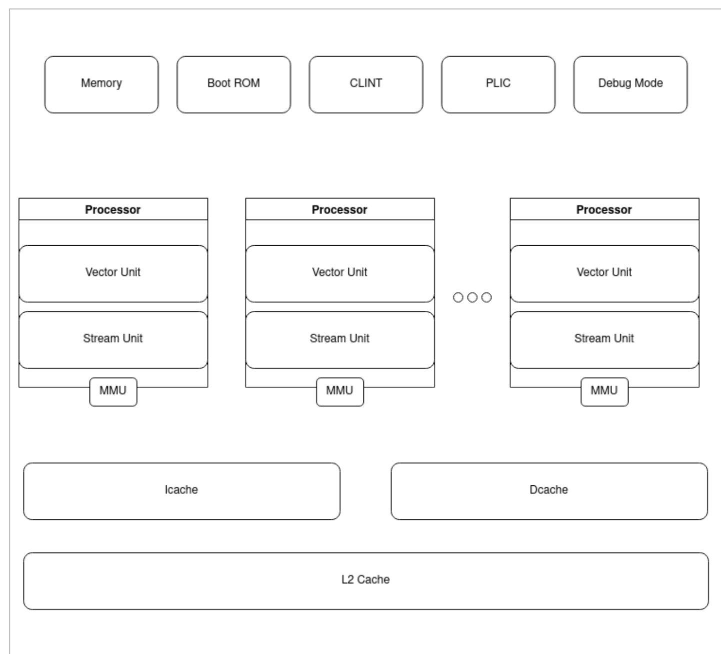


Figure 4.1: Spike Overview with the addition of the Stream Unit.

The following paragraphs describe how the Stream Unit is added to the Spike environment. From this point, every directory mentioned has per base the *riscv-isa-sim/riscv* local root directory, which denotes

the location of the Spike components code. To do so, it was added to this local directory two new code files, namely the *stream_unit.cc* and the *stream_unit.h*, which respectively denote the actual Stream Unit simulation code and its correspondent header. Referencing first the *stream_unit.h* header code, a new class was implemented, called "*streamUnit_t*", that fully denotes all the components present in the Stream unit representation. Listings 4.1 and 4.2 denote the full code that represents the referenced Stream Unit class.

Listing 4.1: Stream Unit Class (Part 1)

```
1 class streamUnit_t
2 {
3 public:
4   processor_t* p;
5   int stream_type[NSR];
6   stream streams[NSR];
7   bool scalar_cooperation;
8   bool rvv_enabled;
```

Listing 4.2: Stream Unit Class (Part 2)

```
1 public:
2   streamUnit_t():
3     p(0),
4     stream_type{0},
5     scalar_cooperation(true),
6     rvv_enabled(true) {
7   }
8   void init_stream_unit();
9 };
```

In the Stream Unit class, first, a pointer "p" is declared, which later will be associated with the processor class, denoting their association. Next, the "stream_type" array is defined. The "stream_type" array represents the actual type of each defined stream, denoting whether it is a loading or a storing stream. Since in the majority of the Instruction Set Architectures (ISAs) the number of provided registers is 32 and a stream will always need to be associated with a register, the "stream_type" array is composed of 32 elements. Moreover, an array of 32 "stream" structures is defined, which will be later detailed and explained. Finally, two booleans are defined, the "scalar_cooperation" and the "rvv_enabled", whose function is to respectively represent, if the stream unit class can be associated with the RISC-V scalar instructions and the RISC-V Vector (RVV) extension's instructions.

Referencing now the *stream_unit.cc* code, a function called "init_stream_unit" was implemented, which will initiate all the necessary components for the stream class to work as the streaming mechanism. Listing 4.3 denotes the referenced function.

Listing 4.3: Stream Unit Init Function

```
1 void streamUnit_t::streamUnit_t::init_stream_unit()
2 {
3   for(int i=0 ; i<NSR; i++){
4     for(int j=0; j<8; j++){
```

```
5         streams[i].dimensions[j].offset = 0;
6         streams[i].dimensions[j].size = 0;
7         streams[i].dimensions[j].stride = 0;
8         streams[i].dimensions[j].dim_elements_counter = 0;
9         streams[i].dimensions[j].total_dim_elements = 0;
10        streams[i].modifiers[j].target = 5;
11        streams[i].modifiers[j].behavior = 0;
12        streams[i].modifiers[j].displacement = 0;
13        streams[i].modifiers[j].size = 0;
14      }
15      streams[i].dimension_counter = 0;
16      streams[i].elements_counter = 0;
17    }
18  }
```

Up to this point, although the definition of the stream unit component is done, the rest of the spike simulation environment still needs to be connected with this streaming mechanism. Therefore, an instance of the stream unit class needs to be created inside of the spike processor class structure. To do so, the following line was added to the structure of the processor class, implemented in the *processor.h* header code:

```
streamUnit_t SU;
```

By doing so, an instance of the stream unit class was created under the actual processor class. To make sure that whenever the processor class is created and initiated, the stream unit class is also created and initiated, a function called *pinit_stream_unit* was also added in the processor constructor class in the *processor.cc* code. Listing 4.4 denotes the actual code inside of this function.

Listing 4.4: Processor Function initiating the Stream Unit

```
1  void processor_t::pinit_stream_unit()
2  {
3      SU.p = this;
4      SU.init_stream_unit();
5  }
```

With the first line of the function, the connection between the stream unit and the processor is done, since by executing it, the stream unit instance will always have access to the actual processor class

instance. With the second line, the stream unit class instance is being initiated by calling the already referenced function in Listing 4.3.

4.2.1 Streams Implementation

Now that the structure of the stream unit class developed was explained and the process of linking the stream unit with the actual Spike processor simulation was denoted, the way the actual streams were simulated can be explained. In the defined extension, a stream is always associated with a respective register. Remembering the code in Listing 4.1, the stream unit class defined an array composed of 32 ”stream” structures. Each one of these structures represents an actual stream. Since the majority of the ISAs provide 32 registers and a stream will always be associated with a register, the maximum number of streams that can be represented is 32. Listing 4.5 denotes the actual code of the ”stream” structure.

Listing 4.5: Stream Structure Code

```
1  struct stream
2  {
3    std::vector<uint64_t> addresses;
4    std::variant<
5        std::vector<uint32_t>,
6        std::vector<uint16_t>,
7        std::vector<uint8_t>,
8        std::vector<uint64_t>
9      > values;
10   dimension dimensions[8];
11   modifier modifiers[8];
12   int dimension_counter;
13   int elements_counter;
14 };
```

Examining the Stream structure code, first an array called ”addresses” is defined, which later will be composed of unsigned elements of 64 bits. This array holds the various memory positions that constitute the correspondent stream being defined. Depending on which memory access pattern the stream is representing, this array will hold the respective necessary memory positions. Therefore, this array is not composed of a fixed number of elements. Moreover, the array ”values” is defined, which, like the previous one, is not composed of a fixed number of elements. This array can be constituted of four different types of elements. Each one of these types represents one of the possible element widths representations of the defined extension, namely the 8-bit for the Byte type (B), the 16-bits for the Half-

Word type (H), the 32-bits for the Word type (W) and the 64-bits for the Double-Word type (D). Therefore, the array ”values” will hold the respective values of the memory positions of the stream. Although the values are being kept in this stream structure, since the defined extension is integrated with the rest of the RISC-V ISA, depending on if the defined extension is cooperating with RVV or with the standard scalar instructions, these values will also be outputted to the correspondent registers.

