1 2 9 0

UNIVERSIDADE Ð
COIMBRA

Ana Beatriz Simões Fernandes

A functional validation framework for the Unlimited Vector Extension

Dissertation in the context of the Master in Electrical and Computer Engineering, specialisation in Computers, advised by Prof. Doctor Gabriel Falcão Paiva Fernandes and Prof. Doctor Nuno Filipe Simões Santos Moraes da Silva Neves and presented to the Department of Electrical and Computer Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

February 2024

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Ana Beatriz Simões Fernandes

A FUNCTIONAL VALIDATION FRAMEWORK FOR THE UNLIMITED VECTOR EXTENSION

Dissertation in the context of the Master in Electrical and Computer Engineering, specialisation in Computers, advised by Prof. Doctor Gabriel Falcão Paiva Fernandes and Prof. Doctor Nuno Filipe Simões Santos Moraes da Silva Neves and presented to the Department of Electrical and Computer Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

Examination Committee

Chairperson: Prof. Doctor Fernando Manuel dos Santos Perdigão
Supervisor: Prof. Doctor Gabriel Falcão Paiva Fernandes
Member of the Committee: Prof. Doctor Vítor Manuel Mendes da Silva

February 2024

BibTEX:

```
@mastersthesis{absf_msc_thesis,
    author      = {Ana Beatriz Simões Fernandes},
    title       = {A functional validation framework for the Unlimited Vector Extension},
    school      = {University of Coimbra},
    year        = {2024},
    month       = feb,
    keywords    = {ISA SIMD Extensions, Data Streaming, RISC-V, Unlimited Vector Extension,
    ↪  Simulation Tools}
}
```

Neo, sooner or later you're going to realise just as I did that there's a difference between knowing the path and walking the path.

Morpheus, *The Matrix*

This work was financially supported by Instituto de Telecomunicações through projects "Bin-NET: PiM-Enabled Binary Neural Network Inferencing at the Edge" (EXPL/EEI-HAC/1511/2021) and "UNIFY: Compilation Abstraction and Hardware Adaptation for Specialized and General-Purpose Computing Unification" (2022.06780.PTDC), funded by the Portuguese Foundation for Science and Technology (FCT).

Agradecimentos

Aos meus orientadores Prof. Doutor Gabriel Falcão e Prof. Doutor Nuno Neves, deixo um profundo obrigado por me terem proporcionado a oportunidade de trabalhar neste tema que tanto me entusiasma.

Ao Prof. Doutor Pedro Tomás, ao Prof. Doutor Nuno Roma e ao Luís Crespo, membros do grupo HPCAS do INESC-ID, agradeço por toda a ajuda e conhecimento que partilharam comigo.

Ao Instituto de Telecomunicações de Coimbra, que apoiou este projeto através da atribuição das bolsas de investigação B-0124-22 (EXPL/EEI-HAC/1511/2021) e B-0144-23 (2022.06780.PTDC), financiadas pela Fundação para a Ciência e Tecnologia (FCT), deixo também o meu agradecimento.

À Quantunna e todos os amigos que lá fiz, agradeço por me terem mostrado o que é viver Coimbra e por terem dado música à minha vida académica.

A todos os amigos que fiz na faculdade, deixo um agradecimento por terem feito parte da minha jornada. Cada um de vocês deixou a sua marca, foi um gosto partilhar estes anos de estudante convosco.

Aos amigos de sempre, Gabriel, Ju e Lima, que há tantos anos me apoiam em qualquer coisa que faça, deixo um obrigado por nunca terem deixado o meu lado. É um gosto ver como crescemos juntos e tornamos os nossos sonhos realidade. Que a distância nunca nos separe e que continuemos a partilhar as nossas vitórias e derrotas.

Ao Bruno, que acompanhou de perto este trabalho, ouvindo as minhas ideias e desabafos, deixo um enorme obrigado, por todo o carinho e compreensão. Obrigada por sempre teres mostrado interesse e orgulho no que faço, que me incentivou a dar o meu melhor. Ver o que conquistamos juntos no último ano é a melhor recompensa e não podia estar mais grata por te ter ao meu lado.

À Patrícia, a melhor colega de casa, amiga e irmã que poderia ter, agradeço profundamente. Sem ti não estaria aqui, és a maior inspiração e motivaste-me sempre a dar o meu melhor. Obrigada por sempre me teres ouvido e acalmado nos meus momentos mais difíceis. Quando para mim tudo parece impossível, trazes-me sempre uma perspetiva mais otimista, mas sempre realista, de ver as coisas.

Aos meus pais, a quem devo tudo o que sou hoje, deixo o meu maior obrigado. À minha mãe, obrigada por todas as palavras de encorajamento e por ser a melhor ouvinte de todos os meus desabafos. Ao meu pai, agradeço por me ter mostrado tudo o que sei sobre o que é ser Engenheiro, mesmo sem diploma. Obrigada por terem sempre acreditado em mim e incentivado a que dê o meu melhor em tudo o que faço, tal como vocês.

Abstract

In order to tackle the limitations of current state-of-the-art Vector-Length Agnostic (VLA) extensions, the RISC-V Unlimited Vector Extension (UVE) was created. This is a new Instruction Set Architecture (ISA) extension that aims to reduce loop control and memory access indexations associated overheads, decreasing the average memory access latency. It achieves this by relying on Single Instruction, Multiple Data (SIMD) processing and the emerging data streaming paradigm. SIMD is ideal for data-centric applications, such as Machine Learning (ML), which are increasingly more popular in embedded and low-power devices. On the other hand, data streaming allows for memory access patterns to be described at the software level, fed to a dedicated co-processor that fetches the data in the background, effectively decoupling and masking memory accesses from computation. An initial proof-of-concept implementation of this extension was made on an Out-of-Order (OoO) processor model, based on the ARM Cortex-A76, in the cycle-accurate simulator *gem5*. Compared with the state-of-the-art Scalable Vector Extension (SVE), results showed that the proposed solution attains performance speedups between 2$\times$ and 4$\times$. Since the initial proposal of UVE, some shortcomings and limitations were identified, such as a lack of support for more fields of application, such as Sparse Linear Algebra, which is characterised by complex scatter-gather memory accesses. Moreover, there was a need for an improved simulation environment, as the *gem5* implementation has since been deprecated. This work presents a new development and functional validation framework for the extension, based on the *Spike* simulator, the golden reference functional RISC-V ISA simulator. This simulator was modified and extended to support data streaming and more than 150 new instructions. In parallel, the extension's specification was revised and updated, to support higher memory access pattern complexity and cover new fields of application. The new simulation framework was used to perform a functional validation of the extension and a comparison with the state-of-the-art RISC-V Vector Extension (RVV). This assessment revealed an average instruction reduction of 75% relative to RVV.

Keywords

ISA SIMD Extensions, Data Streaming, RISC-V, Unlimited Vector Extension, Simulation Tools.

Resumo

Para fazer face às limitações das atuais extensões vetoriais escaláveis, uma nova extensão para RISC-V foi criada: Unlimited Vector Extension (UVE). Esta é uma nova extensão do Instruction Set Architecture (ISA) que visa reduzir o impacto do controlo de fluxo e indexação, diminuindo a latência média de acesso à memória. Isto é conseguido através do processamento de instrução única em múltiplos dados (SIMD, em inglês) e do paradigma emergente de computação em fluxo de dados (*data streaming*, no original). Este tipo de processamento vetorial é ideal para aplicações com elevado volume de dados, como Aprendizagem Computacional, cada vez mais populares em dispositivos embebidos e de baixo consumo. Por outro lado, o fluxo de dados permite que os padrões de acesso à memória sejam descritos ao nível do *software* e alimentados a um co-processador dedicado, que carrega os dados em segundo plano, desassociando os acessos à memória da computação. Uma primeira implementação de prova de conceito desta extensão foi efetuada num modelo de processador fora de ordem, baseado no ARM Cortex-A76, no simulador *gem5*. Em comparação com uma extensão estabelecida no estado-da-arte, Scalable Vector Extension (SVE), os resultados mostraram que a solução proposta alcança aumentos de desempenho entre 2× e 4×. Desde a proposta inicial do UVE, foram identificadas algumas falhas e limitações, tais como a falta de suporte para mais áreas de aplicação, como a Álgebra Linear Esparsa, que se caracteriza por acessos dispersos à memória. Além disso, havia a necessidade de um ambiente de simulação melhorado, uma vez que a implementação do *gem5* foi entretanto descontinuada. Este trabalho apresenta uma nova estrutura de desenvolvimento e validação funcional para a extensão, baseada no simulador *Spike*, o simulador funcional de referência para RISC-V. Este simulador foi modificado e alargado para suportar fluxo de dados e mais de 150 novas instruções. Paralelamente, a especificação da extensão foi revista e atualizada para suportar uma maior complexidade de padrões de acesso à memória e abranger novas áreas de aplicação. O novo ambiente de simulação foi utilizado para efetuar uma validação funcional da extensão e uma comparação com a extensão vetorial do RISC-V, RISC-V Vector Extension (RVV). Esta avaliação revelou uma redução média do número de instruções no valor de 75% em relação ao RVV.

Palavras-Chave

Extensões ISA SIMD, Computação em Fluxo de Dados, RISC-V, Simulação.

Contents

Glossary

AI Artificial Intelligence.

ALU Arithmetic Logic Unit.

AR Augmented Reality.

AVX Advanced Vector Extensions.

CISC Complex Instruction Set Computer.

CPU Central Processing Unit.

CSR Control Status Register.

DLP Data-Level Parallelism.

DSP Digital Signal Processing.

EOD End-of-Dimension.

EOS End-of-Stream.

FIFO First-In, First-Out.

GCC GNU Compiler Collection.

GPU Graphics Processing Unit.

HPC High-Performance Computing.

ILP Instruction-Level Parallelism.

IOT Internet of Things.

ISA Instruction Set Architecture.

ISS Instruction Set Simulator.

ML Machine Learning.

MMU Memory Management Unit.

OoO Out-of-Order.

PC Program Counter.

RAT Register Alias Table.

RISC Reduced Instruction Set Computer.

RTL Register Transfer Level.

RVV RISC-V Vector Extension.

SAT Stream Allocation Table.

SCROB Stream Configuration Reorder Buffer.

SE Streaming Engine.

SIMD Single Instruction, Multiple Data.

SIMD&FP Single Instruction, Multiple Data and Floating-Point.

SISD Single Instruction, Single Data.

SSE Streaming SIMD Extensions.

SU Streaming Unit.

SVE Scalable Vector Extension.

TLB Translation Lookaside Buffer.

UVE Unlimited Vector Extension.

VLA Vector-Length Agnostic.

VR Virtual Reality.

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In the last few decades, there has been an increasing need to improve the performance of a processor. This is due to computational and memory-intensive applications having become more common, with the rise of Machine Learning (ML), Augmented Reality (AR), Virtual Reality (VR), among other technological developments, involving image and video processing with increasing resolution. However, with the end of Dennard Scaling and the presumed slowdown of Moore's Law, traditional methods solely based on increasing clock frequency and the use of cache memory have been revealed to be insufficient for improving performance.

Several solutions have been proposed and are now widely used, such as Instruction-Level Parallelism (ILP) and Data-Level Parallelism (DLP), common in modern high-performance processors. The latter, hidden in Single Instruction, Multiple Data (SIMD) units [1], has proven to be successful in improving the performance of modern processors by allowing the simultaneous processing of multiple data elements. To take advantage of DLP, a plethora of SIMD Instruction Set Architectures (ISAs) has been developed, such as Arm Neon [2] and x86 AVX [3]. However, these approaches present some limitations, such as portability issues, because they exclusively operate on fixed-size registers. Although increasing vector size was a tendency for some time [1, 4], Vector-Length Agnostic (VLA) extensions have emerged to overcome this problem, particularly RISC-V Vector Extension (RVV) [5] and Scalable Vector Extension (SVE) [6], which allow for the size of the vector register to be defined at runtime. This means that different processors with different requirements can adopt distinct vector sizes, with no need to modify application source code. However, a new problem arises with these extensions, as predicate [7] and/or vector control instructions become necessary to disable elements outside loop bounds. This can lead to additional loop instructions [8], and thus more overhead and lower attainable performance.

The RISC-V Unlimited Vector Extension (UVE), proposed and developed by Domingos et al. [9], joins two promising solutions for improving performance:

scalable SIMD extensions and data streaming. RISC-V was chosen as the base ISA due to its open-source nature, as well as its simple and extensible instruction set. By relying on data streaming, this novel RISC-V ISA extension contains several upgrades compared to the ones mentioned above, such as decoupled memory accesses, indexing-free loops, simplified vectorisation, and implicit load/store operations [9]. The streaming paradigm allows for the configuration of memory access patterns at the software level and the fetching of the data in the background, a clear step towards increasing memory access latency and increasing throughput. This work has already demonstrated the paradigm shift in a proof-of-concept *gem5* implementation of the UVE on an Out-of-Order (OoO) processor model, based on the ARM Cortex-A76. It was shown that the performance of a processor can be improved by an average of 2.4× (speed-up), compared to other state-of-the-art implementations.

1.1 Motivation and Objectives

UVE is still in its early stages of development. Its initial proposal lacked support for several applications that involve complex memory accesses, mainly in the Sparse Linear Algebra domain, due to constraints in its pattern descriptors. Moreover, because the ISA was only validated on *gem5*, an independent functional validation of the specification was required, free from the constraints imposed by this simulator. In accordance, this work aims to create a new modelling, simulation, and validation tool to support the development of UVE, by not only independently validating the existing specification, but also introducing streaming support on *Spike* [10]. By creating a functional validation tool, it becomes possible to focus solely on the instructions' behaviour, detaching the ISA development from implementation details, which can be prone to errors. In fact, while adding this new extension to *Spike*, this work also aims to identify and correct any errors that may exist in the current UVE specification, as well as to extend it with new instructions and features, to widen its applicability.

Additionally, so that the features added on *Spike* can be tested and validated, a diverse set of benchmarks is required, composed of commonly found operations and compute kernels in the signal and image processing domains, which are great candidates for UVE applications. Some benchmarks were already considered in the UVE's proposal [9], so one goal of this work is to revise the available code to benefit from whichever corrections and improvements are made to the specification. Furthermore, new benchmarks from fields of application that were not previously considered should be added to demonstrate new features.

Finally, as the second main iteration of UVE is proposed, it is important to create comprehensive documentation of its specification, to both aid its future development and ease its adoption. This document aims to therefore include a detailed description of the functional behaviour of each instruction, as well as an account of the streaming mechanisms and the memory access patterns supported by the UVE, including old and new features. As the chosen simulator also lacks documentation, its alteration process is also detailed, so that it can be easily reproduced and extended by future developers.

The general overview of the flow of the presented work is illustrated in Figure 1.1.

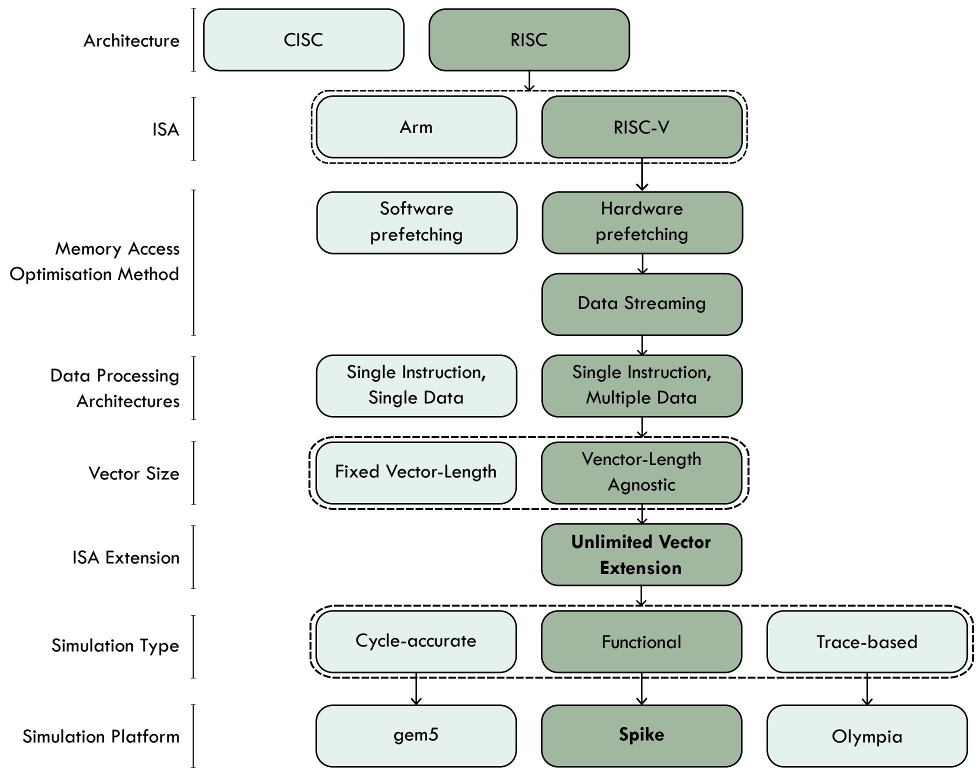


Figure 1.1: Outline of the flow of the presented work.

1.2 Contributions

This work resulted in several artifacts that contribute to the development and validation of the UVE ISA. The main contributions are:

- A revised and improved UVE specification, which includes the correction of errors and the addition of new instructions and features, to widen its applicability.
- A new UVE implementation on *Spike*, the main reference for RISC-V development and validation, which includes data streaming support for over 150 instructions, implemented, tested, and validated.

- A diverse set of benchmarks composed of commonly found High-Performance Computing (HPC) operations and compute kernels that has been revised and expanded to demonstrate the corrections and improvements made to the UVE specification.
- A new functional development and validation framework for UVE, whose main component is the *Spike* simulator, complemented by the validation benchmarks and testing scripts, including support for instruction counting. This material was all made publicly available[1].
- Comprehensive documentation of the UVE specification, which includes a detailed description of the functional behaviour of each instruction.
- An evaluation of the instruction count improvements of UVE over RVV, the official vector extension of the base ISA, RISC-V.

During the development of this work, some preliminary results included in Appendix D were presented at:

- Ana Beatriz Fernandes, Nuno Neves, Luís Crespo, Pedro Tomás, Nuno Roma and Gabriel Falcao (University of Coimbra), *"A functional validation framework for the Unlimited Vector Extension"*, CAMS 2023 - The 1st Workshop on Computer Architecture Modeling and Simulation, hosted by the 56th IEEE/ACM International Symposium on Microarchitecture (MICRO 2023)

  Presented on the 28th of October 2023, Toronto, Canada

1.3 Document Outline

This document starts by offering the necessary background and state-of-the-art in Chapter 2, with an introduction to vector processing, data streaming, and pattern representation concepts that are essential to understanding the UVE extension. Afterwards, Chapter 3 presents the chosen base simulation environment, as well as the modifications and additions that were made to achieve the desired functionality. Chapter 4 details the proposed optimised UVE specification, with the identification of caveats in the original extension that lead to the performed revision. In Chapter 5, the complete framework is presented and used to obtain an ISA performance evaluation. In this chapter, the UVE extension is compared to the official vector extension of RISC-V, RVV, recurring to a set of meaningful benchmarks. Finally, Chapter 6 summarises the work and presents the conclusions and potential future work.

[1]`https://github.com/hpc-ulisboa/UVE2`

Chapter 2

Background and State of the Art

To understand the Unlimited Vector Extension (UVE) [9] and how it improves on other proposed extensions, it is necessary to comprehend other Single Instruction, Multiple Data (SIMD) technologies that preceded it. Furthermore, some context surrounding the data streaming paradigm and memory access pattern description methods is required. This section covers the fundamental concepts of these areas. Lastly, it gives an overview of UVE and shows how it combines or improves on the technologies mentioned above.

2.1 SIMD Architectures and Fixed-Length Vector Extensions

Data-Level Parallelism (DLP) is exploited in many ways in most modern computers. From vector processors, such as the Cray series, to the ubiquitous Graphics Processing Units (GPUs), the idea of executing the same operation on multiple data elements is not new. While these represent specialised computer architectures, it is also possible for the Central Processing Unit (CPU) to take advantage of DLP. This is done through SIMD units made of wide Arithmetic Logic Units (ALUs) that allow for the simultaneous processing of multiple data elements [1]. For this purpose, several SIMD Instruction Set Architecture (ISA) extensions have been developed, at first focusing on operating on fixed-length vector registers. This section describes the most relevant ones, both for Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC) architectures.

2.1.1 Arm Neon and Helium

Intended for accelerating Digital Signal Processing (DSP), Arm Neon technology is the packed Advanced SIMD architecture extension for the A (application) and R-profile (real-time) RISC processors developed by ARM for Armv8 and subsequent implementations [2]. Compiler autovectorisation support is available, as are Neon intrinsics, which are function calls the compiler appropriately replaces with Neon instructions, giving the programmer low-level control over the vectorisation and Neon operations from C/C++ code.

This extension operates on the Single Instruction, Multiple Data and Floating-Point (SIMD&FP) register file of 32 registers that are 128 bits wide. The SIMD unit is fully integrated into the processor, sharing its resources for integer operations, loop control, and caching. This is translated into a considerable reduction of area and power cost, as opposed to a dedicated hardware accelerator [2]. Each register can be divided into lanes, with size depending on the data type that it holds (resulting in a vector of elements). Particularly, when accessed as a vector register by a Neon SIMD instruction, a SIMD&FP vector register may contain:

- Sixteen 8-bit (byte) elements;
- Eight 16-bit (half-word) elements;
- Four 32-bit (word) elements;
- Two 64-bit (double-word) elements.

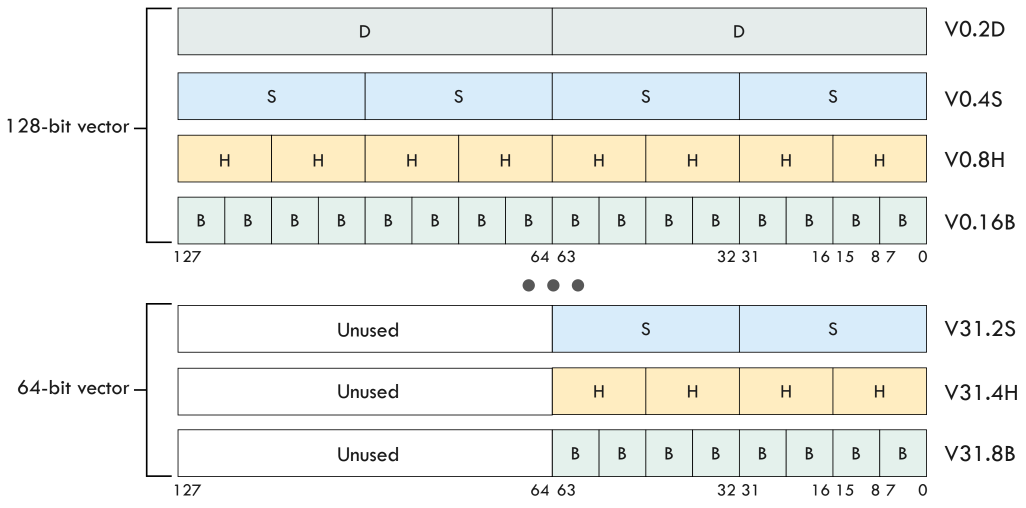
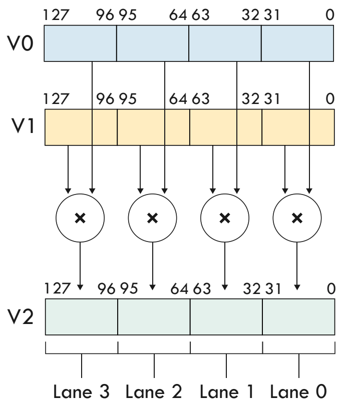


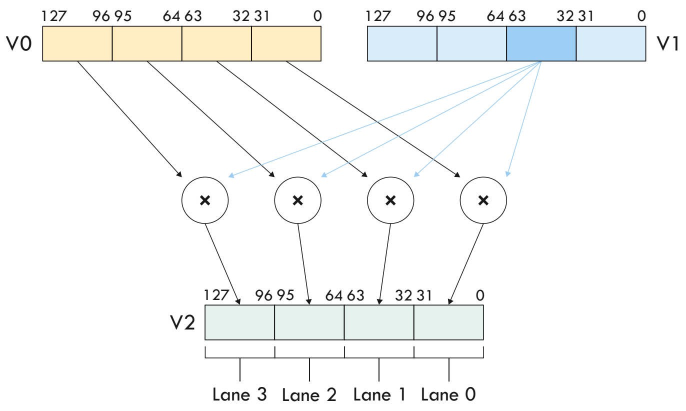
Figure 2.1: Neon 128 and 64-bit vector registers, adapted from [2].

The upper half of the register (64 bits) may also be cleared to zero, so it can behave as a 64-bit vector register. Figure 2.1 shows the layout of Neon registers, where each suffix indicates the number of lanes and the element width (*D* for double-word, *S* for word, *H* for half-word and *B* for byte).

A simple element-wise product can be used as an example to understand how vector operations work. Each lane of a vector, holding an element, is multiplied by the corresponding lane of the other, storing the result in the same lane of the destination vector, as represented in Figure 2.2a. Operations with scalars are also available, such as the example instruction `MUL V2.4S, V0.4S, V1.S[2]`, which multiplies each element of the first operand with the element in index 2 of the second operand, as shown in Figure 2.2b.



(a) Vector-vector multiplication



(b) Vector-Scalar multiplication

Figure 2.2: Neon SIMD multiplication operations, adapted from [2].

It should be noted that this is how most SIMD instructions work, not exclusively on Neon. Furthermore, ARM also provides the Helium extension, which is very similar but targets M-profile architectures, which are meant to be used in embedded systems [11]. Some key Helium features that differ from Neon are listed below.

- Fewer vector registers, but some instructions can access vector and scalar registers simultaneously;
- Loop and lane predication, as well as complex math operations and scatter-gather memory accesses;
- Optimised for low-power and small-area implementations, maximising the usage of all available hardware resources.

2.1.2 Intel SSE and AVX

On the CISC side, targeting x86 architectures, there are two main SIMD extensions: Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX), proposed by Intel but also used by AMD [12]. Before these two groups of extensions emerged, there was MMX, commercialised by Intel in 1997, which introduced integer SIMD instructions. However, because its 64-bit registers were aliased upon the x86 floating-point stack, executing floating-point and MMX routines was impossible at that time [13]. AMD later introduced 3DNow! [14], which improved on MMX, targeting mainly the gaming and 3D graphics markets, and already supported floating-point operations. Intel then released SSE, which added eight new independent 128-bit vector registers and supported floating-point operations, besides expanding the MMX instruction set. With its fourth iteration, first released in 2007, SSE4 is supported by both Intel and AMD processors, ultimately replacing MMX and 3DNow!, which have since been deprecated. While specific features vary between AMD and Intel implementations of SSE4, it supports over 350 instructions, including integer and floating-point operations, as well as new instructions for string and text processing [15].

In 2011, Intel released AVX, which increased the vector register size to 256 bits, allowing for the processing of 8 single-precision or 4 double-precision floating-point numbers per instruction, but not supporting integer operations until a later version, AVX2, made available in 2013. Three years later, AVX-512 further increased the vector register size to 512 bits, while also doubling the number of registers, now at 32, and adding many more features [3, 4].

2.1.3 Discussion

In order to fully understand how SIMD extensions work from the perspective of the software developer, a simple code example is given in Listing 2.1. Assuming that 128-bit SIMD registers are used, each vector register can hold four single-precision floating-point values, as they are 32-bits wide. This means that in one software loop iteration, four elements of the `X` and `Y` arrays are processed, as represented in Listing 2.2. This is called *loop unrolling*, a common technique used to increase performance, as it reduces the number of loop iterations (usually costly in terms of performance due to loop control overhead). However, it is not always possible to include all the processing in an unrolled loop, as the number of elements in the array may not be divisible by the vector length (i.e., the packing factor). In such cases, the last elements must be processed separately. This is the case in the example, where the last two elements are processed outside the loop, as the size of each array is `N=18` and is therefore not a multiple of the packing factor, 4.

```c
int N = 18;
void saxpy(float *X, float *Y, float a, int N) {
    for (int i = 0; i < N; ++i)
        Y[i] += X[i] * a;
}
```

Listing 2.1: Original *SAXPY* C/C++ code.

```c
int N = 18, M = N - N % 4;
for (int i = 0; i < M; i+=4) {
    Y[i] += X[i] * a;
    Y[i+1] += X[i+1] * a;
    Y[i+2] += X[i+2] * a;
    Y[i+3] += X[i+3] * a;
}
Y[N-2] += X[N-2] * a;
Y[N-1] += X[N-1] * a;
```

Listing 2.2: *SAXPY* unrolled loop C/C++ code, assuming a packing factor of 4.

The computations inside the resulting loop can be replaced by a single SIMD instruction (besides the necessary load and store instructions) as shown in Listing 2.3. Instructions outside the loop continue to be performed as usual. An assembly-like pseudo-code based on Arm Neon is presented in Listing 2.4 to show the assembled code of a possible implementation of the *SAXPY* function with SIMD technology. As pointed out by Domingos [1], loop control and memory address calculation instructions are undesirably cluttering the loop and make up most of the code. Furthermore, the edge cases also take up a considerable amount of code, which is not ideal and leads to a large overhead.

```c
int N = 18, M = N - N % 4;
for(int i = 0; i < M; i+=4){
    SIMD_LOAD(vY, Y, i)
    SIMD_LOAD(vX, X, i)
    vY = SIMD_ADD(SIMD_MUL(vX, vA), vY) // vA is a vector filled with 'a'
    SIMD_STORE(Y, vY, i)
}
Y[N-2] += X[N-2]*a;
Y[N-1] += X[N-1]*a;
```

Listing 2.3: *SAXPY* SIMD loop pseudo-code.

```
V.DUP_32 V0, R4  ; Loading value of A (R4) into all elements of V0 (vector
 ↪  register) with elements of size 32-bits

Loop:
    ; R2 contains X address, R1 is the indexer i; R5 is the number of
     ↪  iterations N
    VLD     V1, [R2+R1] ; Load 4 elements of 32-bits of the array X to V1
    VMUL_SP V1, V0, V1
    VLD     V2, [R3+R1] ; R3 contains Y address
    VADD_SP V2, V2, V1
    VST     [R3+R1], V2 ; Store the result of the 4 parallel operations
    ADD     R1, 16      ; Increment i by 4 floating point elements (4-bit
     ↪  addressing)
    SUB     R5, #4      ; Decrement number of iterations
    CMP     R5, #4      ; Compare N with 4
BGE Loop ; Repeat until N < 4

LD      R12, [R2+R1]  ; Case Y[n-1]
MUL_SP  R12, R4, R12
LD      R13, [R3+R1]
ADD_SP  R13, R12, R13
ST      [R3+R1], R13

ADD R1, R1, 1

LD      R12, [R2+R1]  ; Case Y[n]
MUL_SP  R12, R4, R12
LD      R13, [R3+R1]
ADD_SP  R13, R12, R13
ST      [R3+1], R13
```

Listing 2.4: *SAXPY* loop assembly pseudo-code for SIMD processing [1].

As such, it is clear how SIMD architectures achieve higher performance in data parallelism-rich applications, as fewer loop iterations are executed when compared to Single Instruction, Single Data (SISD) architectures. This also means that functional units will be less strained and Instruction Memory will be accessed fewer times. There impact of wrong branch predictions is also lower, resulting from the reduced number of loop iterations that leads to less speculative execution.

While the aforementioned extensions are quite powerful, they do not come without limitations. Particularly, the limited vector width is of great concern, as it forces the implementations to adopt the static length specified in the ISA, hindering portability [8]. Furthermore, different applications benefit from different vector widths. For example, High-Performance Computing (HPC) applications achieve higher throughput with wider vectors. On the other hand, low-power processors usually make use of smaller vectors so that power and resource requirements are met [9]. Moreover, complex memory access patterns usually lower the performance of the architecture, which otherwise performs well with linear memory accesses. Lastly, the growth of vector register widths in SIMD extensions means that for every different vector size, a novel extension that supports the new size must be developed. Consequently, application code must be recompiled, which is not always possible with proprietary software, making it obsolete [1].

2.2 Vector-Length Agnostic Extensions

To overcome these drawbacks, Vector-Length Agnostic (VLA) extensions were introduced, and are now state-of-the-art in vector processing, tending to result in more straightforward and efficient code [16]. VLA programming makes it possible to run the same program on different hardware platforms with distinct vector lengths, such as embedded and HPC processors, with small and large vector registers, with no need for re-coding or re-compilation [8].

Two main scalable vector extensions are worth mentioning: the RISC-V Vector Extension (RVV) [5] and the Scalable Vector Extension (SVE)‘[6]. These served as a base for the development of UVE, so their main characteristics are described below.

2.2.1 RISC-V Vector Extension

The RISC-V Vector Extension (RVV)[5] follows the same principles as its base ISA, RISC-V: simplicity and generality. In accordance, vector length constraints are very loose, not limiting any application to a particular configuration. Its specification states that a single vector element has size `ELEN` (unspecified in the specification, but not less than 8 bits) and that a vector register must have length `VLEN` superior to `ELEN`, where both of these values are powers of two and `VLEN` must not exceed 2^{16} [5]. There are 32 vector registers available.

Each vector register must be configured with its element and vector sizes, which can change on runtime. This is accomplished with the `vsetvli`, `vsetivli` and `vsetvl` instructions, as specified in [5]. These instructions allow the modification of the vector length `vl` and vector type `vtype`, two of the seven Control Status Registers (CSRs) added by this extension.

```
vsetvli rd, rs1, vtypei
```

In this instruction, `rd` is the destination register where the new vector length `vl` will also be written, `rs1` is the register where `AVL`, the application vector length, is provided, and `vtypei` is the immediate vector type. Usually, `vl` is set to `AVL`, if it is inferior or equal to the maximum value that can be represented by the vector type, `VLMAX = LMUL*VLEN/SEW`, where `SEW` is the selected element width. In case it is greater than this value, but still inferior to `2*VLMAX`, the resulting vector length will be `AVL/2`, rounded up. Otherwise, `VLMAX` will be used. The "grouping factor", `LMUL`, 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 [1].

The `vtypei` argument is an immmediate which configures the selected element width `SEW` and vector register group multiplier `LMUL`, as shown in Listing 2.5, besides the *vector mask*, through the mandatory `ta/ma` and `ma/mu` arguments, which can be combined to achieve various behaviours. RISC-V V establishes that if a set is marked as *undisturbed*, the corresponding set of destination elements in a vector register group remains unchanged, retaining the value they already had. If *agnostic* is specified, the destination elements can either retain their values or be overwritten with 1s, depending on the implementation. *Tail* elements correspond to those past the set `vl` during execution, and *mask* refers to an optional vector mask, a regular vector register passed with the suffix `.t`, that can be used in some instructions, allowing lane predication on body elements.

```
# SEW
# e8 (8b), e16 (16b), e32 (32b), e64 (64b)

# LMUL
# mf8 (1/8), mf4 (1/4), mf2 (1/2)
# m1  (1, assumed if m setting absent)
# m2  (2), m4 (4), m8 (8)

# Mandatory flags
# ta # Tail agnostic
# tu # Tail undisturbed
# ma # Mask agnostic
# mu # Mask undisturbed

# Examples:
vsetvli t0, a0, e8, ta, ma        # SEW= 8, LMUL=1
vsetvli t0, a0, e8, m2, ta, ma    # SEW= 8, LMUL=2
vsetvli t0, a0, e32, mf2, ta, ma  # SEW=32, LMUL=1/2
```

Listing 2.5: RVV suggested assembler names and examples for `vset{i}vli` instructions [5].

The other two mentioned instructions work similarly but with an immediate value for `AVL`, or `vtype` encoded in a second source register instead of an immediate.

One of the main benefits of RVV is that it does not force the last lanes of the vector (sections of the physical register with the configured sizes) to be populated. This is useful in the case where the register holds fewer values than the vector size, through these vector control mechanisms. To further understand how this extension works, the *SAXPY* example is employed, this time with RVV, in Listing 2.6.

```
# y = a * x + y; x and y are vectors of length n, a is a scalar
# register arguments:
#   a0: n; fa0: a; a1: x; a2: y

saxpy_:
    vsetvli   a4, a0, e32, m8, ta, ma # a4 = new vl, a0 = n, vtypei = new
     ↪  vtype setting
    vle32.v   v0, (a1)      # v0 = x (word)
    sub       a0, a0, a4    # a0 = n - vl (decrement n by vl)
    slli      a4, a4, 2     # a4 = vl * 4 (4 bytes per word)
    add       a1, a1, a4    # a1 = x + vl * 4 (address increment)
    vle32.v   v8, (a2)      # v8 = y (word)
    vfmacc.vf v8, fa0, v0   # v8 = a * x + y (FP multiply and accumulate)
    vse32.v   v8, (a2)      # y = v8 (store result)
    add       a2, a2, a4    # a2 = y + vl * 4 (address increment)
    bnez      a0, saxpy_
```

Listing 2.6: RVV *SAXPY* loop assembly code [5].

It is clear that the vector control instruction is responsible for the loop iteration, determining how many elements can be processed in a single iteration and repeating this process each time, as the total number of values to process, `AVL`, is decremented by the vector length. Once there are no more elements to be processed, the loop halts. Just as in the previous example from Listing 2.3, the loop also contains the necessary instructions for address calculation, memory access, and computation.

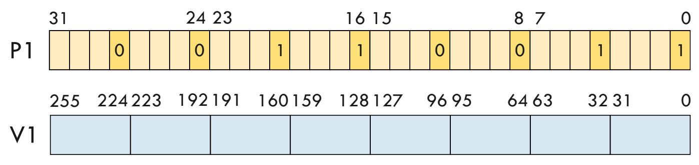
2.2.2 Arm Scalable Vector Extension

Another vector extension worth mentioning is the Arm SVE [6], currently in its second iteration, which adds 32 new scalable registers that extend the already present SIMD register bank (see Section 2.1.1). Each register must have a length between 128 and 2048 bits, in increments of 128 bits, sharing the bottom 128 bits with the fixed-length vector registers introduced by Neon [17]. This extension also adds 16 scalable predicate registers, although only eight are available for arithmetic and memory instructions, reducing the predicate register pressure observed in predicate-centric architectures[6].

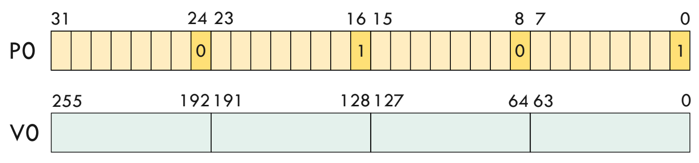
Contrary to what happens in RVV, loop control relies on a predicate register, holding a mask, and not on vector register configuration. This predicate mask indicates for which vector register lanes operations will be executed. Scalability of the vector registers is achieved through the `while` and `inc` instruction families.

```
while{le|lo|ls|lt} pd.<t>, rn, rm
```

Using `whilelt` as an example, this instruction takes the destination predicate register, along with the size specifier (`<t>`), the start value of an iterator in `rn`, and the comparison value in `rm`. This instruction will then populate the predicate register with ones as long as the iterator value is lower than the comparison value. Other similar instructions exist, with different comparison operations. The `inc` instruction family is used to increment the iterator value, taking the iterator register as an argument and incrementing it by the current vector length, so that the next loop iteration performs `whilelt` with the correct starting value, not taking into account already processed elements. The resulting predicate is used to control loop execution (as exemplified in Listing 2.7). With these instructions, there is already a small code reduction inside the loop when compared to RVV (see Listing 2.6).