Up to this point, in the stream representation, the simulated components that will denote the actual stream memory access positions and the respective values were already explained. However, there is still a need to simulate the various descriptors, defined in Section 3.2, that allow to arrive at the memory positions that constitute the stream. The ”dimension_counter” and the ”elements_counter” variables, which are lastly defined in the Stream structure code, will respectively hold the number of dimensions and the number of total elements present in the correspondent stream.

As denoted in Subsection 3.4.1, to represent some streams, a lot of descriptors (each denoted by one stream configuration instruction) are needed. In the defined extension, there are three possible descriptors, the simple three-element tuple descriptor that represents a dimension, the four-element tuple that represents a modifier and the other four-element tuple that represents a dynamic modifier. To simulate the referenced descriptors that constitute a stream, two arrays of ”dimension” and ”modifier” structures were defined. In the defined extension, the maximum dimensions and modifiers a stream can have are respectively eight and seven. Therefore, the ”dimensions” array will have eight ”dimension” structures, and to simplify, the ”modifiers” array will also have eight ”modifier” structures. Listing 4.6 and 4.7 denote respectively the actual code of the ”dimension” and the ”modifier” structure.

Listing 4.6: Dimension Structure Code

```
1 struct dimension
2 {
3   uint64_t offset;
4   uint64_t size;
5   uint64_t stride;
6   int dim_elements_counter;
7   int total_dim_elements;
8 };
```

Listing 4.7: Modifier Structure Code

```
1 struct modifier
2 {
3   uint64_t target;
4   uint64_t behavior;
5   uint64_t displacement;
6   uint64_t size;
7 };
```

In the defined extension, to define a stream dimension three elements are needed, the offset, the size and the stride. Therefore, in the dimension structure, these same elements are defined. Moreover, since there are examples where the number of elements present in each dimension of a stream needs to be tracked, two integer variables are defined. While the ”total_dim_elements” variable will hold the total number of elements the dimension holds, the ”dim_elements_counter” will track, at some point of execution, the number of remaining dimension elements left to process.

To define a modifier four elements are needed, the target, the behaviour, the size and the displacement. Therefore, in the modifier structure, these same elements are defined. In this case, the target variable will have three possible values, zero denoting a size modification, one denoting a stride modification and two denoting an offset modification. The behaviour variable will have two possible values, zero denoting an increment behaviour and one denoting a decrement behaviour.

There is no specific structure to define the dynamic modifier because, at the time of this Thesis work, the dynamic modifier descriptor was not simulated.

With that stated, the Stream Unit Innit function referenced in the Listing 4.3 can now be fully explained. This function simply initiates all the dimensions and modifiers of all the possible streams with empty values. For example, regarding the dimensions, the "total_dim_elements" variable is being set to zero. Moreover, when a definition of a dimension occurs by executing the respective stream configuration instruction, this value will be changed to the actual number of elements of the correspondent dimension. The same occurs for the modifiers. For example, the "target" variable is being set to five when the only possible combinations for this variable are zero, one or two. Furthermore, when a definition of a modifier occurs, this value will be changed to the actual target of the modifier descriptor.

4.3 Instruction Macros

With the streaming mechanisms simulated and explained, this section will explain how the referenced components are used to actually describe the behaviour of the instructions present in the defined extension. First, all the steps needed to actually make sure that the Spike simulator recognizes the extension's instructions and behaviour is denoted. Next, the Macros elaborated to simulate the full behaviour of the instructions will be described.

To include a new instruction to the Spike simulator, the following three guidelines need to be fulfilled:

- Spike does not hold specifically the opcodes of the instructions. These are in fact kept at another repository, the *riscv-opcodes* repository [49]. Therefore, the actual opcode of the instruction needs to be added to the respective riscv-opcodes local repository.

- Next, the actual name of the instruction needs to be added to the list of instructions in the *riscv.mk.in* file. By doing so, the added instruction is now declared under Spike.

- Finally, an *instruction_name.h* needs to be added under the *insns* folder. The *insns* folder holds all the definitions of Spike's supported instructions. The added *instruction_name.h* code needs to define the instruction behaviour, as is this respective code that will be executed when the instruction is called for execution.

Since a new instruction set extension is going to be included, a new folder called *rv_stream* was created in the riscv-opcodes local repository, which holds all the opcodes of the instructions present in the defined extension. For example, the first two stream configuration instructions previously defined in Subsection 3.4.1, have the following opcodes:

```
scrt.ld.w vd rs1 rs2 rs3 26..25=3 14=1 13..12=2 6..2=0x2 1..0=3
```

```
scrt.sta.ld.w vd rs1 rs2 rs3 26..25=2 14=1 13..12=2 6..2=0x2 1..0=3
```

Now that is known how to add the actual extension to the Spike simulator, the behaviour of each instruction, denoted in the *instruction_name.h* code file, needs to be defined. Since the majority of the already supported instructions are defined with a set of Macros, a file code called *stream_ext_macros.h* was added, which holds all the Macros denoting the behaviour of the instructions. Moreover, describing all the implemented vast code would be redundant. Therefore, to simply understand the used implementation techniques, only the code that simulates the "scrt.sta.ld.w" instruction will be explained.

First, in the actual *scrt_sta_ld_w.h* code file the Macro, named "STREAM_START_LOAD", that will actually hold all the code denoting this instruction behaviour will be called. Listing 4.8 denotes the referenced code.

Listing 4.8: scrt.sta.ld.w instruction Macro code

```
1 #define STREAM_START_LOAD \
2   STREAM_TYPE_SIZE \
3   STREAM_GET_PARAMS \
4   P.SU.stream_type[rd_num] = 1; \
5   STREAM_ASSIGN_TYPE(size) \
6   STREAM_UPDATE_DIMENSIONS
```

From Listing 4.8, it is observed that by unfolding the main Macro, more Macros are executed. These Macros together hold the sequence of code that simulates the respective instruction. Listing 4.9 and Listing 4.10 denote respectively the unfolded code present in the "STREAM_TYPE_SIZE" and "STREAM_GET_PARAMS" Macros.

Listing 4.9: STREAM_TYPE_SIZE Macro Code

```
1 #define STREAM_TYPE_SIZE \
2   uint64_t streamTypeSize =
3   insn.stream_type_size(); \
4   int size = 0; \
5   if(streamTypeSize == 0) size = 1; \
6   else if(streamTypeSize == 1) size = 2; \
7   else if(streamTypeSize == 2) size = 4; \
```

```
8    else if(streamTypeSize == 3) size = 8;
```

Listing 4.10: STREAM_GET_PARAMS Macro Code

```
1  #define STREAM_GET_PARAMS \
2    reg_t rd_num = insn.rd(); \
3    uint64_t rs1_num = RS1; \
4    uint64_t rs2_num = RS2; \
5    uint64_t rs3_num = RS3;
```

First, regarding the code present in Listing 4.9, this Macro execution retrieves from the instruction's opcode, the width of the element. In this case, since the instruction denotes that the respective element width is 32 bits corresponding to a word, the "streamTypeSize" variable will hold the value two. Therefore, the "size" variable will hold the value four, denoting the number of bytes the correspondent element width has. This "size" variable is needed when retrieving the actual memory positions of the stream. Regarding the code present in Listing 4.10, by executing this Macro, the parameters of the instruction are being retrieved from the instruction's opcode. These parameters are the destination register and the three source registers. To get the values of the offset, size and stride, the values present in the denoted source registers are outputted. Therefore, the "rd_num", the "rs1_num", the "rs2_num" and the "rs3_num" will respectively be composed of the number of the destination register, the offset value, the size value and the stride value.

Next, by executing the following line of the First Macro code, the "stream_type" variable is set to one. By doing so, the destination register is associated with a load stream.

Moreover, Listing 4.11 and 4.12 denote the code present in the final Macros.

Listing 4.11: STREAM_ASSIGN_TYPE Macro Code

```
1  #define STREAM_ASSIGN_TYPE(size) \
2    if(size == 4){ \
3      P.SU.streams[rd_num].values =
4      std::vector<uint32_t>(); \
5    } \
6    else if(size == 1){ \
7      P.SU.streams[rd_num].values =
8      std::vector<uint8_t>(); \
9    } \
10   else if(size == 2){ \
11     P.SU.streams[rd_num].values =
12     std::vector<uint16_t>(); \
13   } \
14   else if(size == 8){ \
15     P.SU.streams[rd_num].values =
16     std::vector<uint64_t>(); \
17   }
```

Listing 4.12: STREAM_UPDATE_DIMENSIONS Macro Code

```
#define STREAM_UPDATE_DIMENSIONS \
  P.SU.streams[rd_num].dimension_counter += 1; \
  int aux_dim_index =
  P.SU.streams[rd_num].dimension_counter - 1; \
  P.SU.streams[rd_num].dimensions[aux_dim_index]
  .offset = rs1_num; \
  P.SU.streams[rd_num].dimensions[aux_dim_index]
  .size = rs2_num; \
  P.SU.streams[rd_num].dimensions[aux_dim_index]
  .stride = rs3_num;
```

Referencing the code in Listing 4.11, by executing this Macro, the ”values” array elements type is set with the element width apriori obtained. For example, regarding the instruction being defined, since the apriori ”size” parameter was set to four, the ”values” array of the correspondent stream will be composed of elements of 32 bits denoting words. Finally, regarding the code in Listing 4.12, by executing this Macro, the actual parameters of the denoted dimension are saved.

This Macros behaviour is pretty similar for all the stream configuration instructions, where the final goal is to save all the dimensions and modifiers descriptions parameters in the correspondent stream structure. Therefore, by the end, when the configuration of the full stream is done, all the parameters required to actually access and manipulate the memory positions present in the correspondent stream are saved.

4.3.1 Integration with RISC-V Paradigm

The defined stream-based extension is integrated with the rest of the RISC-V paradigm. More specifically, the defined extension can cooperate with both the scalar RISC-V instructions and the RVV extension instructions. In both cases, the process of integrating the defined extension is very similar. When cooperating with the scalar instructions, it is important that the integer and floating point registers are able to iterate through the defined streams. By doing so, it will be possible to first retrieve the source stream values and then save the output value in the correspondent memory position denoted by the destination stream. Moreover, when cooperating with RVV, it is important that the vector registers are able to also iterate through the defined streams. The only difference regarding both cooperations is that while with scalar instructions each stream iteration consumes only one element, with RVV instructions, each stream iteration consumes more than one element. Due to the similarities presented, only the integration with RVV will be discussed.

As mentioned, to integrate the defined extension with RVV, the RVV instructions need to make use of the streams defined by the stream configuration instructions. To do so, Spike's RVV instructions were extended to couple with the streaming behaviour. As mentioned, Spike works with a set of Macros to define the instructions. As so, all the RVV's instructions code is present in the *v_ext_macros.h* code file. To specifically describe how the RVV Macros were extended, all the changes made to the *Vadd*

instruction code will be described. Listing 4.13 and 4.14 respectively denote the unchanged code in the actual *vadd.h* code file and the correspondent Macro code called.

Listing 4.13: *vadd.h* file Code

```
1 // vadd.vv vd, vs1, vs2, vm
2 VI_VV_LOOP
3 ({
4   vd = vs1 + vs2;
5 })
```

Listing 4.14: Vadd Macro Code

```
1  #define VI_VV_LOOP(BODY) \
2    VI_CHECK_SSS(true) \
3    VI_LOOP_BASE \
4    if (sew == e8) { \
5      VV_PARAMS(e8); \
6      BODY; \
7    } else if (sew == e16) { \
8      VV_PARAMS(e16); \
9      BODY; \
10   } else if (sew == e32) { \
11     VV_PARAMS(e32); \
12     BODY; \
13   } else if (sew == e64) { \
14     VV_PARAMS(e64); \
15     BODY; \
16   } \
17   VI_LOOP_END
```

The code referenced in Listing 4.14 describes the processing execution of the arithmetic add operation. In the RVV extension, this instruction needs to be antecedent by the actual Load instructions that transfer the data to the vector source registers. In this case, the Load instructions are substituted by the streams configured with the stream configuration instructions. Therefore, in terms of the RVV instruction referenced, only the mechanism of iterating through the streams needs to be added. Listing 4.15 and 4.16 denote the *Vadd* Macro code with the modifications needed for streaming.

Listing 4.15: Vadd Modified Macro Code (Part 1)

```
1  #define VI_VV_LOOP(BODY) \
2    if(P.SU.rvv_enabled){ \
3      ADD_ITERATE_LOAD_STREAMS \
4    } \
5    VI_CHECK_SSS(true) \
6    VI_LOOP_BASE \
7    if (sew == e8) { \
8      VV_PARAMS(e8); \
9      BODY; \
10     if(P.SU.rvv_enabled){ \
11       ADD_VALUE_DEST_STREAM \
12     } \
13   } else if (sew == e16) { \
14     VV_PARAMS(e16); \
15     BODY; \
```

```
16     if(P.SU.rvv_enabled){ \
17       ADD_VALUE_DEST_STREAM \
18     } \
```