(a) 32-bit elements (word)



(b) 64-bit elements (double-word)

Figure 2.3: SVE predicate organisation in 256-bit vector registers.

One interesting feature of SVE is that each predicate has eight bits per 64-bit vector element, allowing control down to byte granularity [6]. Because only the least significant bit of the mask element is used for control of the corresponding operand vector element, using multiple data element widths becomes possible. This behaviour is represented in Figure 2.3. Additionally, two predication modes are available, *merging* and *zeroing*, with the former being the default. In the *merging* mode, predicated lanes are left unchanged, while in the *zeroing* mode, predicated lanes are set to zero in the destination vector. This configuration is done through suffixes `/m` and `/z`, respectively, added to the register name, as seen in Listing 2.7.

Lastly, it should be mentioned that SVE2 adds several important features to its predecessor, Neon, namely gather-load and scatter-store instructions, per-lane predication, and speculative vectorisation [17, 18]. The main difference between SVE and SVE2 is purely functional, with the latter spanning a wider range of applications, not limiting itself to HPC and Machine Learning (ML) workloads [17]. Despite improving on Neon, these extensions are only related at hardware level. Finally, it is worth noting that this extension can also be used through intrinsic functions, just like Neon.

```
# y = a * x + y; x and y are vectors of length n, a is a scalar
# register arguments:
#   x0: x; x1: y; x2: a; x3: n

saxpy_:
    ldrsw   x3, [x3]          // x3 = n
    mov     x4, #0            // x4 = 0 (iteration counter)
    ld1rw   z0.s, p0/z, [x2] // p0 = vector filled with a, zeroing
     ↪  predication
.loop:
    whilelt p0.s, x4, x3                  // p0 = while(i++ < n)
    ld1w    z1.s, p0/z, [x0, x4, lsl #2] // p0: z1 = x[i]
    ld1w    z2.s, p0/z, [x1, x4, lsl #2] // p0: z2 = y[i]
    fmla    z2.s, p0/m, z1.s, z0.s       // p0? z2 += x[i]*a
    st1w    z2.s, p0, [x1, x4, lsl #2]   // p0? y[i] = z2
.cond:
    incw    x4    // i += (VL/32)
    b.first .loop // more to do?
    ret
```

Listing 2.7: SVE *SAXPY* loop assembly code [6, 19].

2.2.3 Discussion

Although both RVV and SVE have their advantages, several shortcomings can be identified. While they greatly improve on fixed-length SIMD extensions and mitigate some of the highlighted issues, such as loop edge case control, this comes at the cost of a large instruction overhead, related to memory indexing, loop control, and memory access, which antagonise the improvement of data processing throughput [9].

In both cases, configuration and memory indexing/loop control instructions (e.g., incrementing indexing variables) dominate the loop code, which is undesirable. While these solutions typically reduce the number of loop iterations, there is room for improvement in this behaviour. Although memory address calculation and update of indexing variables may seem indispensable, several works decouple memory accesses from computation, achieving promising results [20–27]. Furthermore, both hardware and software prefetching mechanisms have been employed to improve memory access latency [28–42]. It is with this in mind that the analysis of both code snippets (Listing 2.6 and Listing 2.7) leads to the conclusion that each loop only has one actually useful instruction, excluding branches, and everything else should ideally be moved elsewhere. This is the main motivation behind UVE, which relies on data streaming to preempt the acquisition of the necessary data in the correct order, leaving the loop only to perform the necessary computation. Further analysis and comparison between these extensions is presented in Section 2.4.5.

2.3 Data Streaming and Pattern Description

Traditional memory access description is achieved through the use of loops and standard instructions, to calculate the correct sequence of memory addresses that are then loaded/stored from memory. However, other approaches exist, such as data streaming, a paradigm that has been gaining traction in recent years. It relies on the description of memory access patterns that are then fetched in order and fed to the processor, allowing more organised memory transfers, especially on more complex patterns.

Several works already rely on this concept to improve memory access efficiency, namely Imagine [43], the Reconfigurable Streaming Vector Processor (RSVP) [44], Q100 [45], VEAL [46], Stream-dataflow [47], and CoRAM++ [48], which nevertheless do not target general-purpose Out-of-Order (OoO) cores/speculation mechanisms [1].

More recently, the work from Wang et al. [49] introduced stream-based access specialisation for general-purpose processors. They identified that numerous common memory access patterns could benefit from data streaming, offering an execution-driven prefetching mechanism for repeated access patterns. This work was later extended to enable cache optimisations [50] and near-data computation [51], all depending on scalar stream-based ISAs and supporting microarchitecture. Works such as HotStream [52] and Prodigy [53] also propose ISA extensions to support data streaming, although the latter uses the term Data Indirection Graph (DIG) to describe the same concept.

Additionally, Schuiki et al. [54] proposed that the RISC-V register file was extended to include streaming semantics, later improved to support indirection [55].

2.3.1 Pattern Description Model

In order to employ streaming-based mechanisms, it is important to choose a way to represent memory accesses through streams. A stream is essentially a predicable vector of data elements that are processed sequentially. Each element of a stream is usually subject to the same set of operations and is discarded after the computation is complete. If data accesses are deterministic, the order in which the data is going to be consumed can be specified beforehand. This is possible through data pattern descriptors, such as those proposed and developed in [9, 56, 57]. Understanding this representation model is pivotal to comprehend UVE. Hence, the fundamentals of data streaming and pattern description are described next.

Any regular n-dimensional access sequence can be represented by the following affine function:

$$y(X) = y_{base} + \sum_{k=0}^{dim_y} x_k \times S_k, \tag{2.1}$$

with $X = x_0, ..., x_{dim_y}$ and $x_k \in [O_k, E_k + O_k]$, where a stream access $y(X)$ is described as the sum of the base address of an n-dimensional variable (y_{base}) with dim_y pairs of indexing variables (x_k) and their respective strides (S_k), each k corresponding to a dimension of the pattern. E_k corresponds to the number of elements in each k dimension and O_k to the indexing offset. Because x_0, the first dimension of X, has $O_0 = 0$, it is equal to the base address of the variable [9]. Moreover, through a combination of affine functions of this kind, highly complex patterns can be attained, by assigning some parameter of a function to the result of another one. Lastly, indirect memory accesses can also be described by taking the data obtained by the addresses generated by an affine function and injecting them into the aforementioned variables of another function.

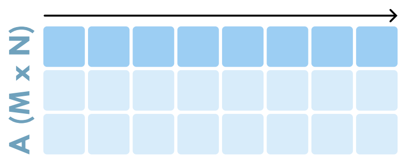
The proposed pattern representation model results from the encoding of the variables associated with each pattern dimension of the function described in Equation (2.1). This representation is based on descriptors and modifiers, defined in a set of dedicated instructions in UVE, which are explained in the next sections.

2.3.2 Linear Patterns – Dimensions

The simplest memory access pattern that can originate a stream is a linear pattern, which is exemplified in Figure 2.4a. This pattern has only one dimension, characterised by three parameters - *offset*, *size* and *stride*.

```
Dimension: {<offset>, <size>, <stride>}
```

It is trivial to represent a similar multi-dimensional pattern, as it results from the addition of more similar dimensions. To exemplify, Figure 2.5a shows a 2D linear pattern, which is a combination of two 1D linear patterns. Additionally, it should be noted that each dimension *offset* is the base address of the descriptor, but each base address is calculated based on all the dimensions *offsets* and *strides*. In the case of a 2D pattern, $y_{base} = O_j + (S_i \times O_i)$. Lastly, it can be seen in Figure 2.6 that by changing the dimension *stride*, more complex patterns can already be described.



(a) 1D pattern

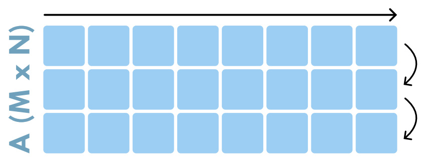
```
for(i = 0; i < N; ++i)
    A[i];
```

(b) C loop

Dimension 1:
Offset = A
Size = N
Stride = 1

(c) Stream representation

Figure 2.4: 1D linear memory access representation.



(a) 2D Pattern

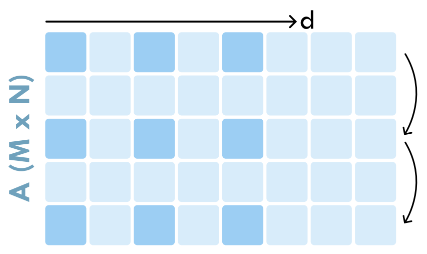
```
for(i = 0; i < M; ++i)
    for(j = 0; j < N; ++j)
        A[i*N+j];
```

(b) C loop

Dimension 1:	**Dimension 2:**
Offset = A	Offset = 0
Size = N	Size = M
Stride = 1	Stride = N

(c) Stream representation

Figure 2.5: 2D linear memory access representation.



(a) 2D Scattered pattern

```
for(i = 0; i < M; i+=2)
    for(j = 0; j < d; j+=2)
        A[i*N+j];
```

(b) C loop

Dimension 1:	**Dimension 2:**
Offset = A	Offset = 0
Size = d/2	Size = M/2
Stride = 2	Stride = 2N

(c) Stream representation

Figure 2.6: 2D scattered memory access representation.

2.3.3 Complex Memory Accesses – Modifiers

Static Modifiers

Manipulating the parameters of each dimension of a pattern allows for the description of higher-complexity patterns. This is done through the use of modifiers, which are descriptors that manipulate the arguments of a dimension, allowing the modelling of inter-loop control dependencies that arise when loop conditions are affected by an outer loop.

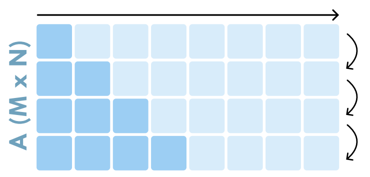
```
Static Modifier: {<target>, <behaviour>, <displacement>, <size>}
```

The importance of these modifiers becomes clear when analysing the memory access pattern represented in Figure 2.7a. In this example, it is evident that the outer loop changes the parameters of the data access that the inner loop describes, as observed in the C code snippet that defines the memory access of the triangular pattern. This access can be divided into two parts, a bi-dimensional stream, just like the ones analysed before, and non-linear behaviour that increments the inner *size* for each outer iteration.

In order to build these modifiers, four parameters must be defined:

- **Target**: The parameter that will be modified (either *offset*, *size* or *stride*).
- **Behaviour**: The type of manipulation (either *increment* or *decrement*).
- **Displacement**: The constant amount of increment or decrement.
- **Size**: The number of iterations where the modifier will be applied.

With these parameters defined, it is possible to build a static modifier that can describe the lower triangular pattern. The full description is represented in Figure 2.7c.



(a) Triangular pattern

```
for (i = 0, k = 1; i < M; ++i, ++k)
    for (j = 0; j < k; ++j)
        A[i*N+j];
```

(b) C loop

Dimension 1:	**Dimension 2:**
Offset = A	Offset = 0
Size = 1	Size = M
Stride = 1	Stride = N

Modifier 2: *Size, Inc, 1, N*

(c) Stream representation

Figure 2.7: Triangular memory access representation.

Dynamic Modifiers

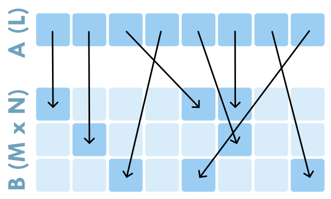
There are several applications whose memory accesses are indirect. That is, the accesses are determined by the values obtained by a different access, such as keeping an array of indices to reference data stored in a different array. The description of such indirect accesses needs a dynamic modifier able to use the values of a different stream as new parameters for the current memory access descriptor.

```
Dynamic Modifier: {<target>, <behaviour>, <displacement>}
```

This is very similar to the previously described static modifiers, but instead of using a constant *displacement*, the source value is now dynamic, obtained from a different stream. This modifier does not need a *size* parameter, as the size of the stream depends on the size of the origin stream and can be inferred. Hence, there are five possible *behaviours* for this type of descriptor:

- **Add**: Adds the dynamic displacement to the base *target*.
- **Subtract**: Subtracts the dynamic displacement from the base *target*.
- **Increment**: Adds the the dynamic displacement to the *target*.
- **Decrement**: Subtracts the the dynamic displacement from the *target*.
- **Set**: Sets the *target* to the dynamic displacement.

It is important to distinguish *add/subtract* from *increment/decrement*. The former takes the dynamically obtained value and uses it as a displacement to *target* field. As an example, the source stream can have values to be added to the target stream original *offset*, corresponding to the indexing variable in a typical C/C++ access loop, as seen in Figure 2.8. On the other hand, the latter takes the dynamically obtained value and uses it as a displacement to the dimension accumulated *target*. This is a counter that is updated during the stream iteration and is reset when the associated dimension is complete, until now referred to as simply *target*. As such, they can be seen as accumulating modifiers, while the former simply replace this counter, adding/subtracting to/from the base value of the selected *target*. These modifiers work slightly differently from *static* ones, in the way that they are associated directly with the target dimension, iterating with it, and not the dimension above it.



(a) Indirection pattern

```
for (i = 0; i < L; ++i)
    B[A[i]];
```

(b) C loop

Stream A	**Stream B**
Dimension 1:	**Dimension 1:**
Offset = A	Offset = B
Size = L	Size = 1
Stride = 1	Stride = 0
	Dynamic modifier 1:
	Offset, Add, A

(c) Stream representation

Figure 2.8: Indirect memory access representation.

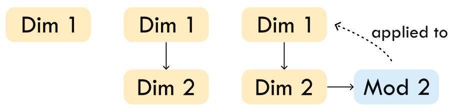
The example in Figure 2.8 represents a scatter-gather memory access, which relied on the assumption that the dimension *size* was ignored and the stream was iterated along the dynamic modifier source stream. However, the original UVE specification did not clarify how this would be done, nor how a modifier would be applied to a single stream element instead of the whole dimension. The defined behaviour of a dynamic modifier is that it is applied at each iteration of a dimension, which means that it would only accumulate the dynamically *offset* with the dimension base *offset*, conflicting with the desired element-wise modifier application. This caveat is explored in Section 4.1.3.

2.3.4 Descriptor Organisation

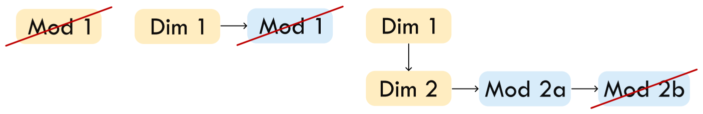
The described pattern representation model results from the encoding of the variables associated with each pattern dimension of the function described in Equation (2.1). This representation is based on descriptors and modifiers that must be organised in a specific way. It was first proposed in [57] that each descriptor was composed of a set of N dimensions and an optional modifier chain, as well as optional references to other descriptors, allowing for a graph-based hierarchical descriptor tree, capable of describing very complex patterns. However, this representation method would require complex hardware support, which is not ideal for a RISC ISA extension, mainly due to the limited encoding space. This makes it impossible to define an N-dimensional descriptor with a single instruction.

As a consequence, UVE was proposed with a simplified version of this organisation scheme, which is already implicit in the examples shown thus far. Instead of having a pattern represented by a set of descriptors, each with N dimensions, in this work each pattern corresponds to a single descriptor, with a set of up to eight dimensions and seven optional modifiers. These are placed in order, in a list-like manner, each node containing a dimension and, optionally, a modifier. The first node (i.e., the first added dimension) corresponds to the lower-level loop in a C/C++ implementation. The last node is therefore the outer loop. Descriptor organisation examples following these rules are represented in Figure 2.9.

As already mentioned, each modifier is associated with the dimension immediately above the one it is meant to modify, so that alterations can occur with each "outer loop iteration". Consequently, it is important to mention that the first dimension of a stream cannot have a modifier, as it has no dimension below that it can modify, and each dimension can only be coupled with one modifier. While this restriction did not seem to pose major problems in the original UVE implementation, it is a limitation that translates into the inability to describe certain patterns, such as the ones characteristic of Sparse Linear Algebra application [58–60]. Therefore, this is addressed further in this work, when adding new features and instructions to the extension in Section 4.1.



(a) Allowed configurations



(b) Illegal configurations

Figure 2.9: Examples of possible descriptor configurations.

2.3.5 Summary

In this section, the pattern description method employed in UVE was explained in detail. Although it is based on the work by Neves et al. [57], it is simplified in order to fit in the developed ISA, resulting in new limitations. However, it should be noted that the majority of complex memory access patterns can still be described in this more compact way, and some can be attained by using more than one stream (and thus, descriptor). Each descriptor is composed of a dimension, defined by an *offset*, *stride*, and *size*, and an optional modifier, static or dynamic. With dimensions, simple linear patterns can be described, while modifiers introduce non-linear behaviour, necessary for complex patterns. Each modifier updates the target value at every iteration of the dimension it is associated with, either statically or with another stream as its source.

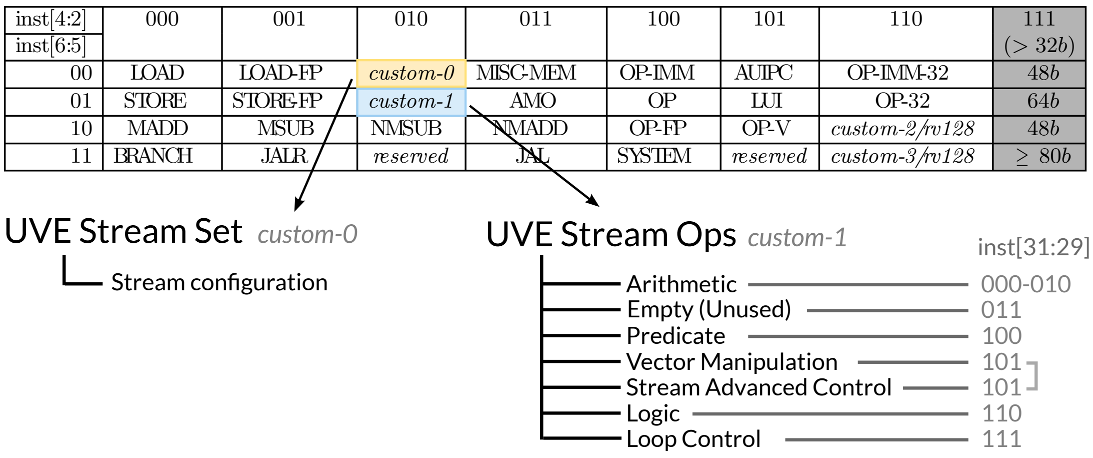
This method is sufficient to describe ubiquitous patterns in HPC applications, with its limitations directly resulting from the desire for hardware simplicity. However, some caveats have since been identified and cannot be overlooked. These are discussed in Section 4.1.3, along with proposed solutions.

2.4 Unlimited Vector Extension

The UVE is the RISC-V extension born from the combination of SIMD computing and data streaming. Proposed in [9], this extension has proven to improve on the previously mentioned technologies. This is a VLA ISA extension, with stream configuration instructions that allow the previously described streaming paradigm to be implemented with data pre-fetching.

RISC-V was chosen as the base ISA not only due to its open-source nature. Although x86 is the dominant ISA in the personal computer market, as well as datacentres and supercomputers, this work targets mainly embedded and low-power systems, for which CISC architectures are usually not suitable. While Arm is at the forefront of mobile and embedded devices, it is also true that the popularity of RISC-V is very high, and continuously growing, especially in academic works [61]. It is simple, yet very powerful and easily extendable. The 32-bit encoding space was chosen in order to target both HPC and embedded processors [1]. Some encoding regions of the base ISA are free for custom extensions, and UVE uses *custom-0* and *custom-1*, as shown in Section 2.4 (*custom-2* and *custom-3*, while available, are reserved for RISC-V 128-bit instructions).

There are two groups of instructions, each occupying a custom region of the encoding space. On the one hand, *stream configuration* instructions, responsible for the definition of each stream as described in Section 2.3, occupy the *custom-0* slot, hereby called *Stream Set*. On the other hand, every other instruction is encoded in *custom-1*, named *Stream Ops*. The encoding of every UVE instruction can be consulted in Appendix C, one of the documents resulting from the work developed in this dissertation.



inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Figure 2.10: RISC-V base opcode map, inst[1:0]=11, with UVE, adapted from [62]

2.4.1 Registers

This extension adds 32 vector registers to the base ISA (named from *"u0"* to *"u31"*). The length of each vector is unlimited[1], but a minimum value is defined, equal to the maximum width of the supported data types (*byte*, *half-word*, *word*, and *double-word*). This means that supporting these data types makes it so the minimum vector length is 64 bits, the length of a *double-word* element. Similarly to RVV, the operating vector length is encoded in a CSR, `vlen`, accessible through a set of vector configuration instructions (see Section 2.4.2 - *Vector Control*). In contrast, the operating *width* of a vector register is encoded independently for each one. Additionally, each vector register must store the information about how many of its bits are valid, for edge cases where a vector is not entirely filled with data. Each of these vectors is associated with a data stream, meaning that *streaming registers* and *vector registers* are no different, simply have different configurations at software level. Possible configurations of vector registers are illustrated in Figure 2.11.

In addition, sixteen predicate registers are present, named *"p0"* to *"p15"*, although only eight can be used in arithmetic and regular memory instructions (`p0-7`). Register `p0` is hardwired to 1, which means it can be used in operations where predication is not necessary (i.e., non-conditional loops), as all valid lanes of the operating streams execute. The remaining predicate registers are either used in the configuration of the other eight or to allow for context saving. These predicate registers always hold *bytes* and are evaluated in a very similar fashion to SVE predicates [6] (see Section 2.2.2, Figure 2.3).

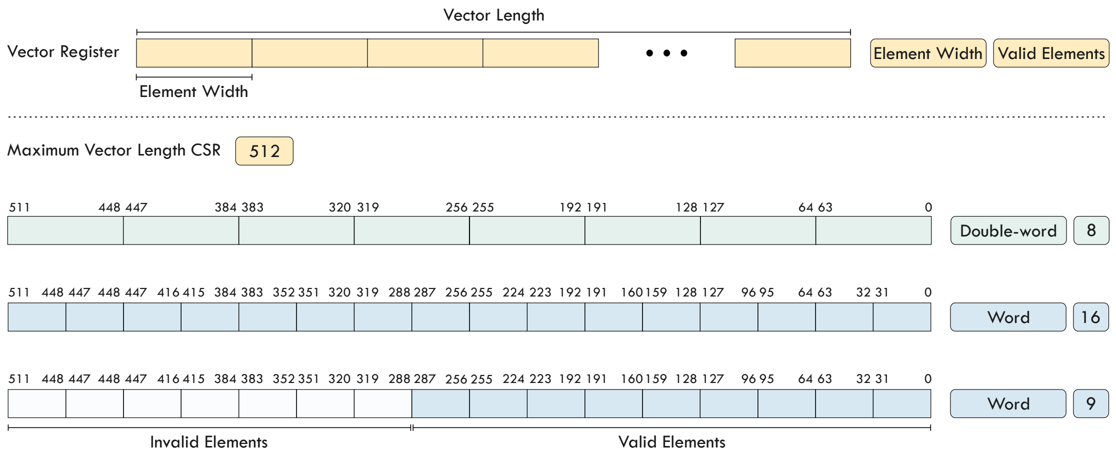


Figure 2.11: UVE registers.

[1]In the specification, there is no upper bound for the vector length, but it must be a limited value in hardware implementations.

2.4.2 Instruction Set

There are currently 60 major instructions, out of which 26 correspond to integer operations, 15 to floating-point operations, and 19 are related to memory manipulation, totalling about 450 instructions when considering the variations of each one. Below, the main types of instructions are described.

- ***Stream Configuration***

  These are the instructions responsible for the configuration of stream descriptor parameters, and they are identified by the prefix `ss` (*Stream Set*). While simple 1D patterns can be configured with the `ss.ld` and `ss.st` instructions, more complex patterns require multiple configuration instructions, in particular, one per descriptor. For this purpose, there are specific stream configuration instructions: for *start*, *append*, and *end*, the latter two with the possibility to have modifiers (static or dynamic). Each *start* configuration instruction must also specify the data width of its elements, with suffix `b`, `h`, `w` or `d`, for *byte*, *half-word*, *word*, or *double-word*, respectively.

- ***Loop Control***

  There are three different conditional branch formats: End-of-Stream (EOS), a condition on the end of the stream, and End-of-Dimension (EOD), a condition on the end of a stream dimension. This allows for straightforward control over iterations of arbitrarily sized streams.

- ***Predicate Configuration***

  As already mentioned, UVE supports instruction predication. Hence, it provides instructions to configure predicate registers, either with fixed values, or based on vector/scalar comparisons.

- ***Vector Manipulation***

  This is the set of instructions responsible for the manipulation of the vector registers, such as moving, duplicating, and width-converting. It should also be noticed that regular vector operations are still supported within the UVE, with no data streaming.

- ***Vector Control***

  These instructions are responsible for the reading/writing of the vector length CSR, as well as for the direct control of running streams, allowing for their suspension, resumption, and termination, as well as loading/storing from previously suspended streams.

- ***Arithmetic and Logic Operations***

  A wide range of arithmetic and logic operations is supported, dealing with both integer and floating-point operands. These instructions can be employed in scalar and vectorial operations without explicit indication.

Every instruction, apart from the *stream configuration* ones, contains the prefix so (*Stream Ops*). Several aspects mentioned in this section were reconsidered throughout this work and will be discussed in detail in Chapter 4.

2.4.3 Supporting Microarchitecture

Data streaming operations are performed within a Streaming Engine (SE), which corresponds to most of the microarchitecture support necessary for UVE. Although this work proposes a functional revision and validation of the extension, it is important to understand the hardware that will support it in the future. Furthermore, the simulation environment proposed in Chapter 3 takes many concepts defined in the proposed microarchitecture, such as the *stream iterator*, and uses them to provide a more realistic simulation environment, despite not providing performance insights.

The proposed microarchitecture targets an OoO processing pipeline, introducing several modifications and additions, such as support for the decoding of UVE instructions, vector register renaming, and the SE. The detailed description of the architecture beyond the SE is presented in detail in Appendix A.

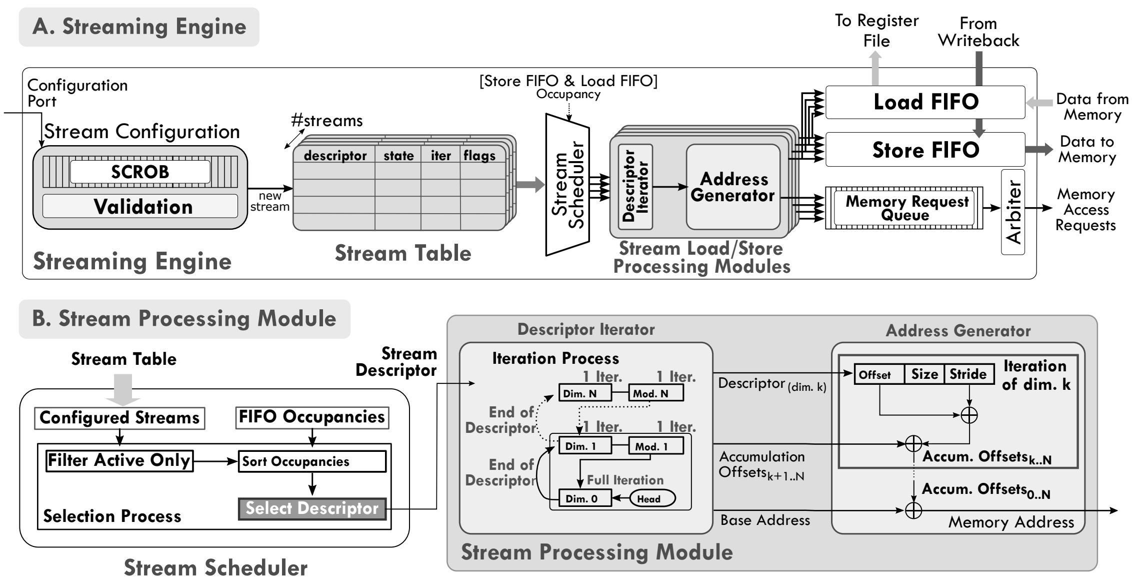


Figure 2.12: Streaming Engine and detailed Stream Processing Module logical block diagrams [9].

The SE is the unit responsible for the management of every stream and it is illustrated in Figure 2.12. It is composed of several structures:

- **Stream Configuration Module and Stream Table:** The former is responsible for the in-order registration of stream configuration instructions on the

Stream Configuration Reorder Buffer (SCROB), which is a part of this module. They are kept here until every data dependency is resolved. Instructions are then taken one per clock cycle and in order, validated, and decoded, so that the corresponding pattern configuration can be written to the *Stream Table*. This table keeps track of every stream's descriptors, state (active/suspended), iterators, and flags (EOD/EOS).

- **Stream Scheduler and Stream Load/Store Processing Modules:** This scheduler is the structure that manages the processing of every stream. In each clock cycle, it selects a number of streams from the *Stream Table* to be iterated. This process is taken care of by the *Address Generators* that exist in each of the *Stream Load/Store Processing Modules*. After the iteration of a stream, a new stream state is generated, which is then registered in the *Stream Table*, as long as it is not selected for processing.

  As detailed below, each stream is associated with a First-In, First-Out (FIFO), and their occupancy is the criterion used by the scheduler to select the streams. Particularly, it chooses the streams with the lowest occupancy to be iterated, so that the most consumed FIFO are prioritised, thus avoiding longer wait times for streams that are being consumed.

  The *Stream Load/Store Processing Modules* are also composed of a *Descriptor Iterator*, which selects which dimension of the descriptor is to be processed and passes it, as well as the accumulation offsets from higher dimensions and base address, to the *Address Generator*, which performs the necessary computations to generate the memory address from the next element to be loaded to a load stream or to which an element from a store stream must be written. After a dimension (and optional modifier) j is fully iterated, the next dimension $j+1$ is selected for processing and, after its first iteration, j is reset to the original state, as does the modifier if it exists.

  The resulting memory addresses are written to the *Load/Store FIFOs*. In the case of load streams, a load request is also generated and registered on the *Memory Request Queue*. Then, an *Arbitrer* takes those requests and issues them to the memory, after the virtual-to-physical page translation that relies on a Translation Lookaside Buffer (TLB) [63, pp. 105-106]. Once these load requests are attended by the memory and data is received, it is written to the *Load FIFOs*. On the other hand, store streams write the generated addresses to the *Store FIFOs*.

- **Load/Store FIFOs:** As already mentioned, for each stream there is an associated FIFO, which keeps the loaded elements for input streams until they are consumed, and memory addresses for output streams until data is written to memory. However, once a data element reaches this point, it is treated as if it has already been consumed, working as a buffer for the vector registers, as the SE in which it is inserted is already part of the core.

These fixed-length FIFO queues are limited to a depth of eight, in order to avoid more complex hardware requirements.

All these components were designed and tuned in order to minimise the architectural impact of the extension, while still providing a functional proof-of-concept implementation with promising results.

2.4.4 Compiler Support

For the development and porting of UVE applications, compiler support is required. Backend support for this novel ISA extension has already been added to two popular C/C++ compilers, both GNU Compiler Collection (GCC) [64] and LLVM [65], which are the ones worth considering for this extension [1]. Although GCC is an official compiler for the GNU and Linux systems, and therefore widely used, it is notoriously effortful to modify and extend. The newer LLVM compiler is modularised, as opposed to GCC, which has a monolithic architecture, and thus more adequate for the development of the UVE extension, as well as other extensions with a stream-based execution model.

Furthermore, the LLVM infrastructure allows for the automatic extraction and encoding of the memory access pattern and computation data-flow graph with streaming representations, a solution proposed by Neves et al. [66]. However, UVE support has not been fully added to LLVM yet, whereas GCC already features a working implementation of the existing instructions. As such, GCC was used for the compilation of applications to run on *Spike* in the first stage of this work, while the focus was on the validation of the existing instructions, as this compiler is already fully functional and is straightforward to use.

In spite of this, it is important to note that, parallel to this work, the development of the LLVM compiler is ongoing, and some benchmarks presented in Section 5.1.2 were already directly obtained from C/C++ code, as support for automatic vectorisation was made available. This compiler will be a crucial element of the UVE development framework that this work aims to create, as autovectorisation is a key feature to turn a vector ISA into a viable solution for the general public [67]. The reason for this is that, without it, the programmer is left with the task of manually vectorising the code, a time-consuming and error-prone process. Besides, complex memory access patterns are extremely difficult to vectorise by hand. Nevertheless, this work has to be done for the validation of the extension until the compiler is fully functional.

Regarding GCC, the modified compiler was obtained by extending the assembler tool, to support the UVE instructions. The work by Domingos [1] makes it so that inline assembly directives can be included inside the C/C++ code, which can also directly interact with it. This is the only way to compile UVE code while LLVM autovectorisation is not supported for every memory access pattern.

```
void uve_saxpy(float *x, float *y, uint64_t sN, float a) {
    asm volatile(
        "ss.ld.w    u1, %[x], %[n], %[z] \n"
        "ss.ld.w    u2, %[y], %[n], %[z] \n"
        "ss.st.w    u3, %[y], %[n], %[z] \n"
        "so.v.dp.w  u4, %[a], p0 \n"
        "fLoop1: \n"
            "so.a.mul.fp u5, u1, u4, p0 \t"
            "so.a.add.fp u3, u2, u5, p0 \n"
            "so.b.nc         u3, fLoop1 \n"
    :
    : [x] "r"(x), [y] "r"(y), [n] "r"(sN), [z] "r" (1), [a] "r" (a));
}
```

Listing 2.8: *SAXPY* C/C++ code with extended assembly directives support.

Even so, the use of C inline assembly facilitates development, as opposed to assembly-only programming, mainly because the C variables become directly usable by the assembly code, with a somewhat high level of abstraction, as all the conversion details are left to the compiler. It should be noticed that the specification of whether a C variable should be written to or read from in the assembly code is the responsibility of the programmer. The output operands must be included after the first colon and the input operands after the second colon. In the case of Listing 2.8 there are no output operands, so that line of code is left blank. Each operand is specified as `[ref] "mode" [cvar]`, where `ref` is its identifier in the assembly code and `mode` indicates what type of register it should use ("r" means that it should be put in a scalar register). As for `cvar`, it is the name of the C variable to be used[2].

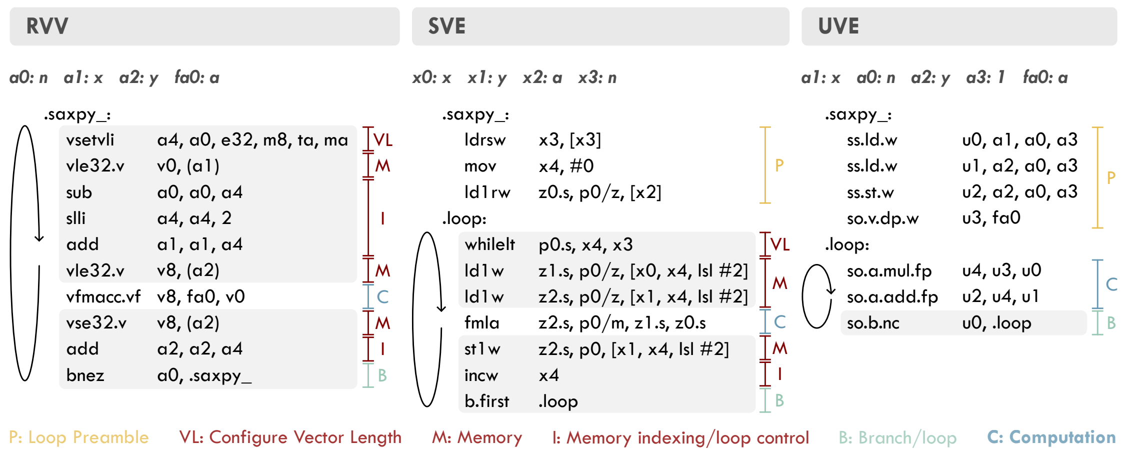
2.4.5 Discussion

In this section, the Unlimited Vector Extension (UVE) was presented, a novel SIMD ISA extension for the RISC-V ISA that exploits data streaming to minimise loop overhead in SIMD loops. To better visualise the improvements achieved by this extension, Figure 2.13 presents the already familiar *SAXPY* kernel in assembly code, but this time with RVV, SVE and UVE, and the type of each instruction identified.

Although UVE requires one more computation instruction than the other two

[2]More information about the extended assembly can be found in `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`.

implementations[3], it is still clear that it significantly reduces loop overhead. The only instruction that is present in the loop and does not contribute to the computation is a branch, which is necessary for each of the three implementations. This is possible because loads and stores are performed implicitly by the SE and configuration instructions are part of the loop preamble, which is executed only once. In large datasets, the number of configuration instructions remains the same, although loop iterations would significantly increase, which means that the loop overhead is reduced to a single branch instruction in this example. In more complex memory access patterns the same would happen: although more configuration instructions are necessary, that number becomes negligible when compared to the number of loop iterations. Moreover, implicit predication is employed, eliminating the need for vector length configuration instructions inside the loop. Finally, loop control is also based on the stream state. In this case, the loop halts once the stream is complete (i.e., every element has been processed, EOS flag is set and caught by branch instruction).



```
a0: n   a1: x   a2: y   fa0: a

.saxpy_:
    vsetvli     a4, a0, e32, m8, ta, ma
    vle32.v     v0, (a1)
    sub         a0, a0, a4
    slli        a4, a4, 2
    add         a1, a1, a4
    vle32.v     v8, (a2)
    vfmacc.vf   v8, fa0, v0
    vse32.v     v8, (a2)
    add         a2, a2, a4
    bnez        a0, .saxpy_
```

```
x0: x   x1: y   x2: a   x3: n

.saxpy_:
    ldrsw       x3, [x3]
    mov         x4, #0
    ld1rw       z0.s, p0/z, [x2]
.loop:
    whilelt     p0.s, x4, x3
    ld1w        z1.s, p0/z, [x0, x4, lsl #2]
    ld1w        z2.s, p0/z, [x1, x4, lsl #2]
    fmla        z2.s, p0/m, z1.s, z0.s
    st1w        z2.s, p0, [x1, x4, lsl #2]
    incw        x4
    b.first     .loop
```

```
a1: x   a0: n   a2: y   a3: 1   fa0: a

.saxpy_:
    ss.ld.w         u0, a1, a0, a3
    ss.ld.w         u1, a2, a0, a3
    ss.st.w         u2, a2, a0, a3
    so.v.dp.w       u3, fa0
.loop:
    so.a.mul.fp     u4, u3, u0
    so.a.add.fp     u2, u4, u1
    so.b.nc         u0, .loop
```

Figure 2.13: *SAXPY* assembly code with RVV, SVE and UVE.