Listing 4.16: Vadd Modified Macro Code (Part 2)

```
1   } else if (sew == e32) { \
2     VV_PARAMS(e32); \
3     BODY; \
4     if(P.SU.rvv_enabled){ \
5       ADD_VALUE_DEST_STREAM \
6     } \
7   } else if (sew == e64) { \
8     VV_PARAMS(e64); \
9     BODY; \
10    if(P.SU.rvv_enabled){ \
11      ADD_VALUE_DEST_STREAM \
12    } \
13  } \
14  VI_LOOP_END \
15  if(P.SU.rvv_enabled) { \
16    ITERATE_STORE_STREAMS \
17  }
```

Referencing the previous code, it is important to mention that the streaming mechanisms are only executed with the condition that in the Stream unit class, the cooperation with the RVV is enabled. Examining the code, first in order to populate the source registers with the data, the correspondent streams need to be iterated. This is done by executing the "ADD_ITERATE_LOAD_STREAMS" Macro, which code is denoted in Listing 4.17

Listing 4.17: ADD_ITERATE_LOAD_STREAMS Macro Code

```
1  #define ADD_ITERATE_LOAD_STREAMS \
2    int vec_len = P.VU.vl->read(); \
3    if(P.SU.stream_type[rs1_num] == 1){ \
4      CHECK_VECTOR_DIM_COUPLING(rs1_num) \
5      if(isVecDimCoupled){ \
6        EXECUTE_ITERATE_LOAD_DIM_STREAM(rs1_num) \
7      } \
8      else{ \
9        EXECUTE_ITERATE_LOAD_STREAM(rs1_num) \
10     } \
11     EXECUTE_RVV_COOPERATION_FILL_V_REGISTER(rs1_num); \
```

```
12   } \
13   if(P.SU.stream_type[rs2_num] == 1){ \
14     CHECK_VECTOR_DIM_COUPLING(rs2_num) \
15     if(isVecDimCoupled){ \
16       EXECUTE_ITERATE_LOAD_DIM_STREAM(rs2_num) \
17     } \
18     else{ \
19       EXECUTE_ITERATE_LOAD_STREAM(rs2_num) \
20     } \
21     EXECUTE_RVV_COOPERATION_FILL_V_REGISTER(rs2_num); \
22   }
```

Examining the code present in Listing 4.17, the "vec_len" variable is defined. This variable is set to have the actual vector length defined by the RVV extension. This is the variable that will denote in each iteration of the stream, how much data is going to be retrieved to the source registers. Next, the actual iteration through the streams is executed by the "EXECUTE_ITERATE_LOAD_STREAM" Macro, which is performed in both the source streams. A different case is present if Vector Couple Streaming is set, but only the normal example will be demonstrated. Listing 4.18 denote the "EXECUTE_ITERATE_LOAD_STREAM" Macro code.

Listing 4.18: EXECUTE_ITERATE_LOAD_STREAM Macro Code

```
1 #define EXECUTE_ITERATE_LOAD_STREAM(stream_number) \
2   int total_number_elements = P.SU.streams[stream_number].addresses.size(); \
3   for(int i = 0; i < vec_len; i ++){ \
4     if(P.SU.streams[stream_number].elements_counter == 0) break; \
5     int addr_idx = total_number_elements - P.SU.streams[stream_number].elements_counter; \
6     STREAM_GET_VALUE(addr_idx,stream_number); \
7     P.SU.streams[stream_number].elements_counter -= 1; \
8   }
```

Examining the code present in Listing 4.18, the total number of elements present in the stream is counted. Next, depending on the "vec_len" variable, the corresponding number of elements will be retrieved from the stream to the associated vector register. This is done by the "STREAM_GET_VALUE" Macro. Finally, the number of elements processed is discounted from the "elements counter" variable, which denotes the number of elements that are still to be processed in the stream.

Going back to the code present in Listing 4.15 and Listing 4.16, now that the source streams were iterated, so that the vector source registers were populated with the corresponding data, the rest of the code executing the actual add operation was not changed. The only addition is that by executing

the "ADD_VALUE_DEST_STREAM" Macro, the add operation result is also saved to the stream structure. Finally, with the add operation result in the destination register, the destination stream needs to be iterated to save the outputted results to the memory. This is done by executing the "ITERATE_STORE_STREAMS" Macro, which is the final Macro code and the last modification made to the RVV "Vadd" code.

Summing up, it was possible to integrate the defined stream-based extension with the RVV extension by defining streams and letting the RVV processing instructions iterate through the defined streams to retrieve and output the results.

4.4 Summary

This Chapter described the implementation of the defined stream-based extension on the Spike functional Simulator. First, it was described how Spike was extended to simulate the streaming mechanisms. Next, it was detailed the inclusion of the proposed set of instructions and their integration with the remaining RISC-V paradigm.

5 Results Overview and Discussion

Contents

5.1 Overview 60
5.2 Methodology 60
5.3 Results 61

5.1 Overview

This Chapter provides a comprehensive validation of the proposed RISC-V stream-based extension. First, it is described the validation methodology, including the adopted toolchains. Then it is presented a detailed breakdown and characterization of the set of benchmarks used in the validation steps. Finally, the proposed extension is evaluated in terms of code efficiency in comparison with the standard RISC-V scalar and vector extensions.

5.2 Methodology

As mentioned, in this section all the tools that helped in the development of this Thesis work will be exposed. Moreover, it will be explained how they were used to achieve the pretended milestones. This section will specifically tackle the Spike Simulator, the RISC-V Opcodes repository, the RISC-V Proxy Kernel, the compiler with assembler support for stream approaches and a set of benchmarks that were used in the validation of the proposed extension.

5.2.1 Spike Simulator

As already denoted throughout the presentation of this work, the Spike simulator was the principal tool that led to the viability of this Thesis. Therefore, the Spike simulator does not need any further presentations. Section 2.6 describes the vast study made during this work to fully understand how this tool worked. Moreover, the whole Chapter 4 is focused on how the streaming mechanisms, denoted in the stream-based extension, were simulated under the Spike simulator.

5.2.2 RISC-V Opcodes

The RISC-V Opcodes repository [49] not only holds all the standard RISC-V instructions opcodes but also all the opcodes of the instructions present in the official RISC-V extensions. As already stated, the Spike simulator does not hold the opcodes of the instructions itself. In fact, the RISC-V opcodes repository works alongside the Spike simulator providing the opcodes of the instructions supported in Spike. In the Thesis work, this repository was of extreme importance since it was extended to accommodate all the opcodes of the instructions present in the defined extension.

5.2.3 RISC-V Proxy Kernel

The RISC-V Proxy Kernel [50] is a lightweight application execution environment that can host statically-linked RISC-V ELF binaries. Moreover, it mimics an Operating System's Kernel and overviews and

registers information of an execution of a given binary. In the Thesis work, this tool was essential to test the extension's instructions benchmarks that will be denoted later in this document. The benchmarks ran did not simulate bare-metal executions because targeting these would be an extremely complex issue to deal with in the available time frame. The Proxy Kernel itself didn't need to understand the instructions of the proposed extension as it relays their execution to the Spike simulator.

5.2.4 Compiler

At the start of the project, it was given access to resources built during the development of the Unlimited Vector Extension (UVE) extension. One of these resources was a customized version of the LLVM compiler with assembler support for the UVE extension. The customized compiler is a version of the LLVM compiler, that can assemble UVE instructions into the equivalent opcodes. This project cannot directly generate UVE code from C source code and, instead, must translate from inline assembly instructions.

Due to the similarities between the UVE extension and the defined extension, this tool can be repurposed to compile the stream-based extension's instructions as if they were UVE instructions, but keeping the stream-based extension's instructions execution behaviour. This compiler was modified to provide assembler support for the defined stream-based extension. Therefore, the modified compiler can translate from inline assembly instructions the defined extension's instructions.

5.2.5 Benchmarks

Another of the resources provided by the UVE repository was a repository with several hand-made kernels that used UVE for their implementation. The repository with the kernels is based on the Polybench suite [51] and contains two versions of each benchmark: one with the original C source code and another with the corresponding UVE version. The Polybench project is a benchmark suite with 30 algorithms based on mathematical utilities to evaluate applications and their capabilities. Most of the provided algorithms include complex memory access patterns that showcase the UVE's potential. These benchmarks were modified in order to respect the defined stream-based extension instructions and were of extreme importance to validate the results that will be further explained in this document.

5.3 Results

In this section, all the results obtained during the work of this Thesis will be showcased. One of the major milestones of this work was to define a stream-based extension and prove its functional behaviour by adding it to the Spike Simulator. Another goal was to make sure that the defined extension's streaming mechanisms could cooperate with the RISC-V Vector (RVV) extension. Moreover, it was also wanted

that the streaming mechanisms simulated could cooperate with the standard scalar RISC-V operations. As so, this section will be divided into two major subsections. Subsection 5.3.1 will focus on the results obtained in terms of implementation. It will demonstrate how it was acted to prove that the defined extension was successfully defined and extended to the Spike Simulator. Moreover, in Subsection 5.3.2, the focus is on showing, with relevant benchmark code examples, the improvements delivered by the stream-based extension. A vast analysis will be done, discussing whether the extension's mechanisms are an asset in terms of performance delivery.

5.3.1 Implementation Results

Despite having the extension instructions code already in the Spike Simulator, a testing process was needed to validate the proposed extension. The testing protocol denoted in figure 5.1 was adopted.

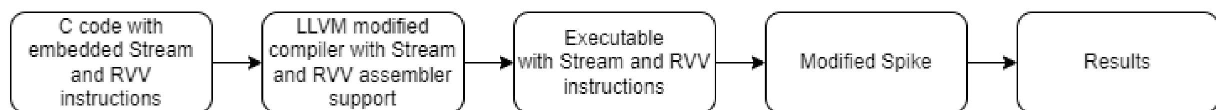


Figure 5.1: Testing Protocol used to validate the modified Spike with the defined Stream-based Extension.

This testing protocol is defined by sequentially performing the following steps:

- Implement a C code denoting a benchmark. The benchmark is executed with the use of the defined stream-based extension's instructions and the RVV's instructions, the C code will be implemented by using the ASM [52] mechanism that allows reading and writing of C variables using assembly code.
- By using the modified version of the LLVM compiler that has assembly support for stream and RVV instructions, compile the benchmark created, which will give the correspondent benchmark executable.
- With the help of the RISC-V Proxy Kernel tool, execute the benchmark under the Spike simulator. By doing so, the respective benchmark will be executed by the simulated functional model of the proposed stream-based extension.
- Finally, check the outputted results by executing the benchmark with the stream-based instructions and confirm if it corresponds to the expected results. If it is confirmed, it can be assumed that the correspondent instructions used in the benchmark were successfully simulated in the Spike environment.

It is important to note that, to test the stream instructions defined in Chapter 3, there is a need to make use of the RVV or the standard scalar RISC-V processing instructions. The instructions defined

in the proposed stream-based extension only configure and control the actual streams. To make use of the configured streams, for example, the arithmetic instructions present in the RVV extension need to be used, which will iterate through the load/store streams consuming their data. Therefore, the compiler used also has assembler support for the RVV extension. In order to test that all the instructions present in the defined stream-based extension were successfully added, the denoted testing process was repeated to a lot of benchmarks. Table 5.1 denotes all the benchmarks that were tested, the correspondent behaviour being tested, information about the benchmarks and if the expected results were able to be obtained. The behaviour column denotes a brief explanation of what the correspondent benchmark consists of. The information column gives more detailed information about the number of streams, the maximum loop nesting and the type of memory access pattern present in the correspondent benchmark.

Table 5.1: Benchmarks Executed with the defined Stream-based Extension

Benchmarks	Behaviour	Information			Spike Support
		# streams	Loop Nesting	Mem Acc Pattern	
SimpleStreamWordExe	Memory Copy to test the correct configuration of simple single-dimension streams with Word data width	2	1	1D	✓
SimpleStreamHalfExe	Memory Copy to test the correct configuration of simple single-dimension streams with Half-Word data width	2	1	1D	✓
SimpleStreamDoubleExe	Memory Copy to test the correct configuration of simple single-dimension streams with Double-Word data width	2	1	1D	✓
SimpleStreamByteExe	Memory Copy to test the correct configuration of simple single-dimension streams with Byte data width	2	1	1D	✓
1DimStreamStrideExe	Memory Copy to test the correct configuration of single-dimension streams with stride different than one	2	1	1D	✓
SignedAddExe	Signed addition to test the configuration and iteration of single-dimension streams	3	1	1D	✓
FloatingPointAddExe	Floating point addition to test the configuration and iteration of single-dimension streams	3	1	1D	✓
2DimStreamExe	Memory Copy to test the correct configuration of simple 2-Dimensional streams	2	2	2D	✓

Continued on the next page

Continuation					
Benchmarks	**Behaviour**	**Information**			**Spike Support**
		# streams	**Loop Nesting**	**Mem Acc Pattern**	
2DimStream-RectScatter-Exe	Memory Copy to test the correct configuration of 2-Dimensional streams denoting the complex Rectangular-Scattered Pattern	2	2	2D	✓
3DimStream-Exe	Memory Copy to test the correct configuration of 3-Dimensional streams	2	3	3D	✓
SimpleDim-CoupleExe	Inner loop processing with outer loop memory access to test Vector-Coupled Streaming	2	2	2D	✓
LowerTrian-gularExe	Memory Copy to test the correct configuration of modifiers denoting the complex Lower Triangular Pattern	2	2	2D + MOD	✓
ComplexDim-CoupleExe	Inner loop processing with outer loop memory access in conjunction with Lower Triangular access pattern to test Vector-Coupled Streaming with modifier	2	2	2D + MOD	✓
3mm	3 Matrix Multiplications using streams	3	3	4D	✓
covariance	Covariance Computation using streams	7	3	4D + MOD	✓
jacobi-1d	1-D Jacobi stencil computation using streams	4	1	1D	✓
jacobi-2d	2-D Jacobi stencil computation using streams	6	2	2D	✓
gemm	Matrix-multiply C=alpha.A.B+beta.C using streams	4	2	4D	✓
gemver	Vector Multiplication and Matrix Addition using streams	16	2	2D	✓
mvt	Matrix Vector Product and Transpose using streams	8	2	2D	✓
saxpy	Single Precision A times X plus Y using streams	3	1	1D	✓
seidel-2d	2-D Seidel stencil computation using streams	10	2	2D	✓
trisolv	Triangular solver using streams	5	2	2D + MOD	✓