Although reducing the number of instructions is a remarkable improvement, it does not imply that the overall performance of the processor in which UVE is used improves. This depends on the implementation of the required architecture model, described in Section 2.4.3. In the extension proposal [9], such an implementation is presented, based on the ARM Cortex-A76 and simulated on *gem5*. This proof-of-concept model was benchmarked against the same processor with SVE and Neon, revealing an average speed-up of 2.4× relative to SVE on vectorised benchmarks and much higher when compared to Neon (the available SVE compiler failed to do so in some cases). It proved an average of 60.9% less committed instructions than SVE, 93.1% relative to Neon, which combined with the

[3]A single fused multiply-add instruction cannot be used in this case, as it takes only three operands and would have to function as a load and store stream simultaneously, which is not possible in the current specification.

```c
void floyd_warshall(float *path) {
    for (k = 0; k < N; k++) {
        for(i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                path[i][j] = path[i][j] < path[i][k] + path[k][j] ?
                    path[i][j] : path[i][k] + path[k][j];
    }
}
```

Listing 2.9: Original *Floyd-Warshall* C/C++ kernel code [68].

streaming mechanisms and consequent reduction of load-to-use latency, results in this significant speed-up.

However, this implementation is not without limitations. On the microarchitecture side, one of the main shortcomings that were identified is related to the lack of memory coherence in the Load/Store FIFOs of the SE. Because they are assumed to have been consumed once they enter these buffers, loaded data elements that may be actively being changed by other output streams (or even non-stream instructions) in memory are not updated. This becomes a problem in data-parallel kernels, where iterations depend on previous ones. Moreover, memory incoherencies can occur inside a vector register itself, in cases where each new data element depends on new values for previous elements, obtained in previous loop iterations after a vector has already been preloaded. This is the case in the example in Listing 2.9, where the core kernel of the *Floyd-Warshall* algorithm is implemented. In this case, the UVE implementation would not work, as `path[i][j]` is updated and used in every loop iteration, a problem when outdated values of these elements have already been loaded to the SE. In this case, the code is considered to be unvectorisable, as operations simply cannot be performed on multiple elements at once. However, memory incoherences in the stream buffers can be avoided. A possible solution is to only allow data to leave them and be written to the vector registers at the moment when they are to be consumed by the processor. While inside the buffers, allowing the Load and Store FIFOs to communicate, each element is not assumed to be consumed and can be overwritten in the case of a memory write. While this is the general idea, its implementation is not straightforward and is out of the scope of this work, so it is left to future work, as well as any microarchitecture-related optimisations.

On the other hand, the main focus of this work is the ISA specification of UVE, on which several improvements can be identified and are detailed in Chapter 4.

Chapter 3

Simulation Infrastructure

Since the Unlimited Vector Extension (UVE) development is in its early stages, the specification is undergoing several improvements and corrections, and a real hardware implementation is not yet available. Therefore, a software simulator is the most adequate tool to continue the development and validation of the extension. In accordance, this chapter starts by presenting the chosen simulator, *Spike*, and the reasons for its selection over the one used in the original work, *gem5*. Then, the developed infrastructure is described, as well as the modifications and additions made to the simulator in order to support existing and new UVE instructions.

3.1 The RISC-V ISA Simulator: *Spike*

The Instruction Set Simulator (ISS) *Spike* has been chosen as the appropriate tool to validate instructions and overall behaviour of UVE, as well as to continue the development of its specification. *Spike* is the golden reference functional RISC-V ISA simulator and is widely used as the proof-of-concept target for every RISC-V extension [10, 61].

As mentioned in Section 2.4.5, the UVE proof-of-concept was implemented and validated on a different simulator, *gem5*, which is a cycle-accurate simulator whose purpose is to mimic real hardware behaviour. The work by Domingos et al. [9] implements the supporting microarchitecture, namely the Streaming Engine (SE) and the necessary CPU processing pipeline modifications. Although much of the base work is done and available for further development, the base *gem5* simulator does not provide extensive enough documentation available and its base code is constantly evolving between releases, so the original implementation is now considered deprecated. In addition, some benchmarks used to evaluate the extension return unexpected results, due to the internal workings of the simulator, while others take some time to be executed, which is not ideal for an early development tool. Moreover, a validation that is completely independent of implementation details is necessary (i.e., microarchitecture and pipeline modifications to a specific processor), as it is the only way to ensure that the extension is correctly formalised and that the instructions behave and interact as expected. All of this led to the decision to opt for a simpler and purely functional simulator, thus speeding up the development process and allowing the focus to be solely on the specification, key for future implementations.

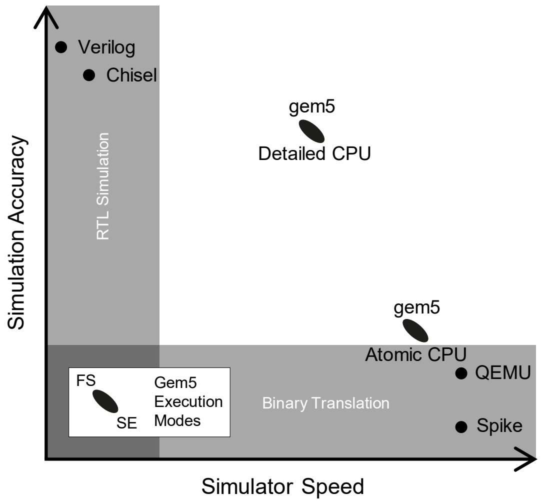


Figure 3.1: Illustration of simulation accuracy vs. speed of multiple simulation platforms [69].

The choice of *Spike* as the base simulation environment for this work resulted from an initial assessment of the advantages and disadvantages between several simulation platforms. As can be seen in Figure 3.1, there is usually a compromise between simulation accuracy and speed when choosing between the available RISC-V simulators available. Whereas Register Transfer Level (RTL) Simulation is the most accurate, binary translation is the fastest. It is clear that *gem5* is the most adequate tool when time performance analysis is pivotal but it is not feasible to create a proper RTL Simulation, as it requires the development and implementation of the entire system. Because the main objective is to perform a functional validation, binary translation is not only enough, as it is preferable. Therefore, although *Spike* does not allow cycle-by-cycle precision, it is suitable for this work. Despite QEMU appearing to be slightly more accurate, it is a much bigger project, as it targets multiple architectures, not only RISC-V, and is thus more difficult to modify, something that is necessary in order to create UVE support. This is pointed out by Henriques [70], who already used the *Spike* simulator to implement a few UVE instructions and whose work was made available for the initial development of this dissertation.

Spike is currently at Version 1.1.0 and already supports many RISC-V ISA features, including the RISC-V Vector Extension (RVV) which served as a base for some of the developed modules. However, upon analysing the implementation of several extensions on the simulator, it became clear that the UVE implementation structure would be very different. This is mainly due to the way the simulator source code is written, heavily dependent on macros defined in multiple files and with little to no documentation. This resulted in code structured in a very different way than the rest of the simulator and its supported extensions, albeit more comprehensible.

3.2 Simulator Files and Code Structure

The *Spike* simulator is a complex piece of software, with several files and classes, and a large codebase. It emulates a processor through the `processor_t` class, declared and implemented in files `processor.h/cc`. Alterations to this class were minimal, limited to the addition of the Streaming Unit (SU). Another key component which the processor has access to is the Memory Management Unit (MMU), implemented through the `mmu_t` class.

Only a few original files were modified, with the majority of the new code being added in new files. The emulated SE structures were implemented in C++ classes as well, and the *Standard Library* was extensively used. The main classes and their attributes are shown in Figure 3.2, which also indicates in which files each definition and implementation can be found.

The Streaming Unit (SU) and its supporting classes, described in detail in Section 3.3, are defined and implemented in the files `descriptors.h/cc` (dimensions and modifiers) and `streaming_unit.h/cc` (registers and SU). Moreover, files containing the implementation of each instruction were created. These must be inserted in the `riscv/insns` directory and have the same name as the corresponding instruction (e.g., `so.a.add.fp` is implemented in file `so_a_add_fp.h`). The necessary decoding functions are included in the file `decode.h`, described in Section 3.4.1. For the simulator to recognise the new instructions, the file that holds the ISA encoding, `encoding.h`, was updated. To obtain the necessary code, the official RISC-V Opcodes project [71] was used, where the encoding of each instruction was added and UVE's predicate registers and immediate encoding was added to the file `constants.py`.

Furthermore, the new extension was added to file `riscv/riscv.mk.in`, identically to what is done to the native ones, so each new instruction was included in the variable `riscv_insn_ext_uve`. In this file, every new source and header file was also added to variables `riscv_srcs` and `riscv_install_hdrs`, respectively, so that they could be recognised during the compilation of the simulator.

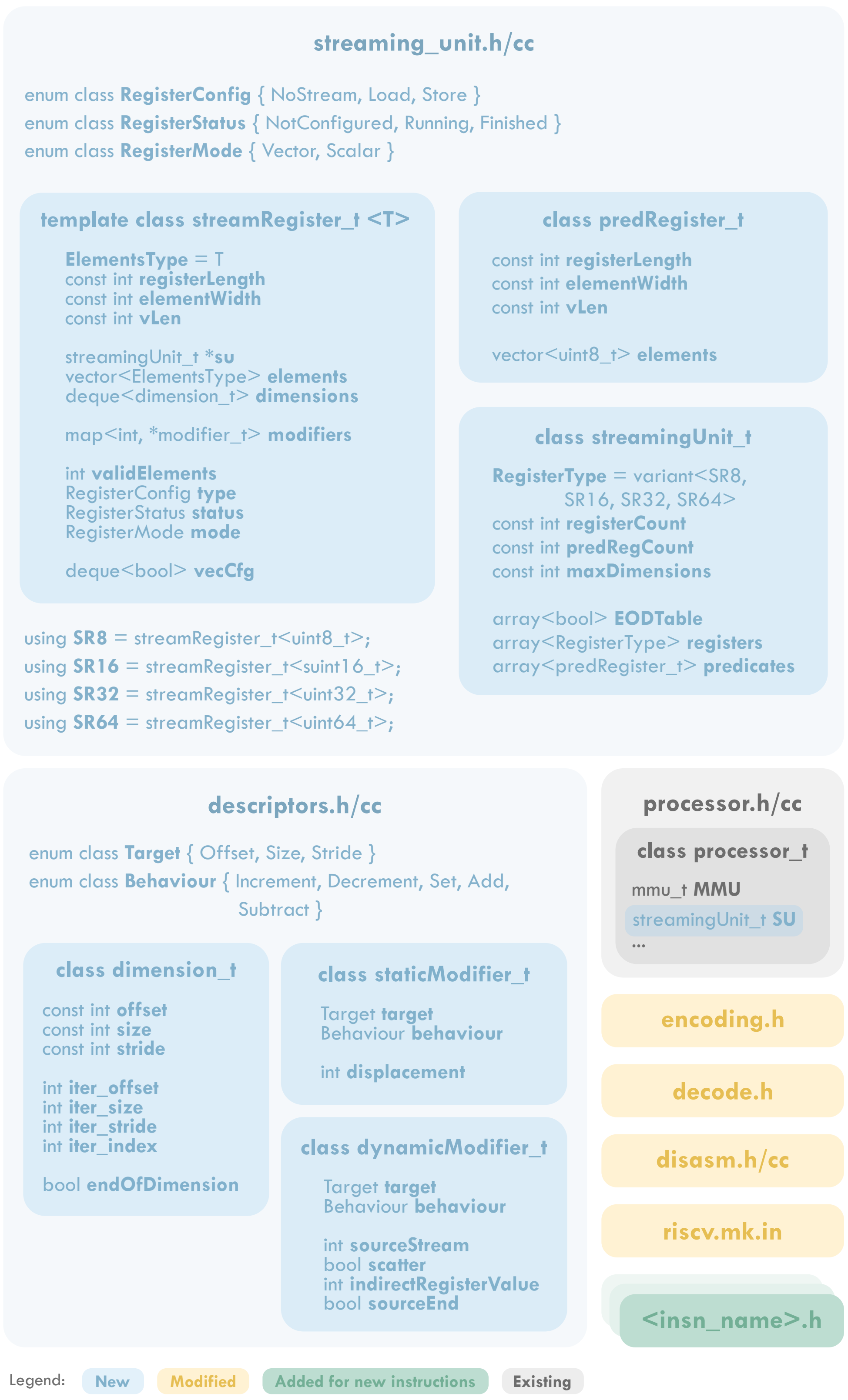


Figure 3.2: Diagram of added and modified structures and files on *Spike*.

Lastly, the disassembler was extended so that the new registers and instruction formats could be recognised by *Spike's* debugger and trace generator. Code related to the disassembler is in files `regnames.cc` and `disasm.cc`, both in the `disasm` directory, and `disasm.h`, which is in the `riscv` folder. These changes are detailed in Section 3.5.

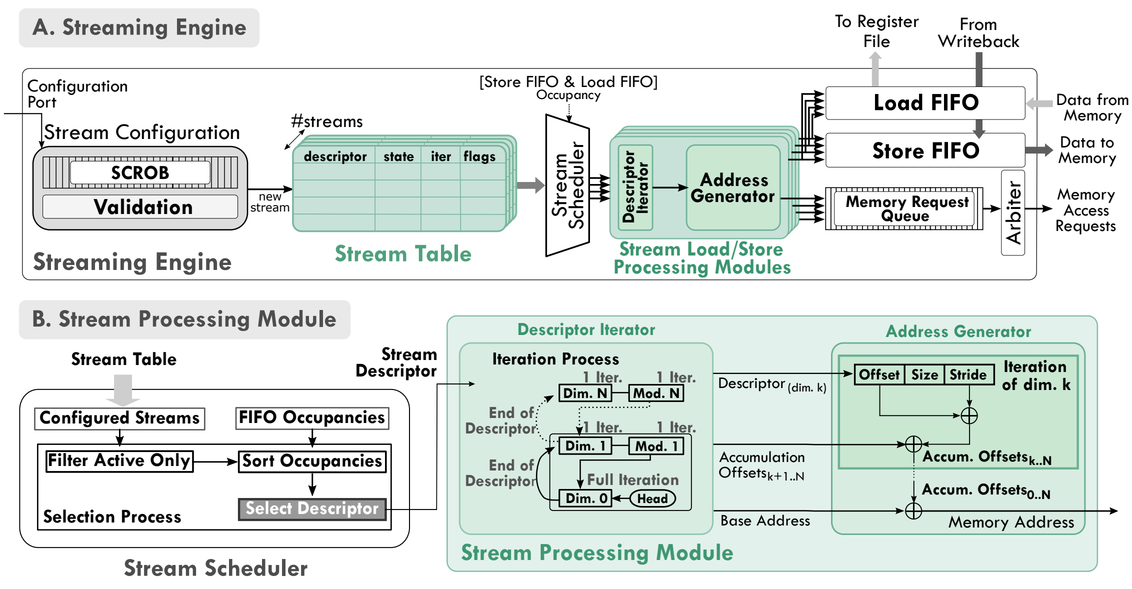
3.3 Streaming Simulation Infrastructure

In order to add UVE to *Spike*, the key necessary addition to the simulator is a set of mechanisms that emulate the SE, responsible for streaming operations. As such, the focal component of the new simulation framework is the Streaming Unit (SU), a new class that has access to the streaming and predicate registers. This unit mimics some parts of the original SE [9], specifically the *Stream Tables* and the *Stream Processing Modules* (see Figure 3.3). Each UVE register may or may not be associated with a stream, and this module is responsible for the implicit loading and storing of data, as well as the iteration of the streams (by the *Address Generator*). For the desired functional evaluation, the *Load/Store FIFOs*, the SCROB, and the *Stream Scheduler*, represented in gray in Figure 3.3, were not needed, as streams are iterated as they are being consumed, with each computation instruction triggering the iteration of the source streams (implicit loading) and the destination streams (implicit storing). The resulting elements are immediately placed in the associated registers and the *End Of Dimension* flags are updated and saved. The iteration and address generation parts work very similarly to the proposed configuration and are implemented in a different class, *Dimension*, which has access to the *Modifier* class, where static and dynamic modifiers are implemented. Each streaming register, when associated with a stream, is therefore also associated with n dimensions and respective modifiers if such is the case.

3.3.1 Stream Iteration and Load/Store Mechanisms

The most important part of the streaming process is the implicit loading/storing of new elements. This was the process that required the most changes to *Spike* to correctly implement and that allowed for the identification of some issues in the old specification. It should be noticed that the current *Spike* implementation works sequentially and not in parallel, which means that these operations are not performed while other instructions are being executed, as would be expected in a real processor. However, that is not necessary for an ISS to perform as desired. With this in mind, a simplistic view of this process is described in the flowchart of Figure 3.4.

As a first approach to the implementation of the stream iteration and element



The components which were implemented on the proposed framework are represented in green, while the ones in **grey** are implementation specific, and thus not needed to fully describe UVE functional behaviour.

Figure 3.3: (A) Streaming Engine and (B) Stream Processor Module proposed in [9], now emulated on *Spike*.

load operations, after the stream configuration was complete, the SU immediately loaded the first elements to the associated register. Then, each computation instruction directly read its operands and, once they had been consumed, the stream was iterated and new values were loaded to the register, a similar process to what was specified and implemented on *gem5*. However, this highlighted the memory coherence problem already mentioned in Section 2.4.5, despite not having *Load/Store FIFOs*, because the memory was accessed before a consuming instruction was executed, leaving plenty of time for other instructions to make changes to the elements in memory which were not reflected in the register. Although it is a different issue from the one identified in the originally proposed SE, this was what led to its identification, one of the many instances throughout this work where the implementation of the extension has led to the identification of issues in the specification. While in real hardware a solution for this issue will require more complex changes to the SE, it was noticed that it is necessary to delay the register load operation until the consuming instruction is executed. This is the solution that was implemented and that is currently in use: each instruction that takes a load stream as an operand starts by requesting the SU to load elements from memory to the register.

As previously detailed, the *Stream Processing Modules* are responsible for the iteration and flag setting of each stream. This process was implemented in the SU, at the register level, accessing each dimension of the configured pattern to perform necessary checks and computations. As such, several methods were implemented, namely the ones responsible for offset generation, available in List-

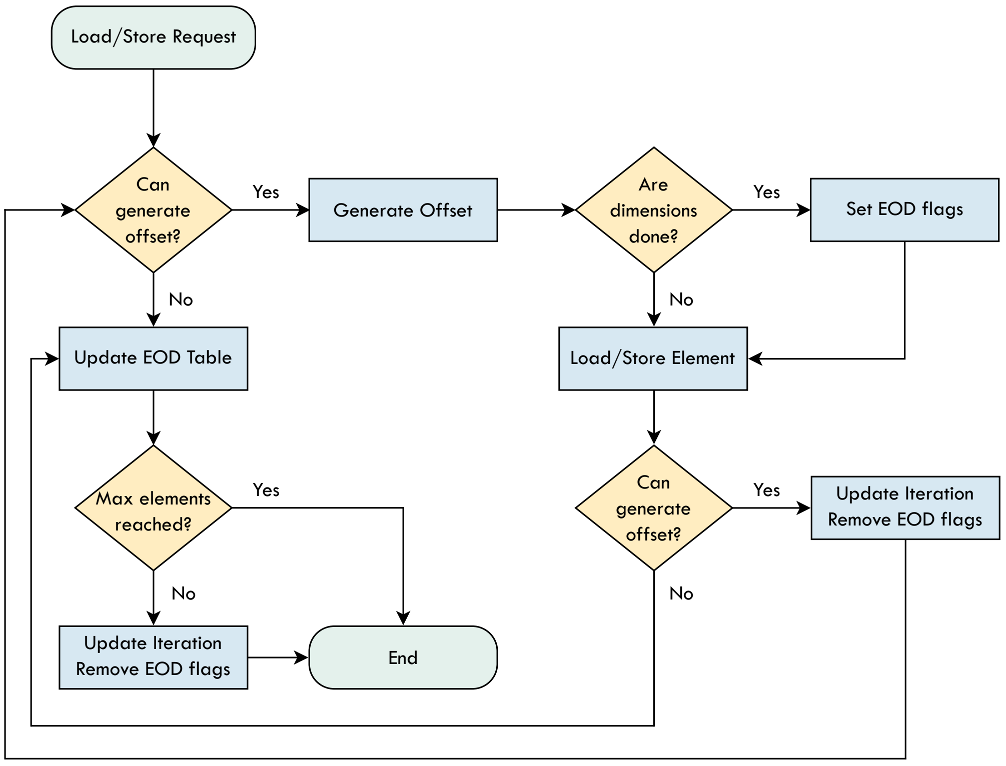


Figure 3.4: Flowchart of a high-level overview of the loading and storing of elements to/from a stream, as implemented on *Spike*.

ing 3.1.

According to the UVE specification, each register can hold values of four different widths (*byte*, *half-word*, *word* and *double-word*). As such `streamRegister_t` was implemented as a template class, allowing for type flexibility. Because of this, *variants* are often used, namely in the `streamingUnit_t` class, to be able to have an array of registers (emulating a *Register File*) of different and unknown types. Accordingly, to access a register the `std::visit()` *callable* must be used[1]. In contrast, the `predRegister_t` class is a regular one because predicates are always composed of *bytes*, as seen in Section 2.4.1.

The `streamRegister_t` class contains a *vector* of `dimension_t` objects, each corresponding to a configured dimension of the memory pattern of the stream, in case the register is configured as `Load` or `Store`. When no stream is associated with a register, it is configured as `NoStream`, and none of these attributes are used, it simply saves values in the `elements` vector. This class also has a structure holding modifiers, which are mapped to the dimension they are associated with. Lastly, `vecCfg` is a mask that indicates which dimensions are vector coupled. This functionality is further explained in Section 4.3.1.

[1]Documentation on these C++ utilities can be found at `https://en.cppreference.com/w/cpp/utility/variant`

The `streamRegister_t<T>::generateAddress()` method is responsible for the accumulation of all offsets calculated per dimension, as well as the setting of the EOD flag. Other methods used in this piece of code have very straightforward implementations, returning exactly what is expected from their names. The variables used by `dimension_t::calcAddress()` are attributes of this class and are updated during the stream iteration process (*Update Iteration* in Figure 3.4):

- `iter_index` is incremented by 1 after each iteration, and is reset to 0 when the EOD flag is reset, signaling the start of a new full iteration of the dimension.
- `iter_size` is the number of elements in the dimension and is set during the stream configuration process. It is used as the upper limit for the `iter_index` iterator.
- `iter_size`, `iter_stride`, and `iter_offset` are set during the stream configuration process and are only changed during the dimension iteration if a modifier is applied.

```cpp
size_t dimension_t::calcOffset(size_t width) const {
    return iter_offset + iter_stride * iter_index * width;
}

template <typename T>
size_t streamRegister_t<T>::generateAddress() {
    /* Result will be the final accumulation of all offsets calculated per
    ↪ dimension */
    size_t init = 0;
    int dimN = 0;

    return std::accumulate(dimensions.begin(), dimensions.end(), init,
    ↪ [&](size_t acc, Dimension &dim) {
        if (dim.isLastIteration() &&
        ↪ isDimensionFullyDone(dimensions.begin(), dimensions.begin() +
        ↪ dimN)) {
            dim.setEndOfDimension(true);
        }
        ++dimN;
        return acc + dim.calcAddress(elementWidth);
    });
}
```

Listing 3.1: Offset computation C/C++ code.

It is assumed that the *offset* of each dimension is already the value in bytes, as it can be the base address of a descriptor. The remaining values correspond to an element count and must be multiplied by the element width, which is passed as an argument to methods that require it, as dimensions and modifiers have no information about the element width of the stream they are associated with.

Before an *address* is generated, a check is performed, as two situations prevent the SU from generating a new load/store *address*:

- The last iteration of the outermost dimension has been reached, signaling the EOS flag. In this case, the *status* of the register is set to *finished* and its *type* is set to *NoStream*, as the stream has ended and it can be used as a regular vector register again.
- A vector coupled dimension has its EOD flag set, signaling the end of the dimension. In this case, iteration can only resume once the EOD flag is reset, which is done by a new iteration of the stream.

Finally, throughout the iteration of a stream, if present, modifiers are also applied. Dynamic and static modifiers are applied in different moments, as the former must be applied before the start of a dimension, while the latter are applied after a dimension ends. Furthermore, new scatter-gather descriptors were added to the specification and were implemented in the simulator. These behave differently from previously existing modifiers and are therefore also applied in different moments. Their functioning and possible applications are detailed in Section 4.1.3.

3.3.2 Stream Table

In the proposed simulation environment, the *Stream Table* is not a single structure. Instead, the information it keeps is divided into various attributes of the SU and its registers. Besides variables that have information about its state and configuration (*type*, *status*, and *mode*), or if a register is associated with a stream, it has a list of dimensions that build the stream memory access pattern, which each have an EOD flag, `bool endOfDimension` of `dimension_t`, as seen in Figure 3.2. As indicated in the flowchart of Figure 3.4, these flags are set and reset during the iteration process. This means that they can be set and reset several times during the execution of a single instruction, as a dimension may come to an end while there is still space for more elements in the vector register, in which case the iteration process continues and a new dimension is processed, unless the one that ends is configured as vector coupled. However, if a dimension still came to an end, that information cannot be lost, or branch instructions will not be able to capture the EOD signal. As such, the EOD flags are saved in a structure called `EODTable`, which belongs to the SU. This 2D array is responsible for saving all EOD flags for every stream and is also updated during the iteration process. These updates are performed in a way that, in case a given dimension ends, this signal is saved in the table before a new iteration, which inherently resets all EOD flags. This way, they are not lost and can be used by branch instructions, which access `EODTable` instead of the registers.

3.4 Instruction Implementation

Following the standard implementation of instructions on *Spike*, each new instruction was implemented in its separate file. Each instruction has a corresponding *header* file in the `riscv/insns` folder. While compiling the simulator, these files will be used to create copies of the `riscv/insn_template.cc` file for each instruction, responsible for the generation of the various versions of the instruction (e.g., 32/64 bit). The obvious implication is that the developed code for an instruction exists inside an external function, therefore header file inclusion is not allowed and only some variables are accessible, namely the processor, the instruction object, and the Program Counter (PC). It is through the processor that each instruction can access the MMU, as well as the SU and its registers. The instruction that is being executed, an `insn_t` object, has access to the operand decoding functions, and the PC is mainly used in branching instructions.

Furthermore, predication support was developed at the instruction level, which means that the predicate values never reach the SU, for simplicity. A predicate register has a fixed vector size of 64 *bytes*, and a predicate is thus evaluated according to the element width of the instruction's source operands. As a result, in each predicated instruction the predicate register is read for each active lane, and the operation is only performed if it evaluates to 1, as stated by the ISA specification [9].

3.4.1 Operand Decoding

To execute an instruction, the simulator must first be able to decode its arguments. For this purpose, decoding functions for each operand type were created, according to the ISA encoding. These functions, divided into different types of instructions, followed the same pattern as already existing ones (for other extensions), some even being direct copies so that there is complete flexibility in case the UVE encoding is changed. In case that happens, it is not necessary to alter every instruction if, for example, one of the source registers is differently encoded, and only the decoding function corresponding to its type requires updates. These functions are defined in file `decode.h` and some are shown in Listing 3.2.

The first three functions presented in Listing 3.2 were already implemented on the simulator. The `x()` function takes the first bit to be read and the length of the operand, both defined in the ISA encoding, which can be consulted in Appendix C and is further detailed in Chapter 4. It discards the lower `lo` bits by performing a right shift on `b`, the instruction bits. Then, it applies a mask to the result, which is obtained by subtracting 1 from the result of a left shift of 1 by `len` bits. This way, the function returns the desired operand, which is then used in the instruction's implementation. The `xs()` function is similar, but handles signed values, useful for immediate operands. These functions are used in the decoding of the UVE instructions in a very similar fashion to already existing instructions.

The least straightforward decoding function is the one that handles the immediate operand of the branching instructions. The `uve_branch_imm()` function is used to calculate the offset to be added to the program counter in case the branch is taken. Because the immediate operand is not contiguous in the instruction, the function performs a series of shifts and adds them together to obtain the final value. Because a branch can jump either to a previous or a following instruction, the immediate operand is signed, and the function `xs()` is used to obtain the sign bit in position 28. As a final note, the lower bit of the immediate operand is not used, as it is always 0. This is due to each instruction being 32 bits long (4 bytes), and the PC being incremented by 4 after each instruction is executed. Because RISC-V also has a 16-bit (2 bytes) instruction format, the PC is incremented by 2

```
// Spike defined functions
typedef uint64_t insn_bits_t;
insn_bits_t b; // Current instruction bits
uint64_t x(int lo, int len) { return (b >> lo) & ((insn_bits_t(1) << len) -
↪  1); }
uint64_t xs(int lo, int len) { return int64_t(b) << (64 - lo - len) >> (64
↪  - len); }

// Registers for arithmetic and logic instructions
uint64_t uve_rd() { return x(7, 5); }
int64_t uve_rs1() { return x(15, 5); }
int64_t uve_rs2() { return x(20, 5); }
int64_t uve_rs3() { return x(27, 5); }
uint64_t uve_pred() { return x(25, 3); }

// Calculate offset for UVE branching instruction
int64_t uve_branch_imm() { return (x(8, 4) << 1) + (x(22, 6) << 5) + (x(7,
↪  1) << 11) + ( xs(28, 1) << 12); }
```

Listing 3.2: Operand decoding function examples.

in that case. For compatibility, the RISC-V specification states that branch offsets are then always scaled by 2 bytes, even when no 16-bit instructions are used [62]. This means that there is no need to encode the least significant bit in the instruction and that the offset must always suffer a left shift by 1 before being added to the PC, which is performed by the decode instructions on *Spike*.

3.5 Disassembler

The disassembler is a key component of the *Spike* simulator, as it is responsible for translating the binary instructions into human-readable assembly code. This is particularly important important for the desired simulation and validation framework, as they are required for a readable debugger and trace output, which represent important tools for the development and validation of UVE. It should also be mentioned that the *Spike*-generated trace can be used in other different tools to provide other types of simulations, such as [72].

Several functions and macros were added to the source code so that new instructions and operands were correctly recognised and printed in the output trace, as well as during any debugging session. They follow the same pattern as already existing ones, for other extensions.

First, the new register types must be added, as well as the size of the register file. Register names are defined in the `regnames.cc` file, which lists the `u0-31` and `p0-15` registers in two different character arrays, `ur_name` and `pr_name`, respectively. The size of each register file (streaming and predicate) is defined in file `decode.h`.

The operand disassembling functions they take the instruction object and use the decoding functions described in Section 3.4.1 to obtain the index of the register, whose name they get from the previously defined arrays. Immediate operands in UVE are only used for branch instructions and are directly printed in the disassembled instruction, as they are not associated with any register.

To facilitate the disassembly, instructions can be grouped by types with a similar encoding (i.e., operands in the same bit positions). Each type of instruction has a corresponding function where its operands are indicated and macros are used to further simplify the code. UVE instructions are added to the disassembler as illustrated in the example from Listing 3.3, showing the entire process for the `so.a.add` instructions.

```
struct : public arg_t {
  std::string to_string(insn_t insn) const {
    return ur_name[insn.uve_rd()];
  }
} urd;
```

```
struct : public arg_t {
  std::string to_string(insn_t insn) const {
    return ur_name[insn.uve_rs1()];
  }
} urs1;

struct : public arg_t {
  std::string to_string(insn_t insn) const {
    return ur_name[insn.uve_rs2()];
  }
} urs2;

struct : public arg_t {
  std::string to_string(insn_t insn) const {
    return pr_name[insn.uve_pred()];
  }
} upred;

static void NOINLINE add_uve_arith_insn(disassembler_t* d, const char*
 ↪  name, uint32_t match, uint32_t mask) {
  d->add_insn(new disasm_insn_t(name, match, mask, {&urd, &urs1, &urs2,
   ↪  &upred}));
}

void disassembler_t::add_instructions(const isa_parser_t* isa) {
  #define DEFINE_UATYPE(code) add_uve_arith_insn(this, #code, match_##code,
   ↪  mask_##code);
  DEFINE_UATYPE(so_a_add_fp);
  DEFINE_UATYPE(so_a_add_us);
  DEFINE_UATYPE(so_a_add_sg);
}
```

Listing 3.3: Disassembler structures and functions for UVE *add* instructions.

3.6 Summary

In this chapter, the most relevant modifications and additions to the *Spike* simulator were described in detail. The used files and structures were listed and key algorithms that were developed to effectively emulate a Streaming Engine (SE) were described, accompanied by examples in code snippets. Also, instruction implementation and necessary decoding functions were explained. Lastly, changes made to the disassembler were shown, a key component of the framework, as it is required by the debugger and allows trace generation.

Chapter 4

Unlimited Vector Extension Specification Revision

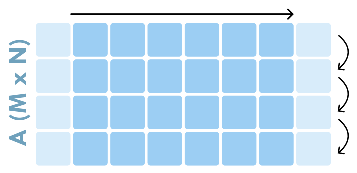
Throughout the simulator development procedure, several limitations were found in the original UVE specification, mainly related to the behaviour of specifi instructions, the stream execution model, and the set of benchmark applications. Therefore, the extension was fully revised and several improvements were introduced. Furthermore, some functional aspects that previously were either only envisioned or implicit in instruction definitions, were now formalised. This chapter describes the newly proposed modifications and additions made to the UVE specification, as well as the reasoning behind them.

4.1 Stream Configuration

According to the UVE specification, registers can be associated with streams, which are managed by the Streaming Engine (SE) and are implicitly loaded/stored from/to memory when read or written to. To configure each stream, a dedicated set of instructions is used, as described in Section 2.4.2. As the instruction set was tested, some shortcomings were found in the streaming interface, which led to de modifications proposed in this section.

4.1.1 Base Address and Offset

In its first version, UVE allowed any dimension of a stream pattern to be configured with the base address of the access as its *offset*. Consequently, while generating memory addresses, the SE would add the *offsets* of the dimensions, assuming they all corresponded to a *byte* value, as that is how memory addresses are interpreted. This did not pose a problem until now, as every tested pattern had dimension *offsets* equal to zero, except the one holding the memory base address for the stream data. However, when testing a *convolution* kernel, which involves the padding of the source matrix, the descriptor encoding requires setting *offsets* to non-zero values. A simplified version of an access of this kind is represented in Figure 4.1. It was found that the SE would not correctly compute the memory addresses in this case. Because this value corresponds to an element count, it is an integer and not a *byte* value. As such, to correctly compute the memory address of each element of the stream, the SE would need to multiply it by the element size (in bytes) and add it to the base address. This multiplication was not present in the original specification and would be impossible to perform correctly unless the dimension whose *offset* corresponded to the base address was distinguishable from the others.



(a) Padded pattern.

```
for (i = 0; i < M; ++i)
    for (j = 1; j < N-1; ++j)
        A[i*N+j];
```

(b) C loop.

Dimension 1:	**Dimension 2:**
Offset = A	Offset = 1
Size = N-2	Size = M
Stride = 1	Stride = N

(c) Stream representation.

Figure 4.1: Padded memory access pattern example, where the *offset* is non-zero.

An initial approach to solving this issue consisted of always defining the base address as the *offset* of the first dimension to be configured. This meant that `stream-start` configuration instructions would receive a memory address that did not require any additional computation. On the other hand, `stream-append` and `stream-end` configuration instructions would always be given *offset* values that required the multiplication by the element size, which was performed immediately. This way, the SE would have correct *offset* values, without additional computation.

Eventually, because stream configuration instructions suffered restructuring, as shown in Section 4.3.1, this idea was maintained and implemented in the new *header* instructions, which replace stream start configuration instructions.

4.1.2 Scalar Streams

Although UVE is a vector extension, it is primarily a streaming extension. With this in mind, there are many complex patterns that, while not vectorisable, can still benefit from data streaming. To support these cases, both the SE and the ISA must handle scalar streams. Although it was possible to implement scalar streams by configuring the stream length to 1, scalar code was not formally supported. Three possible approaches were studied:

- Extending the register bank with new streaming scalar registers, completely separate from the streaming vector registers;
- Adding streaming support to the native RISC-V scalar registers;
- Modifying the streaming vector registers to support scalar elements, with the SE handling the scalar elements as if they were vector elements with a single element;

Of these alternatives, the last one is the simplest and requires little hardware modifications relative to the existing specification, as opposed to the other two. From the SE perspective, everything works the same, as its control is always dependent on the vector length. This puts the burden of handling scalar streams on the ISA, which must guarantee their correct configuration. As such, new information is added to the stream table: a single flag that indicates whether the stream is scalar or vectorial. Because the base ISA is scalar, UVE streaming registers are by default also scalar.

Lastly, the scalar/vector property of a register is transient from source to destination. In accordance, when reduction or scalar instructions are used, the destination vector is always scalar (despite any previous configuration). However, if the destination is a vectorial load stream, an exception must be raised, even though it should not happen in well-structured code. Furthermore, if an arithmetic or logic instruction has at least one scalar source operand, the destination register is also scalar, even if previously configured as vectorial. When the operands are vectorial, the destination register is also configured as vectorial.

4.1.3 Dimensions and Modifiers

Order of dimension configuration

One of the main changes in the new specification is related to the order in which dimensions are appended to a descriptor. To match the order in which they are presented in a typical C/C++ `for` loop, this order has been inverted. This is because the original UVE specification stated that the first dimension is the innermost one (in Listing 4.1 the first dimension of the streams described in Listing 4.2 is the one equivalent to the loop in line 3). Previously, UVE code would present dimensions in the opposite order of what is expected, by appending dimensions from the first to the last one. This often led to confusion and errors, so it was changed. While this minor change does not affect the functionality of the extension, its usability was improved. Furthermore, it led to the elimination of certain instructions, as explained in Section 4.3.1.

```
// data is a N x M matrix; cov is a M x M matrix; double_n is (double)N

for (i = 0; i < M; i++)
    for (j = i; j < M; j++) {
        cov[i * M + j] = 0;
        for (k = 0; k < N; k++)
            cov[i * M + j] += data[k * M + i] * data[k * M + j];
        cov[i * M + j] /= double_n - 1.0;
        cov[j * M + i] = cov[i * M + j];
}
```

Listing 4.1: Snippet of the *covariance* C/C++ kernel code [68].

```
# cov[i * M + j]
ss.sta.st.d         u1, cov, M, M     # D2
ss.app.mod.ofs.inc u1, M, one
ss.app.mod.siz.dec u1, M, one
ss.end              u1, zero, M, one # D1

# cov[j * M + i]
ss.sta.st.d         u2, cov, M, one # D2
ss.app.mod.ofs.inc u2, M, M
ss.app.mod.siz.dec u2, M, one
ss.end              u2, zero, M, M   # D1
```

Listing 4.2: *Covariance* kernel UVE pseudo-assembly store stream configuration of streams with two modifiers per dimension.

Multiple Modifiers per Dimension

The original specification of UVE stated that only one modifier was allowed per dimension of a stream descriptor. This was found to be a limitation when trying to implement the *covariance* kernel (specifically, in the loop presented in Listing 4.1).

For the storing of the `cov` matrix pattern to be correctly defined, both the *offset* and the *size* of a dimension must be modified simultaneously. This is because, in each outer loop iteration, both the writing matrix index and the number of elements to be stored are changed. This is due to the indexing dependence on the second nested loop: in each outer loop iteration, the lower limit of the iteration is increased, which means that one less iteration is performed each time. The UVE stream configuration of accesses to the `cov` matrix is represented in Listing 4.2, already with two modifiers appended to the outermost dimension, which will affect the innermost one.

Explicit Target Dimension

When testing a new kernel with three nested loops, shown in Listing 4.3, it was found that some patterns could not be described with the current set of modifiers. This is because a dimension may need to be updated by a modifier that is iterated with a dimension of a higher order and not the dimension directly above it. This limitation can be solved by explicitly indicating the target dimension in the modifier appending instruction. This way, the modifier is still associated with the correct dimension, iterating along with it, but can affect an arbitrary dimension of the pattern. Formally, it also moves the iteration variable updates mirror the variable update order found in the equivalent `for` loop code.

```
1 // C is an N x N matrix; A is an N x M matrix; alpha and beta are scalars
2 for (i = 0; i < N; i++) {
3     for (j = 0; j <= i; j++)
4         C[i*N+j] *= beta;
5     for (k = 0; k < M; k++)
6         for (j = 0; j <= i; j++)
7             C[i*N+j] += alpha * A[i*M+k] * A[j*M+k];
8 }
```

Listing 4.3: *SYRK* (Symetric Rank-K Update) C/C++ computation kernel code [68].

As such, in the example from Listing 4.3, to accurately describe the 3D memory access pattern of `C[i*N+j]`, the *size* of the first dimension (correspondent to the loop in line 7) must be incremented every time the third dimension (correspondent to the loop in line 3) is iterated. This is due to the innermost loop being bounded by `i`, which is incremented two loops above. The result is that while the second dimension is iterated, the size of the first one remains unchanged, as `i` is not updated in the loop on line 6.

Consequently, the `C[i*N+j]` stream access pattern needs a *size* modifier appended to the last dimension, but targeting the first one, as shown in Listing 4.4. For this to be possible, modifier configuration instructions need to be extended to include the target dimension, as presented in Section 4.3.1.

Scatter-gather Dynamic Modifiers

Although the desired behaviour and description of scatter-gather accesses are the ones described in Figure 2.8 of Chapter 2, the original specification lacked ded-

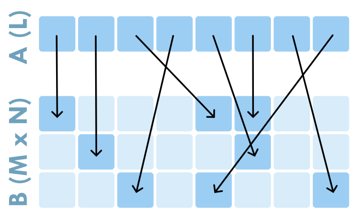
```
// C[i*N]+j
ss.sta.st.d          u3, C, N, N         # D3 (i)
ss.app.mod.siz.inc.1 u3, N, one          # Target: D1
ss.app               u3, zero, M, zero   # D2 (k)
ss.end               u3, zero, one, one  # D1 (j)
ss.cfg.vec           u3                  # vectorial stream
```

Listing 4.4: *SYRK* kernel UVE pseudo-assembly store stream configuration with explicitly defined target modifier.

icated support for this type of memory access. Traditional modifiers are applied when the dimension they are associated with is iterated. This means that, in order to describe scatter-gather accesses, the affected dimension must be scalar and the one above it iterates the number of elements to be accessed. This is a poor way of describing accesses that could be vectorial, as it wastes the SIMD capabilities of the UVE extension.

To solve this issue, a new kind of dynamic modifier was introduced. These modifiers are associated directly with the dimension that they affect, meaning that they are applied at each iteration (and each element), allowing registers to be filled with vectors. These modifiers are only defined for *offset* targets, as these are enough to perform scatter-gather accesses. However, other targets may be added in the future, if necessary. The updated example from Figure 2.8 is presented in Figure 4.2. This new descriptor is configured through a new set of scatter-gather modifier instructions.

```
ss.<app/end>.ind.ofs.sg.<behaviour>
```



(a) Scatter-gather pattern

```
for (i = 0; i < L; ++i)
    B[A[i]];
```

(b) C loop

Stream A
Dimension 1:
Offset = A
Size = L
Stride = 1

Stream B (vector)
Dimension 1:
Offset = B
Size = L
Stride = 0
Scatter-gather 1:
Offset, Add, A

(c) Stream representation

Figure 4.2: Scatter-gather memory access representation.

Support for this new descriptor type was added on *Spike* and was used in one of the benchmarks presented in Section 5.1.2, *SpMV-2*.

4.2 Predication Policies

As mentioned throughout this work, UVE supports lane control through predication. This means that in any SIMD instruction, there is the possibility of controlling which elements are operated on or not. This is achieved through the use of predicate registers, which are used to mask the operation of the instruction. The original specification simply stated that a predicated element would not take part in the computation, and the corresponding lane in the destination vector would be left unchanged. As seen in Section 2.2.2 and Section 2.2.1, this is known as *merging* predication in Arm SVE [6] and *undisturbed* tail/mask in RISC-V RVV[5].

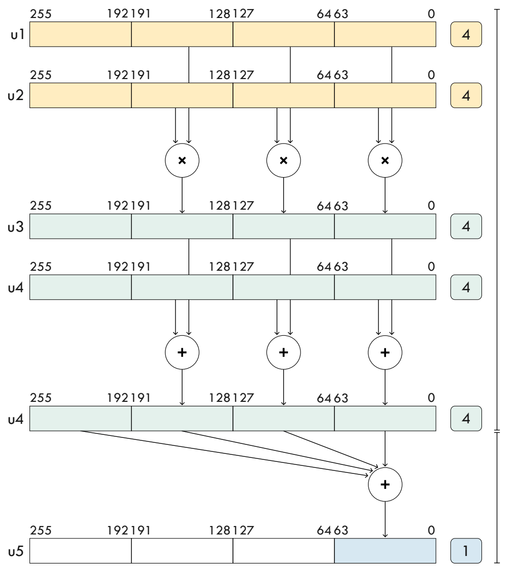
However, a different kind of predication exists, known as *zeroing* predication in SVE, where predicated lanes are set to 0 in the destination. This mode was not present in the original UVE specification, having been left for future work. Furthermore, no implicit vector predication policy was defined or specified, which was revealed to be an issue in most applications.

In particular, although considered in the original specification, an important aspect is related to what happens in the execution of instructions when non-complete vectors are used as operands. Until now, there were no explicit rules on how the number of valid elements of a vector register was managed, so it was entirely possible to have less data in a vector than the maximum allowed (e.g., a vector register configured to handle 8 elements but only 4 are valid). The initial approach was to simply take the number of valid elements of the source registers (i.e., the minimum between the valid elements from the source registers) and only operate on those. Then, this value was copied to the destination register, leaving the remaining elements unchanged. This was defined as *implicit vector predication*.

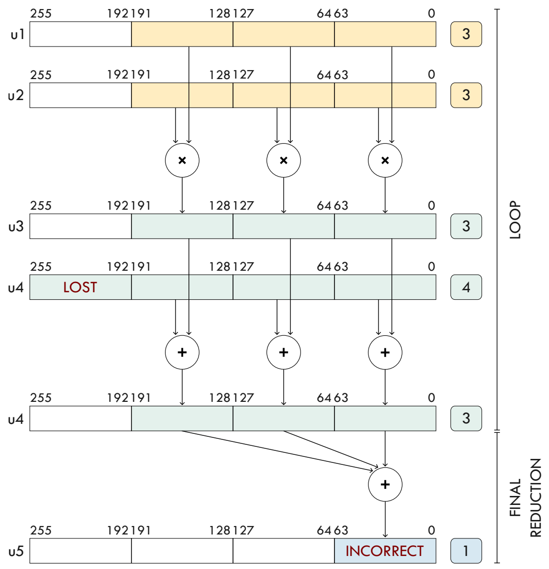
```
// u1 and u2 are load streams; u5 is a store stream

so.v.dp.d  u4, zero, p0 // fill u4 with 0s

jloop1 :
    so.a.mul.fp u3, u1, u2, p0 // u3 = u1 * u2
    so.a.add.fp u4, u4, u3, p0 // u4 = u4 + u3
    so.b.ndc.1 u1, .jloop1

so.a.adde.fp   u5, u4, p0      // reduce vector to scalar
```

Listing 4.5: Example of a reduction loop in UVE assembly code.

To understand where problems may arise, one can analyse Listing 4.5, where a common reduction loop is presented. In this case, both source operands of the multiplication are streams, which means that in each iteration, new values are implicitly loaded to these registers. However, the number of valid elements in these registers is not known, and although the vector is full in most iterations (Figure 4.3a), edge cases can occur where the vector is not full. This may lead to incorrect results, as partial sums are being stored in u4 at each iteration. The original specification states that the valid elements of the destination are set to the minimum between the valid elements of the sources. As Figure 4.3b illustrates, this results in incorrect values, as the partial sums in invalid lanes of u4 will be lost. On the other hand, if the valid elements of this register remain untouched, the result is correct, as the partial sums are not lost before the horizontal add in line 7.



(a) Computation with full vectors and correct result.



(b) Computation with incomplete vectors, which lead to incorrect result.

Figure 4.3: Reduction loops with flawed original implicit predication policy, assuming a vector length of 256 bits. Load streams are in yellow, auxiliary registers in green, and store streams in blue.

One possible solution to this issue is to redefine the implicit vector predication policy. It was defined that the only way to have invalid elements in a vector register is if is *a)* a scalar register, *b)* a load stream. In every other case, the vector register is always full. In particular, a store stream register may have valid elements with irrelevant data without issues, as the store pattern is assumed to be well-defined and those elements will not be stored in memory by the SE. This begs the question: if a vector register is the destination of an instruction with streams as sources, what results will be stored in lanes where the source register has invalid elements? The simple answer is to fill these lanes with zeroes, i.e., *zeroing* vector predication. With this policy, the reduction loop in Listing 4.5 would always produce correct results, as zero is the neutral element for addition.

This is illustrated in Figure 4.4.

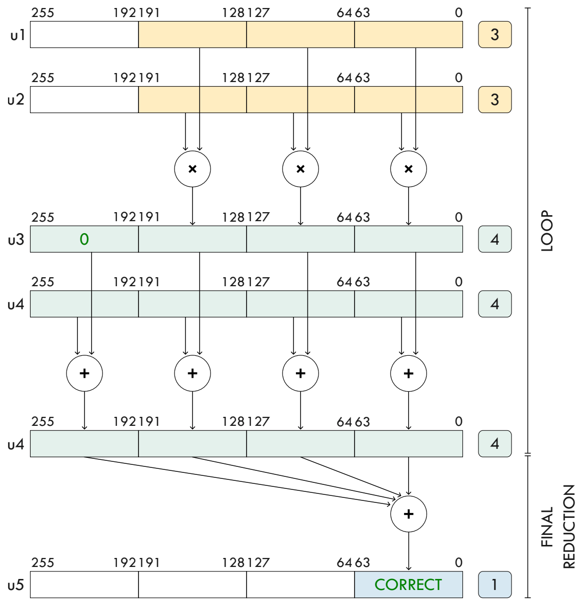


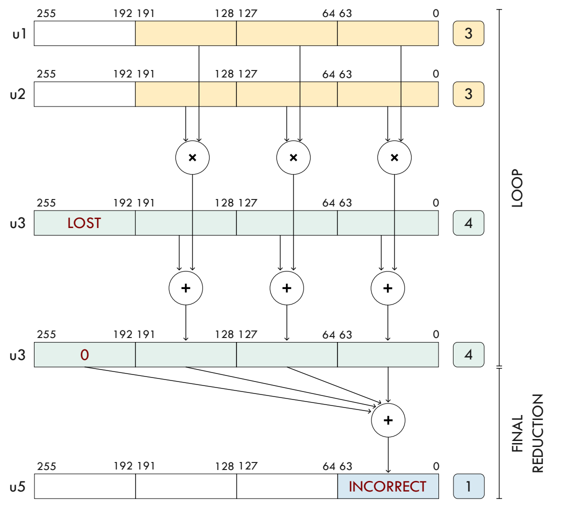
Figure 4.4: Reduction loop with *zeroing* predication.

With this modification, two new scenarios arise: what happens in accumulation instructions, and what happens if zero is not the neutral element for the operation (e.g., multiplication).

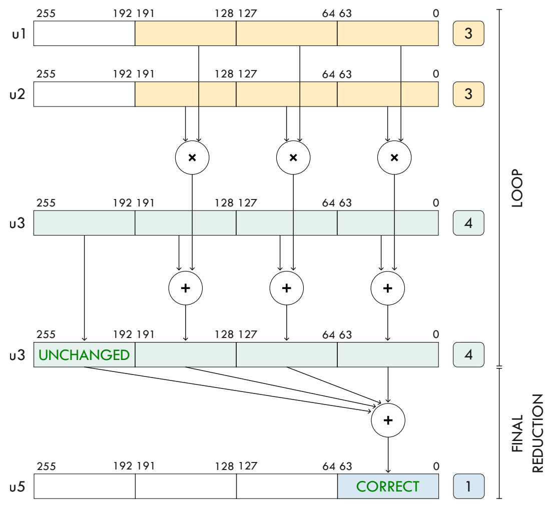
The first scenario presents itself in instructions such as `so.a.mac`, which could easily replace the two computation instructions in Listing 4.5, as it performs a multiply-accumulate operation. With *zeroing* predication, the behaviour of a reduction loop is presented in Figure 4.5a. In this case, the register that is subject to predication is itself the accumulator, which means that values must not be lost in any iteration. However, that is exactly what happens with *zeroing* predication in cases where, after full-vector iterations, the source registers have fewer elements. To solve this issue, the previous *merging* policy would be more adequate, while still marking all lanes as valid in the destination. This is illustrated in Figure 4.5b.

To handle the second scenario, two possible paths are possible: either fill invalid lanes with another value (e.g., 1 for multiplication) or reorganise the loop code and perform *merging* predication. The first approach was discarded, as a new predication mode for such specific situations would only convolute the specification. The second approach is ideal and easy to implement, as shown in Listing 4.6. By simply reorganising the multiplication instructions, the values accumulated in `u4` are never lost, and the result is always correct. This is possible whenever operations are commutative, which covers most accumulation cases. In reality, this is the order in which the multiplication would be performed in a scalar code, so it is a natural solution.

The devised solution aims to solve all the issues presented so far, by allowing



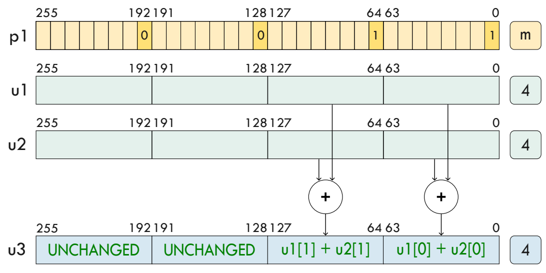
(a) Incorrect result with *zeroing* predication.



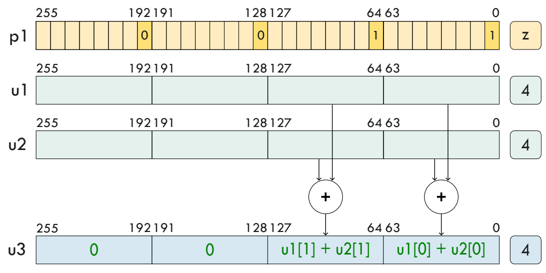
(b) Correct result with *merging* predication.

Figure 4.5: Reduction loop with *multiply-accumulate* instruction and both possible predication policies.

both *zeroing* and *merging* predication policies to be used. As observed, different situations require different approaches, and a single policy would not be enough to cover all of them. Moreover, a single rule for predication type would not guarantee that the correct policy would be applied every time. In accordance, it was decided that the predication policy would be explicitly encoded in each streaming register. This way, when a register is used as a source operand, the chosen predication policy is used to determine the behaviour of the instruction in the destination. Additionally, both modes are also added to explicit instruction predicates, which means that each predicate register is also configured to use a specific predication policy, as shown in Figure 4.6. This way, the predication policy is always explicit, and thus correctly applied. As a final note, *zeroing* was the policy chosen as the default for all streaming registers, as this is the most common case observed in studied kernels. For predicate registers, the default policy is *merging*, preserving its original behaviour, but allowing for the newly added *zeroing* predication. When executing a predicated instruction with source streams, the



(a) Merging predication.



(b) Zeroing predication.

Figure 4.6: Illustration of explicit predication in the `so.a.add` instruction.

predication policy of the stream prevails over the policy of the predicate register passed to the instruction.

```
// u1 and u2 are load streams; u5 is a store stream

so.v.dp.d  u4, zero, p0 // fill u4 with 0s

jloop1 :
    so.a.mul.fp u3, u1, u2, p0 // u3 = u1 * u2 (zeroing)
    so.a.mul.fp u4, u4, u3, p0 // u4 = u4 * u3
    so.b.nc u1, .jloop1

so.v.dp.d  u4, zero, p0 // fill u4 with 0s

jloop2:
    so.a.mul.fp u3, u4, u1, p0 // u3 = u4 * u1 (merging)
    so.a.mul.fp u4, u3, u2, p0 // u4 = u4 * u2 (merging)
    so.b.nc u1, .jloop1
```

Listing 4.6: Example of product accumulation loops in UVE assembly code.

4.3 Instruction Set Overview

Having laid out all the modifications to the UVE specification, this section shows how they are reflected in the encoding of the instructions. Because a thorough definition of the encoding was not provided in the original specification, this section will also serve as a reference for the encoding of instructions that did not suffer any alterations.

The encoding of these instructions is divided into several fields that are used to specify the operation to be performed and its operands. To simplify the representation of the encoding, some fields are presented with a specific notation. Additionally, the full encoding and bit fields are presented in Appendix C. This section closely follows the notation and conventions used in the official RISC-V specification.

4.3.1 Stream Configuration

Instructions responsible for the configuration of a stream, including its dimensions and modifiers, are at the core of data streaming from the programmer perspective and constitute the preamble of any streaming computational kernel.

As mentioned in Section 2.4, these instructions are found in the *custom-2* opcode region of RISC-V, called *StreamSet* in the context of UVE, with the mnemonic SS. To differentiate the different types of stream configuration instructions, the *tc* field is used. The *tc* field is a 2-bit field that encodes the type of stream configuration instruction, detailed in Table 4.1.

Table 4.1: Original *tc* field encoding.

***tc* field**	**Prefix**	**Meaning**
00	APP	Append dimension/modifier to configuration
01	END	Append dimension/modifier and end configuration
10	STA	Start configuration and append dimension
11	LD/ST	1D Load/Store configuration

Table 4.2: *width* field encoding.

***width* field**	**Suffix (WTH)**	**Meaning**
00	B	Byte (8 bits)
01	H	Half-word (16 bits)
10	W	Word (32 bits)
11	D	Double-word (64 bits)

Original Specification

In the original UVE, instructions that solely configure dimensions needed four operands: the destination register, the *size* and *stride* of the dimension, and the base address or *offset*. The `ss.ld`, `ss.st`, `ss.sta.ld`, and `ss.sta.st` instructions were used to start a stream associated with the given destination register. Since they were the first (or only) instructions of the stream definition, they also required the element width to be defined in the encoding, through a suffix in the instruction name (e.g., `ss.ld.w` or `ss.sta.st.d`). This information occupied the two least-significant bits of *funct3* and is here represented by the mnemonic WTH, detailed in Table 4.2.

The `ss.app` and `ss.end` instructions appended a dimension or modifier (distinguished in the *funct3* field) to the stream configuration associated with the given destination register.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	tc	rs2	rs1	funct3	vd	opcode
5	2	5	5	3	5	7
stride	11	size	base addr	LD.[WTH]	dest	SS
stride	11	size	base addr	ST.[WTH]	dest	SS
stride	STA	size	base addr	LD.[WTH]	dest	SS
stride	STA	size	base addr	ST.[WTH]	dest	SS
stride	APP	size	offset	000	dest	SS
stride	END	size	offset	000	dest	SS

Each static modifier configuration instruction took three operands: the destination register, the *size*, and the *displacement* value, whereas a dynamic modifier only required two: the destination and the source registers. The *funct3* and *l* fields encoded the type of modifier. The latter was a single bit field present in *dynamic modifier* instructions encoding, indicating whether the modifier was linked to the coupled dimension or if it required a *dimension hop (dhop)* which indicated to which dimension the modifier was linked. In the case of static modifiers, this information was encoded in the *funct3* field as well, and the only difference from typical modifiers was that the *size* operand was omitted and instead inferred from the dimension it was appended to. Furthermore, there were several different instructions for each type of modifier, with different *behaviour* and *target* values. These distinctions were made in the homonymous fields detailed in Table 4.3[1] and Table 4.4.

Table 4.3: *behaviour* field encoding.

***b* field**	**Suffix (B)**	**Meaning**
000	INC	Increment
001	DEC	Decrement
010	ADD	Add to base value
011	SUB	Subtract from base value
100	SET	Set to value

Table 4.4: *target* field encoding.

***ta* field**	**Suffix (T)**	**Meaning**
00	SIZ	Size
01	STR	Stride
10	OFS	Offset

31 27	26 25	24 22	21 20	19 15	14 12	11 7	6 0
rs3	tc	b	ta	rs1	funct3	vd	opcode
5	2	3	2	5	3	5	7
displacement	APP	B	T	size	MOD	dest	SS
displacement	END	B	T	size	MOD	dest	SS
displacement	APP	B	T	0	MODL	dest	SS
displacement	END	B	T	0	MODL	dest	SS

31	30 28	27	26 25	24 22	21 20	19 15	14 12	11 7	6 0
-	dh	l	tc	b	ta	rs1	funct3	vd	opcode
1	3	1	2	3	2	5	3	5	7
0	dimh	0	APP	B	T	src	IND	dest	SS
0	dimh	0	END	B	T	src	IND	dest	SS
0	000	1	APP	B	T	src	IND	dest	SS
0	000	1	END	B	T	src	IND	dest	SS

[1]The presented encoding of this field is updated relative to the original, which had two different encodings for INC and DEC in each type of modifier, despite having the same behaviour.

Finally, three additional configuration instructions existed, which either targeted a whole stream (`ss.cfg.ind` and `ss.cfg.mem`) or a single stream dimension (`ss.cfg.vec`). The first one was used to indicate if a stream was to be used as the source of a dynamic modifier of another stream, which in a real implementation would mean that data does not need to be moved to the CPU, never leaving the Streaming Engine (SE). The second one configured the cache-level access for that stream. The last one was used to indicate that a certain stream dimension was vector-coupled. A vector-coupled dimension stops the stream iteration once it reaches EOD, resuming once another request is made to the SE. This differs from the default behavior, which fills the register with values from the next dimension iteration. This instruction allowed for dimensions to be consumed individually, without mixing elements from other iterations to fill the vector. It was also first proposed that this instruction indicated that a stream is vectorial and not scalar, as discussed in Section 4.1.

31 27	26 25	24 18	17 12	11 7	6 0
-	tc	-	funct6	vd	opcode
5	2	7	6	5	7
0	00	0	CFG.IND	dest	SS
0	00	0	CFG.MEM[l]	dest	SS
0	00	0	CFG.VEC	dest	SS

Updates to the Specification

To accommodate for the newly proposed changes and additions, such as scatter-gather modifiers and the inversion of the dimension/modifier appending order, as well as to reduce the number of required configuration instructions, the encoding of the stream configuration instructions was modified.

Firstly, several configuration instructions were collapsed in one single *header* instruction, which is used to start a stream configuration, similarly to the `ss.sta` instructions, whose name was kept, but without a dimension or modifier. Instead, the encoding space was used to define what previously required the `ss.cfg` instructions, as well as new information about the predication mode (see Section 4.2). This was done by adding several new fields:

- ***pm* field**: a 1-bit field that indicates the predication mode of the stream. If set, the stream suffers from merging predication, otherwise, it suffers from zeroing predication, the default. If set, this field translates into the optional M suffix in the instruction name (i.e., `ss.sta.m`).

- ***vec* field**: a 1-bit field that indicates if the stream is vectorial. If set, the stream is vectorial, otherwise, it is scalar. If set, this field translates into the optional V suffix in the instruction name (i.e., `ss.sta.v`).

- ***vdim* field**: a 3-bit field that indicates the vector-coupled dimension of the stream. If no dimension is vector-coupled, this field is set to 111, as the default behaviour of the outermost dimension is similar to that of a vector-coupled dimension. A number is added to the instruction name to indicate the vector-coupled dimension (i.e., `ss.sta.v.1`), unless the field is set to 8, in which case the number is omitted.
- ***inds* field**: a 1-bit field that indicates if the stream is to be used as the source of a dynamic modifier of another stream. If set, this field translates into the optional INDS suffix in the instruction name (i.e., `ss.sta.inds`). For now, a stream can only be the source of a dynamic modifier if it is scalar, as only one value is used to modify the target per iteration, so an entire vector cannot be loaded. Improvements to this behaviour are left to future work, as the SE architecture complexity should be taken into account.
- ***mem* field**: a 2-bit field that indicates the cache-level access for that stream, from 0 (default) to 3. This field translates into the optional MEM[l] suffix in the instruction name (i.e., `ss.sta.mem2`), absent when it is 0.

While these instructions do not configure any dimension or modifier, they still configure the base address of the stream, consistent with the behaviour proposed in Section 4.1.1. Because of the removal of `ss.<ld/st>` instructions, the *tc* field encoding was updated and is summarised in Table 4.5.

Table 4.5: Updated *tc* field encoding.

***tc* field**	**Prefix**	**Meaning**
00	STA	Stream header instructions
01	APP	Append dimension/modifier
10	END	Append dimension/modifier and end configuration
11	-	Reserved

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0
pm	vec	vdim	tc	inds	mem	-	rs1	funct3	vd	opcode
1	1	3	2	1	2	2	5	3	5	7
PM	V	vdim	00	INDS	MEM[l]	00	base addr	LOAD.[WTH]	dest	SS
PM	V	vdim	00	0	MEM[l]	00	base addr	STORE.[WTH]	dest	SS

While instructions that append dimensions did not suffer any alterations, modifier instructions were updated to include the *tdim* field, replacing the operand that previously indicated the *size* of the modifier. This is a 3-bit field that encodes that *target* dimension of a modifier (i.e., the dimension it modifies). This takes advantage of the fact that the *size* of the modifier is in most cases the same as the *size* of the dimension it is linked to to solve the target dimension issue highlighted in Section 4.1.3. As such, previously existing `ss.app.modl` and `ss.app.indl` were removed, since every modifier instruction now has an explicitly defined target dimension and implicit *size*. This decision was made after observing that no studied pattern required a modifier to be applied to only a few iterations of a dimension. This behaviour is directly related to the induction variable dependencies in the loops that modifiers represent. In fact, the modifier *size* field was never used on *Spike* and has since been removed. Furthermore, due to the inversion of the order in which dimensions appear, `ss.end.modl` and `ss.end.indl` instructions were removed, as they are no longer necessary.

31 27	26 25	24 22	21 20	19 18	17 15	14 12	11 7	6 0
rs3	tc	b	ta	-	tdim	funct3	vd	opcode
5	2	3	2	2	3	3	5	7
displacement	APP	B	T	0	tdim	MOD	dest	SS

Lastly, with the encoding space freed by the removal of the aforementioned instructions, it was possible to add the new scatter-gather instructions. These instructions have the same encoding as ordinary dynamic modifiers but with the *sg* field set to 1. In this case, the suffix SG is added to the instruction name (e.g., `ss.app.ind.sg`). These instructions do not need a *tdim* field, as they are linked to the target dimension, as described in Section 4.1.

31	30 28	27	26 25	24 22	21 20	19 15	14 12	11 7	6 0
-	tdim	sg	tc	b	ta	vs1	funct3	vd	opcode
1	3	1	2	3	2	5	3	5	7
0	tdim	0	APP	B	T	origin	IND	dest	SS
0	0	SG	APP	B	10	origin	IND	dest	SS

4.3.2 Loop Control – Branching

These instructions control the flow of every UVE computation kernel, using EOD and EOS flags raised by the SE to branch accordingly, allowing to loop over data streams. These instructions belong to the *StreamOps* opcode region, with the mnemonic SO, and take two operands: the destination register and the offset to the target instruction, an immediate value scattered across the instruction encoding, similar to the original RISC-V *B-type* instructions [62]. The *n* field is a single bit that differentiates between dimension *complete* and *not complete* instructions (i.e., `so.b.dc.3` and `so.b.ndc.3`). Any pattern dimension can be used in the branch comparison, and its index is encoded in the 3-bit *d* field. In the case where this field is 111, the EOS flag is used to determine the branch outcome, and both the D prefix and the dimension index are omitted from the instruction name (i.e., `so.b.c` and `so.b.nc`).

31 29	28	27 22	21	20	19 15	14 12	11 8	7	6 0
funct3	imm[12]	imm[10:5]	n	d[2]	vs1	d[1:0] -	imm[4:1]	imm[11]	opcode
3	1	6	1	1	5	3	4	1	7
111	offset[12 | 10:5]		N	d[2]	src1	d[1:0]	offset[11 | 4:1]		SO

While revising the specification, it became clear that the *d* field could be encoded contiguously, as there was an unused bit at the right of *d[1:0]*. As such, the instruction was updated and *n* was also moved to the right, leading to a clearer encoding.

31 29	28	27 22	21	20	19 15	14 12	11 8	7	6 0
funct3	imm[12]	imm[10:5]	-	n	vs1	d	imm[4:1]	imm[11]	opcode
3	1	6	1	1	5	3	4	1	7
111	offset[12 | 10:5]		0	N	src1	d	offset[11 | 4:1]		SO

As a final note, some initially proposed predicate-based branch instructions were removed, as they were deemed unnecessary and redundant, given the new predication policies and valid element settings on vector registers.

4.3.3 Lane Control – Predication

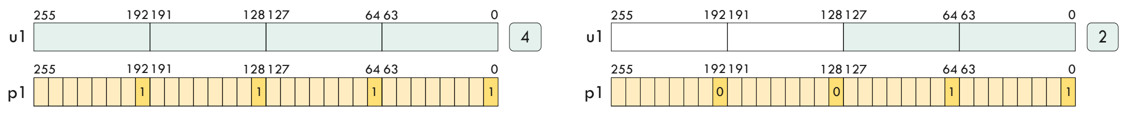
One key aspect of UVE is the ability to predicate instructions, which is done through the use of predicate registers that need to be configured before use. Belonging to the *StreamOps* opcode region, these instructions are responsible for the population of predicates.

Original Specification

Most instructions from the original specification retain their functionality in the new specification. The most simple predicate instructions, `so.p.zero` and `so.p.one`, simply take the destination register and set it to all zeroes or ones, respectively. They can also be predicated themselves, which means that a *predicate* is an operand of these instructions. The `so.p.not` instruction takes two operands: the destination register and the source register, and sets the former to the bitwise negation of the latter. The `so.p.mv` and `so.p.mvt` instructions take the same operands and move the source predicate into the destination, either directly or reversed, respectively.

There is also another instruction that takes an additional argument, a source vector register, and creates a mask from all its valid elements, which is then stored in the destination predicate register. This instruction is called `so.p.vr` and its behaviour is illustrated in Figure 4.7. This instruction can also be predicated, in which case the destination predicate register is always set to zero in lanes where the predicate mask is null.

```
so.p.vr p1, u1, p0 #dest, src1, pred
```



(a) Full source vector. (b) Incomplete source vector.

Figure 4.7: Illustration of `so.p.vr` instruction with different source vectors, assuming true instruction predicate (`p0`).

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	-	vs1	funct4	pd	opcode
4	3	5	5	4	4	7
1000	pred	0	0	ZERO	dest	SO
1000	pred	0	0	ONE	dest	SO
1000	pred	0	src1	VR	dest	SO

31 28	27 25	24 19	18 15	14 12	11 7	6 0
funct4	ps3	-	ps1	funct4	pd	opcode
4	3	6	4	4	4	7
1000	pred	0	src1	NOT	dest	SO
1000	pred	0	src1	MV	dest	SO
1000	pred	0	src1	MVT	dest	SO

Originally, predicates could also be generated from vector-vector and vector-scalar comparisons, which took an extra operand for the second source register. Additionally, because arithmetic operations were involved, the type of computation was encoded in the *funct3* field, according to Table 4.6. It should be noted that two versions of each instruction were available, one that took two vector registers and performed an element-wise comparison, and one that took a scalar register whose value was compared to each value in the *src1* vector register.

Table 4.6: *fps* field encoding.

***fps* field**	**Suffix (FPS)**	**Meaning**
00	US	Unsigned integer operation
01	FP	Floating-point operation
10	SG	Signed integer operation
11	-	Reserved

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	vs2	vs1	funct4	pd	opcode
4	3	5	5	4	4	7
1000	pred	src2	src1	EGT.[FPS]	dest	SO
1001	pred	src2	src1	EQ.[FPS]	dest	SO
1001	pred	src2	src1	LT.[FPS]	dest	SO

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	rs2	vs1	funct4	pd	opcode
4	3	5	5	4	4	7
1000	pred	src2	src1	EGTS.[FPS]	dest	SO
1001	pred	src2	src1	EQS.[FPS]	dest	SO
1001	pred	src2	src1	LTS.[FPS]	dest	SO

Lastly, because predicates may result from arithmetic operations on vectors with an arbitrary element width, despite not having a defined data type themselves, conversion instructions are necessary to adapt a previously defined predicate to be applied to other vector operands of a different data type. This conversion was done through the use of the `so.p.cv` instruction, which took the source predicate and the destination register as operands. It also encoded the element width to which the source predicate was to be converted in the *width* field, according to Table 4.2, which was traduced into a suffix in the instruction name (e.g., `so.p.cv.b`).

31 28	27 25	24 22	21 20	19	18 15	14 11	10 7	6 0
funct4	ps3	-	width	-	ps1	funct4	pd	opcode
4	3	3	2	1	4	4	4	7
1000	0	0	WTH	0	src1	CV	dest	SO

Updates to the Specification

Starting with the conversion instructions, it was noticed during the development of this work that these instructions lacked information about the data type used to configure the source register. While this is not a problem in vector conversion instructions (see Section 4.3.4), predicate registers do not contain any information about the data type, contrary to UVE vector registers. To perform a conversion, the data width of the source predicate must be known, as it determines how the source predicate is interpreted and converted. As such, the instruction now features two width fields, *dw* and *sw*, for the destination and source predicates, respectively. The behaviour of these instructions is exemplified in Figure 4.8.

`so.p.cv.d.w p2, p1` `so.p.cv.w.d p2, p1`

31 28	27 25	24	23 22	21 20	19	18 15	14 11	10 7	6 0
funct4	-	pm	dw	sw	-	ps1	funct4	pd	opcode
4	3	1	2	2	1	4	4	4	7
1000	0	PM	WTH	WTH	0	src1	CV	dest	SO

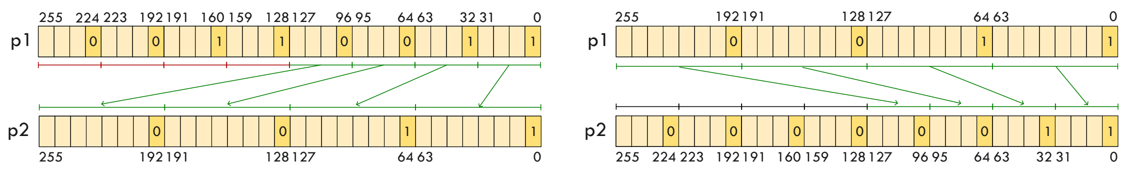
Additionally, a change that is present in every predicate instruction is the inclusion of the *pm* field, which indicates the predication mode of the stream, similar to the stream configuration instructions in Section 4.3.1, and described in Section 4.2. In this case, because merging is the default behaviour, if the bit is set then zeroing is chosen and a suffix is added to the instruction name (i.e., `so.p.zero.z`).

31 28	27 25	24	23 20	19 15	14 12	11 7	6 0
funct4	ps3	pm	-	vs1	funct4	pd	opcode
4	3	1	4	5	4	4	7
1000	pred	PM	0	0	ZERO	dest	SO
1000	pred	PM	0	0	ONE	dest	SO
1000	pred	PM	0	src1	VR	dest	SO

31 28	27 25	24	23 19	18 15	14 12	11 7	6 0
funct4	ps3	pm	-	ps1	funct4	pd	opcode
4	3	1	5	4	4	4	7
1000	pred	PM	0	src1	NOT	dest	SO
1000	pred	PM	0	src1	MV	dest	SO
1000	pred	PM	0	src1	MVT	dest	SO

The comparison instructions were also revised, as there was no available encoding space for the required *pm* field. Upon analysis of the instruction set, scalar comparisons were deemed dispensable, as they can be replaced with an `so.v.mvsv` instruction followed by a regular vector predicate comparison. As such, the scalar comparison instructions were removed, freeing up a bit 11, previously belonging to *funct4* field. The naming of the *greater or equal* comparison instruction was also changed to more closely resemble typical ISA naming conventions.

31 28	27 25	24 20	19 15	14 12	11	10 5	6 0
funct4	ps3	vs2	vs1	funct3	pm	pd	opcode
4	3	5	5	3	1	4	7
1000	pred	src2	src1	GE.[FPS]	PM	dest	SO
1001	pred	src2	src1	EQ.[FPS]	PM	dest	SO
1001	pred	src2	src1	LT.[FPS]	PM	dest	SO



(a) Element widening (word to double-word). (b) Element narrowing (double-word to word).

Figure 4.8: Illustration of `so.p.cv` instruction with different source and destination widths.

4.3.4 Vector Manipulation

These instructions allow for the transferring of data between vector registers, as well as the conversion between vectors of different data types. They belong to the *StreamOps* opcode region, with the mnemonic SO, and remain unchanged from the original specification, apart from the removal of some instructions. Vector load/store instructions were deemed unnecessary, as they were simple non-streaming vector memory access instructions, which can be easily replicated with linear streams. Moreover, move instructions had *"no stream"* variants, which did not trigger the iteration of source/destination streams, as UVE instructions do by default when reading or writing from/to a register associated with a stream. Because data streaming is the main focus of UVE, these instructions were deemed deprecated for now, simplifying the instruction set.

The remaining instructions take up to three operands: the destination, source, and predicate registers. Two simple *move* instructions are available, `so.v.mv`, which moves the source vector into the destination, and `so.v.mvt` which first reverses the vector. To perform vector-to-scalar and scalar-to-vector moves, `so.v.mvvs` and `so.v.mvsv` are available, respectively. It should be noted that in these cases, the destination register remains configured as scalar. Another instruction is available to create a vector from a scalar, `so.v.dp`, which broadcasts the scalar value to the destination vector register, which is configured as vectorial.

31 27	26 23	22 20	19 15	14 12	11 7	6 0
funct5	funct4	ps2	vs1	funct3	vd	opcode
5	4	3	5	3	5	7
10101	MV	pred	src1	0	dest	SO
10101	MVT	pred	src1	0	dest	SO

31 27	26 23	22 20	19 15	14 12	11 7	6 0
funct5	funct4	ps2	vs1	funct3	rd	opcode
5	4	3	5	3	5	7
10101	DP	pred	src1	WTH	dest	SO

31 27	26 23	22 20	19 15	14 12	11 7	6 0
funct5	funct4	ps2	vs1	funct3	rd	opcode
5	4	3	5	3	5	7
10101	MVVS	0	src1	0	dest	SO

31 27	26 23	22 20	19 15	14 12	11 7	6 0
funct5	funct4	ps2	rs1	funct3	vd	opcode
5	4	3	5	3	5	7
10101	MVSV	0	src1	WTH	dest	SO

To be able to convert the elements of a vector to different data types, the `so.v.cv` instruction is available, which takes the source vector and the destination register as operands. This instruction performs the necessary narrowing or widening of the source vector to fit the destination element width, which is encoded in the *width* field, according to Table 4.2. These operations have a similar behaviour as predicate conversion ones, illustrated in Figure 4.8. However, the behaviour of the block of data that would be lost is not clearly defined yet. It was initially proposed that the elements in the source register were implicitly right-shifted after the conversion and, if associated with a stream, new values were loaded to the now empty lanes. However, this behaviour has not been validated and further testing is necessary, which is left for future work.