5.3.2 Benchmark Results

Spike is deterministic in its executions, meaning that either the code is successfully executed and the outputted results match the expected, or the code is unsuccessfully executed and the final results do

not match the expected. This characteristic was rather useful in the previous subsection to prove that the defined stream-based extension was successfully extended on the Spike Simulator. However, to test the actual performance enhancements brought by the defined stream-based extension, Spike does not give enough details to work around. As so, the only inherent way to prove the improvements in using the defined extension is to analyse the reduction of instructions needed to execute certain benchmarks. As mentioned, the goal of the defined extension is to increase performance in memory access, by reducing the processor's workload associated with memory addressing/indexing. Therefore, a simple analysis of the number of instructions needed to attain the total number of memory addressing/indexing actions in a respective benchmark already provides enough conclusions to measure performance enhancement.

Based on what was stated, a decent amount of benchmarks were analysed to count the number of executed instructions needed with and without the use of the defined stream-based extension. As so, two benchmark analyses will be denoted. In the first analysis, the number of instructions needed to execute the Memory Copy benchmark will be compared between using only the RVV extension and using the RVV extension plus the defined steam-based extension. In the second analysis, the number of instructions needed to execute the Memory Copy benchmark will be compared between using only the scalar RISC-V instructions and using the scalar RISC-V instructions plus the defined steam-based extension.

Starting by the first analysis, Listing 5.1 and Listing 5.2 respectively denote the Memory Copy kernel using only the RVV extension and the Memory Copy kernel using the RVV extension and the stream-based extension.

Listing 5.1: Memory Copy Kernel with RVV

```
1  asm volatile (
2  "mv %[dest], %[result]\n"
3  "loop:\n"
4      "vsetvli t0, %[n], e8, m1, ta, ma\n"
5      "vle8.v v0, (%[src])\n"
6      "add %[src], %[src], t0\n"
7      "sub %[n], %[n], t0\n"
8      "vse8.v v0, (%[dest])\n"
9      "add %[dest], %[dest], t0\n"
10 "bnez %[n], loop\n"
11
12 :: [result] "=r" (result),
13 [src] "+r" (src), [dest] "+r" (dest),
14 [n] "+r" (n)
15 );
```

Listing 5.2: Memory Copy Kernel with RVV and the stream-based extension

```
1  asm volatile(
2      "li s10, 8 \t\n "
3      "scrt.ld.b v1, %[src], %[sn], %[one] \t\n"
4      "scrt.st.b v30, %[dest], %[sn], %[one] \t\n"
5
6      :: [src]"r"(src), [dest]"r"(dest),
7      [sn]"r"(n), [one]"r"(1)
8  );
9  asm volatile(
10     "loop: \t\n"
11         "vsetvli t0, s10, e8, m1, ta, ma \t\n "
12         "vmv.v.v v30, v1 \t\n"
13     "sb.nc v1, loop \n\t"
14 );
```

Comparing both kernels, while in Listing 5.1 the memory accesses are done inside the loop by executing the load and store instructions, in Listing 5.2 the memory accesses are pre-configured by the use of the stream-based extension before the loop. Imagine that the vector length used is 8 and that the number of elements to be copied is 64. To copy all elements from the source to the destination, in both kernels the loop needs to be executed 8 times. Since in Listing 5.1, the loop has 7 instructions including the branch, the total number of instructions needed to execute the code would be $1 + 8 \times 7 = 57$. On the other hand, in Listing 5.2, the loop has 3 instructions including the branch, so the total number of instructions would be $3 + 8 \times 3 = 27$. From this simple made-up example, the enhancement delivered by the defined stream-based in terms of executed instructions is already being highlighted. Figure 5.2 denotes the real number of instructions executed in Spike when executing two of the studied benchmarks. In this case, the numbers show the difference between executing the benchmarks using only the RVV extension and using the RVV extension plus the defined stream-based extension. The numbers represented make reference to an execution where the vector length was 8 and the number of elements to be processed was 256.

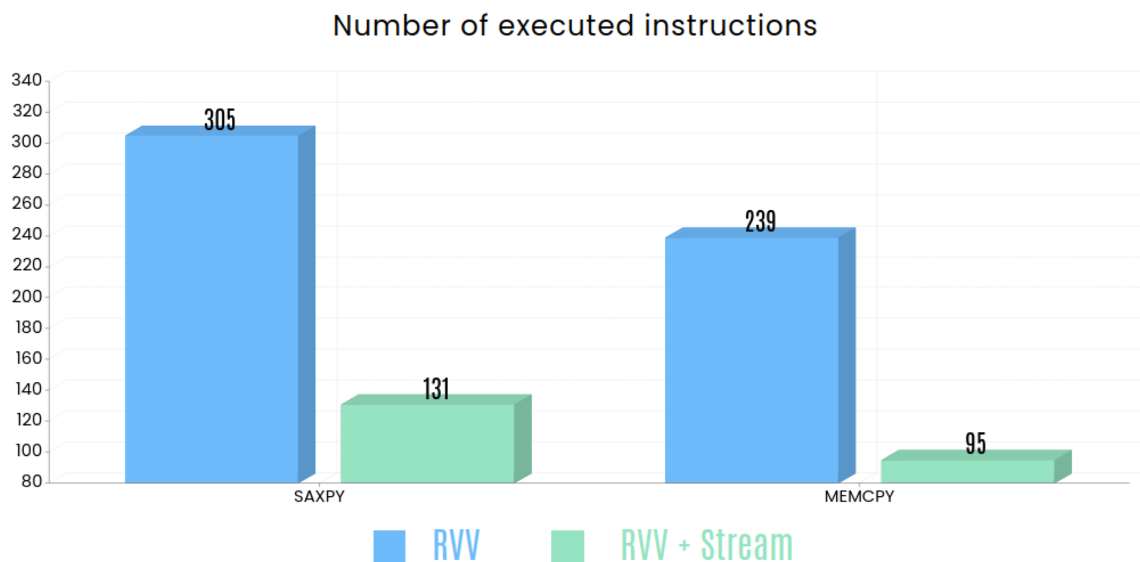


Figure 5.2: RVV versus RVV + Stream.

Comparing the numbers presented in Figure 5.2, the efficiency delivered in terms of the number of executed instructions from using the defined stream-based extension in conjunction with the RVV extension is clearly highlighted.

Moving on to the second analysis, Listing 5.3 and Listing 5.4 respectively denote the Memory Copy kernel using only scalar RISC-V instructions and the Memory Copy kernel using scalar RISC-V instructions and the stream-based extension.

Listing 5.3: Memory Copy Kernel with scalar RISC-V instructions

```
asm volatile (
"1:\n"
    "lb %[a0], 0(%[src])\n"
    "sb %[a0], 0(%[dest])\n"
    "addi %[src], %[src], 1\n"
    "addi %[dest], %[dest], 1\n"
    "addi %[n], %[n], -1\n"
"bnez %[n], 1b\n"
:: [n] "+r" (n), [src] "+r" (src),
[dest] "+r" (dest)
);
```

Listing 5.4: Memory Copy Kernel with scalar RISC-V instructions and the stream-based extension

```
asm volatile(
    "scrt.ld.b a1, %[src], %[sn], %[one] \t\n"
    "scrt.st.b a30, %[dest], %[sn], %[one] \t\n"

    :: [src]"r"(src), [dest]"r"(dest),
    [sn]"r"(n), [one]"r"(1)
);
asm volatile(
    "loop: \t\n"
        "mv a30, a1 \t\n"
    "sb.nc a1, loop \n\t"
);
```

Comparing both kernels, once again, while in Listing 5.3 the memory accesses are done inside the loop by executing the load and store instructions, in Listing 5.4 the memory accesses are pre-configured by the use of the stream-based extension before the loop. Imagine that the number of elements to be copied is 64. To copy all elements from the source to the destination, in both kernels the loop needs to be executed 64 times. Since in Listing 5.3, the loop has 6 instructions including the branch, the total number of instructions needed to execute the code would be $64 \times 6 = 384$. On the other hand, in Listing 5.4, the loop has 2 instructions including the branch, so the total number of instructions would be $2 + 64 \times 2 = 130$. Once more, from this simple made-up example, the enhancement delivered by the defined stream-based in terms of executed instructions is denoted. Figure 5.3 denotes the real number of instructions executed in Spike when executing two of the studied benchmarks. In this case, the numbers show the difference between executing the benchmarks using only the scalar RISC-V instructions and using the scalar RISC-V instructions plus the defined stream-based extension. The numbers represented make reference to an execution where the number of elements to be processed was 64.

Comparing the numbers presented in Figure 5.3, the efficiency delivered in terms of the number of executed instructions from using the defined stream-based extension in conjunction with the scalar RISC-V instructions is clearly highlighted.

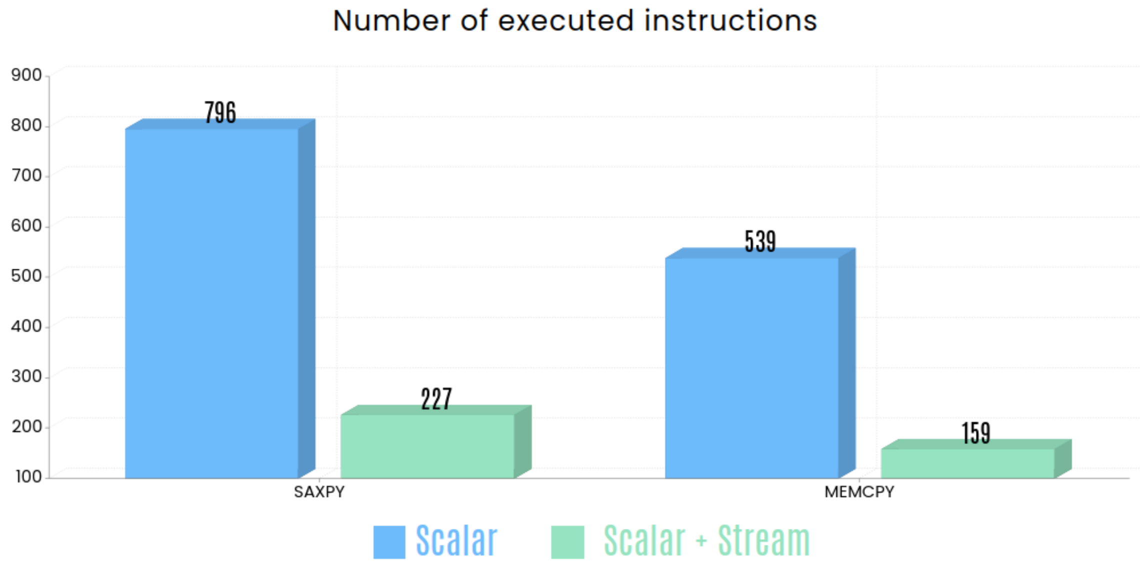


Figure 5.3: Scalar versus Scalar + Stream.

Regarding the examples referenced so far, it is obvious that the defined stream-based extension delivers a real enhancement in terms of code efficiency, as the number of executed instructions in a correspondent benchmark is significantly reduced by the use of the defined extension. Moreover, it is also stated that with the increase in the number of elements to be processed, the delivered enhancement also increases. This statement is based on the fact that by using the stream-based extension, the instructions associated with memory addressing/indexing are successfully removed from the loop processing code.

6

Conclusions and Future Work

Contents

6.1 Conclusions 70
6.2 Future Work 71

6.1 Conclusions

Recently, the number of data streaming techniques implemented into general-purpose processors has increased considerably [5, 19]. This is due to the fact that the performance of memory devices has not improved at the same rate as that of processors, leading to an increase in the gap between Central Processing Unit (CPU) and memory speeds. Recent works show that data streaming applied to a processor memory structure improves memory access performance, both in latency and bandwidth. Moreover, streaming is an excellent opportunity to decouple the memory access procedures from the computational operations.

This work defines a new stream-based extension, whose goal is to increase performance in memory access. This extension counts with a set of specialized stream configuration and manipulation instructions. By decoupling the memory access from the core processing, the defined stream-based extension can remove the instructions associated with control and memory indexation.

Additionally, on the computational side, Single-Instruction-Multiple-Data (SIMD) extensions are becoming predominant in general-purpose processors [1, 10–12]. This is due to the proliferation of certain application domains, such as deep learning and artificial intelligence, with increasing computational performance demands, leaning on the need to further exploit Data-Level Parallelism (DLP). Furthermore, with the emergence of Vector-Length Agnostic (VLA) SIMD extensions [2, 3], each processor implementation can be tuned to achieve any desired SIMD behaviour. One example of these extensions is the recent RISC-V Vector (RVV) extension developed by the RISC-V organization.

The proposed stream-based extension is integrated with the subsets of the RISC-V Instruction Set Architecture (ISA), which means it can interact and interoperate with the RVV extension. By doing so, it is possible to combine the performance enhancement delivered by RVV, adding a way to decrease the memory access latency by using the defined stream-based extension. Furthermore, the defined stream-based extension can also cooperate with the scalar RISC-V instructions.

Attached with the RISC-V paradigm, Spike is a RISC-V ISA simulator, which implements a functional model of one or more RISC-V harts. Despite being recognized as the golden reference ISA simulator by RISC-V, Spike still does not support any extension that accommodates streaming techniques.

This work extended Spike with the defined stream-based extension. By doing so, the functional behaviour of the defined extension was validated.

6.2 Future Work

The defined stream-based extension and the correspondent extended support on the Spike simulator can already be used in the future to enhance certain applications' performance. However, due to the time constraints of the developed work, there were still some possible improvements that were not implemented and are kept for future work. Therefore, further improvements that can be developed into the base of the developed work are:

- To further improve the proposed stream-based extension, it is important to further tune essential characteristics such as the number of configurable descriptors and the available descriptors.
- To make the defined stream-based extension more reliable, it is important to create full compilation support for the extension.
- In the extended Spike Simulator, there were still a few instructions present in the defined extension that were not supported. One example is the support for the dynamic modifier descriptor. It is important to further extend Spike with such modifications to have a fully functional model of the defined extension.
- Spike is a deterministic tool, which does not provide enough details to measure the full enhancements delivery by the defined extension. It is important to further extend the defined extension in other simulation tools.

Bibliography

[1] ARM, “Neon programmer’s guide.”

[2] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, “The arm scalable vector extension,” *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[3] A. Waterman and K. Asanovic, ““risc-v ”v” vector extension”,” 2021.

[4] J. M. Domingos, N. Neves, N. Roma, and P. Tomás, “Unlimited vector extension with data streaming support,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 209–222.

[5] Z. Wang and T. Nowatzki, “Stream-based memory access specialization for general purpose processors,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 736–749.

[6] N. Neves, J. M. Domingos, N. Roma, P. Tomas, and G. Falcao, “Compiling for vector extensions with stream-based specialization,” *IEEE Micro*, 2022.

[7] G. E. Moore *et al.*, “Progress in digital integrated electronics,” in *Electron devices meeting*, vol. 21. Washington, DC, 1975, pp. 11–13.

[8] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted mosfet’s with very small physical dimensions,” *IEEE Journal of solid-state circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[9] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[10] A. Peleg and U. Weiser, “Mmx technology extension to the intel architecture,” *IEEE micro*, vol. 16, no. 4, pp. 42–50, 1996.

[11] S. Thakkur and T. Huff, “Internet streaming simd extensions,” *Computer*, vol. 32, no. 12, pp. 26–34, 1999.

[12] C. Lomont, “Introduction to intel advanced vector extensions,” *Intel white paper*, vol. 23, 2011.

[13] N. C. Crago and S. J. Patel, “Outrider: Efficient memory latency tolerance with decoupled strands,” in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 117–128.

[14] T. J. Ham, J. L. Aragón, and M. Martonosi, “Desc: Decoupled supply-compute communication management for heterogeneous architectures,” in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 191–203.

[15] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, “Imagine: Media processing with streams,” *IEEE micro*, vol. 21, no. 2, pp. 35–46, 2001.

[16] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, “The reconfigurable streaming vector processor (rsvp/spl trade/),” in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 141–150.

[17] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-dataflow acceleration,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 416–429.

[18] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” *ACM SIGARCH computer architecture news*, vol. 44, no. 3, pp. 367–379, 2016.

[19] 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.

[20] R. Organization, “riscv-isa-sim,” https://github.com/riscv-software-src/riscv-isa-sim, January 2023.

[21] ARM, “Instruction set architecture,” in *ARM Glossary*.

[22] R.-V. International, “The risc-v instruction set manual, volume i: User-level isa, version 2.2,” RISC-V International, Tech. Rep., 2020.

[23] R. Bhargava, L. K. John, B. L. Evans, and R. Radhakrishnan, “Evaluating mmx technology using dsp and multimedia applications,” in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1998, pp. 37–46.

[24] A. Strey and M. Bange, "Performance analysis of intel's mmx and sse: A case study," in *European conference on parallel processing*. Springer, 2001, pp. 142–147.

[25] I. Corporation, "Intel sse2 (streaming simd extensions 2)," Intel Corporation, Tech. Rep., 2001.

[26] ——, "Intel sse3 (streaming simd extensions 3)," Intel Corporation, Tech. Rep., 2004.

[27] ——, "Intel sse4 (streaming simd extensions 4)," Intel Corporation, Tech. Rep., 2007.

[28] G. Conte, S. Tommesani, and F. Zanichelli, "The long and winding road to high-performance image processing with mmx/sse," in *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, 2000, pp. 302–310.

[29] I. Corporation, "Intel avx2 (advanced vector extensions 2)," Intel Corporation, Tech. Rep., 2013.

[30] ——, "Intel avx-512 (advanced vector extensions 512)," Intel Corporation, Tech. Rep., 2016.

[31] A. Tanikawa, K. Yoshikawa, T. Okamoto, and K. Nitadori, "N-body simulation for self-gravitating collisional systems with a new simd instruction set extension to the x86 architecture, advanced vector extensions," *New Astronomy*, vol. 17, no. 2, pp. 82–92, 2012.

[32] D. Gregg and A. Forin, "Dynamically-sized vector operations on simd architectures," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 1, pp. 267–275, 1993.

[33] A. Pohl, M. Greese, B. Cosenza, and B. Juurlink, "A performance analysis of vector length agnostic code," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2019, pp. 159–164.

[34] T. Odajima, Y. Kodama, and M. Sato, "Power performance analysis of arm scalable vector extension," in *2018 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*. IEEE, 2018, pp. 1–3.

[35] V. Maisto and A. Cilardo, "A pluggable vector unit for risc-v vector extension," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 1143–1148.

[36] V. Razilov, E. Matúš, and G. Fettweis, "Communications signal processing using risc-v vector extension," in *2022 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 2022, pp. 690–695.

[37] W. A. Wulf, "The memory wall," *ACM SIGARCH Computer Architecture News*, 1995.

[38] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 393–405.

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

[40] D. Graham, "riscv-ovpsim/imperas-riscv-tests," https://github.com/riscv-ovpsim/imperas-riscv-tests, October 2022.

[41] B. Perez, A. Fell, and J. D. Davis, "Coyote: An open source simulation tool to enable risc- v in hpc," in *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021, pp. 130–135.

[42] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. Califor-nia, USA, 2005, pp. 10–5555.

[43] M. W. Michael Clark, Andy Wright, "rv8," https://github.com/michaeljclark/rv8, September 2018.

[44] X. Guo and R. Mullins, "Accelerate cycle-level full-system simulation of multi-core risc-v systems with binary translation," in *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.

[45] Antmicro, "renode," https://github.com/renode/renode, January 2023.

[46] Agra, "risc-vp," https://github.com/agra-uni-bremen/riscv-vp, December 2022.

[47] M. Montón, "A RISC-V SystemC-TLM simulator," in *Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, 2020.

[48] S. Eyck Jentzsch, "Hifive1-vp," https://github.com/Minres/HIFIVE1-VP, November 2022.

[49] R. Organization, "riscv-opcodes," https://github.com/riscv/riscv-opcodes, October 2023.

[50] R.-V. Organization, "riscv-pk," https://github.com/riscv-software-src/riscv-pk, October 2023.

[51] Pouchet Louis-Noël, "Polybench/c - the polyhedral benchmark suite," 2015.

[52] GNU, "Gcc assembler instructions with c expression operands," 2020.