31 27	26 23	22 20	19 15	14 12	11 7	6 0
funct5	funct4	ps2	rs1	funct3	vd	opcode
5	4	3	5	3	5	7
10110	CV.[FPS]	0	src1	WTH	dest	SO

4.3.5 Vector Control

In the *StreamOps* encoding space, two instructions to configure the vector length are available, both to set and get the value from the `VLEN` CSR, `so.c.setvl` and `so.c.getvl`, respectively. The `so.c.setvl` instruction takes the source register as an operand, which contains the new value for `VLEN`, in bytes, and the destination register, which is used to store the new value of `VLEN`. This value is the minimum between the requested value and the maximum allowed, `VLMAX`, similar to the equivalent RVV instructions (see Section 2.2.1). After the execution of the instruction, the `VLEN` CSR is updated with the new value, and every vector register is configured to the new length.

31 27	26 20	19 15	14 12	11 7	6 0
funct5	-	rs1	funct3	rd	opcode
5	7	5	3	5	7
10110	0	src1	SETVL	dest	SO
10110	0	src1	GETVL	dest	SO

The remaining instructions allow for explicit control over the streams, such as suspension and resuming, as well as definite breaking of the stream: `so.v.suspd`, `so.v.resum`, and `so.v.break`, respectively. Additionally, the `so.v.vload` and `so.v.vstor` instructions are available to load and store data from and to suspended streams.

31 27	26 15	14 12	11 7	6 0
funct5	-	funct3	vd	opcode
5	12	3	5	7
10110	0	SUSPD	dest	SO
10110	0	RESUM	dest	SO
10110	0	BREAK	dest	SO
10110	0	VLOAD	dest	SO
10110	0	VSTOR	dest	SO

4.3.6 Arithmetic and Logic Instructions

The instructions that perform SIMD arithmetic and logic operations on UVE vector registers (part of the *StreamOps* encoding space) are the most common in computation loops. Although the element width is encoded in each register, the data type as seen in Table 4.6 is not, so it must be present in each arithmetic instruction. Simple instructions take two source operands and a destination, as well as a predicate register, and perform an element-wise operation. Particularly, `so.a.mac` is a multiply-accumulate instruction, which multiplies the source operands and adds the result to the value already in the destination register.

There is also an instruction to obtain the absolute value of every vector element, which takes only one source operand, `so.a.abs`, which does not have an *unsigned* version, as it can only operate on *signed* types. The `so.a.inc` and `so.a.dec` instructions increment or decrement its only source operand, respectively, storing the result in the destination register.

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	vs2	vs1	funct3	vd	opcode
4	3	5	5	3	5	7
0000	pred	src2	src1	ADD.[FPS]	dest	SO
0000	pred	src2	src1	SUB.[FPS]	dest	SO
0001	pred	src2	src1	MUL.[FPS]	dest	SO
0001	pred	src2	src1	DIV.[FPS]	dest	SO
0011	pred	0	src1	ABS.[FP]	dest	SO
0011	pred	src2	src1	MAC.[FPS]	dest	SO
0100	pred	src2	src1	MIN.[FPS]	dest	SO
0100	pred	src2	src1	MAX.[FPS]	dest	SO
0110	pred	0	src1	INC.[FP]	dest	SO
0110	pred	0	src1	DEC.[FPS]	dest	SO

Some reduction instructions are also present in this extension, which instead of taking two source operands and performing an element-wise operation, take only one source operand and a destination, and perform a reduction operation on the vector elements, storing the scalar result in the destination stream register, which is therefore configured as scalar.

Addition is a special case in this set of instructions, as it is the only reduction operation that can accumulate the result with the destination register (`so.a.adde.acc`), which can also be a regular RISC-V scalar register (`so.a.adds.acc` and `so.a.adds.acc`). In particular, in the floating-point variants of these instructions (`so.a.adds.fp` and `so.a.adds.acc.fp`), a RISC-V floating-point register is required as the destination. This versatility is useful for accumulating the result of a reduction operation in a loop, as it can be done in a single instruction, something very common in computation kernels (e.g., *SGD, GEMVER, covariance* used in Section 5.1.2).

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	acc	vs1	funct3	vd	opcode
4	3	5	5	3	5	7
0010	pred	ACC	src1	ADDE.[FPS]	dest	SO
0101	pred	0	src1	MINE.[FPS]	dest	SO
0101	pred	0	src1	MAXE.[FPS]	dest	SO

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	acc	vs1	funct3	fd	opcode
4	3	5	5	3	5	7
0010	pred	ACC	src1	ADDS.[FPS]	dest	SO

Lastly, several SIMD logic instructions are available, which perform bit-wise operations on the source operands, storing the result in the destination register. In detail, both logical (zero-extending) and arithmetic (sign-extending) shift right instructions are available, as seen in RVV, (`so.a.srl`) and `so.a.sra`, respectively. The shift-amount value is encoded in the second source operand, which can be a vector or regular RISC-V register.

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	vs2	vs1	funct3	vd	opcode
4	3	5	5	3	5	7
1100	pred	src2	src1	NAND	dest	SO
1100	pred	src2	src1	AND	dest	SO
1100	pred	src2	src1	OR	dest	SO
1100	pred	src2	src1	NOR	dest	SO
1100	pred	src2	src1	XOR	dest	SO
1100	pred	0	src1	NOT	dest	SO
1101	pred	src2	src1	SLL	dest	SO
1101	pred	src2	src1	SRL	dest	SO
1101	pred	src2	src1	SRA	dest	SO

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	rs2	vs1	funct3	vd	opcode
4	3	5	5	3	5	7
1101	pred	src2	src1	SLLS	dest	SO
1101	pred	src2	src1	SRLS	dest	SO
1101	pred	src2	src1	SRAS	dest	SO

4.4 Summary

This chapter detailed how some issues were found, through tests performed on the simulator presented in Chapter 3, which revealed caveats in the original specification. Furthermore, while attempting to describe more complex patterns in new computation kernels, several features were found to be missing. These include:

- The need for multiple modifiers per dimension;
- The necessity of explicitly indicating the target dimension of a modifier;
- Scatter-gather dynamic modifiers;
- The need for scalar streams;
- New predication policies.

Then, the new ISA encoding was presented, with all the necessary changes to support these features. Most of these features were also implemented on the simulator, which was used to test the new specification. The results of these tests are presented in Section 5.1.2.

It is important to note that the new specification is not yet final, and as more complex patterns are tested, new features may be added. However, the current specification is already a significant improvement over the original one, and it represents the base for the first stable release of the UVE extension. Moreover, support for some of the new features, such as the new scatter-gather dynamic modifiers, has yet to be added to the microarchitecture of UVE and tested in hardware. While this is outside the scope of this work, it is expected that the new features will be implemented in the future, and hardware constraints may lead to a new revision of the extension.

Chapter 5

Experimental Results and Discussion

In order to provide a meaningful evaluation of any instruction set, ISA simulators are widely used and indispensable tools [73]. One of the main goals of this work was to provide a framework that allowed the development and validation of the UVE ISA, based on *Spike*, the official RISC-V ISA simulator, as described in Chapter 3. This chapter presents the proposed framework beyond the simulator, as well as ISA performance evaluation results it provides.

5.1 Framework

After extending the *Spike* simulator to support data streaming and UVE instructions, other components were added to the framework to provide a complete environment for the development and validation of the UVE ISA. The complete design of the framework is shown in Figure 5.1.

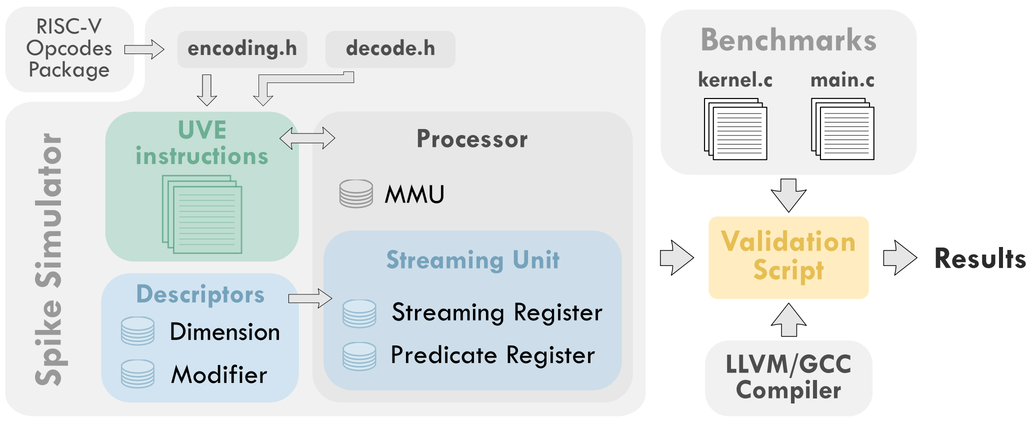


Figure 5.1: Framework structure.

5.1.1 Supported Features

As seen in Chapter 4, the specification of the extension was updated in parallel to the development of the simulator. Most of the described features were implemented in the simulator, which currently supports the following instructions:

- 58 arithmetic and logic instructions;
- 16 branch instructions;
- 24 predicate instructions;
- 11 vector manipulation instructions;
- 49 stream configuration instructions.

This totals 158 instructions across all types except *vector control*, which was not implemented, as it is more related to the architecture and not the functional behaviour of the extension, the focus of this work. Stream configuration instructions are by far the largest group of instructions, both in the old specification and in the updated version. Now deprecated *modl* and *ind* instructions were never implemented in the simulator, which amounted to 240 indirection instructions and 12 for static modifiers. This left 30 instructions for dynamic modifiers and 12 for static modifiers, which were all implemented on the simulator. However, because of the changes described in Section 4.3.1, END instructions were later removed, cutting this number to half. With the addition of 10 new scatter-gather instructions, a total of 31 modifier-related instructions exist and are all implemented in the simulator. Despite this, the new explicitly defined target modifier appending ones are not yet present in the simulator (amounting to 48 instructions for static modifiers and 120 for dynamic modifiers). These were left out due to the time constraints of this work, but they are expected to be implemented in the near future. As the only studied benchmark that required this functionality was *SYRK* (Symmetric Rank-K Update) (see Section 4.1.3), it is not used in this work.

As it stands, the simulator is still lacking support for some new features, such as explicit predication policy configuration, new predicate width-conversion instructions, and header instructions. However, it supports the most important features of the new specification, as well as all the maintained features from the original one, excluding cache and direct stream control (`so.c` instructions).

5.1.2 Benchmarks

In order to validate UVE, as any other ISA extension, a wide set of benchmarks is necessary, which allows all features to be tested and instructions to be executed.

This extension is intended to be used in applications that can benefit from the vectorisation of their data, which are present in a variety of domains, including memory-intensive applications such as Artificial Intelligence (AI)-based ones, linear algebra, stencils, and data mining. As such, several benchmarks from these fields of applications were implemented by making use of UVE, not only to showcase its usage but also to demonstrate its potential in instruction reduction. The original C/C++ code was taken from public benchmark suites [68, 74].

The complete set of used benchmarks and their characteristics are shown in Table 5.1. An attempt to include a representative set of benchmarks was made both in terms of complexity and memory access patterns. There is at least one kernel where each type of modifier is employed, *covariance* was successfully implemented with two modifiers in a single dimension, and the novel scatter-gather dynamic modifier was included in a version of *SpMV* (Sparse Vector-Matrix Multiplication). This is an important benchmark, as it was only possible to implement

Table 5.1: Benchmarks used for testing and respective characteristics.

	Benchmark	Num. of Streams	Num. of Kernels*	Max. Loop Nesting	Memory Access Pattern
Memory	**Memcpy**	1	1	1	1D
	Stream	10	4	2	2D
BLAS	**SAXPY**	3	1	1	1D
	GEMM	6	2	3	3D
Algebra	**3MM**	3	3	3	3D
	MVT	8	2	2	2D
	GEMVER	17	4	2	2D
	Trisolv	5	1	2	2D + Static Modifier
Stencil	**Jacobi-1D**	8	2	1	1D
	Jacobi-2D	12	2	2	2D
ML/AI	**Convolution**	10	1	1	2D
	SGD	9	3	3	3D
Data Mining	**Covariance**	9	3	3	4D + 2 Static Modifiers
	SpMV-1	4	1	2	3D + Dynamic Modifier
	SpMV-2	6	1	2	2D + Dynamic SG Modifier

* The number of kernels corresponds to the number of disjunct loop statements (i.e., excluding nested loops).

due to the proposed changes to the original specification, representing a common Sparse Linear Algebra computation kernel, one of the main fields of application targeted by this work.

In each benchmark, RISC-V instruction `rdinsret` was added before and after each kernel. This instruction reads the `INSTRET` CSR, which is part of the base ISA specification and supported by *Spike*. The value that is stored in the CSR is the number of retired instructions[1] at the point of execution. By reading this value in these two moments, it is possible to obtain the dynamic instruction count, which differs from a static count that is obtained by simply disassembling a program with a compiler. It is a much more meaningful metric for ISA performance evaluation, as it is the exact number of executed instructions (e.g., with executed loops). This instruction count was manually validated in the *SAXPY* kernel, to ensure that the number of executed instructions was as expected.

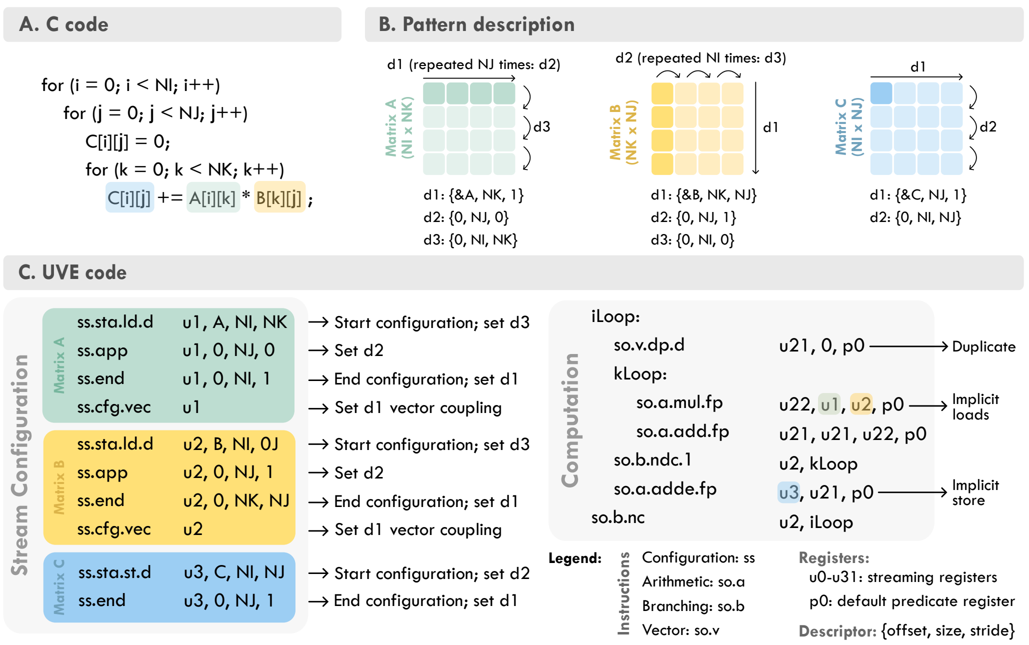
Matrix Multiplication UVE Implementation Example

To better understand how C/C++ code can be ported to UVE with the specification that is currently supported by the simulator, Figure 5.2 illustrates how a matrix multiplication kernel was implemented in the *3mm* benchmark (three-matrix-multiplication).

Each source matrix is described in three dimensions, each equivalent to a loop in the original C/C++ code. The innermost loop loads matrix `A` line by line, and matrix `B` column by column. In order to correctly perform the matrix multiplication, each line of `A` is loaded `NJ` consecutive times, while `B` is loaded column by column `NI` times. Each line is multiplied by the respective column, `A[i][k]*B[k][j]`, and the result is accumulated for each `C[i][j]` element. This matrix only requires two dimensions to be described, as it is accessed element by element. In fact, it could also be described in one dimension, as it is accessed sequentially (`d1:{&C, NI*NJ, 1}`). The processing of each line and column must be performed in separate iterations, not to mix elements from other lines/columns in the vector register. This is why the `ss.cfg.vec` instruction is employed in the first dimension of both `A` and `B`, as it stops the SE from proceeding with the iteration process and filling the vector register once a line/column has been completely loaded.

The translation of the descriptors in Figure 5.2B into UVE code is shown in Figure 5.2C, which also includes the computation kernel. This kernel performs the multiplication of the vectors and the accumulation of the results, which is then stored in the `C` matrix. Innermost loop control is achieved by checking if the first dimension of `B` has ended, i.e., if a single column has been consumed (`A`

[1]Instructions Retired is a commonly used performance evaluation metric which corresponds to the number of instructions that were completely executed (i.e., excluding speculatively executed instructions that end up not being necessary for the program flow).



```
for (i = 0; i < NI; i++)
  for (j = 0; j < NJ; j++)
    C[i][j] = 0;
    for (k = 0; k < NK; k++)
      C[i][j] += A[i][k] * B[k][j] ;
```

```
ss.sta.ld.d   u1, A, NI, NK    → Start configuration; set d3
ss.app        u1, 0, NJ, 0     → Set d2
ss.end        u1, 0, NI, 1     → End configuration; set d1
ss.cfg.vec    u1               → Set d1 vector coupling
ss.sta.ld.d   u2, B, NI, 0J    → Start configuration; set d3
ss.app        u2, 0, NJ, 1     → Set d2
ss.end        u2, 0, NK, NJ    → End configuration; set d1
ss.cfg.vec    u2               → Set d1 vector coupling
ss.sta.st.d   u3, C, NI, NJ    → Start configuration; set d2
ss.end        u3, 0, NJ, 1     → End configuration; set d1
```

```
iLoop:
  so.v.dp.d       u21, 0, p0
  kLoop:
    so.a.mul.fp   u22, u1, u2, p0
    so.a.add.fp   u21, u21, u22, p0
  so.b.ndc.1      u2, kLoop
  so.a.adde.fp    u3, u21, p0
so.b.nc           u2, iLoop
```

Figure 5.2: Matrix multiplication kernel as implemented on the developed framework.

could have similarly been used, instead checking if a line has been consumed, which is equivalent as both have size `NK`). The final reduction is performed after this loop, with implicit storing of the result in the store stream. The outermost loop executes until the `B` load stream finishes, i.e., until each column of matrix `B` has been accessed `NI` times (which is also when the `A` load stream and `C` store stream finish).

5.1.3 Additional Artefacts

To make the proposed framework work seamlessly and to provide a complete environment for the development and validation of the UVE ISA, several additional artefacts were included. These include a set of scripts to automate the process of compiling, running, and validating the results from each benchmark. Developed in JavaScript, they allow the developer to easily compare results from different runs, different data types, and different compilation flags[2][3].

Both GCC and LLVM are included in the framework, but only GCC has full UVE backend support (i.e., assembly instruction recognition). While the LLVM compiler is still under development, autovectorisation of some patterns is already possible. Benchmarks *GEMM*, *GEMVER*, *Convolution*, *Covariance*, and *SpMV-1/2* were autovectorised by LLVM and hand-verified and corrected. This means that

[2]`https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html`
[3]`https://clang.llvm.org/docs/CommandGuide/clang.html`

the developed framework was already used to validate UVE autovectorisation support by LLVM, which was also one of its intended purposes. It is expected that this compiler will soon be completely integrated into the framework, replacing GCC as the main compiler.

5.2 Instruction Count Evaluation

The developed framework makes it possible to assess the performance of the UVE ISA in terms of instruction count. Originally, this evaluation was performed using a modified version of the *gem5* simulator, which did not support RVV at that time. As such, the original specification was compared with Arm SVE and Neon, which were supported by the simulator. This evaluation was dependent on the implemented microarchitecture, based on the Arm Cortex-A76.

In this work, by using the RISC-V ISA simulator, *Spike*, it is possible to validate the ISA independently from a chosen microarchitecture. Moreover, it supports RVV, the official vector extension of the base ISA. As such, each benchmark presented in Table 5.1 from Section 5.1.2 was executed using UVE and RVV. Each RVV kernel was obtained directly from the scalar C/C++ code through the LLVM compiler with autovectorisation support[4]. With this in mind, RVV code was not hand-optimised, opposite to UVE which was manually vectorised directly in assembly. To minimise the impact of this difference, the RVV kernels were compiled with full optimisation (`-O3` flag activated), with and without loop unrolling. On *Spike*, the RVV extension was enabled with `VLMAX=512` and `ELEN=64`, to match UVE configurations.

Each kernel was written for the four data widths supported by UVE: *byte* (8-bit), *half-word* (16-bit), *word* (32-bit), and *double* (64-bit) whenever possible. The first two were only validated for *signed integer* data type operations, while *word* kernels were validated for both *signed integer* and *floating-point* operations. The *double* data type was only validated for *floating-point* operations. The size of the dataset was 2500 elements for all benchmarks, except for *SGD* and *SpMV-1/2*, which used specific datasets. *SGD* (Stochastic Gradient Descent) used a dataset with 4420 elements and was executed during 100 epochs. *SpMV-1/1* (Sparse Matrix-Vector Multiplication) used an *ellpack* format sparse matrix [75, 76] with 1666 non-zero elements[5]. The *convolution* kernel performed a 2D convolution with a 3x3 kernel on a 50x50 image.

Figure 5.3 shows the retired instruction count for each benchmark, for each data type. The number of instructions tends to increase with the data width for UVE, which is expected, as the number of elements processed in parallel de-

[4]`https://llvm.org/docs/Vectorizers.html`

[5]`https://sparse.tamu.edu/HB/494_bus`

creases. This leads to more loop iterations being necessary to process the same chunk of data. However, this behaviour is not observed for RVV, whose behaviour is less consistent. This is presumably due to the autovectorisation process, which was not hand-optimised. The comparison between scalar, RVV and UVE code is presented in Figure 5.4 and Figure 5.5.

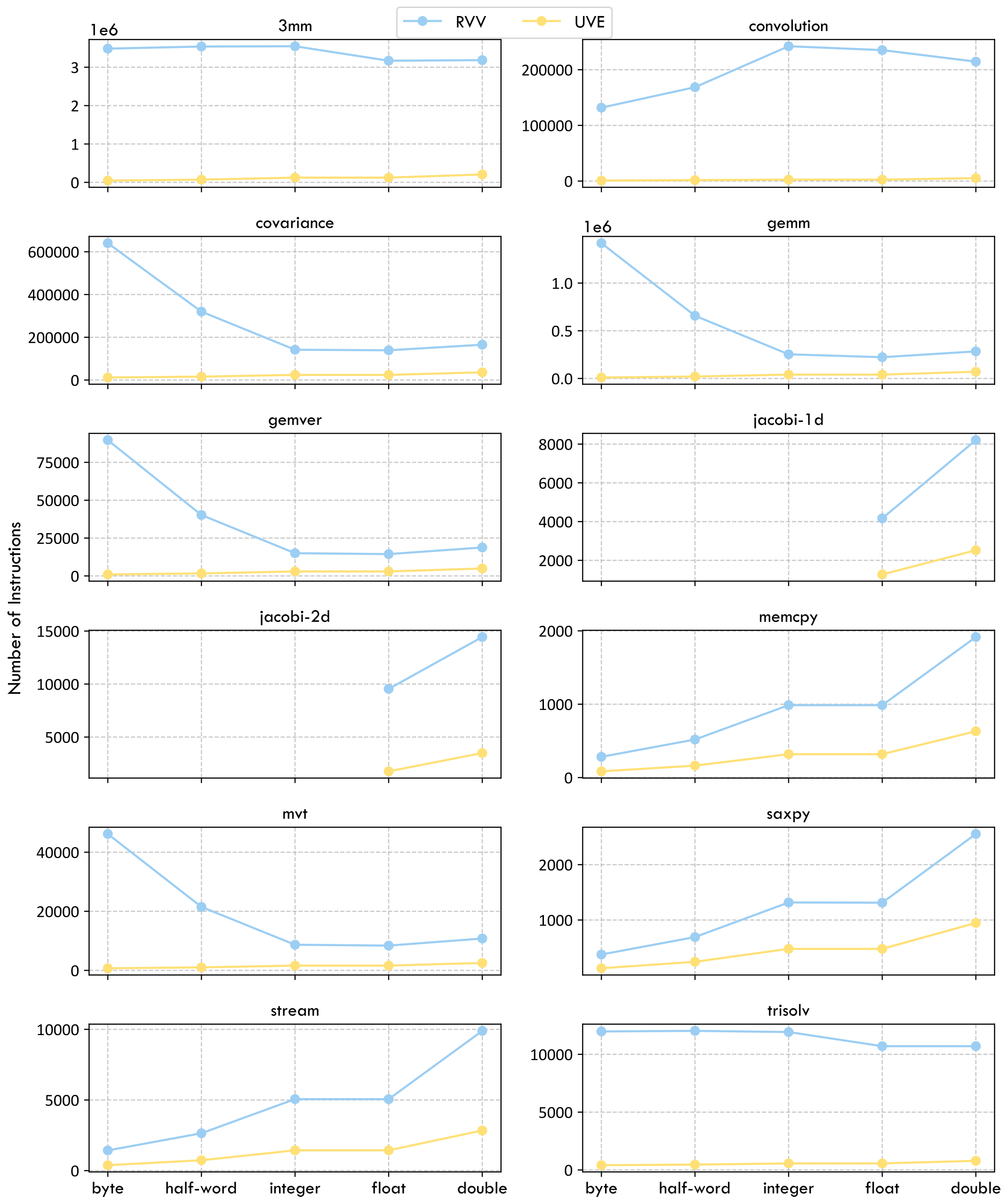


Figure 5.3: UVE and RVV (with loop unrolling) instruction count of executed benchmarks for each data type. Benchmarks that are only available for one data type were omitted.

In order to improve RVV performance, loop unrolling was enabled during compilation, to observe if this way it could match UVE performance. The comparison results with RVV without loop unrolling are presented in Appendix B. It was not possible to evaluate RVV against UVE on the *trisolv* benchmark, as the compiler failed to vectorise this kernel.

In some *byte*-type benchmarks, RVV performs very poorly, executing more instructions than scalar code, as seen in Figure 5.5a. It fails to improve in every instance of the *3mm* kernel, which may be due to autovectorisation limitations. Moreover, it extremely underperforms in the *SpMV-2* double-precision benchmark, which includes scatter-gather memory accesses. While it is evident that both RVV and UVE generally improve over scalar code, executing a very small fraction of instructions, it is observed that UVE outperforms RVV in all cases. It also seems to support more complex patterns, as it consistently succeeds when RVV fails to improve over scalar code. It is observed that RVV performance improves in some benchmarks, but it still underperforms UVE in all cases.

To more accurately compare the two extensions, the UVE instruction reduction relative to RVV was also calculated and is presented in Figure 5.6 for the double-precision benchmarks. For other data widths, the results are presented in Appendix B. With this metric, it is possible to verify that UVE executes fewer instructions than RVV in all cases. The average improvement of UVE over RVV is 78.42% without loop unrolling, and 74.84% with loop unrolling in double-precision applications. This shows that UVE is consistently more efficient than RVV in terms of instruction count, even when loop unrolling is enabled.

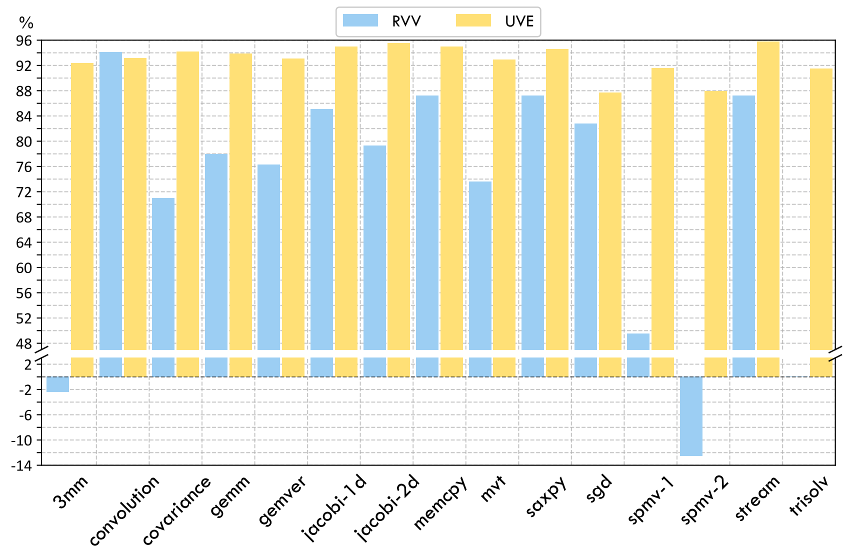
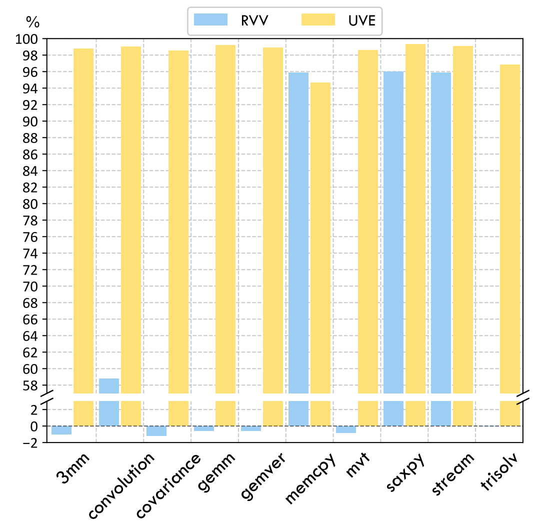
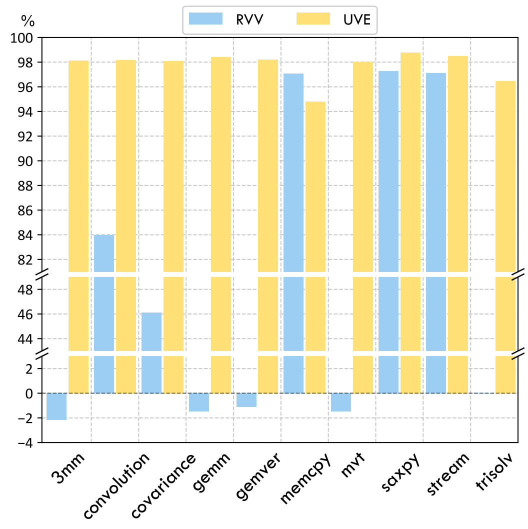


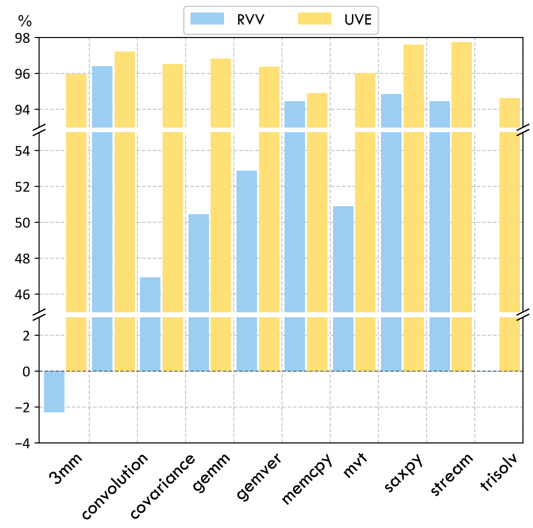
Figure 5.4: Percentage of reduction of instructions in double-precision floating-point RVV with loop unrolling and UVE benchmarks, relative to scalar code $\left(1 - \frac{Inst_{extension}}{Inst_{scalar}}\right)$. To exemplify, the UVE *SpMV-2* kernel achieved a 92% instruction reduction, while RVV executed 12% more instructions than scalar code.



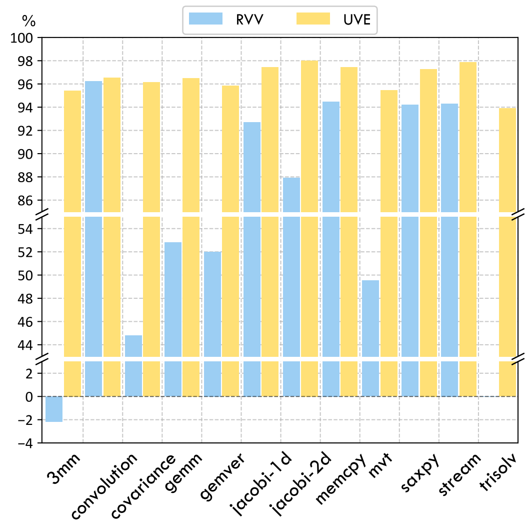
(a) Signed char (8 bits).



(b) Signed short integer (16 bits).



(c) Signed integer (32 bits).

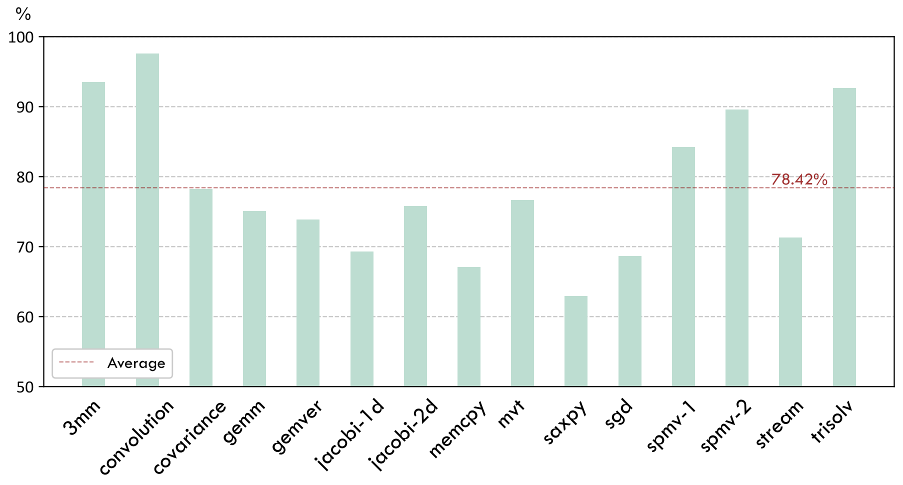


(d) Single-precision floating-point (32 bits).

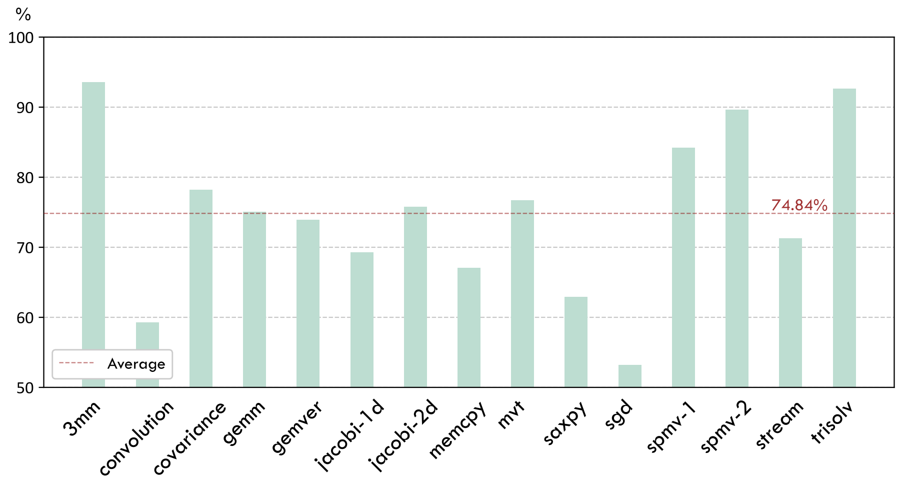
Figure 5.5: Percentage of reduction of instructions in different data type RVV with loop unrolling and UVE benchmarks, relative to scalar code $\left(1 - \frac{Inst_{extension}}{Inst_{scalar}}\right)$.

5.3 Summary

This chapter showed the two main results from this work: the complete functional simulation environment for UVE and a quantitative evaluation of the ISA, through the counting of retired instructions. This evaluation was performed with a wide set of benchmarks and used both scalar and RVV code as the baseline for comparison. The results show that UVE consistently outperforms RVV in terms



(a) UVE vs RVV without loop unrolling.



(b) UVE vs RVV with loop unrolling.

Figure 5.6: Double-precision floating-point UVE reduction of retired instructions, relative to RVV $\left(1 - \frac{Inst_{UVE}}{Inst_{RVV}}\right)$. (a) shows the percentual reduction of instructions relative to RVV without loop unrolling, while (b) shows the reduction of instructions relative to RVV with loop unrolling.

of executed instruction. The UVE extension achieves an average improvement of 78.42% relative to RVV without loop unrolling in double-precision benchmarks. Even when performing loop unrolling on the latter, UVE is still able to reduce the number of retired instructions by an average of 74.84%.

Chapter 6

Conclusion

By relying on data streaming, the Unlimited Vector Extension decouples memory accesses from computation, which allows memory access logic to be moved to a co-processor, the Streaming Engine (SE). On the other hand, it is also a vectorial ISA extension, which improves computation performance through SIMD operations. Several emerging applications, such as machine learning and image processing, are particularly well-suited to SIMD computation, and these extensions allow performance improvements without the need for dedicated external accelerators. Moreover, due to the rise of the Internet of Things (IOT), these applications are evermore common in portable and embedded devices, where power consumption is a critical factor. The predominant architecture in these devices is RISC, where the increasingly popular and open-source RISC-V stands out. This is the target architecture for the UVE extension, making it a relevant contribution to the field.

While the extension was validated through a proof-of-concept implementation on the *gem5* simulator, the development of the UVE specification is still ongoing. Because of the constraints imposed by this simulator, the first UVE implementation has since been considered deprecated, creating a need for a new simulation and validation framework that is also independent of the microarchitecture and pipeline modifications to a specific processor. This framework is presented in this work, complete with validation mechanisms, debugging support, and instruction counting capabilities. This new simulation environment is based on the *Spike* simulator, which is a functional RISC-V Instruction Set Simulator (ISS) that is widely used as the proof-of-concept target for every RISC-V extension ISA validation. The simulator was extended to support data streaming through a Streaming Unit (SU) that emulates the functional behaviour of the SE, the main component of the UVE supporting microarchitecture. In this work, over 150 instructions were implemented on *Spike*, which allowed 15 benchmarks from a wide range of applications to be executed and validated.

A revision of the specification was equally presented, with new instructions and features, such as scatter-gather descriptors, scalar streaming processing, and overall higher attainable memory pattern access complexity. Unnecessary instructions were removed from the extension and existing ones were improved to further reduce instruction overhead. This resulted in a more efficient and effective UVE ISA that is suitable for a wider range of applications.

The developed framework provided an independent functional validation of UVE. The used set of benchmarks proved the correctness of the specification and provided a quantitative evaluation of the ISA extension against RVV, which had not yet been made. The results showed that the UVE extension can decrease the number of retired instructions by up to 99% when compared to scalar code, and by an average of 75% when compared to the RVV extension in double-precision applications. This is a significant improvement, consistent with the obtained results in the original work, and it is a strong indicator of the potential of UVE.

6.1 Future Work

While the initial goals of this work were achieved, there are other aspects that need to be addressed in the future. Firstly, the proposed modifications to the extension have yet to be entirely implemented on *Spike*, as new header and modifier configuration instructions were left out due to time constraints. Adding these new features will allow the new specification to be fully validated before it is implemented on a real hardware platform. Moreover, data width conversion instructions need more thorough evaluation, as they have not been tested in the current framework and their behaviour is not yet fully formalised.

One aspect that can be evaluated and improved is the size of the predicate registers. Currently, they have the same size as the streaming registers, but an approach similar to SVE could be more beneficial. This extension maps a single bit of the predicate to each value in the vector register, effectively reducing the necessary size of the predicate register to the maximum *byte*-sized elements a vector register can hold (in the proposed UVE implementation it would be 64 bits, as vector registers are 512 bits wide). Moreover, explicit predication should be more thoroughly evaluated, as it was not explored extensively enough in the validated benchmarks.

To improve the portability of UVE, the LLVM compiler with UVE autovectorisation support must be developed. This work is currently under development and is expected to be released in the near future.

Finally, the proposed specification must be implemented in a new cycle-accurate simulator. This will provide an assessment of its real performance beyond instruction counts, such as power consumption and speed-up, which are more realistic evaluation metrics of the complete extension.

References

[1] João Mário Domingos. 'Unlimited Vector Extension with Data Streaming Support'. MA thesis. Instituto Superior Técnico, Oct. 2020.

[2] ARM. *Learn the architecture - Introducing Neon*. 2020. URL: https://developer.arm.com/documentation/102474.

[3] Chris Lomont. 'Introduction to Intel® Advanced Vector Extensions'. In: (2011).

[4] Hossein Amiri and Asadollah Shahbahrami. 'SIMD programming using Intel vector extensions'. In: *Journal of Parallel and Distributed Computing* 135 (Jan. 2020), pp. 83–100. ISSN: 07437315. DOI: 10.1016/j.jpdc.2019.09.012.

[5] RISC-V. *Working draft of the proposed RISC-V V vector extension*. 2023. URL: https://github.com/riscv/riscv-v-spec.

[6] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico and Paul Walker. 'The ARM Scalable Vector Extension'. In: *IEEE Micro* 37.2 (2017), pp. 26–39. DOI: 10.1109/MM.2017.35.

[7] Adrian Barredo, Juan M. Cebrian, Miquel Moreto, Marc Casas and Mateo Valero. 'Improving Predication Efficiency through Compaction/Restoration of SIMD Instructions'. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. San Diego, CA, USA: IEEE, Feb. 2020, pp. 717–728. ISBN: 978-1-72816-149-5. DOI: 10.1109/HPCA47549.2020.00064. URL: https://ieeexplore.ieee.org/document/9065430/.

[8] Angela Pohl, Mirko Greese, Biagio Cosenza and Ben Juurlink. 'A Performance Analysis of Vector Length Agnostic Code'. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. Dublin, Ireland: IEEE, July 2019, pp. 159–164. ISBN: 978-1-72814-484-9. DOI: 10.1109/HPCS48598.2019.9188238. URL: https://ieeexplore.ieee.org/document/9188238/.

[9] Joao Mário Domingos, Nuno Neves, Nuno Roma and Pedro Tomás. 'Unlimited Vector Extension with Data Streaming Support'. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, June 2021, pp. 209–222. ISBN: 978-1-66543-333-4. DOI: `10.1109/ISCA52012.2021.00025`. URL: `https://ieeexplore.ieee.org/document/9499750/`.

[10] RISC-V. *RISC-V ISA Simulator*. C/C++. Dec. 2021. URL: `https://github.com/riscv-software-src/riscv-isa-sim`.

[11] ARM. *Helium Architecture*. URL: `https://developer.arm.com/Architectures/Helium`.

[12] Alexander Heinecke, Thomas Auckenthaler and Carsten Trinitis. 'Exploiting State-of-the-Art x86 Architectures in Scientific Computing'. In: *2012 11th International Symposium on Parallel and Distributed Computing*. 2012, pp. 47–54. DOI: `10.1109/ISPDC.2012.15`.

[13] 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*. Padova, Italy: IEEE Comput. Soc, 2000, pp. 302–310. ISBN: 978-0-7695-0740-8. DOI: `10.1109/CAMP.2000.875989`. URL: `http://ieeexplore.ieee.org/document/875989/`.

[14] S. Oberman, G. Favor and F. Weber. 'AMD 3DNow! technology: architecture and implementations'. In: *IEEE Micro* 19.2 (1999), pp. 37–48. DOI: `10.1109/40.755466`.

[15] R.M. Ramanathan, Ron Curry, Srinivas Chennupaty, Robert L. Cross, Shihjong Kuo and Mark J. Buxton. 'Extending the World's Most Popular Processor Architecture'. In: *White Paper* (2006).

[16] Adrian Barredo, Juan M. Cebrian, Mateo Valero, Marc Casas and Miquel Moreto. 'Efficiency analysis of modern vector architectures: vector ALU sizes, core counts and clock frequencies'. In: *The Journal of Supercomputing* 76.3 (Mar. 2020), pp. 1960–1979. ISSN: 0920-8542, 1573-0484. DOI: `10.1007/s11227-019-02841-6`.

[17] Arm. 'Learn the architecture - Introducing SVE2 guide'. In: (2023). URL: `https://developer.arm.com/documentation/102340/0100/`.

[18] Peng Sun, Giacomo Gabrielli and Timothy M. Jones. 'Speculative Vectorisation with Selective Replay'. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 223–236. DOI: `10.1109/ISCA52012.2021.00026`.

[19] Arm. *SVE Optimization Guide*. 2021. URL: `https://developer.arm.com/documentation/102699/0100`.

[20] N. Clark, M. Kudlur, Hyunchul Park, S. Mahlke and K. Flautner. 'Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization'. In: *37th International Symposium on Microarchitecture (MICRO-37'04)*. Portland, OR, USA: IEEE, 2004, pp. 30–40. ISBN: 978-0-7695-2126-8. DOI: `10.1109/MICRO.2004.5`. URL: `http://ieeexplore.ieee.org/document/1550980/`.

[21] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam and Changkyu Kim. 'DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing'. In: *IEEE Micro* 32.5 (2012), pp. 38–51. DOI: `10.1109/MM.2012.51`.

[22] Snehasish Kumar, Nick Sumner, Vijayalakshmi Srinivasan, Steve Margerm and Arrvindh Shriraman. 'Needle: Leveraging Program Analysis to Analyze and Extract Accelerators from Whole Programs'. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 565–576. DOI: `10.1109/HPCA.2017.59`.

[23] Amirali Sharifian, Snehasish Kumar, Apala Guha and Arrvindh Shriraman. 'CHAINSAW: von-neumann accelerators to leverage fused instruction chains'. In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016.

[24] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson and Michael Bedford Taylor. 'Conservation cores: reducing the energy of mature computations'. In: *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2010, pp. 20–218. ISBN: 9781605588391. DOI: `10.1145/1736020.1736044`. URL: `https://doi.org/10.1145/1736020.1736044`.

[25] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher and Joel Emer. 'Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration'. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 137–151. ISBN: 9781450362405. DOI: `10.1145/3297858.3304025`. URL: `https://doi.org/10.1145/3297858.3304025`.

[26] Neal Clayton Crago and Sanjay Jeram Patel. 'OUTRIDER: efficient memory latency tolerance with decoupled strands'. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 117–128. ISBN: 9781450304726. DOI: `10.1145/2000064.2000079`. URL: `https://doi.org/10.1145/2000064.2000079`.

[27] Tae Jun Ham, Juan L. Aragón and Margaret Martonosi. 'DeSC: decoupled supply-compute communication management for heterogeneous architectures'. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 191–203. ISBN: 9781450340342. DOI: 10.1145/2830772.2830800. URL: https://doi.org/10.1145/2830772.2830800.

[28] K.J. Nesbit and J.E. Smith. 'Data Cache Prefetching Using a Global History Buffer'. In: *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. 2004, pp. 96–96. DOI: 10.1109/HPCA.2004.10030.

[29] S. Somogyi, T.F. Wenisch, A. Ailamaki, B. Falsafi and A. Moshovos. 'Spatial Memory Streaming'. In: *33rd International Symposium on Computer Architecture (ISCA'06)*. 2006, pp. 252–263. DOI: 10.1109/ISCA.2006.38.

[30] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi and Andreas Moshovos. 'Practical off-chip meta-data for temporal memory streaming'. In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 2009, pp. 79–90. DOI: 10.1109/HPCA.2009.4798239.

[31] Yasuo Ishii, Mary Inaba and Kei Hiraki. 'Access map pattern matching for data cache prefetch'. In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09. Yorktown Heights, NY, USA: Association for Computing Machinery, 2009, pp. 499–500. ISBN: 9781605584980. DOI: 10.1145/1542275.1542349. URL: https://doi.org/10.1145/1542275.1542349.

[32] Yao Guo, Pritish Narayanan, Mahmoud Abdullah Bennaser, Saurabh Chheda and Csaba Andras Moritz. 'Energy-Efficient Hardware Data Prefetching'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19.2 (Feb. 2011), pp. 250–263. ISSN: 1063-8210, 1557-9999. DOI: 10.1109/TVLSI.2009.2032916.

[33] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow and Rajeev Balasubramonian. 'Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers'. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 626–637. DOI: 10.1109/HPCA.2014.6835971.

[34] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley and Zeshan Chishti. 'Efficiently prefetching complex address patterns'. In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 141–152. DOI: 10.1145/2830772.2830793.

[35] Leeor Peled, Shie Mannor, Uri Weiser and Yoav Etsion. 'Semantic locality and context-based prefetching using reinforcement learning'. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: Association for Computing Machinery, 2015,

pp. 285–297. ISBN: 9781450334020. DOI: `10.1145/2749469.2749473`. URL: `https://doi.org/10.1145/2749469.2749473`.

[36] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish and Srinivas Devadas. 'IMP: Indirect memory prefetcher'. In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 178–190. DOI: `10.1145/2830772.2830807`.

[37] Pierre Michaud. 'Best-offset hardware prefetching'. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 469–480. DOI: `10.1109/HPCA.2016.7446087`.

[38] Mohammad Bakhshalipour, Pejman Lotfi-Kamran and Hamid Sarbazi-Azad. 'An Efficient Temporal Data Prefetcher for L1 Caches'. In: *IEEE Computer Architecture Letters* 16.2 (2017), pp. 99–102. DOI: `10.1109/LCA.2017.2654347`.

[39] Sam Ainsworth and Timothy M. Jones. 'Software prefetching for indirect memory accesses'. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 305–317. DOI: `10.1109/CGO.2017.7863749`.

[40] Sushant Kondguli and Michael Huang. 'Division of Labor: A More Effective Approach to Prefetching'. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 83–95. DOI: `10.1109/ISCA.2018.00018`.

[41] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran and Hamid Sarbazi-Azad. 'Bingo Spatial Data Prefetcher'. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 399–411. DOI: `10.1109/HPCA.2019.00053`.

[42] Ioan Hadade, Timothy M. Jones, Feng Wang and Luca di Mare. 'Software Prefetching for Unstructured Mesh Applications'. In: *ACM Trans. Parallel Comput.* 7.1 (Mar. 2020). ISSN: 2329-4949. DOI: `10.1145/3380932`. URL: `https://doi.org/10.1145/3380932`.

[43] 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'. In: *IEEE Micro* 21.2 (2001), pp. 35–46. DOI: `10.1109/40.918001`.

[44] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette and Ali Saidi. 'The Reconfigurable Streaming Vector Processor (RSVPTM)'. In: (2003).

[45] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim and Kenneth A. Ross. 'Q100: the architecture and design of a database processing unit'. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 255–268. ISBN:

9781450323055. DOI: 10.1145/2541940.2541961. URL: https://doi.org/10.1145/2541940.2541961.

[46] Nathan Clark, Amir Hormati and Scott Mahlke. 'VEAL: Virtualized Execution Accelerator for Loops'. In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA '08. USA: IEEE Computer Society, 2008, pp. 389–400. ISBN: 9780769531748. DOI: 10.1109/ISCA.2008.33. URL: https://doi.org/10.1109/ISCA.2008.33.

[47] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani and Karthikeyan Sankaralingam. 'Stream-Dataflow Acceleration'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 416–429. ISBN: 9781450348928. DOI: 10.1145/3079856.3080255. URL: https://doi.org/10.1145/3079856.3080255.

[48] Gabriel Weisz and James C. Hoe. 'CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing'. In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–8. DOI: 10.1109/FPL.2015.7294017.

[49] Zhengrong Wang and Tony Nowatzki. 'Stream-Based Memory Access Specialization for General Purpose Processors'. In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 736–749. ISBN: 9781450366694. DOI: 10.1145/3307650.3322229. URL: https://doi.org/10.1145/3307650.3322229.

[50] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur and Tony Nowatzki. 'Stream Floating: Enabling Proactive and Decentralized Cache Optimizations'. In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021, pp. 640–653. DOI: 10.1109/HPCA51647.2021.00060.

[51] Zhengrong Wang, Jian Weng, Sihao Liu and Tony Nowatzki. 'Near-Stream Computing: General and Transparent Near-Cache Acceleration'. In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022, pp. 331–345. DOI: 10.1109/HPCA53966.2022.00032.

[52] Sérgio Paiágua, Frederico Pratas, Pedro Tomás, Nuno Roma and Ricardo Chaves. 'HotStream: Efficient Data Streaming of Complex Patterns to Multiple Accelerating Kernels'. In: *2013 25th International Symposium on Computer Architecture and High Performance Computing*. 2013, pp. 17–24. DOI: 10.1109/SBAC-PAD.2013.17.

[53] Nishil Talati, Kyle May, Armand Behroozi et al. 'Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design'. In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021, pp. 654–667. DOI: 10.1109/HPCA51647.2021.00061.

[54] Fabian Schuiki, Florian Zaruba, Torsten Hoefler and Luca Benini. 'Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores'. In: *IEEE Transactions on Computers* 70.2 (2021), pp. 212–227. DOI: `10.1109/TC.2020.2987314`.

[55] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler and Luca Benini. 'Indirection Stream Semantic Register Architecture for Efficient Sparse-Dense Linear Algebra'. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 1787–1792. DOI: `10.23919/DATE51398.2021.9474230`.

[56] Nuno Neves, Pedro Tomás and Nuno Roma. 'Efficient data-stream management for shared-memory many-core systems'. In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–8. DOI: `10.1109/FPL.2015.7293960`.

[57] Nuno Neves, Pedro Tomás and Nuno Roma. 'Adaptive In-Cache Streaming for Efficient Data Management'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.7 (July 2017), pp. 2130–143. ISSN: 1063-8210, 1557-9999. DOI: `10.1109/TVLSI.2017.2671405`.

[58] Siying Feng, Zhang Zhengya, Mahlke Scott and Trevor N Mudge. 'Acceleration Techniques of Sparse Linear Algebra on Emerging Architectures'. AAI30353402. PhD thesis. USA, 2022. ISBN: 9798368476452.

[59] Guoqing Xiao, Chuanghui Yin, Tao Zhou, Xueqi Li, Yuedan Chen and Kenli Li. 'A Survey of Accelerating Parallel Sparse Linear Algebra'. In: *ACM Comput. Surv.* 56.1 (Aug. 2023). ISSN: 0360-0300. DOI: `10.1145/3604606`. URL: `https://doi.org/10.1145/3604606`.

[60] Vidushi Dadu, Jian Weng, Sihao Liu and Tony Nowatzki. 'Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms'. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 924–939. ISBN: 9781450369381. DOI: `10.1145/3352460.3358276`. URL: `https://doi.org/10.1145/3352460.3358276`.

[61] Benjamin W. Mezger, Douglas A. Santos, Luigi Dilillo, Cesar A. Zeferino and Douglas R. Melo. 'A Survey of the RISC-V Architecture Software Support'. In: *IEEE Access* 10 (2022), pp. 51394–51411. DOI: `10.1109/ACCESS.2022.3174125`.

[62] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA*. 2019. URL: `https://github.com/riscv/riscv-isa-manual/`.

[63] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[64] Richard Stallman. *Using GNU CC*. Boston, MA: Free Software Foundation, 1995. ISBN: 978-1-882114-66-5.

[65] C. Lattner and V. Adve. 'LLVM: A compilation framework for lifelong program analysis & transformation'. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. San Jose, CA, USA: IEEE, 2004, pp. 75–86. ISBN: 978-0-7695-2102-2. DOI: `10.1109/CGO.2004.1281665`. URL: `http://ieeexplore.ieee.org/document/1281665/`.

[66] Nuno Neves, João Mário Domingos, Nuno Roma, Pedro Tomás and Gabriel Falcao. 'Compiling for Vector Extensions With Stream-Based Specialization'. In: *IEEE Micro* 42.5 (Sept. 2022), pp. 49–58. ISSN: 0272-1732, 1937-4143. DOI: `10.1109/MM.2022.3173405`.

[67] Neil Adit and Adrian Sampson. 'Performance Left on the Table: An Evaluation of Compiler Autovectorization for RISC-V'. In: *IEEE Micro* 42.5 (Sept. 2022), pp. 41–48. ISSN: 0272-1732, 1937-4143. DOI: `10.1109/MM.2022.3184867`.

[68] Louis-Noël Pouchet. *PolyBench/C*. 2012. URL: `https://web.cs.ucla.edu/~pouchet/software/polybench/`.

[69] Alec Roelke and Mircea R Stan. *RISC5: Implementing the RISC-V ISA in gem5*. 2017.

[70] Luís Henriques. 'Automatic Streaming for RISC-V via Source-to-Source Compilation'. MA thesis. Porto: Universidade do Porto, July 2022. URL: `https://hdl.handle.net/10216/142750`.

[71] RISC-V. *RISC-V Opcodes*. URL: `https://github.com/riscv/riscv-opcodes`.

[72] *Olympia*. C++. Feb. 2024. URL: `https://github.com/riscv-software-src/riscv-perf-model`.

[73] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr and Andreas Hoffmann. 'A universal technique for fast and flexible instruction-set architecture simulation'. In: *Proceedings of the 39th Annual Design Automation Conference*. DAC '02. New Orleans, Louisiana, USA: Association for Computing Machinery, 2002, pp. 22–27. ISBN: 1581134614. DOI: `10.1145/513918.513927`. URL: `https://doi.org/10.1145/513918.513927`.

[74] Jack Dongarra, Michael A. Heroux and Piotr Luszczek. 'HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems'. In: *National Science Review* 3.1 (Mar. 2016), pp. 30–35. ISSN: 2053-714X, 2095-5138. DOI: `10.1093/nsr/nwv084`.

[75] Nathan Bell and Michael Garland. 'Efficient Sparse Matrix-Vector Multiplication on CUDA'. In: (2008).

[76] F. Vázquez, G. Ortega, J.J. Fernández and E.M. Garzón. 'Improving the Performance of the Sparse Matrix Vector Product with GPUs'. In: *2010 10th IEEE International Conference on Computer and Information Technology*. 2010, pp. 1146–1151. DOI: `10.1109/CIT.2010.208`.

[77] Dezsö Sima. 'The Design Space of Register Renaming Techniques'. In: *IEEE Micro* 20.5 (Sept. 2000), pp. 70–83. ISSN: 0272-1732. DOI: `10.1109/40.877952`. URL: `https://doi.org/10.1109/40.877952`.

Appendices

Appendix A

Unlimited Vector Extension Supporting Microarchitecture

The Streaming Engine (SE) is responsible for most streaming operations and is the main hardware component of the proposed extension. Aside from that, some modifications were made to the Central Processing Unit (CPU) processing pipeline, in order to support streaming. An Out-of-Order (OoO) processing pipeline was chosen as the target of the proof-of-concept implementation of Unlimited Vector Extension (UVE), as it is more common in High-Performance Computing (HPC), thus providing a more valid assessment of the extension in its main target applications, albeit more complex to implement. The proposed microarchitecture is illustrated in Figure A.1, where the modifications listed below are also highlighted.

- **Decoders, register file and execution units:** Support for the decoding of added instructions, vector registers, as well as necessary logic, arithmetic, and branch functional units (similar to RISC-V Vector Extension (RVV) and Scalable Vector Extension (SVE)).
- **Rename stage:** Support for vector register and stream renaming, allowing for speculative configuration of streams. Register renaming is an important mechanism that is already present in most processors. It allows for the elimination of certain data dependencies, by separating architectural and physical registers [77]. This principle is applied to stream configurations, thus making it possible to speculatively configure new streams while others that share the same logical name are still executing.
- **Commit stage:** Support for the commit and squash of streams, through the signaling of all misspeculation and commit events related to the processing of streams to the SE. As a result of an eventual misspeculation, stream configurations or iterations may be incorrectly performed. In the first case, all

the structures involved are released and a new configuration may be accepted by the SE. However, two actions are necessary in the second case. On the one hand, the pipeline is responsible for reverting the physical register to the previously committed value. On the other hand, the SE must be notified of the squash so that it can revert the speculated pointers on the load/store circular buffers to the current commit point. This means that buffered data is never impacted by misspeculations on loads, as well as generated addresses on stores. As streaming data patterns are deterministic, the fact that they are consumed in the wrong order does not change data validity inside said buffers, and it can be re-used with no need for new loads.

In order to fully understand the role of the SE, it is important to first understand how a stream is supposed to behave from the moment it is configured until it is terminated. Furthermore, its implementation on an OoO core is not straightforward. The most important aspects of stream operation in the proposed model are hereby detailed.

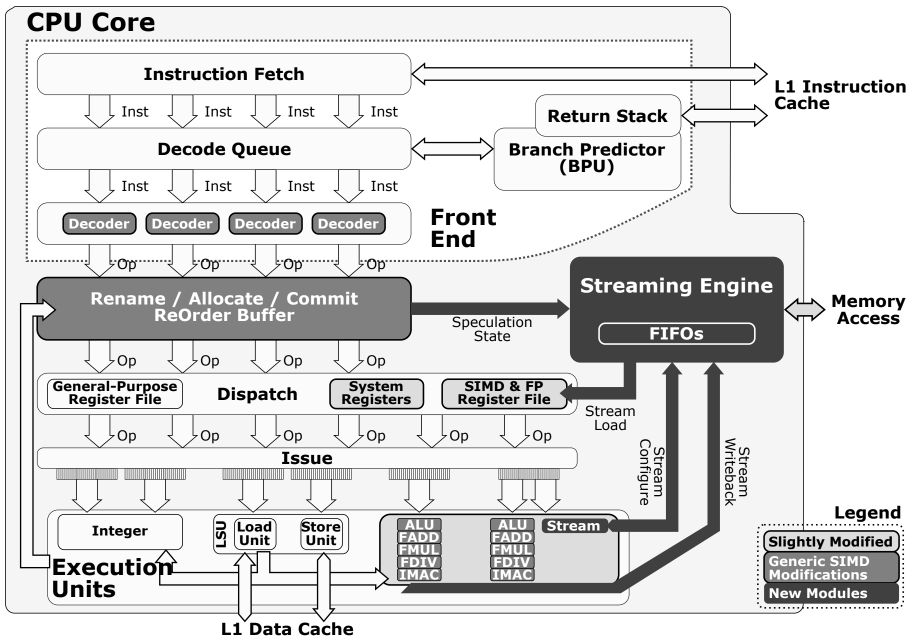


Figure A.1: UVE supporting microarchitecture overview, highlighting modifications introduced in a traditional OoO processing pipeline [9].

Stream Configuration

As mentioned in Section 2.3.4 and Section 2.4.2, multiple instructions are needed to configure any pattern that is not trivial, and these instructions must be executed in order so that the descriptors are correctly chained. In an OoO architecture, this means that some mechanism must be put in place to ensure in-order execution. However, speculative configuration of streams is desirable as it improves performance. As previously mentioned, stream renaming was implemented to support this. At this stage, each stream configuration instruction is inserted in the *Stream Configuration Reorder Buffer (SCROB)*, a new structure embedded in the SE. It processes each configuration instruction in order as soon as the corresponding operands are available, similarly to a re-order buffer. Then, after its configuration, the stream is processed by the SE. This encompasses the pre-loading of data in the case of a load stream, or the computation of store addresses for store streams, that will then await for the committing of store data.

Lastly, each stream configuration also results in two stream state iterators: *speculative* or *commit*. These are dynamically iterated once a stream manipulation instruction reaches the rename and commit stages, respectively, to allow for speculative execution.

Stream Renaming

When a certain stream is beginning the respective configuration, its corresponding identification register may still be occupied by another running stream, due to misspeculation or even pipeline latency. To mitigate possible pipeline blockings, a Stream Allocation Table (SAT) is included. This structure, very similar to a Register Alias Table (RAT), is responsible for the mapping between each physical and logical stream identification register. Moreover, the SAT is also designated for keeping information about which registers are currently associated with active streams, which is necessary for the distinction between stream operations (involving reads/writes from a stream) and regular register operations, which are not handled by the SE.

Stream Iteration

Iterating a stream is the process of reading from input streams (read streams) and writing to output streams (store streams). This occurs during the rename stage, where the *speculative* iterator is incremented. In the case of a load stream, when an instruction that consumes values from a stream enters the rename stage, they are immediately read from a register that holds the pre-loaded data, all while new data is already being pre-loaded to a different physical register. This is thanks to vector register renaming, which was extended beyond the standard of only performing renaming for destination registers to support the renaming of source registers as well. The newly renamed physical register is passed to the load queue of the SE, which then loads the next values from memory.

Stream Termination

The end of a stream is reached at the commit stage, either by an explicit termination instruction or because an instruction with an End-of-Stream (EOS) signal was committed, due to the reaching of the end of the streaming pattern. When this happens, every structure associated with the stream is released.

Memory hierarchy

As mentioned in Section 2.4.2, UVE allows for the configuration of data loading from any cache level. Besides, to simplify the implementation, as well as minimising the impact on caches, the proposed model joins stream requests and typical memory loads/stores before the L1 cache is accessed. This is possible because, most of the time, conventional memory accesses are mutually exclusive from stream accesses, as streaming loops generally do not require scalar memory operations. The memory hierarchy is illustrated in Figure A.2.

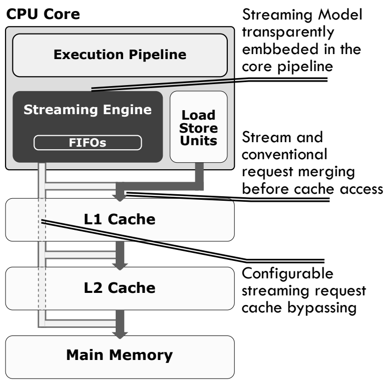

CPU Core
Streaming Model transparently embbeded in the core pipeline
Execution Pipeline
Streaming Engine
FIFOs
Load Store Units
Stream and conventional request merging before cache access
L1 Cache
L2 Cache
Configurable streaming request cache bypassing
Main Memory

Figure A.2: System overview, featuring the SE embedded in an OoO core and respective connections to the memory hierarchy [9].

Memory coherence

Eventual stream load/store dependencies are handled through typical mechanisms found in modern architectures: request delays, replays, and squashes. This ensures that data resulting from conventional operations can be read by input streams straight away. Likewise, data written by store streams is readily available for conventional load instructions to use. Cache coherence is guaranteed by a MOESI protocol. Coherence mechanisms at the level of the stream First-In, First-Out (FIFO) buffers are not defined, under the assumption that data that has been preloaded has already been consumed by the core, as it would with register pre-fetching or loop unrolling. However, this raises the issue of how load and store streams from the same memory access will behave. This problem is discussed in Section 2.4.5.

Appendix B

Instruction Counting Results

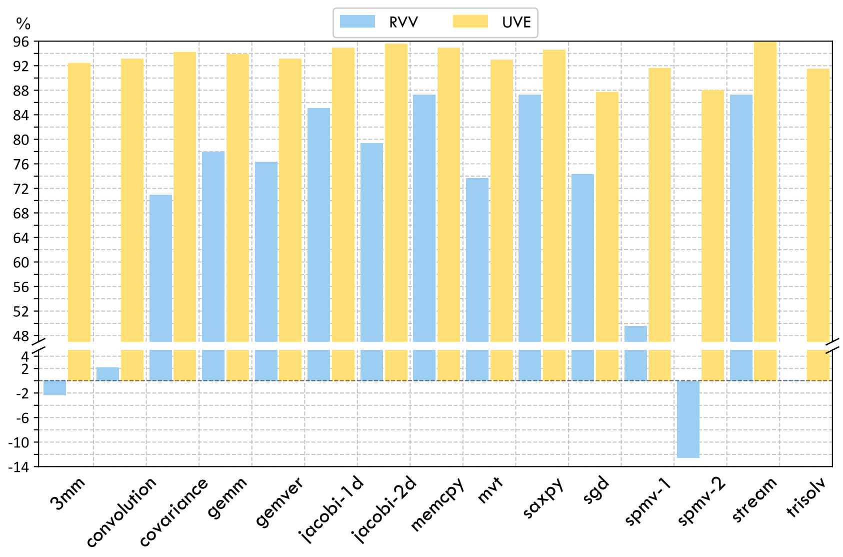
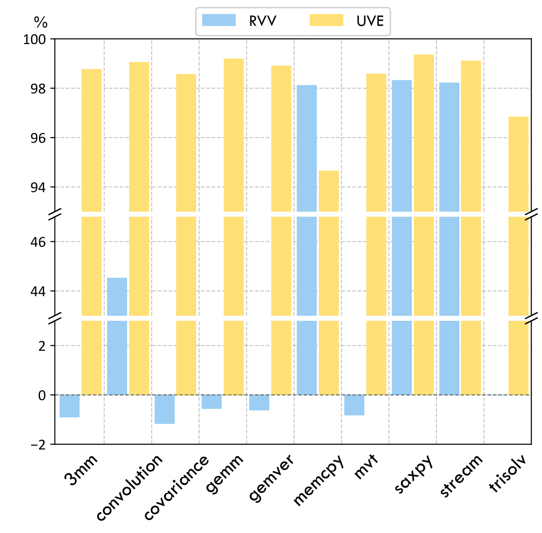
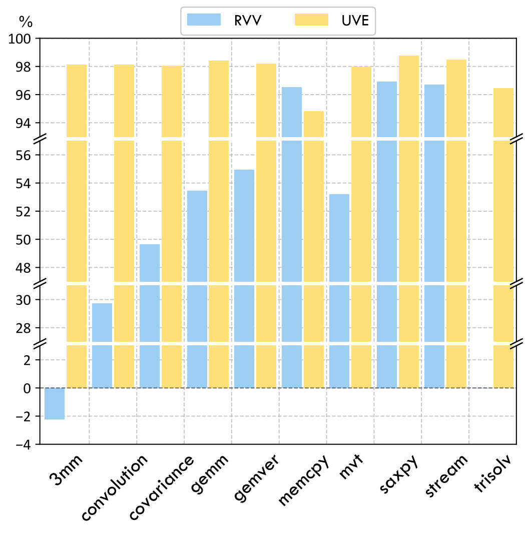


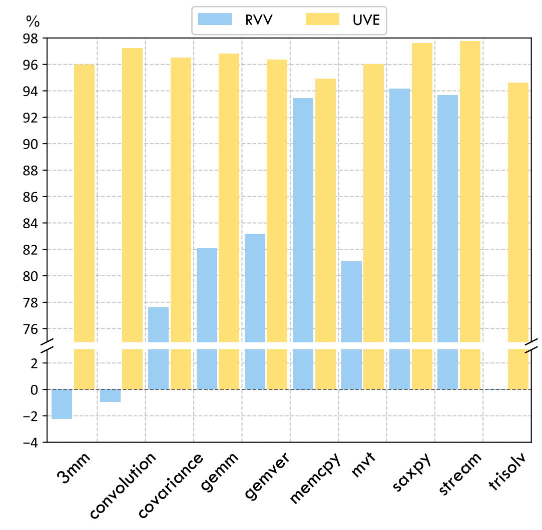
Figure B.1: Percentage of reduction of instructions in double-precision floating-point RVV without loop unrolling and UVE benchmarks, relative to scalar code $\left(1 - \frac{Inst_{extension}}{Inst_{scalar}}\right)$.



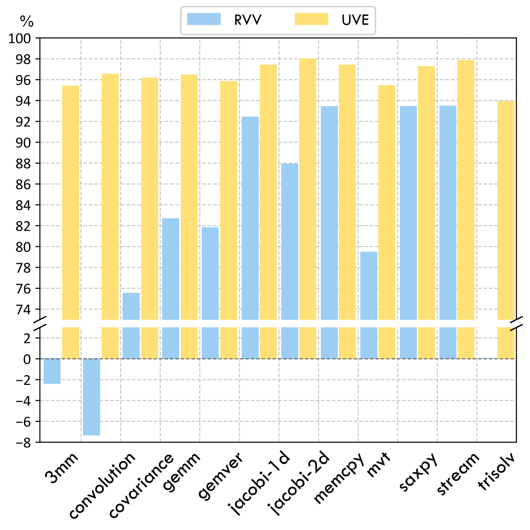
(a) Signed char (8 bits).



(b) Signed short integer (16 bits).

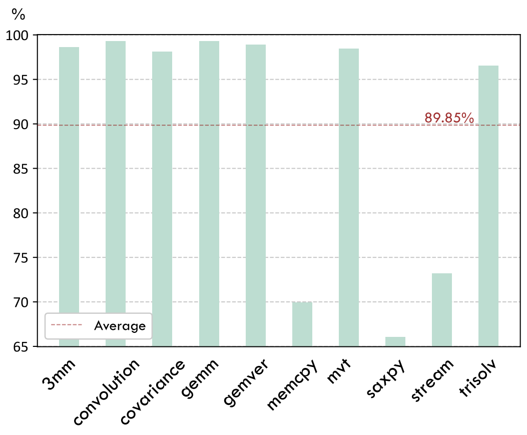


(c) Signed integer (32 bits).

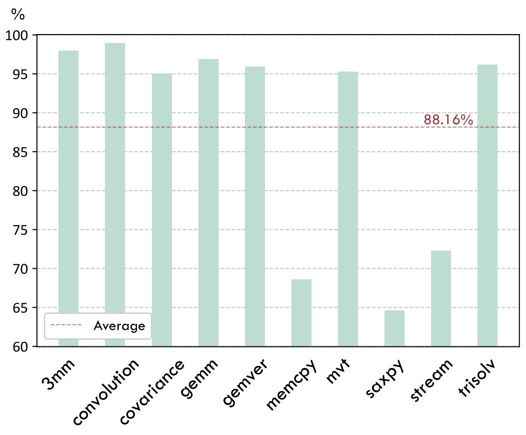


(d) Single-precision floating-point (32 bits).

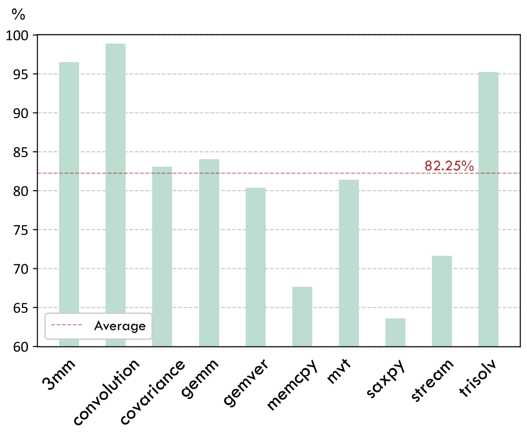
Figure B.2: Percentage of reduction of instructions in different data type RVV without loop unrolling and UVE benchmarks, relative to scalar code $\left(1 - \frac{Inst_{extension}}{Inst_{scalar}}\right)$.



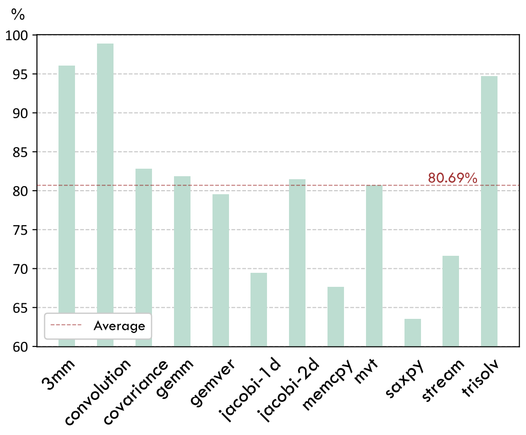
(a) Signed char (8 bits).



(b) Signed short integer (16 bits).

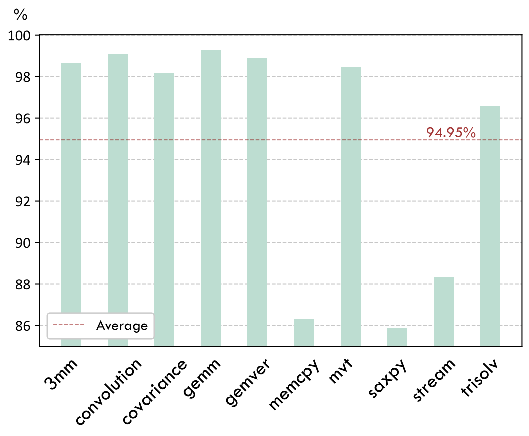


(c) Signed integer (32 bits).

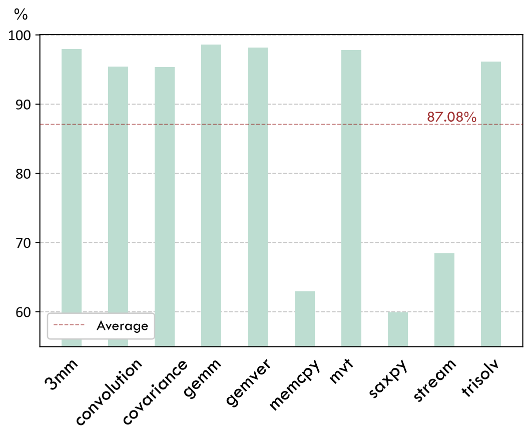


(d) Single-precision floating-point (32 bits).

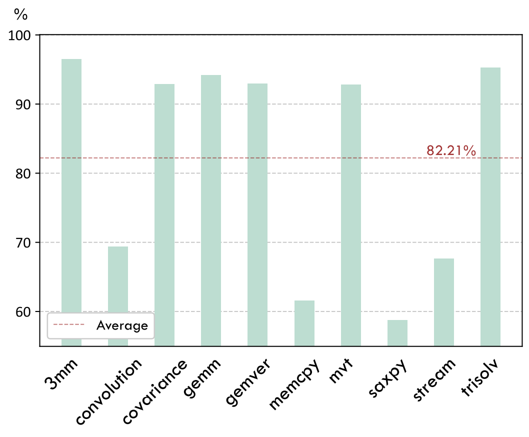
Figure B.3: UVE reduction of retired instructions, relative to RVV without loop unrolling $(1 - \frac{Inst_{UVE}}{Inst_{RVV}})$.



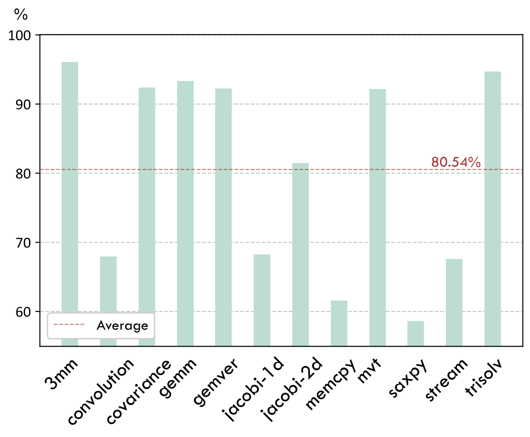
(a) Signed char (8 bits).



(b) Signed short integer (16 bits).



(c) Signed integer (32 bits).



(d) Single-precision floating-point (32 bits).

Figure B.4: UVE reduction of retired instructions, relative to RVV with loop unrolling $\left(1 - \frac{Inst_{UVE}}{Inst_{RVV}}\right)$ for different data types.

Appendix C

UVE Instruction Listing

31 28	27 25	24 20	19 15	14 12	11 7	6 0	
funct4	ps3	vs2	vs1	funct3	vd/rd	opcode	UA-type

Arithmetic and Logic Instructions

0000	ps3	vs2	vs1	000	vd	0101011	SO.A.ADD.US
0000	ps3	vs2	vs1	001	vd	0101011	SO.A.ADD.FP
0000	ps3	vs2	vs1	010	vd	0101011	SO.A.ADD.SG
0000	ps3	vs2	vs1	100	vd	0101011	SO.A.SUB.US
0000	ps3	vs2	vs1	101	vd	0101011	SO.A.SUB.FP
0000	ps3	vs2	vs1	110	vd	0101011	SO.A.SUB.SG
0001	ps3	vs2	vs1	000	vd	0101011	SO.A.MUL.US
0001	ps3	vs2	vs1	001	vd	0101011	SO.A.MUL.FP
0001	ps3	vs2	vs1	010	vd	0101011	SO.A.MUL.SG
0001	ps3	vs2	vs1	100	vd	0101011	SO.A.DIV.US
0001	ps3	vs2	vs1	101	vd	0101011	SO.A.DIV.FP
0001	ps3	vs2	vs1	110	vd	0101011	SO.A.DIV.SG
0010	ps3	00000	vs1	000	vd	0101011	SO.A.ADDE.US
0010	ps3	00000	vs1	001	vd	0101011	SO.A.ADDE.FP
0010	ps3	00000	vs1	010	vd	0101011	SO.A.ADDE.SG
0010	ps3	00001	vs1	000	vd	0101011	SO.A.ADDE.ACC.US
0010	ps3	00001	vs1	001	vd	0101011	SO.A.ADDE.ACC.FP
0010	ps3	00001	vs1	010	vd	0101011	SO.A.ADDE.ACC.SG
0010	ps3	00000	vs1	100	rd	0101011	SO.A.ADDS.US
0010	ps3	00000	vs1	101	rd	0101011	SO.A.ADDS.FP
0010	ps3	00000	vs1	110	rd	0101011	SO.A.ADDS.SG
0010	ps3	00001	vs1	100	rd	0101011	SO.A.ADDS.ACC.US
0010	ps3	00001	vs1	101	rd	0101011	SO.A.ADDS.ACC.FP
0010	ps3	00001	vs1	110	rd	0101011	SO.A.ADDS.ACC.SG
0011	ps3	00000	vs1	001	vd	0101011	SO.A.ABS.FP
0011	ps3	00000	vs1	000	vd	0101011	SO.A.ABS.SG
0011	ps3	vs2	vs1	100	vd	0101011	SO.A.MAC.US
0011	ps3	vs2	vs1	101	vd	0101011	SO.A.MAC.FP
0011	ps3	vs2	vs1	110	vd	0101011	SO.A.MAC.SG

31 29	28	27 25	24 22	21	20	19 15	14 12	11 7	6 0	
funct4		ps3	vs2			vs1	funct3	vd/rd	opcode	UA-type
funct3	imm[12|10:5]			-	n	vs1	funct3	imm[4:1|11]	opcode	UB-type

Arithmetic and Logic Instructions (Continuation)

1100	ps3	vs2	vs1	000	vd	0101011	SO.A.NAND
1100	ps3	vs2	vs1	001	vd	0101011	SO.A.AND
1100	ps3	vs2	vs1	010	vd	0101011	SO.A.NOR
1100	ps3	vs2	vs1	011	vd	0101011	SO.A.OR
1100	ps3	00000	vs1	100	vd	0101011	SO.A.NOT
1100	ps3	vs2	vs1	101	vd	0101011	SO.A.XOR
1101	ps3	vs2	vs1	000	vd	0101011	SO.A.SLL
1101	ps3	rs2	vs1	001	vd	0101011	SO.A.SLLS
1101	ps3	vs2	vs1	010	vd	0101011	SO.A.SRL
1101	ps3	rs2	vs1	011	vd	0101011	SO.A.SRLS
1101	ps3	vs2	vs1	100	vd	0101011	SO.A.SRA
1101	ps3	rs2	vs1	101	vd	0101011	SO.A.SRAS
0100	ps3	vs2	vs1	000	vd	0101011	SO.A.MIN.US
0100	ps3	vs2	vs1	001	vd	0101011	SO.A.MIN.FP
0100	ps3	vs2	vs1	010	vd	0101011	SO.A.MIN.SG
0100	ps3	vs2	vs1	100	vd	0101011	SO.A.MAX.US
0100	ps3	vs2	vs1	101	vd	0101011	SO.A.MAX.FP
0100	ps3	vs2	vs1	110	vd	0101011	SO.A.MAX.SG
0101	ps3	00000	vs1	000	vd	0101011	SO.A.MINE.US
0101	ps3	00000	vs1	001	vd	0101011	SO.A.MINE.FP
0101	ps3	00000	vs1	010	vd	0101011	SO.A.MINE.SG
0101	ps3	00000	vs1	100	vd	0101011	SO.A.MAXE.US
0101	ps3	00000	vs1	101	vd	0101011	SO.A.MAXE.FP
0101	ps3	00000	vs1	110	vd	0101011	SO.A.MAXE.SG
0110	ps3	00000	vs1	000	vd	0101011	SO.A.INC.US
0110	ps3	00000	vs1	001	vd	0101011	SO.A.INC.FP
0110	ps3	00000	vs1	010	vd	0101011	SO.A.INC.SG
0110	ps3	00000	vs1	100	vd	0101011	SO.A.DEC.US
0110	ps3	00000	vs1	101	vd	0101011	SO.A.DEC.FP
0110	ps3	00000	vs1	110	vd	0101011	SO.A.DEC.SG

Loop Control Branching Instructions

111	imm[12|10:5]	0	1	vs1	000	imm[4:1|11]	0101011	SO.B.NDC.1
111	imm[12|10:5]	0	1	vs1	001	imm[4:1|11]	0101011	SO.B.NDC.2
111	imm[12|10:5]	0	1	vs1	010	imm[4:1|11]	0101011	SO.B.NDC.3
111	imm[12|10:5]	0	1	vs1	011	imm[4:1|11]	0101011	SO.B.NDC.4
111	imm[12|10:5]	0	1	vs1	100	imm[4:1|11]	0101011	SO.B.NDC.5
111	imm[12|10:5]	0	1	vs1	101	imm[4:1|11]	0101011	SO.B.NDC.6
111	imm[12|10:5]	0	1	vs1	110	imm[4:1|11]	0101011	SO.B.NDC.7
111	imm[12|10:5]	0	0	vs1	000	imm[4:1|11]	0101011	SO.B.DC.1
111	imm[12|10:5]	0	0	vs1	001	imm[4:1|11]	0101011	SO.B.DC.2
111	imm[12|10:5]	0	0	vs1	010	imm[4:1|11]	0101011	SO.B.DC.3
111	imm[12|10:5]	0	0	vs1	011	imm[4:1|11]	0101011	SO.B.DC.4
111	imm[12|10:5]	0	0	vs1	100	imm[4:1|11]	0101011	SO.B.DC.5
111	imm[12|10:5]	0	0	vs1	101	imm[4:1|11]	0101011	SO.B.DC.6
111	imm[12|10:5]	0	0	vs1	110	imm[4:1|11]	0101011	SO.B.DC.7
111	imm[12|10:5]	0	1	vs1	111	imm[4:1|11]	0101011	SO.B.NC
111	imm[12|10:5]	0	0	vs1	111	imm[4:1|11]	0101011	SO.B.C

31 28	27 25	24	23 22	21 20	19	18 15	14 12	11	10 7	6 0	
funct4	ps3	z	dw	sw	-	ps1	funct4		pd	opcode	UP1-type
funct4	ps3	z	-		vs1		funct4		pd	opcode	UP2-type
funct4	ps3	vs2			vs1		funct3	z	pd	opcode	UP3-type
funct5	-				rs1		funct3	rd		opcode	UC-type

Lane Control Predication Instructions

31 28	27 25	24	23 22	21 20	19	18 15	14 12	11	10 7	6 0	
1000	ps3	0	000000000				0000		pd	0101011	SO.P.ZERO
1000	ps3	1	000000000				0000		pd	0101011	SO.P.ZERO.Z
1000	ps3	0	000000000				0001		pd	0101011	SO.P.ONE
1000	ps3	1	000000000				0001		pd	0101011	SO.P.ONE.Z
1000	ps3	0	0000		vs1		0010		pd	0101011	SO.P.VR
1000	ps3	1	0000		vs1		0010		pd	0101011	SO.P.VR.Z
1000	ps3	0	00000			ps1	0011		pd	0101011	SO.P.NOT
1000	ps3	1	00000			ps1	0011		pd	0101011	SO.P.NOT.Z
1000	ps3	0	00000			ps1	0100		pd	0101011	SO.P.MV
1000	ps3	1	00000			ps1	0100		pd	0101011	SO.P.MV.Z
1000	ps3	0	00000			ps1	0101		pd	0101011	SO.P.MVT
1000	ps3	1	00000			ps1	0101		pd	0101011	SO.P.MVT.Z
1000	000	0	00	00	0	ps1	0110		pd	0101011	SO.P.CV.B.B
1000	000	1	00	00	0	ps1	0110		pd	0101011	SO.P.CV.B.B.Z
1000	000	0	01	01	0	ps1	0110		pd	0101011	SO.P.CV.H.H
1000	000	1	01	01	0	ps1	0110		pd	0101011	SO.P.CV.H.H.Z
1000	000	0	10	10	0	ps1	0110		pd	0101011	SO.P.CV.W.W
1000	000	1	10	10	0	ps1	0110		pd	0101011	SO.P.CV.W.W.Z
1000	000	0	11	11	0	ps1	0110		pd	0101011	SO.P.CV.D.D
1000	000	1	11	11	0	ps1	0110		pd	0101011	SO.P.CV.D.D.Z
1000	ps3	vs2			vs1		100	0	pd	0101011	SO.P.GE.US
1000	ps3	vs2			vs1		100	1	pd	0101011	SO.P.GE.US.Z
1000	ps3	vs2			vs1		101	0	pd	0101011	SO.P.GE.FP
1000	ps3	vs2			vs1		101	1	pd	0101011	SO.P.GE.FP.Z
1000	ps3	vs2			vs1		110	0	pd	0101011	SO.P.GE.SG
1000	ps3	vs2			vs1		110	1	pd	0101011	SO.P.GE.SG.Z
1001	ps3	vs2			vs1		000	0	pd	0101011	SO.P.EQ.US
1001	ps3	vs2			vs1		000	1	pd	0101011	SO.P.EQ.US.Z
1001	ps3	vs2			vs1		001	0	pd	0101011	SO.P.EQ.FP
1001	ps3	vs2			vs1		001	1	pd	0101011	SO.P.EQ.FP.Z
1001	ps3	vs2			vs1		010	0	pd	0101011	SO.P.EQ.SG
1001	ps3	vs2			vs1		010	1	pd	0101011	SO.P.EQ.SG.Z
1001	ps3	vs2			vs1		100	0	pd	0101011	SO.P.LT.US
1001	ps3	vs2			vs1		100	1	pd	0101011	SO.P.LT.US.Z
1001	ps3	vs2			vs1		101	0	pd	0101011	SO.P.LT.FP
1001	ps3	vs2			vs1		101	1	pd	0101011	SO.P.LT.FP.Z
1001	ps3	vs2			vs1		110	0	pd	0101011	SO.P.LT.SG
1001	ps3	vs2			vs1		110	1	pd	0101011	SO.P.LT.SG.Z

Vector Control Instructions

10110	0000000	rs1	000	rd	0101011	SO.C.SETVL
10110	000000000000		111	rd	0101011	SO.C.GETVL
10110	000000000000		001	vd	0101011	SO.C.SUSPD
10110	000000000000		010	vd	0101011	SO.C.RESUM
10110	000000000000		011	vd	0101011	SO.C.BREAK
10110	000000000000		100	vd	0101011	SO.C.VLOAD
10110	000000000000		101	vd	0101011	SO.C.VSTOR

Appendix C

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
funct5			funct4			ps2	vs1/rs1	funct3	vd/rd	opcode	UV1-type
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Vector Manipulation Instructions

10101	1000	ps2	rs1	000	vd	0101011	SO.V.DP.B
10101	1000	ps2	rs1	001	vd	0101011	SO.V.DP.H
10101	1000	ps2	rs1	010	vd	0101011	SO.V.DP.W
10101	1000	ps2	rs1	011	vd	0101011	SO.V.DP.D
10101	0010	000	vs1	000	rd	0101011	SO.V.MVVS
10101	0011	000	rs1	000	vd	0101011	SO.V.MVSV.B
10101	0011	000	rs1	001	vd	0101011	SO.V.MVSV.H
10101	0011	000	rs1	010	vd	0101011	SO.V.MVSV.W
10101	0011	000	rs1	011	vd	0101011	SO.V.MVSV.D
10101	0100	000	vs1	000	vd	0101011	SO.V.CV.US.B
10101	0100	000	vs1	001	vd	0101011	SO.V.CV.US.H
10101	0100	000	vs1	010	vd	0101011	SO.V.CV.US.W
10101	0100	000	vs1	011	vd	0101011	SO.V.CV.US.D
10101	0101	000	vs1	000	vd	0101011	SO.V.CV.FP.B
10101	0101	000	vs1	001	vd	0101011	SO.V.CV.FP.H
10101	0101	000	vs1	010	vd	0101011	SO.V.CV.FP.W
10101	0101	000	vs1	011	vd	0101011	SO.V.CV.FP.D
10101	0110	000	vs1	000	vd	0101011	SO.V.CV.SG.B
10101	0110	000	vs1	001	vd	0101011	SO.V.CV.SG.H
10101	0110	000	vs1	010	vd	0101011	SO.V.CV.SG.W
10101	0110	000	vs1	011	vd	0101011	SO.V.CV.SG.D

Stream Configuration Instructions

0	0	000	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B
0	0	000	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.MEM1
0	0	000	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.MEM2
0	0	000	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.MEM3
0	0	000	00	1	00	00	rs1	100	vd	0001011	SS.STA.LD.B.INDS
0	0	000	00	1	01	00	rs1	100	vd	0001011	SS.STA.LD.B.INDS.MEM1
0	0	000	00	1	10	00	rs1	100	vd	0001011	SS.STA.LD.B.INDS.MEM2
0	0	000	00	1	11	00	rs1	100	vd	0001011	SS.STA.LD.B.INDS.MEM3
1	0	000	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.M
1	0	000	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.M.MEM1
1	0	000	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.M.MEM2
1	0	000	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.M.MEM3
1	0	000	00	1	00	00	rs1	100	vd	0001011	SS.STA.LD.B.M.INDS
1	0	000	00	1	01	00	rs1	100	vd	0001011	SS.STA.LD.B.M.INDS.MEM1
1	0	000	00	1	10	00	rs1	100	vd	0001011	SS.STA.LD.B.M.INDS.MEM2
1	0	000	00	1	11	00	rs1	100	vd	0001011	SS.STA.LD.B.M.INDS.MEM3
0	0	000	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H
0	0	000	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.MEM1
0	0	000	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.MEM2
0	0	000	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.MEM3
0	0	000	00	1	00	00	rs1	101	vd	0001011	SS.STA.LD.H.INDS
0	0	000	00	1	01	00	rs1	101	vd	0001011	SS.STA.LD.H.INDS.MEM1
0	0	000	00	1	10	00	rs1	101	vd	0001011	SS.STA.LD.H.INDS.MEM2
0	0	000	00	1	11	00	rs1	101	vd	0001011	SS.STA.LD.H.INDS.MEM3
1	0	000	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.M
1	0	000	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.M.MEM1
1	0	000	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.M.MEM2
1	0	000	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.M.MEM3
1	0	000	00	1	00	00	rs1	101	vd	0001011	SS.STA.LD.H.M.INDS
1	0	000	00	1	01	00	rs1	101	vd	0001011	SS.STA.LD.H.M.INDS.MEM1
1	0	000	00	1	10	00	rs1	101	vd	0001011	SS.STA.LD.H.M.INDS.MEM2
1	0	000	00	1	11	00	rs1	101	vd	0001011	SS.STA.LD.H.M.INDS.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type
					Stream Configuration Instructions (Continuation)						
0	0	000	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W
0	0	000	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.MEM1
0	0	000	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.MEM2
0	0	000	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.MEM3
0	0	000	00	1	00	00	rs1	110	vd	0001011	SS.STA.LD.W.INDS
0	0	000	00	1	01	00	rs1	110	vd	0001011	SS.STA.LD.W.INDS.MEM1
0	0	000	00	1	10	00	rs1	110	vd	0001011	SS.STA.LD.W.INDS.MEM2
0	0	000	00	1	11	00	rs1	110	vd	0001011	SS.STA.LD.W.INDS.MEM3
1	0	000	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.M
1	0	000	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.M.MEM1
1	0	000	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.M.MEM2
1	0	000	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.M.MEM3
1	0	000	00	1	00	00	rs1	110	vd	0001011	SS.STA.LD.W.M.INDS
1	0	000	00	1	01	00	rs1	110	vd	0001011	SS.STA.LD.W.M.INDS.MEM1
1	0	000	00	1	10	00	rs1	110	vd	0001011	SS.STA.LD.W.M.INDS.MEM2
1	0	000	00	1	11	00	rs1	110	vd	0001011	SS.STA.LD.W.M.INDS.MEM3
0	0	000	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D
0	0	000	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.MEM1
0	0	000	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.MEM2
0	0	000	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.MEM3
0	0	000	00	1	00	00	rs1	111	vd	0001011	SS.STA.LD.D.INDS
0	0	000	00	1	01	00	rs1	111	vd	0001011	SS.STA.LD.D.INDS.MEM1
0	0	000	00	1	10	00	rs1	111	vd	0001011	SS.STA.LD.D.INDS.MEM2
0	0	000	00	1	11	00	rs1	111	vd	0001011	SS.STA.LD.D.INDS.MEM3
1	0	000	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.M
1	0	000	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.M.MEM1
1	0	000	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.M.MEM2
1	0	000	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.M.MEM3
1	0	000	00	1	00	00	rs1	111	vd	0001011	SS.STA.LD.D.M.INDS
1	0	000	00	1	01	00	rs1	111	vd	0001011	SS.STA.LD.D.M.INDS.MEM1
1	0	000	00	1	10	00	rs1	111	vd	0001011	SS.STA.LD.D.M.INDS.MEM2
1	0	000	00	1	11	00	rs1	111	vd	0001011	SS.STA.LD.D.M.INDS.MEM3
0	1	000	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1
0	1	000	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.MEM1
0	1	000	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.MEM2
0	1	000	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.MEM3
1	1	000	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.M
1	1	000	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1
0	1	000	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.MEM1
0	1	000	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.MEM2
0	1	000	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.MEM3
1	1	000	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.M
1	1	000	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type
					Stream Configuration Instructions (Continuation)						
0	1	000	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.MEM1
0	1	000	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.MEM2
0	1	000	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.MEM3
1	1	000	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.M
1	1	000	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1
0	1	000	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.MEM1
0	1	000	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.MEM2
0	1	000	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.MEM3
1	1	000	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.M
1	1	000	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.M.MEM3
0	1	001	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2
0	1	001	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.MEM1
0	1	001	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.MEM2
0	1	001	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.MEM3
1	1	001	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.M
1	1	001	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2
0	1	001	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.MEM1
0	1	001	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.MEM2
0	1	001	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.MEM3
1	1	001	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.M
1	1	001	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2
0	1	001	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.MEM1
0	1	001	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.MEM2
0	1	001	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.MEM3
1	1	001	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.M
1	1	001	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2
0	1	001	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.MEM1
0	1	001	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.MEM2
0	1	001	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.MEM3
1	1	001	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.M
1	1	001	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.M.MEM3
0	1	010	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3
0	1	010	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.MEM1
0	1	010	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.MEM2
0	1	010	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type
Stream Configuration Instructions (Continuation)											
1	1	010	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.M
1	1	010	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3
0	1	010	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.MEM1
0	1	010	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.MEM2
0	1	010	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.MEM3
1	1	010	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.M
1	1	010	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3
0	1	010	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.MEM1
0	1	010	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.MEM2
0	1	010	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.MEM3
1	1	010	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.M
1	1	010	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3
0	1	010	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.MEM1
0	1	010	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.MEM2
0	1	010	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.MEM3
1	1	010	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.M
1	1	010	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.M.MEM3
0	1	011	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4
0	1	011	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.MEM1
0	1	011	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.MEM2
0	1	011	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.MEM3
1	1	011	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.M
1	1	011	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4
0	1	011	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.MEM1
0	1	011	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.MEM2
0	1	011	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.MEM3
1	1	011	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.M
1	1	011	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4
0	1	011	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.MEM1
0	1	011	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.MEM2
0	1	011	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

1	1	011	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.M
1	1	011	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4
0	1	011	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.MEM1
0	1	011	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.MEM2
0	1	011	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.MEM3
1	1	011	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.M
1	1	011	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.M.MEM3
0	1	100	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5
0	1	100	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.MEM1
0	1	100	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.MEM2
0	1	100	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.MEM3
1	1	100	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.M
1	1	100	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5
0	1	100	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.MEM1
0	1	100	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.MEM2
0	1	100	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.MEM3
1	1	100	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.M
1	1	100	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5
0	1	100	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.MEM1
0	1	100	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.MEM2
0	1	100	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.MEM3
1	1	100	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.M
1	1	100	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5
0	1	100	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.MEM1
0	1	100	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.MEM2
0	1	100	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.MEM3
1	1	100	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.M
1	1	100	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.M.MEM3
0	1	101	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6
0	1	101	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.MEM1
0	1	101	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.MEM2
0	1	101	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.MEM3
1	1	101	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.M
1	1	101	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

0	1	101	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6
0	1	101	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.MEM1
0	1	101	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.MEM2
0	1	101	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.MEM3
1	1	101	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.M
1	1	101	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6
0	1	101	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.MEM1
0	1	101	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.MEM2
0	1	101	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.MEM3
1	1	101	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.M
1	1	101	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6
0	1	101	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.MEM1
0	1	101	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.MEM2
0	1	101	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.MEM3
1	1	101	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.M
1	1	101	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.M.MEM3
0	1	110	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7
0	1	110	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.MEM1
0	1	110	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.MEM2
0	1	110	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.MEM3
1	1	110	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.M
1	1	110	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.M.MEM3
0	1	110	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7
0	1	110	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.MEM1
0	1	110	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.MEM2
0	1	110	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.MEM3
1	1	110	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.M
1	1	110	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.M.MEM3
0	1	110	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7
0	1	110	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.MEM1
0	1	110	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.MEM2
0	1	110	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.MEM3
1	1	110	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.M
1	1	110	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type
				Stream Configuration Instructions (Continuation)							
0	1	110	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7
0	1	110	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.MEM1
0	1	110	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.MEM2
0	1	110	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.MEM3
1	1	110	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.M
1	1	110	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.M.MEM3
0	1	111	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V
0	1	111	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.MEM1
0	1	111	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.MEM2
0	1	111	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.MEM3
1	1	111	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.M
1	1	111	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.M.MEM1
1	1	111	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.M.MEM2
1	1	111	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.M.MEM3
0	1	111	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V
0	1	111	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.MEM1
0	1	111	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.MEM2
0	1	111	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.MEM3
1	1	111	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.M
1	1	111	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.M.MEM1
1	1	111	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.M.MEM2
1	1	111	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.M.MEM3
0	1	111	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V
0	1	111	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.MEM1
0	1	111	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.MEM2
0	1	111	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.MEM3
1	1	111	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.M
1	1	111	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.M.MEM1
1	1	111	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.M.MEM2
1	1	111	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.M.MEM3
0	1	111	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V
0	1	111	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.MEM1
0	1	111	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.MEM2
0	1	111	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.MEM3
1	1	111	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.M
1	1	111	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.M.MEM1
1	1	111	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.M.MEM2
1	1	111	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.M.MEM3
0	0	000	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B
0	0	000	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.MEM1
0	0	000	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.MEM2
0	0	000	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.MEM3
1	0	000	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.M
1	0	000	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.M.MEM1
1	0	000	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.M.MEM2
1	0	000	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

0	0	000	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H
0	0	000	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.MEM1
0	0	000	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.MEM2
0	0	000	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.MEM3
1	0	000	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.M
1	0	000	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.M.MEM1
1	0	000	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.M.MEM2
1	0	000	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.M.MEM3
0	0	000	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W
0	0	000	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.MEM1
0	0	000	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.MEM2
0	0	000	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.MEM3
1	0	000	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.M
1	0	000	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.M.MEM1
1	0	000	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.M.MEM2
1	0	000	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.M.MEM3
0	0	000	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D
0	0	000	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.MEM1
0	0	000	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.MEM2
0	0	000	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.MEM3
1	0	000	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.M
1	0	000	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.M.MEM1
1	0	000	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.M.MEM2
1	0	000	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.M.MEM3
0	1	000	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1
0	1	000	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.MEM1
0	1	000	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.MEM2
0	1	000	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.MEM3
1	1	000	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.M
1	1	000	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1
0	1	000	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.MEM1
0	1	000	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.MEM2
0	1	000	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.MEM3
1	1	000	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.M
1	1	000	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1
0	1	000	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.MEM1
0	1	000	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.MEM2
0	1	000	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.MEM3
1	1	000	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.M
1	1	000	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

0	1	000	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1
0	1	000	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.MEM1
0	1	000	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.MEM2
0	1	000	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.MEM3
1	1	000	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.M
1	1	000	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.M.MEM3
0	1	001	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2
0	1	001	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.MEM1
0	1	001	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.MEM2
0	1	001	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.MEM3
1	1	001	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.M
1	1	001	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2
0	1	001	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.MEM1
0	1	001	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.MEM2
0	1	001	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.MEM3
1	1	001	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.M
1	1	001	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2
0	1	001	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.MEM1
0	1	001	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.MEM2
0	1	001	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.MEM3
1	1	001	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.M
1	1	001	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2
0	1	001	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.MEM1
0	1	001	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.MEM2
0	1	001	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.MEM3
1	1	001	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.M
1	1	001	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.M.MEM3
0	1	010	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3
0	1	010	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.MEM1
0	1	010	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.MEM2
0	1	010	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.MEM3
1	1	010	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.M
1	1	010	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

0	1	010	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3
0	1	010	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.MEM1
0	1	010	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.MEM2
0	1	010	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.MEM3
1	1	010	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.M
1	1	010	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3
0	1	010	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.MEM1
0	1	010	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.MEM2
0	1	010	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.MEM3
1	1	010	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.M
1	1	010	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3
0	1	010	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.MEM1
0	1	010	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.MEM2
0	1	010	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.MEM3
1	1	010	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.M
1	1	010	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.M.MEM3
0	1	011	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4
0	1	011	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.MEM1
0	1	011	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.MEM2
0	1	011	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.MEM3
1	1	011	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.M
1	1	011	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4
0	1	011	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.MEM1
0	1	011	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.MEM2
0	1	011	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.MEM3
1	1	011	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.M
1	1	011	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4
0	1	011	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.MEM1
0	1	011	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.MEM2
0	1	011	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.MEM3
1	1	011	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.M
1	1	011	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

0	1	011	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4
0	1	011	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.MEM1
0	1	011	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.MEM2
0	1	011	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.MEM3
1	1	011	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.M
1	1	011	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.M.MEM3
0	1	100	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5
0	1	100	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.MEM1
0	1	100	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.MEM2
0	1	100	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.MEM3
1	1	100	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.M
1	1	100	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5
0	1	100	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.MEM1
0	1	100	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.MEM2
0	1	100	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.MEM3
1	1	100	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.M
1	1	100	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5
0	1	100	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.MEM1
0	1	100	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.MEM2
0	1	100	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.MEM3
1	1	100	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.M
1	1	100	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5
0	1	100	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.MEM1
0	1	100	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.MEM2
0	1	100	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.MEM3
1	1	100	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.M
1	1	100	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.M.MEM3
0	1	101	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6
0	1	101	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.MEM1
0	1	101	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.MEM2
0	1	101	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.MEM3
1	1	101	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.M
1	1	101	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6
0	1	101	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.MEM1
0	1	101	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.MEM2
0	1	101	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

1	1	101	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.M
1	1	101	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6
0	1	101	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.MEM1
0	1	101	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.MEM2
0	1	101	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.MEM3
1	1	101	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.M
1	1	101	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6
0	1	101	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.MEM1
0	1	101	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.MEM2
0	1	101	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.MEM3
1	1	101	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.M
1	1	101	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.M.MEM3
0	1	110	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7
0	1	110	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.MEM1
0	1	110	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.MEM2
0	1	110	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.MEM3
1	1	110	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.M
1	1	110	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.M.MEM3
0	1	110	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7
0	1	110	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.MEM1
0	1	110	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.MEM2
0	1	110	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.MEM3
1	1	110	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.M
1	1	110	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.M.MEM3
0	1	110	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.7
0	1	110	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.7.MEM1
0	1	110	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.7.MEM2
0	1	110	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.7.MEM3
1	1	110	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.7.M
1	1	110	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.7.M.MEM3
0	1	110	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.7
0	1	110	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.7.MEM1
0	1	110	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.7.MEM2
0	1	110	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.7.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 18	17 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1		funct3	vd	opcode	USTA-type
rs3			funct2	b		t	-	tdim	funct3	vd	opcode	UMOD-type
rs3			funct2	rs2			rs1		funct3	vd	opcode	UAE-type
					Stream Configuration Instructions (Continuation)							
1	1	110	00	0	00	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.M
1	1	110	00	0	01	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.M.MEM1
1	1	110	00	0	10	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.M.MEM2
1	1	110	00	0	11	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.M.MEM3
0	1	111	00	0	00	00	rs1		000	vd	0001011	SS.STA.ST.B.V
0	1	111	00	0	01	00	rs1		000	vd	0001011	SS.STA.ST.B.V.MEM1
0	1	111	00	0	10	00	rs1		000	vd	0001011	SS.STA.ST.B.V.MEM2
0	1	111	00	0	11	00	rs1		000	vd	0001011	SS.STA.ST.B.V.MEM3
1	1	111	00	0	00	00	rs1		000	vd	0001011	SS.STA.ST.B.V.M
1	1	111	00	0	01	00	rs1		000	vd	0001011	SS.STA.ST.B.V.M.MEM1
1	1	111	00	0	10	00	rs1		000	vd	0001011	SS.STA.ST.B.V.M.MEM2
1	1	111	00	0	11	00	rs1		000	vd	0001011	SS.STA.ST.B.V.M.MEM3
0	1	111	00	0	00	00	rs1		001	vd	0001011	SS.STA.ST.H.V
0	1	111	00	0	01	00	rs1		001	vd	0001011	SS.STA.ST.H.V.MEM1
0	1	111	00	0	10	00	rs1		001	vd	0001011	SS.STA.ST.H.V.MEM2
0	1	111	00	0	11	00	rs1		001	vd	0001011	SS.STA.ST.H.V.MEM3
1	1	111	00	0	00	00	rs1		001	vd	0001011	SS.STA.ST.H.V.M
1	1	111	00	0	01	00	rs1		001	vd	0001011	SS.STA.ST.H.V.M.MEM1
1	1	111	00	0	10	00	rs1		001	vd	0001011	SS.STA.ST.H.V.M.MEM2
1	1	111	00	0	11	00	rs1		001	vd	0001011	SS.STA.ST.H.V.M.MEM3
0	1	111	00	0	00	00	rs1		010	vd	0001011	SS.STA.ST.W.V
0	1	111	00	0	01	00	rs1		010	vd	0001011	SS.STA.ST.W.V.MEM1
0	1	111	00	0	10	00	rs1		010	vd	0001011	SS.STA.ST.W.V.MEM2
0	1	111	00	0	11	00	rs1		010	vd	0001011	SS.STA.ST.W.V.MEM3
1	1	111	00	0	00	00	rs1		010	vd	0001011	SS.STA.ST.W.V.M
1	1	111	00	0	01	00	rs1		010	vd	0001011	SS.STA.ST.W.V.M.MEM1
1	1	111	00	0	10	00	rs1		010	vd	0001011	SS.STA.ST.W.V.M.MEM2
1	1	111	00	0	11	00	rs1		010	vd	0001011	SS.STA.ST.W.V.M.MEM3
0	1	111	00	0	00	00	rs1		011	vd	0001011	SS.STA.ST.D.V
0	1	111	00	0	01	00	rs1		011	vd	0001011	SS.STA.ST.D.V.MEM1
0	1	111	00	0	10	00	rs1		011	vd	0001011	SS.STA.ST.D.V.MEM2
0	1	111	00	0	11	00	rs1		011	vd	0001011	SS.STA.ST.D.V.MEM3
1	1	111	00	0	00	00	rs1		011	vd	0001011	SS.STA.ST.D.V.M
1	1	111	00	0	01	00	rs1		011	vd	0001011	SS.STA.ST.D.V.M.MEM1
1	1	111	00	0	10	00	rs1		011	vd	0001011	SS.STA.ST.D.V.M.MEM2
1	1	111	00	0	11	00	rs1		011	vd	0001011	SS.STA.ST.D.V.M.MEM3
rs3			01	rs2			rs1		000	vd	0001011	SS.APP
rs3			01	000		00	00	000	100	vd	0001011	SS.APP.MOD.SIZ.INC.1
rs3			01	001		00	00	000	100	vd	0001011	SS.APP.MOD.SIZ.DEC.1
rs3			01	000		01	00	000	100	vd	0001011	SS.APP.MOD.STR.INC.1
rs3			01	001		01	00	000	100	vd	0001011	SS.APP.MOD.STR.DEC.1
rs3			01	000		10	00	000	100	vd	0001011	SS.APP.MOD.OFS.INC.1
rs3			01	001		10	00	000	100	vd	0001011	SS.APP.MOD.OFS.DEC.1
rs3			01	000		00	00	001	100	vd	0001011	SS.APP.MOD.SIZ.INC.2
rs3			01	001		00	00	001	100	vd	0001011	SS.APP.MOD.SIZ.DEC.2
rs3			01	000		01	00	001	100	vd	0001011	SS.APP.MOD.STR.INC.2
rs3			01	001		01	00	001	100	vd	0001011	SS.APP.MOD.STR.DEC.2
rs3			01	000		10	00	001	100	vd	0001011	SS.APP.MOD.OFS.INC.2
rs3			01	001		10	00	001	100	vd	0001011	SS.APP.MOD.OFS.DEC.2

31		27	26 25	24 22	21 20	19 18	17 15	14 12	11 7	6 0	
rs3			funct2	b	t	-	tdim	funct3	vd	opcode	UMOD-type
-	tdim	sg	funct2	b	t	vs1		funct3	vd	opcode	UIND-type

Stream Configuration Instructions (Continuation)

rs3			01	000	00	00	010	100	vd	0001011	SS.APP.MOD.SIZ.INC.3
rs3			01	001	00	00	010	100	vd	0001011	SS.APP.MOD.SIZ.DEC.3
rs3			01	000	01	00	010	100	vd	0001011	SS.APP.MOD.STR.INC.3
rs3			01	001	01	00	010	100	vd	0001011	SS.APP.MOD.STR.DEC.3
rs3			01	000	10	00	010	100	vd	0001011	SS.APP.MOD.OFS.INC.3
rs3			01	001	10	00	010	100	vd	0001011	SS.APP.MOD.OFS.DEC.3
rs3			01	000	00	00	011	100	vd	0001011	SS.APP.MOD.SIZ.INC.4
rs3			01	001	00	00	011	100	vd	0001011	SS.APP.MOD.SIZ.DEC.4
rs3			01	000	01	00	011	100	vd	0001011	SS.APP.MOD.STR.INC.4
rs3			01	001	01	00	011	100	vd	0001011	SS.APP.MOD.STR.DEC.4
rs3			01	000	10	00	011	100	vd	0001011	SS.APP.MOD.OFS.INC.4
rs3			01	001	10	00	011	100	vd	0001011	SS.APP.MOD.OFS.DEC.4
rs3			01	000	00	00	100	100	vd	0001011	SS.APP.MOD.SIZ.INC.5
rs3			01	001	00	00	100	100	vd	0001011	SS.APP.MOD.SIZ.DEC.5
rs3			01	000	01	00	100	100	vd	0001011	SS.APP.MOD.STR.INC.5
rs3			01	001	01	00	100	100	vd	0001011	SS.APP.MOD.STR.DEC.5
rs3			01	000	10	00	100	100	vd	0001011	SS.APP.MOD.OFS.INC.5
rs3			01	001	10	00	100	100	vd	0001011	SS.APP.MOD.OFS.DEC.5
rs3			01	000	00	00	101	100	vd	0001011	SS.APP.MOD.SIZ.INC.6
rs3			01	001	00	00	101	100	vd	0001011	SS.APP.MOD.SIZ.DEC.6
rs3			01	000	01	00	101	100	vd	0001011	SS.APP.MOD.STR.INC.6
rs3			01	001	01	00	101	100	vd	0001011	SS.APP.MOD.STR.DEC.6
rs3			01	000	10	00	101	100	vd	0001011	SS.APP.MOD.OFS.INC.6
rs3			01	001	10	00	101	100	vd	0001011	SS.APP.MOD.OFS.DEC.6
rs3			01	000	00	00	110	100	vd	0001011	SS.APP.MOD.SIZ.INC.7
rs3			01	001	00	00	110	100	vd	0001011	SS.APP.MOD.SIZ.DEC.7
rs3			01	000	01	00	110	100	vd	0001011	SS.APP.MOD.STR.INC.7
rs3			01	001	01	00	110	100	vd	0001011	SS.APP.MOD.STR.DEC.7
rs3			01	000	10	00	110	100	vd	0001011	SS.APP.MOD.OFS.INC.7
rs3			01	001	10	00	110	100	vd	0001011	SS.APP.MOD.OFS.DEC.7
rs3			01	000	00	00	111	100	vd	0001011	SS.APP.MOD.SIZ.INC.L
rs3			01	001	00	00	111	100	vd	0001011	SS.APP.MOD.SIZ.DEC.L
rs3			01	000	01	00	111	100	vd	0001011	SS.APP.MOD.STR.INC.L
rs3			01	001	01	00	111	100	vd	0001011	SS.APP.MOD.STR.DEC.L
rs3			01	000	10	00	111	100	vd	0001011	SS.APP.MOD.OFS.INC.L
rs3			01	001	10	00	111	100	vd	0001011	SS.APP.MOD.OFS.DEC.L
0	000	0	01	000	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.INC.1
0	000	0	01	001	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.DEC.1
0	000	0	01	010	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.ADD.1
0	000	0	01	011	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.SUB.1
0	000	0	01	100	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.SET.1
0	000	0	01	000	01	vs1		110	vd	0001011	SS.APP.IND.STR.INC.1
0	000	0	01	001	01	vs1		110	vd	0001011	SS.APP.IND.STR.DEC.1
0	000	0	01	010	01	vs1		110	vd	0001011	SS.APP.IND.STR.ADD.1
0	000	0	01	011	01	vs1		110	vd	0001011	SS.APP.IND.STR.SUB.1
0	000	0	01	100	01	vs1		110	vd	0001011	SS.APP.IND.STR.SET.1
0	000	0	01	000	10	vs1		110	vd	0001011	SS.APP.IND.OFS.INC.1
0	000	0	01	001	10	vs1		110	vd	0001011	SS.APP.IND.OFS.DEC.1
0	000	0	01	010	10	vs1		110	vd	0001011	SS.APP.IND.OFS.ADD.1
0	000	0	01	011	10	vs1		110	vd	0001011	SS.APP.IND.OFS.SUB.1
0	000	0	01	100	10	vs1		110	vd	0001011	SS.APP.IND.OFS.SET.1

31	30 28	27	26 25	24 22	21 20	19 15	14 12	11 7	6 0	
-	tdim	sg	funct2	b	t	vs1	funct3	vd	opcode	UIND-type
				Stream Configuration Instructions (Continuation)						
0	001	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.2
0	001	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.2
0	001	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.2
0	001	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.2
0	001	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.2
0	001	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.2
0	001	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.2
0	001	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.2
0	001	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.2
0	001	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.2
0	001	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.2
0	001	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.2
0	001	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.2
0	001	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.2
0	001	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.2
0	010	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.3
0	010	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.3
0	010	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.3
0	010	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.3
0	010	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.3
0	010	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.3
0	010	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.3
0	010	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.3
0	010	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.3
0	010	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.3
0	010	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.3
0	010	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.3
0	010	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.3
0	010	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.3
0	010	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.3
0	011	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.4
0	011	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.4
0	011	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.4
0	011	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.4
0	011	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.4
0	011	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.4
0	011	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.4
0	011	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.4
0	011	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.4
0	011	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.4
0	011	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.4
0	011	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.4
0	011	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.4
0	011	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.4
0	011	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.4

31	30 28	27	26 25	24 22	21 20	19 15	14 12	11 7	6 0	
-	tdim	sg	funct2	b	t	vs1	funct3	vd	opcode	UIND-type

Stream Configuration Instructions (Continuation)

0	100	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.5
0	100	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.5
0	100	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.5
0	100	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.5
0	100	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.5
0	100	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.5
0	100	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.5
0	100	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.5
0	100	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.5
0	100	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.5
0	100	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.5
0	100	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.5
0	100	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.5
0	100	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.5
0	100	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.5
0	101	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.6
0	101	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.6
0	101	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.6
0	101	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.6
0	101	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.6
0	101	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.6
0	101	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.6
0	101	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.6
0	101	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.6
0	101	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.6
0	101	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.6
0	101	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.6
0	101	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.6
0	101	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.6
0	101	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.6
0	110	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.7
0	110	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.7
0	110	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.7
0	110	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.7
0	110	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.7
0	110	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.7
0	110	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.7
0	110	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.7
0	110	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.7
0	110	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.7
0	110	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.7
0	110	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.7
0	110	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.7
0	110	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.7
0	110	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.7

31	30 28	27	26 25	24 22	21 20	19 15	14 12	11 7	6 0	
-	tdim	sg	funct2	b	t	vs1	funct3	vd	opcode	UIND-type
rs3			funct2	rs2		rs1	funct3	vd	opcode	UAE-type
Stream Configuration Instructions (Continuation)										
0	111	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.L
0	111	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.L
0	111	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.L
0	111	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.L
0	111	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.L
0	111	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.L
0	111	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.L
0	111	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.L
0	111	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.L
0	111	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.L
0	111	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.L
0	111	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.L
0	111	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.L
0	111	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.L
0	111	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.L
0	000	1	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SG.INC
0	000	1	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SG.DEC
0	000	1	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SG.ADD
0	000	1	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SG.SUB
0	000	1	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SG.SET
0	000	1	10	000	10	vs1	110	vd	0001011	SS.END.IND.OFS.SG.INC
0	000	1	10	001	10	vs1	110	vd	0001011	SS.END.IND.OFS.SG.DEC
0	000	1	10	010	10	vs1	110	vd	0001011	SS.END.IND.OFS.SG.ADD
0	000	1	10	011	10	vs1	110	vd	0001011	SS.END.IND.OFS.SG.SUB
0	000	1	10	100	10	vs1	110	vd	0001011	SS.END.IND.OFS.SG.SET
rs3			10	rs2		rs1	000	vd	0001011	SS.END

Table C.1: Unlimited Vector Extension (UVE) instruction listing for RISC-V

Appendix D

Paper presented at CAMS 2023

A functional validation framework for the Unlimited Vector Extension

Ana Beatriz Fernandes
Instituto de Telecomunicações
Dept. of Electrical and Computer Eng.
University of Coimbra, Portugal
ana.fernandes@co.it.pt

Nuno Neves
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
nuno.neves@inesc-id.pt

Luís Crespo
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
luis.miguel.crespo@tecnico.ulisboa.pt

Pedro Tomás
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
pedro.tomas@inesc-id.pt

Nuno Roma
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
nuno.roma@inesc-id.pt

Gabriel Falcao
Instituto de Telecomunicações
Dept. of Electrical and Computer Eng.
University of Coimbra, Portugal
gff@co.it.pt

ABSTRACT

The Unlimited Vector Extension (UVE) was already proposed to tackle the limitations of current state-of-the-art Vector-Length Agnostic (VLA) extensions. This is a new Instruction Set Architecture (ISA) extension that aims to reduce loop control and memory access indexation overheads, as well as memory access latency, joining data streaming and Single Instruction, Multiple Data (SIMD) processing. This ISA extension has already been validated in a cycle-accurate simulator, *gem5*, with a first implementation made on an out-of-order processor model, based on the ARM Cortex-A76. However, as compilation support is currently being developed, and several shortcomings and improvements on the existing specification have been identified, an increasing need to efficiently run and validate UVE code has surged. As such, support for UVE has been added to the *Spike* simulator. This is the golden reference functional RISC-V ISA simulator, written in C++. To achieve this, the simulator has been extended to accommodate for the necessary architecture changes, such as new registers that hold the data streams (streaming registers) together with a convenient Streaming Unit that emulates the configuration and manipulation of the streams. The result is a powerful tool that provides the possibility to validate all current features and improvements of UVE, along with some preliminary code obtained from the compiler currently under development.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; *Reduced instruction set computing*; **Data flow architectures**; • **Computing methodologies → Simulation tools**.

KEYWORDS

ISA SIMD Extensions, Data Streaming, RISC-V, Unlimited Vector Extension, Simulation Tools, Data Flow Architecture

ACM Reference Format:
Ana Beatriz Fernandes, Nuno Neves, Luís Crespo, Pedro Tomás, Nuno Roma, and Gabriel Falcao. 2024. A functional validation framework for the Unlimited Vector Extension. In *Proceedings of ACM Conference (CAMS'23).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the last few decades, there has been an increasing need to improve processors' performance, as computational and memory-intensive applications become more common (e.g. Machine Learning and Sparse Linear Algebra). However, with the end of Dennard Scaling and Moore's Law, more traditional methods of improving performance have been revealed to be insufficient, such as increasing the clock frequency and the use of cache memory.

Several solutions have been proposed and are now widely established, such as Instruction-Level Parallelism (ILP) and Data-Level Parallelism (DLP), common in modern high-performance processors. The latter, hidden in Single Instruction, Multiple Data (SIMD) units [8], allow the simultaneous processing of multiple data elements. To take advantage of this, a plethora of SIMD Instruction Set Architectures (ISAs) has been developed, such as Arm NEON [1] and x86 AVX [11], focused on operating on fixed-size registers. However, because a vector's optimal size depends on the application, this approach presents some limitations. To overcome this, other vector-length agnostic extensions have emerged, particularly the RISC-V Vector Extension (RVV) [24] and the Scalable Vector Extension (SVE) [22], which allow for the size of the vector register to be defined at runtime. This means that different processors, with different application requirements, can adopt distinct vector sizes, with no need to modify the source code. However, a new problem arises with these extensions, as predicate [2] and vector control instructions become necessary to disable elements outside loop bounds, which leads to more loop instructions [17], and thus more overhead and decrease of performance.

The RISC-V Unlimited Vector Extension (UVE), proposed and developed by Domingos et al. [8], joins two promising solutions for improving performance: scalable SIMD extensions and data streaming. RISC-V was chosen as the base ISA due to its open-source nature, as well as its simple and extensible instruction set. By relying on data streaming, this novel RISC-V ISA extension has several improvements when compared to the ones mentioned above, such as decoupled memory accesses, indexing-free loops, simplified vectorisation, and implicit load/store operations [8]. The streaming paradigm allows for the configuration of memory access patterns at software level and the subsequent data fetching in the background, a clear step towards improving the memory access latency and throughput. This paradigm shift was already demonstrated in a proof-of-concept *gem5* implementation of UVE on an out-of-order processor model, based on the ARM Cortex-A76. It showed that it can improve the performance of a processor by up to 2.4 times when compared to other state-of-the-art implementations [8].

In accordance, this work exploits a prominent research trend that considers the use of unconventional architectures to improve the attained processor's performance. In particular, it presents a new modelling, simulation and validation tool to support the development of UVE, by not only independently validating the existing specification, but also introducing streaming support on *Spike* [18], on which several existing instructions were added, tested and, whenever pertinent, modified. Before this contribution, this process was very time-consuming and tightened to the several constraints imposed by *gem5*. Hence, it now becomes much more efficient with this new tool, as *Spike* offers a simpler instruction implementation pipeline. For this to be possible, the simulator was expanded to include a Streaming Unit (SU), similar to RVV's Vector Unit already present in the simulator. Moreover, a new RISC-V extension was added to the simulator, where many of the existing UVE instructions were added.

In order to test the streaming mechanisms and validate the functional behaviour of the chosen instructions, a subset from the benchmarks that were considered in UVE's proposal [8] was used, mainly based on Polybench/C [1]. It should be noted that for applications it is not yet possible to automatically generate UVE code from regular C code, which means that some benchmarks had to be manually written. This had already been done for previous works, but the available code was revised and compared to code generated from the compiler currently being developed [13] whenever possible.

The developed framework and supporting documentation are publicly available online [2]. This repository contains ongoing work and is thus subject to continual updates.

2 DATA STREAMING AND UVE

Memory access is the most time and energy-consuming operation in modern computer architectures [5], so it is natural that this is the main target of optimisation attempts. While cache structures greatly improve access latencies, they are dependent on data locality, which is not always guaranteed. Moreover, in applications with complex access patterns, it is often not possible to efficiently make use of these structures. Furthermore, if there is a large volume of data to be loaded/stored, particularly in multi-core systems, instances of cache contention (i.e. when multiple cores attempt to update the same cache line) and energy consumption rise [10]. This means that adapting the data communication scheme to the running application is crucial for performance increase. One re-emerging technology aiming to tackle this problem is data streaming [8, 13, 16, 20, 21, 23], which decouples memory accesses from data processing, effectively masking data transfers behind computation [7].

A stream is essentially a predicable vector of data elements that are processed sequentially. Each element of a stream is subject to the same set of operations and is discarded after the computation is complete. These structures rely almost solely on spatial locality, which means that the order in which the data is going to be consumed can be specified beforehand [7]. This is possible through data pattern descriptors, such as those proposed and developed in [8, 14, 15]. Understanding this representation model is pivotal to understanding UVE. Hence, the fundamentals of data streaming and pattern description are described next.

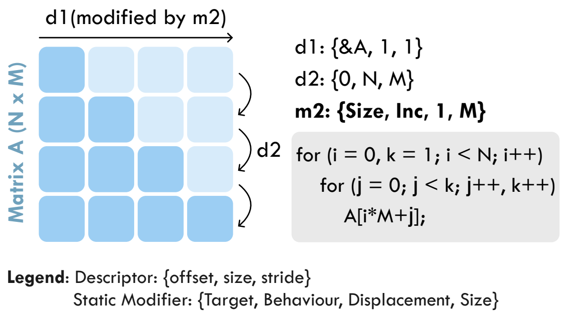
Any regular n-dimensional access sequence can be represented by the following affine function:

$$y(X) = y_{base} + \sum_{k=0}^{dim_y} x_k \times S_k, \quad (1)$$

with $X = x_0, ..., x_{dim_y}$ and $x_k \in [O_k, E_k + O_k]$.

This means that a stream access $y(X)$ is described as the sum of the base address of an n-dimensional variable (y_{base}) with dim_y pairs of indexing variables (x_k) and their respective strides (S_k), each k corresponding to a dimension of the pattern. E_k corresponds to the number of elements in each k dimension and O_k to the indexing offset. Because x_0 has $O_0 = 0$, it is equal to the base address of the variable [8]. Moreover, through a combination of affine functions of this kind, highly complex patterns can be attained, by assigning the base address and/or the offset of a function to the result of another one. Lastly, indirect memory accesses can also be described by taking the data obtained by the addresses generated by an affine function and injecting them into the aforementioned variables of another function.

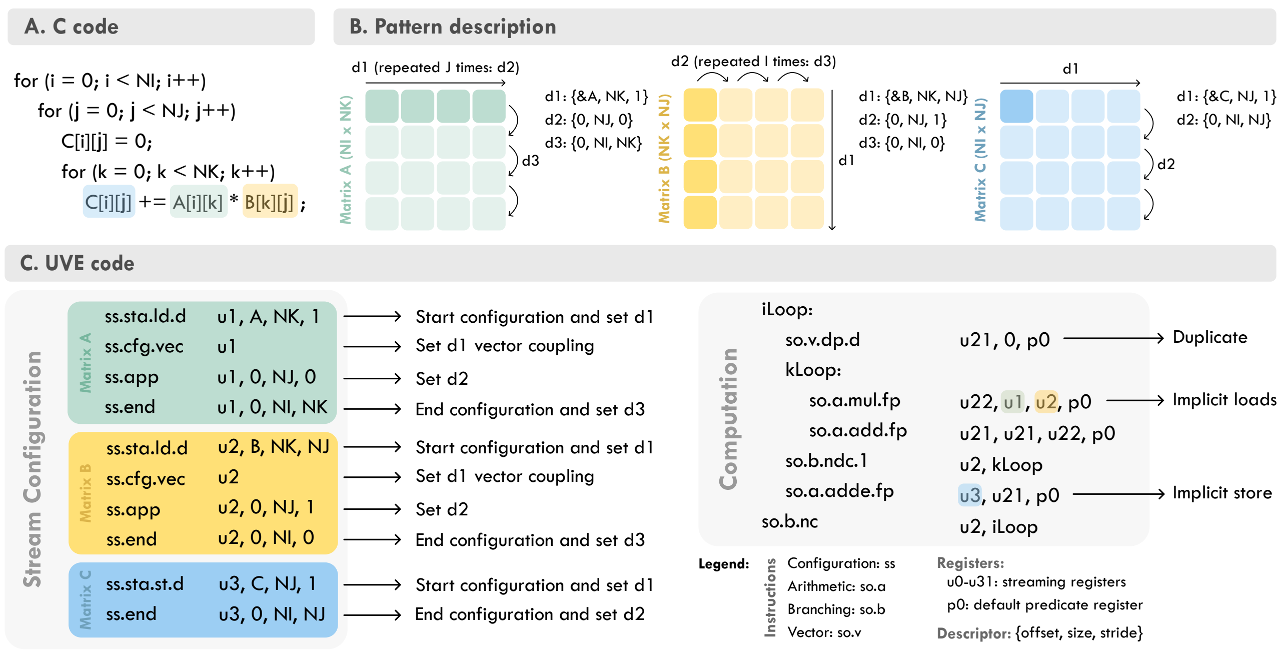
The proposed pattern representation model results from the encoding of the variables associated with each pattern dimension of the function described in Equation (1). This representation is based


d1(modified by m2)
Matrix A (N x M)
d2
d1: {&A, 1, 1}
d2: {0, N, M}
m2: {Size, Inc, 1, M}
Legend: Descriptor: {offset, size, stride}
Static Modifier: {Target, Behaviour, Displacement, Size}

```
for (i = 0, k = 1; i < N; i++)
    for (j = 0; j < k; j++, k++)
        A[i*M+j];
```

Figure 1: Triangular access pattern description, where a static modifier is applied to increment the size of the first dimension.

[1]https://web.cs.ucla.edu/ pouchet/software/polybench/
[2]https://github.com/hpc-ulisboa/UVE2



```
A. C code

for (i = 0; i < NI; i++)
  for (j = 0; j < NJ; j++)
    C[i][j] = 0;
    for (k = 0; k < NK; k++)
      C[i][j] += A[i][k] * B[k][j] ;
```

```
C. UVE code

Stream Configuration
Matrix A
  ss.sta.ld.d   u1, A, NK, 1     ──→ Start configuration and set d1
  ss.cfg.vec    u1               ──→ Set d1 vector coupling
  ss.app        u1, 0, NJ, 0     ──→ Set d2
  ss.end        u1, 0, NI, NK    ──→ End configuration and set d3
Matrix B
  ss.sta.ld.d   u2, B, NK, NJ    ──→ Start configuration and set d1
  ss.cfg.vec    u2               ──→ Set d1 vector coupling
  ss.app        u2, 0, NJ, 1     ──→ Set d2
  ss.end        u2, 0, NI, 0     ──→ End configuration and set d3
Matrix C
  ss.sta.st.d   u3, C, NJ, 1     ──→ Start configuration and set d1
  ss.end        u3, 0, NI, NJ    ──→ End configuration and set d2

Computation
iLoop:
  so.v.dp.d       u21, 0, p0         ──→ Duplicate
  kLoop:
    so.a.mul.fp   u22, u1, u2, p0    ──→ Implicit loads
    so.a.add.fp   u21, u21, u22, p0
  so.b.ndc.1      u2, kLoop
  so.a.adde.fp    u3, u21, p0        ──→ Implicit store
so.b.nc           u2, iLoop
```

Legend:
Instructions: Configuration: ss; Arithmetic: so.a; Branching: so.b; Vector: so.v
Registers: u0-u31: streaming registers; p0: default predicate register
Descriptor: {offset, size, stride}

Figure 2: Exemplification of two-matrix multiplication from (A) the C source code, through (B) the pattern description of each matrix data access and (C) the resulting UVE configuration and computation kernels.

on descriptors and modifiers (see Figure 1), defined in a set of dedicated instructions in UVE. Simple descriptors, that remain constant throughout execution, are exemplified in Figure 2, which is part of the kernel used in the *trisolv* benchmark (see Section 4). There are two types of optional modifiers, which when associated with a certain dimension of the descriptor are able to alter its parameters, allowing the modelling of inter-loop control dependencies that arise when loop conditions are affected by an outer loop. On the one hand, *static modifiers* are able to add or subtract a certain displacement to any of the dimension's parameters. On the other hand, *indirect modifiers* allow for the substitution of these parameters with pointers to data obtained from another stream. This makes it possible to create complex pattern descriptors, which are common in a plethora of applications, such as Sparse Algebra and Data Mining.

UVE adds 32 vector registers to the base ISA (named from *"u0"* to *"u31"*). The length of each vector is unlimited, but a minimum value is defined, equal to the width of the supported data types (*byte*, *half-word*, *word*, and *double-word*), therefore set between 8 and 64 *bytes*, restricted to powers of two. Each of these vectors can be associated with a data stream. In addition, sixteen predicate registers are present, named *"p0"* to *"p15"*, although only eight can be used in arithmetic and regular memory instructions (*p0-p7*). Register *p0* is hardwired to 1, which means it can be used in operations where predication is not necessary (i.e. non-conditional loops), as all valid lanes of the operating streams execute. The remaining predicate registers are used in the configuration of the other eight.

There are currently 60 major instructions, out of which 26 correspond to integer operations, 15 to floating-point operations and 19 are related to memory manipulation, totalling about 450 instructions when considering the variations of each one.

Furthermore, UVE not only lets one describe data streams through the ISA, but it also defines the operation of the supporting microarchitecture to manipulate them, consisting in a dedicated *Streaming Engine*, along with other minor structures that extend the processor in order to fully support this ISA extension.

Because *Spike* is a functional simulator based on a somewhat high level of abstraction from the real hardware, the added structures do not fully mimic the proposed microarchitecture, namely the memory hierarchy, pipelining and Load/Store FIFOs, but are implemented to respect the instruction set extension specification.

3 UVE VALIDATION FRAMEWORK

Having proven its great potential [8], UVE will benefit from an efficient tool to validate every aspect of its specification, so that it can be further improved and expanded to support new and more complex applications. The developed framework is hereby described in detail and its structure is represented in Figure 3.

3.1 Simulator

As noted by Roelke and Stan [19], there is usually a compromise between simulation accuracy and speed when choosing between the various RISC-V simulators available. As such, *Spike* was chosen as the most appropriate tool to continue this development. Although it does not allow cycle-accurate precision, it is the golden reference

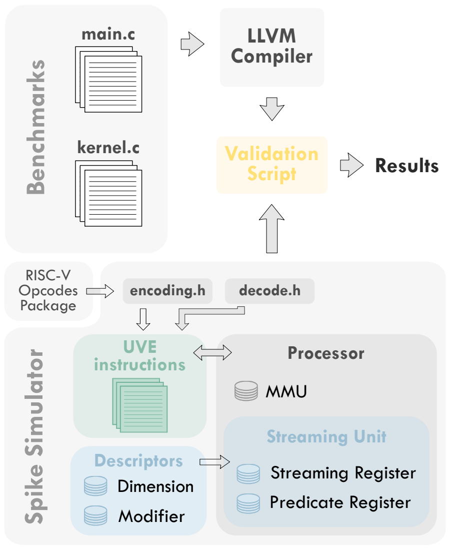


Figure 3: Framework structure.

functional RISC-V ISA software simulator and is widely used as the proof-of-concept target for every RISC-V extension [6, 12]. In fact, despite QEMU appearing to be slightly more accurate [19], it is a much bigger and more complex project, as it targets multiple architectures, not only RISC-V, and is thus more difficult to modify, something that is necessary in order to create UVE support. This is pointed out by Henriques [9], who already used the *Spike* simulator to implement some UVE instructions and whose work laid the foundation for the development of the currently proposed validation framework.

Spike is currently at Version 1.1.0 and already supports many RISC-V ISA features, along which is the RISC-V Vector Extension, which served as a base for the developed Streaming Unit (SU) – the UVE's equivalent to its Vector Unit. However, upon analysing the implementation of several extensions on the simulator, it became clear that UVE's implementation structure would be very different. This is mainly due to the way the simulator's source code is written, heavily dependent on macros defined in multiple files and with little to no documentation. This resulted in code structured in a very different way than the rest of the simulator and its supported extensions, albeit more comprehensible.

3.1.1 Streaming simulation infrastructure. The focal component of the developed simulator is the SU, a new class that has access to the streaming and predicate registers. This unit mimics some parts of the proposed *Streaming Engine* [8], specifically the *Stream Table* and the *Stream Processing Module*, as well as the remaining infrastructure responsible for the memory accesses (see Figure 4). Each register may or may not be associated to a stream, and this module is responsible for the implicit loading and storing of data, as well as the iteration of the streams (by the *Address Generator*). For the desired functional evaluation, the *Load/Store FIFOs* and the *Stream Scheduler*, represented in Figure 4, were not needed, as streams are iterated as they are being consumed, with each computation instruction triggering the iteration of the source streams (implicit loading) and the destination streams (implicit storing). The resulting elements are immediately placed in the associated registers and the *End Of Dimension* flags are updated and saved to the *Stream Table*. The iteration and address generation parts work very similarly to the proposed configuration and are implemented in a different class, *Dimension*, which has access to the *Modifier* class, where static modifiers are implemented. Each streaming register, when associated to a stream, is therefore also associated to n dimensions and respective modifiers, if such is the case.

Furthermore, predication support was developed at the instruction level, which means that the predicate values never reach the SU, for simplicity. A predicate register has a fixed vector size of 64 *bytes*, and a predicate is thus evaluated according to the datatype of the instruction's source operands. As a result, in each predicated instruction the predicate register is read for each active lane, and the operation is only performed if it evaluates to 1, as stated by the ISA specification [8].

3.1.2 Modified files. Several source files were modified to add the necessary structures to support UVE (e.g. decoding functions for each instruction argument), according to the ISA encoding. These functions, divided into different types of instructions, followed the same pattern as already existing ones, some even being direct copies, so that there is complete flexibility in case the UVE encoding is changed. In that case, it is not necessary to alter each instruction if, for example, one of the source registers is differently encoded. It is only required that the decoding function corresponding to its type is updated accordingly.

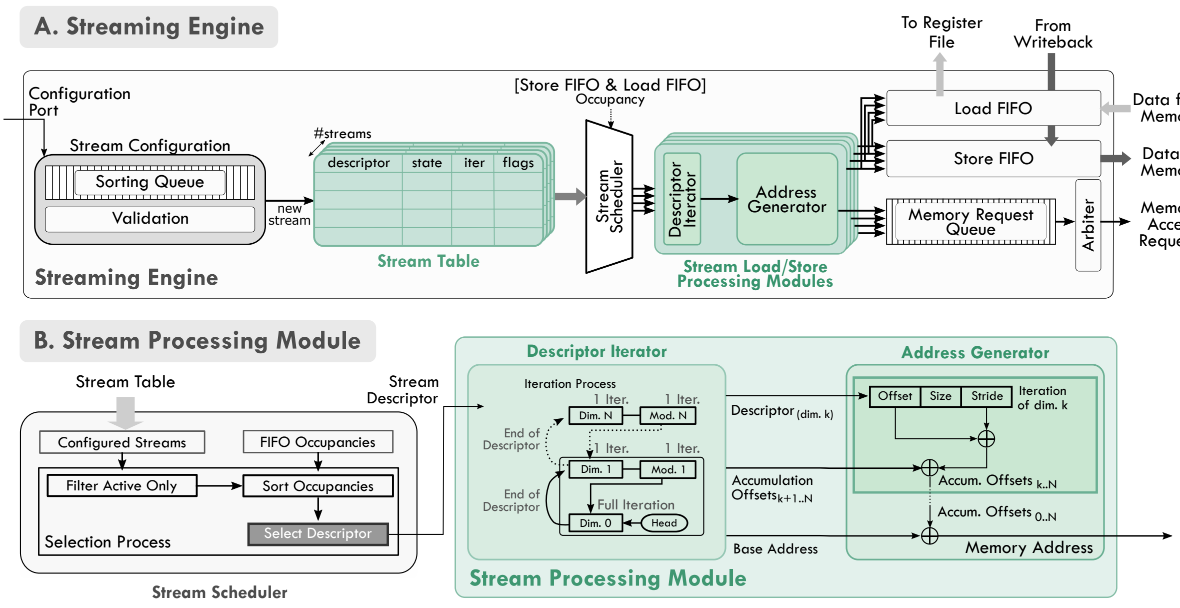
For the simulator to recognise the new instructions, the file that holds all the ISA encoding, `encoding.h`, must be updated. To obtain the necessary code, the official RISC-V Opcodes project [3] was used, where the encoding of each instruction was added to the standard ISA and UVE's predicate registers, and immediate encoding was added to the file `constants.py`.

Lastly, the new extension was added to file `riscv/riscv.mk.in`, identically to what is done to the native ones, so each new instruction was included in the variable `riscv_insn_ext_uve`. In this file every new source and header file was also added to variables `riscv_srcs` and `riscv_install_hdrs`, respectively, so that they could be recognised during the compilation of the simulator.

3.1.3 New files. The various new classes priorly mentioned are defined in files `descriptors.h` (dimensions and modifiers) and `streaming_unit.h` (registers and SU).

Furthermore, each instruction has a corresponding *header* file in the `riscv/insns` folder. While compiling the simulator, these files will be used to create copies of the `riscv/insn_template.cc` file for each instruction, responsible for the generation of the various versions of the instruction (e.g. 32/64 bit). The obvious implication is that the developed code for an instruction exists inside an external function, therefore header file inclusion is not allowed and only some variables are accessible, namely the processor, the executed instruction and the process counter. It is through the processor that each instruction can access the Memory Management Unit (MMU), as well as the SU and its registers. The executed instruction,

[3]Available at https://github.com/riscv/riscv-opcodes



The components which were implemented on the proposed framework are represented in **green**, while the ones in **grey** are implementation specific, and thus not needed to fully describe UVE functional behaviour.

Figure 4: (A) Streaming Engine and (B) Stream Processor Module proposed by Domingos et al. [8], now emulated on *Spike*.

an insn_t object, has access to the opcode decoding functions, allowing the instruction code to access its operands. The process counter is mainly used in branching instructions.

4 EXPERIMENTAL RESULTS

4.1 Methodology

In order to validate the UVE ISA functional simulation, various benchmarks from a wide range of application domains, such as memory access, linear algebra/BLAS and stencil, were chosen. These benchmarks were either hand-coded in order to have its corresponding UVE implementation, inserted in the original source code using *inline assembly*, or even generated by an adaptded version of the LLVM compiler that, while currently unavailable to the public, is undergoing preliminary testing.

Figure 2.C shows how a simple matrix multiplication ($C = A \times B$) can be coded with UVE. In this example, *u1* and *u2* are streaming registers configured with load data streams from matrices A and B and *u3* is associated to the store stream, corresponding to matrix C, which holds the computation results. This example is also part of the kernel used in benchmark *3mm*, already implemented and tested on *Spike* (see Section 4.2 and Figure 5).

4.2 Evaluation

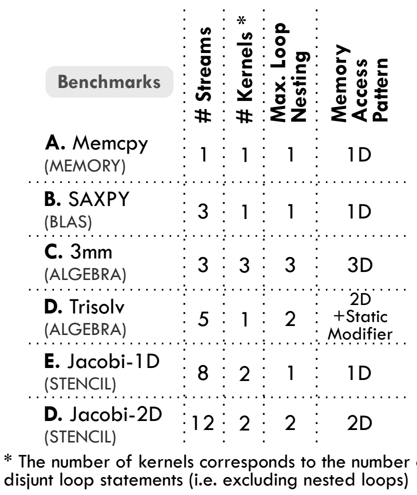
The described framework is currently able of simulating most of the UVE specification, which means that data streaming capability has been successfully added to *Spike*, as well as many instructions from the proposed ISA. It currently supports multi-dimensional pattern descriptors, as well as static modifiers, although indirection is not yet implemented. Stream-based branching and predication are also supported, as well as multiple arithmetic and vector operations on the streaming registers. In total, more than 100 instructions have been implemented and validated on *Spike*, which can be categorised as follows:

- Arithmetic (41)
- Branching (16)
- Predication (15)
- Vector (8)
- Stream Configuration (21)

With these fully functional instructions, several benchmarks can already be ran on the simulator, as summarised in Figure 5. All these benchmarks, which had been previously used for validation of the UVE ISA and supporting microarchitecture on *gem5*, outputted the same expected behaviour in *Spike*, proving the correct functioning of the developed Streaming Unit and the added instructions on the this simulator.

5 RELATED WORK

The main focus of the presented work is the development of a new simulation environment for UVE, where the major difference from previous works is the chosen base tool. In this section, an overview of the legacy *gem5* UVE simulation framework is presented.



Benchmarks	# Streams	# Kernels *	Max. Loop Nesting	Memory Access Pattern
A. Memcpy (MEMORY)	1	1	1	1D
B. SAXPY (BLAS)	3	1	1	1D
C. 3mm (ALGEBRA)	3	3	3	3D
D. Trisolv (ALGEBRA)	5	1	2	2D +Static Modifier
E. Jacobi-1D (STENCIL)	8	2	1	1D
D. Jacobi-2D (STENCIL)	12	2	2	2D

* The number of kernels corresponds to the number of disjunt loop statements (i.e. excluding nested loops)

Figure 5: Benchmarks used for testing and respective characteristics.

gem5 was notoriously used in UVE's proposal by Domingos et al. [8], to provide a reliable performance evaluation with cycle-accurate precision based on the adaption of an out-of-order processor model architecture, as it is extendable to support custom ISAs, as well as microarchitecture models [3, 4, 8, 19].

Despite being an open-source project, *gem5* has little documentation available, similarly to *Spike*. However, it is a much more complex tool, which resulted in a long and difficult process of modifications to support UVE. The source code of the simulator was extensively changed, to support new vectorial and predicate registers, where scalability (e.g. variable vector register length, with the element width as a part of the register) was a major obstacle to overcome. This is due to the simulator not being prepared for this at the ISA level, as well as requiring that the configuration of the architecture is changed at each execution. Furthermore, the instruction set parser was modified to support the new vector registers, as well as the width and valid index register information [8]. Lastly, UVE instructions were added, described in a Domain-Specific Language (DSL) based on C++ and Python, which, on the one hand, allowed templating for multiple instructions and code reuse, but on the other hand required many different templates to be developed, such as for instructions with different operands, which are extremely common.

Because this simulator relied on the implementation of the supporting microarchitecture for the validation of UVE, as it depends on a *Streaming Engine*, the instruction set had not been simulated independently until now. The proposed *Spike*-based framework made it possible to focus solely on the instruction's behaviour, detaching the ISA development from implementation details prone to specification errors.

Lastly, simulation platforms alternative to *Spike* exist, such as QEMU[4] and Chisel[5]. While the former is closely related to *Spike*, it is a more complex project, and therefore lacks the simplicity required for an efficient and continuously changing framework. The latter could be used to create an RTL simulation. While this was not the goal of this work, it could be useful, as the proposed framework exists within a bigger project on UVE currently under development, where this type of validation is appropriate.

[4]https://www.qemu.org/
[5]https://www.chisel-lang.org/

6 CONCLUSION

In this paper, a new validation framework for the UVE ISA extension based on the *Spike* RISC-V simulator is presented. This new simulation tool provides efficient development means and functional evaluation for this ISA extension and of its supporting microarchitecture. A Streaming Unit, responsible for the management of the streams, and most of the existing UVE instructions were added to *Spike*, also including support and ensuring validation for the main functionalities already offered by UVE, such as data streaming with implicit loads/stores, predication, and n-dimensional pattern description with static modifiers. A representative set of benchmarks was tested and verified, confirming the previous results that had been obtained with a *gem5* legacy UVE simulator. Furthermore, this tool was also used to validate the preliminary results from the UVE-LLVM compiler that is currently under development, and has proven to be useful both in the development of UVE applications and in the development of the UVE extension itself.

6.1 Future Work

Although the most part of the instruction set has already been added to the framework, some instructions of the existing ISA are yet to be implemented, namely logical and stream configuration instruction. The latter have to be accompanied by the implementation of indirection, which is not yet supported.

In addition, the *Spike* simulator's disassembler still has no information about the added extension, which makes the use of the *debugger* less straightforward. This can be improved in the future, making the developed tool much more useful by improving code review and correction.

Lastly, the LLVM compiler toolchain currently under development is to be tested on this framework. Once it is released, it will integrate this project, removing the need to hand-code applications where UVE is to be used. This will allow anyone to develop software taking advantage of this new ISA extension.

ACKNOWLEDGMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects UIDB/50008/2020, UIDB/50021/2020, EXPL/EEI-HAC/1511/2021, 2022.06780.PTDC, 2022.11626.BD, and from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 and Specific Grant Agreement No 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland.

REFERENCES

[1] Arm. 2011. *Introducing NEON Development Article.* https://developer.arm.com/documentation/dht0002/a/

[2] Adrian Barredo, Juan M. Cebrian, Miquel Moreto, Marc Casas, and Mateo Valero. 2020. Improving Predication Efficiency through Compaction/Restoration of SIMD Instructions. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, San Diego, CA, USA, 717–728. https://doi.org/10.1109/HPCA47549.2020.00064

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[4] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. 2006. The M5 Simulator: Modeling Networked Systems. *IEEE Micro* 26, 4 (2006), 52–60. https://doi.org/10.1109/MM.2006.82

[5] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18).* Association for Computing Machinery, New York, NY, USA, 316–331. https://doi.org/10.1145/3173162.3173177

[6] Chipyard. 2019. *The RISC-V ISA Simulator (Spike) - Chipyard 1.8.1 documentation.* https://chipyard.readthedocs.io/en/latest/Software/Spike.html

[7] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. 2003. The Reconfigurable Streaming Vector Processor (RSVPTM). (2003).

[8] Joao Mário Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. 2021. Unlimited Vector Extension with Data Streaming Support. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* IEEE, Valencia, Spain, 209–222. https://doi.org/10.1109/ISCA52012.2021.00025

[9] Luís Henriques. 2022. *Automatic Streaming for RISC-V via Source-to-Source Compilation.* Master's thesis. Universidade do Porto, Porto. https://hdl.handle.net/10216/142750

[10] Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob Van Der Wijngaart. 2011. *The Case for Message Passing on Many-Core Chips.* Springer New York, New York, NY, 115–123. https://doi.org/10.1007/978-1-4419-6460-1_5

[11] Chris Lomont. 2009. *Introduction to Intel® Advanced Vector Extensions.* www.obpm.org/download/Intro_to_Intel_AVX.pdf

[12] Christoph Müllner. 2021. *Emulators and Simulators - RISC-V International.* https://wiki.riscv.org/display/HOME/Emulators+and+Simulators#EmulatorsandSimulators-Spike/riscv-isa-sim

[13] Nuno Neves, João Mário Domingos, Nuno Roma, Pedro Tomás, and Gabriel Falcao. 2022. Compiling for Vector Extensions With Stream-Based Specialization. *IEEE Micro* 42, 5 (9 2022), 49–58. https://doi.org/10.1109/MM.2022.3173405

[14] Nuno Neves, Pedro Tomás, and Nuno Roma. 2015. Efficient data-stream management for shared-memory many-core systems. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL).* 1–8. https://doi.org/10.1109/FPL.2015.7293960

[15] Nuno Neves, Pedro Tomás, and Nuno Roma. 2017. Adaptive In-Cache Streaming for Efficient Data Management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 7 (7 2017), 2130–143. https://doi.org/10.1109/TVLSI.2017.2671405

[16] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 416–429. https://doi.org/10.1145/3140659.3080255

[17] Angela Pohl, Mirko Greese, Biagio Cosenza, and Ben Juurlink. 2019. A Performance Analysis of Vector Length Agnostic Code. In *2019 International Conference on High Performance Computing & Simulation (HPCS).* IEEE, Dublin, Ireland, 159–164. https://doi.org/10.1109/HPCS48598.2019.9188238

[18] RISC-V. 2021. *Spike RISC-V ISA Simulator.* https://github.com/riscv-software-src/riscv-isa-sim

[19] Alec Roelke and Mircea R Stan. 2017. RISC5: Implementing the RISC-V ISA in gem5.

[20] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2021. Indirection Stream Semantic Register Architecture for Efficient Sparse-Dense Linear Algebra. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE).* 1787–1792. https://doi.org/10.23919/DATE51398.2021.9474230

[21] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. 2021. Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores. *IEEE Trans. Comput.* 70, 2 (2021), 212–227. https://doi.org/10.1109/TC.2020.2987314

[22] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (3 2017), 26–39. https://doi.org/10.1109/MM.2017.35 arXiv:1803.06185 [cs].

[23] Zhengrong Wang and Tony Nowatzki. 2019. Stream-Based Memory Access Specialization for General Purpose Processors. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19).* Association for Computing Machinery, New York, NY, USA, 736–749. https://doi.org/10.1145/3307650.3322229

[24] A. Waterman and K. Asanovic. 2021. *RISC-V "V" Vector Extension.* https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc