POLITECNICO DI TORINO

Master's Degree in Electronic Engeneering

Master's Degree Thesis

Design of an edge-oriented vector accelerator based on the RISC-V "V" extension

Supervisors
Prof. Maurizio MARTINA
Prof. Guido MASERA
Dott. Michele CAON
Dott. Walid WALID

Candidate
Francesca SICA

October 2022

Abstract

In the last decade, the ever-increasing diffusion of machine learning algorithms for digital signal processing has drastically changed the hardware processing requirements for edge devices. These systems have to manage a large amount of data while often still responding to some events in real-time.
To elaborate the information acquired by these systems, a suitable paradigm can be *edge computing*: instead of directly sending raw data to remote central servers, this can be partially processed close to where it is collected so that a smaller amount of elaborated data is sent to central systems, reducing the response time, the network-overloading and the overall power budget required.
To manage such a high quantity of data, it is important to choose efficient architectures exploiting *parallel computing*: *vector processors* have demonstrated to be a promising solution. Among the advantageous aspects, there is the reduction of the overhead caused by instruction fetch from memory (Von Neumann Bottleneck), which is typical of scalar processors when dealing with data-driven workloads: a single vector instruction can be used to process a very large vector. Moreover, vector processors are characterized by high flexibility since they are programmable: to change functionalities for different application targets, it is sufficient to modify and recompile the code. This cannot be done on custom hardware accelerators that arise for a single specific application. Certainly, the versatility of vector processors inherently brings some area and power consumption overhead.
Considering the most recent ISAs, some allow having "hardware-agnostic" software, such that the same code can run on vector architectures with different parallelism. Among these, RISC-V is one of the most promising: its vector extension ("V") allows to evince the physical length of vectors and elements at run time. So, the same code can be used on processors featuring different vector register sizes, allowing high performance and great versatility in different application domains.
In this thesis, a scalable and highly configurable vector processor based on the RISC-V "V" extension is designed and implemented. Most of its components (i.e., vector register file and processing elements) is made up of a set of identical lanes, each processing different elements of a vector independently from the others. The advantage of this structure is that, depending on the application and power consumption target, priority can be either given to performance using a higher number of lanes and arithmetic operators, or to the area and power having a less performing yet smaller processor.
The performance of the processor is evaluated on a representative workload of machine learning algorithms: matrix convolution is used as case study, considering a 4x4 matrix and a 2x2 filter with 32-bit elements. With a 256-bit vector register and 2 lanes, results show a latency of 412 clock cycles to terminate the program, with a throughput of 128 bits/cycle; from synthesis results, the total cell area is about $1\,137\,755\,\mu m^2$, and the clock frequency is 510 MHz. The same code is run with different hardware configuration: as expected, throughput and area scale almost linearly with the number of lanes.
In conclusion, the designed vector processor is potentially very versatile: it implements a subset of standard instructions used in most use cases; then, it is scalable and highly configurable, being able to choose the number of resources and consequently optimizing the throughput.

Acknowledgements

Vorrei ringraziare i professori Maurizio Martina e Guido Masera per avermi dato la grande opportunità di lavorare a questo progetto.

Un grazie speciale va a Michele, per la disponibilità ed il supporto costante che mi hanno permesso di portare a termine il progetto nel migliore dei modi.

Un grande ringraziamento va a tutta la mia famiglia e a i miei amici per avermi sempre sostenuto ed aiutato, dandomi preziosi consigli e stando al mio fianco in questi anni impegnativi.

Table of Contents

List of Tables v

List of Figures vi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Thesis organization . . . 4

2 Background 5
- 2.1 Data Level Parallelism . . . 5
- 2.2 Architectural models . . . 6
  - 2.2.1 Packed-SIMD architecture . . . 6
  - 2.2.2 Vector processors . . . 8
- 2.3 Related works . . . 9

3 Design and Implementation 13
- 3.1 Top level architecture . . . 13
  - 3.1.1 Design parameters . . . 17
  - 3.1.2 Design choices . . . 20
- 3.2 Instruction Decoder . . . 25
- 3.3 Instruction Status Table . . . 29
- 3.4 Scheduling Logic . . . 33
  - 3.4.1 Hazard Detection Unit . . . 34
  - 3.4.2 Operand Requester . . . 38
- 3.5 Vector Register File . . . 52
- 3.6 Source and Destination Tables . . . 58
  - 3.6.1 Source Table . . . 60
  - 3.6.2 Destination Table . . . 60
  - 3.6.3 Position Index Table . . . 63
  - 3.6.4 Source and Destination Tables Counters . . . 63
- 3.7 Processing Element . . . 65

3.7.1 Arithmetic unit 67
3.7.2 Reduction unit 72
3.8 Load-Store Unit 74
3.8.1 Load-store emulator 75
3.8.2 Load-store unit preliminary design 79

4 Testing and synthesis 87
4.1 Testing methodology 87
4.2 Case study 89
4.3 Testing and synthesis results 90
4.3.1 Baseline 91
4.3.2 Results comparison by changing parameters 93

5 Conclusion 101
5.1 Future improvements 102

A Matrix convolution pseudo-code 105

Acronyms 107

Bibliography 109

List of Tables

4.1 Results with `VLEN` = 64 bits and `NUM_LANE` = 1. 91
4.2 Results from the simulations. For each combination, the latency (expressed in clock cycles) and throughput (expressed in bit/cycle) are reported. 95
4.3 Results from the synthesis. For each combination, the area and cycle time (T_{ck}) are reported. 95

List of Figures

2.1 Vector processor block scheme. The red box highlights the operations that can be executed simultaneously and the *chaining* implementation. 9
2.2 Packed-SIMD architecture block scheme. The red box highlights the operations that can be executed simultaneously. 9

3.1 Top-level view of the vector processing unit 16
3.2 *instruction decoder* block diagram 26
3.3 RISC-V vector extension instruction format (taken from the RISC-V "V" specifications [9]) 27
3.4 *instruction status table* block diagram 30
3.5 *scheduling logic* block diagram 33
3.6 *hazard detection unit* block diagram 34
3.7 Output generator of the *processing element status table* 37
3.8 *operand requester* block diagram 39
3.9 *activation instruction buffer CU* flow chart 40
3.10 Example of how the register sections to read are distributed in the lanes 41
3.11 *arithmetic instruction buffer* block diagram 43
3.12 Data exchange between *arithmetic instructions buffers* for widening operations 45
3.13 *arithmetic instruction buffer* CU flow chart 47
3.14 *load-store instruction buffer* block diagram 49
3.15 Example of *vector register file* configuration with two banks for each lane 52
3.16 Example of *vector register file* configuration, starting from $\texttt{EEW} = \texttt{SEW}$ and arriving to $\texttt{EEW} = 2 \cdot \texttt{SEW}$. 54
3.17 *vector register file* lane block diagram 56
3.18 *source table* and *destination table* allocation 58
3.19 *source table* and *destination table* block diagram 59
3.20 *processing element* block diagram 66
3.21 *arithmetic unit* block diagram 68

3.22 ALU block diagram 70
3.23 Multiplier and divider block diagram 71
3.24 *reduction unit* block diagram 73
3.25 Example of strided load-store operation 74
3.26 Flow chart of the control unit inside the *load-store unit* 78
3.27 *load-store unit* block diagram 80
3.28 L1 Data Cache arbiter 82
3.29 Load strided unit block diagram 85

4.1 Matrix convolution example with 16x16 input matrix, an 8x8 filter and a 9x9 resulting matrix. 90
4.2 Area composition considering `NUM_LANE` = 1 and `VLEN` = 64 bits. . . 92
4.3 Surface chart showing the trend of the area as a function of `VLEN` and `NUM_LANE`. 96
4.4 Surface chart showing the trend of the cycle time as a function of `VLEN` and `NUM_LANE`. 96
4.5 Surface chart showing the trend of the throughput as a function of `VLEN` and `NUM_LANE`. 97
4.6 Trend of area and throughput by varying `NUM_LANE` while fixing `VLEN` = 1024 bits. 98
4.7 Trend of area and throughput varying `VLEN` and fixing `NUM_LANE` = 2. 99

Chapter 1

Introduction

The aim of the thesis project is to design and implement a scalable and highly configurable edge-oriented accelerator based on the RISC-V vector extension (V). It works as a co-processor intended to be integrated inside the *LEN5* core, a modular 64-bit out-of-order (OoO) RISC-V core developed at Politecnico di Torino. The code of the vector processing unit is available on GitLab [1].

1.1 Motivation

Nowadays machine learning [2] and digital signal processing [3] are widespread in different Artificial Intelligence applications, like image and speech recognition, robotics, gameplay and medicine.
These systems must handle a large amount of data while often still responding to some events in real time (for example in applications such as autonomous vehicles, drone navigation and robotics). The raw data collected by these systems can be elaborated in two ways: the first one considers to send data to remote central servers (usually far from the data source), such that the latter can process it returning subsequently the response to the system (*cloud computing paradigm*); the second one considers to partially elaborate data close where it is collected so that a smaller amount of processed data is sent to central servers (*edge computing paradigm*).
Considering data-driven systems, the cloud computing paradigm can lead to high latency and network-overloading problems due to the transmission delay of the network and to the massive quantity of data to be managed. Moreover, taking into account the large distance between the system and the remote central servers, this does not represent an optimal solution from the energy efficiency point of view, since the overall power required to transfer such a high quantity of data over long distances is quite large.

A better solution can be the edge computing paradigm, which can overcome some limits of cloud computing [4]. Since data is partially elaborated close to the system, only a smaller quantity of data is sent to remote central servers allowing to minimize the bandwidth of the network; then, the shorter distances lead to a faster response time (by decreasing the transmission delay) and a reduction of the overall power budget required to transfer data.

In order to manage a high quantity of data, it is important to choose efficient solutions exploiting *parallel computing*.
In 1974 Robert Dennard observed that power density was constant as transistors got smaller [5]. Thus chips could be designed to operate faster and still use less power. However, Dennard scaling ended 30 years after it was observed, not because transistors didn't continue to get smaller but because the threshold voltage decreased so much that static power became a significant part of overall power. Processor frequencies reach a saturation point, leading to an increasing interest in parallel multi-core architectures.
With the MIMD paradigm, instead of a single processor, multiple efficient cores are used, which fetch their own instructions and operate on their own data independently from the others. However, multi-core architectures fail to exploit the regularity of data-parallel applications, as cores execute copies of the same instructions across multiple data elements and the fetch mechanism is one of the most expensive components of processor due to the Von Neumann Bottleneck (VNB) [6]. This term describes the disparity between the speed of computation and the speed of memory access in processors designed using the Von Neumann architecture. The imbalance arises because the speed at which a CPU can perform computation is significantly higher than the speed of memory access. So MIMD paradigm is not the best solution for data-parallel applications. Moreover, MIMD architectures in general can achieve high performance by having multiple cores but they have higher costs in terms of area and power consumption and this is not in line with an edge-oriented solution, where the architecture needs to have a reduced area and power budget to be almost integrated into the system. So, also considering that the multiple cores would execute copies of the same instructions, having multiple fetch units would be a waste of area and energy.
A way to relax the VNB is to exploit the SIMD paradigm which shares the fetch operation among multiple processing units, so instructions operate on vectors of operands. This solution has a very low cost in terms of implementation, then it is potentially energy-efficient since a single instruction can launch many data operations and, finally, it can achieve a smaller area having only one fetch unit for all processing units. The problem introduced by this type of architecture is that it requires dedicated instructions for each combination of element and array size, as well as specific code and compiler optimizations to properly store and load the

data to be processed. As a consequence, hundreds of instructions should be added to the ISA and decoded by the microarchitecture, while a significant overhead in terms of code size is paid at programming or compilation time, often significantly reducing the effectiveness of this approach.
In order to overcome the problems mentioned above and satisfy requirements like high performance and energy efficiency, the right solution to manage data-parallel applications can be the use of *vector processors* [7]. In this architecture, a single vector instruction can be used to process a very large vector, thereby amortizing the instruction fetch overhead. Moreover, vector processors are characterized by high flexibility since they are programmable: to change functionalities for different application targets, it is sufficient to modify and recompile the code. This cannot be done on custom hardware accelerators that arise for a single specific application. Certainly, the versatility of vector processors inherently brings some area and power consumption overhead.

Along with the processor architecture, it is important to select a suitable Instruction Set Architecture (ISA). RISC-V is an ISA born in UC Berkley in 2011 [8] which is gaining popularity over the years both in academic and industrial fields due to some key points that differentiate it from ISA of other processor architectures (e.i. Intel's x86 and ARM ISA). An important aspect is that it is a completely open-source ISA, allowing smaller developers and manufacturers to design and build hardware without the cost of licensing proprietary ISA and paying royalties. This feature has also gained interest in the academic world.
Even though RISC-V is thought for Von Neumann architectures which consequently suffer from the VNB, it features a modular design where, starting from a base ISA, different extensions (among which there is the vector "V" one) can be added in order to accelerate computationally expensive tasks and provide better support for operating systems. This structure makes RISC-V suitable for different kinds of applications ranging from simple, low-power microcontrollers to application-specific processors and high-performance computers.
Another important aspect that makes RISC-V one of the most promising ISA is that it allows having "hardware-agnostic" software, such that the same code can run on vector architectures with different parallelism. Its vector extension ("V") allows to evince the physical length of vectors and elements at run time so the same code can be used on processors featuring different vector register sizes, allowing high performance and great versatility in different application domains.

Taking into account all the previous considerations, it was decided to start a thesis project in order to design an edge-oriented accelerator based on the RISC-V vector extension (RV64V).
The version 1.0 of the specifications [9] was followed: it has been frozen for public review so it is stable enough to begin implementations since it is not expected to

have incompatible changes in future versions.
Starting from the basic motivations, the intent of this work was to perform a horizontal design of the processor architecture starting from scratch, such that it was able to process a subset of the RV64V ISA including instructions used in most of the use cases. So a first working architecture was implemented without optimizing it internally, but this is a good starting point for future work.

1.2 Thesis organization

The thesis work is organized as follows:

- This is the *first* introductory *chapter* including the motivations behind the project.
- The *second chapter* is focused on giving general information about Data Level Parallelism (DLP), looking at different architectures exploiting it.
  Then, a summary of existing vector processor architectures already implemented in the past is presented.
- The *third chapter* is about the object of the thesis, namely the design and implementation of the vector processing unit, explaining the structure and the behavior of each module inside it.
- The *fourth chapter* is focused on the simulation and synthesis results in order to evaluate the performance. For this purpose, the matrix convolution is used as case study, where the same code is run with different implementation parameters in order to compare performance in terms of area and throughput.
- In the *fifth chapter* the final conclusions are presented, together with some further improvements for the future.

Chapter 2

Background

2.1 Data Level Parallelism

In order to manage high quantity of data in data-driven systems, it is necessary to exploit Data Level Parallelism (DLP). With this approach, a processor architecture can operate on multiple elements of data simultaneously.
Among the class of processors working with a DLP approach, there are the MIMD and SIMD architectures. The first one, exploited for example by GPUs, provides multiple efficient cores which fetch their own instructions and operate on their own data independently from the others. This type of architectures is out of the scope of this project as it certainly achieves high performance but it has higher costs in terms of area and power and this is not in line with the goal of having an edge-oriented processor. Also, as mentioned in the motivation section 1.1, MIMD architectures may also suffer from VNB when each core executes a different instruction stream and therefore it has to perform the expensive instruction fetch from memory.
The SIMD architectures, on the other hand, overcome the VNB since they are divided in multiple identical processing units, each of them sharing the instruction fetch and working independently from the others. So all processing units execute the same instruction, each on different data with the advantage of processing a high quantity of data in parallel while accessing to the memory only one time for the instruction fetch. Then, they are potentially energy-efficient since a single instruction can launch many data operations and they can achieve smaller area having only one fecth unit for all processing units.

Going into more details in order to see the advatages of DLP, we can define the performance of a processor as

$$Performance = \frac{1}{CPU\ time} \tag{2.1}$$

where the CPU time is the time required by the processor to complete a given task, not including the time waiting for I/O or running other programs [5].
Then, the CPU time is defined as

$$CPU\ time = N \cdot CPI \cdot T_{ck} \tag{2.2}$$

where

- N is the number of instructions executed in the task.
- *CPI* (Cycles Per Instruction) is the average number of clock cycles required to execute a single instruction.
- T_{ck} is the clock period.

The DLP paradigm allows reducing the N factor since a single instruction can process a large number of elements. However, this can increase the CPI, since a single instruction can take more cycles to complete. Although the reduction of the N factor comes with an increase in the CPI term of the equation, the parallelism implicit in each vector instruction can drastically reduce the CPI factor, simply by adding parallel functional units that process the instruction.
In this way, by reducing the number of instructions and incrementing the number of processing units, the CPU time can be drastically reduced improving the performance.

2.2 Architectural models

In this section different SIMD processor architectural models exploiting DLP are briefly presented.
As already mentioned, this model expects to have a processor divided in multiple identical processing units, each of them executing a single instruction on multiple data in parallel while sharing a single instruction memory and control unit.

2.2.1 Packed-SIMD architecture

This type of processor architecture has several independent and identical processing units, each of them handling a subset of a data vector. So the number of processing

units determines the vector length (a block scheme of the packed-SIMD architecture can be seen in figure 2.2).
The advantage of this architecture is that it requires low costs in terms of implementation but the drawback is that the vector length is fixed by establishing the number of processing units. Since the vector length is commonly encoded in the instruction, this model requires dedicated instructions for different vector size. As a consequence, hundreds of instructions should be added to the ISA and decoded by the microarchitecture, while a significant overhead in terms of code size is paid at programming or compilation time. Certainly this consequence places limits on the maximum vector size so as not to be forced to add a very large amount of instructions.
Some examples of existing solutions requiring different extensions depending on the vector length include Intel's AVX [10], AVX2 [11] and AVX-512 [12] extensions, where the last was used in Intel Xeon Phi product family x200 based on the Knights Landing or Knights Mill architectures. They are used for supercomputers and servers but also for machine learning applications, image elaboration and data compression. The main difference between the AVX/AVX2 and AVX-512 extensions is the vector length: in the former case, it is 256-bit wide, while in the last one it is 512-bit wide, leading to a change of the architecture parallelism (registers widths, functional units parallelism exc.). Moreover, in the AVX-512 extension, the number of registers is doubled and more functionalities are supported, like embedded masking, embedded floating-point rounding control and fault suppression, scatter instructions and high speed math instructions. In all extensions the vector length is fixed either to 256 bits or to 512 bits, while the vector's element length can be configured (for example, if the vector is 256-bits wide, it can contain thirty-two 8-bit elements, sixteen 16-bit elements, eight 32-bit elements or four 64-bit elements).
Another example of packed-SIMD extension is Neon [13], which is produced by ARM and implemented in Cortex-A and Cortex-R series of processors in order to improve use cases on mobile devices, such as multimedia encoding/decoding, user interface and 2D/3D graphics and gaming. Neon registers are 128-bits wide and they are considered as vectors of elements that are processed simultaneously: each register can contain two 64-bit, four 32-bit, eight 16-bit or sixteen 8-bit integer data elements since multiple data types are supported by the technology.
The main limitation of these solutions is given by the fixed length of the vectors, so that, from one extension to the other, new instructions are introduced since the vector length is encoded in the instruction itself; then, the code to be run on the processor must be changed and re-compiled.

2.2.2 Vector processors

Vector processors, like packed-SIMD architectures, provide a single instruction working on vectors of data. They have vector functional units with multiple parallel pipelines (or *lanes*) that process simultaneously different elements of vectors (a block scheme of the vector architecture can be seen in figure 2.1).

Differently from the packed-SIMD architecture, with vector instructions we abstract away how many lanes we have from the number of elements in the vector. In particular, one of the common approaches to manage vector elements is "stripmining": usually, considering an application, the size of the vector to be processed is greater then the physical parallelism of the processor; then multiple iterations are performed, each processing a subset of the original vector, until all elements of the latter have been handled. For example, if a vector is made up by 16 elements and there are only two lanes, a vector processor will simply cycle through all the elements until all of them are processed. So, it performs eight iterations, each of them processing two elements (one for each lane). Considering the same example, in a a packed-SIMD architecture, the number of lanes determines the vector size so that only vectors of 2 elements would be allowed (each lane would manage one element).
So, the advantage of vector processors is that the vector length and the elements length can be *dynamically configured*. Then the same code can be used featuring different number of lanes, allowing a great versatility in different application domains, from simple microcontrollers to high-performance computers.

The fact that the element width can be configured at run time implies that arithmetic units able to perform sub-word parallel computations are the most convenient choice for this architecture. This means that, giving a fixed parallelism of the arithmetic unit, it can process in parallel multiple elements with a parallelism that is smaller than the maximum one.
An example is shown in [14], where a 32-bit arithmetic unit can operate on four 8-bit integer numbers, two 16-bit integer numbers or one 32-bit integer number. This is possible because the datapath is made up by four 8-bit Processing Elements. Each PE accepts two 8-bit operands and outputs a 16-bit result. For the additions/subtractions, two PEs are carry-linked to perform 16-bit operations and four PEs are carry-linked to perform 32-bit operations. For the multiplications, larger data level multiplications are decomposed into small data level multiplications and then the partial products are added together by a 64-bit partitionable adder made up by two 32-bit Kogge-Stone adders.

In order to further improve the performance of a vector processor, *chaining* can be implemented. It is like data forwarding for element-dependent operations from

one vector functional unit to another. It allows a vector operation to start as soon as the individual elements of its vector source operand become available, without waiting for the update of the entire vector: the results from the first functional unit in the chain are "forwarded" to the second functional unit and so on. An example of chaining can be seen in the figure 2.1: in the same clock cycle (red box), while element 2 is loaded, the element 1 (loaded on previous clock cycle) is multiplied and the element 0 (loaded two clock cycles ago, multiplied on the previous clock cycle) is added.
This technique cannot be applied in packed-SIMD processors as it can been seen in figure 2.2: it cannot start the multiplication until the load is done *for all the elements* of the vector.

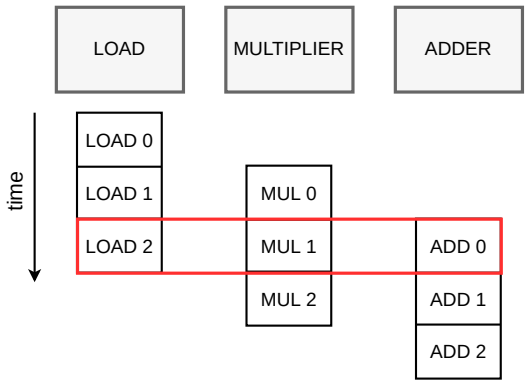

Figure 2.1: Vector processor block scheme. The red box highlights the operations that can be executed simultaneously and the *chaining* implementation.

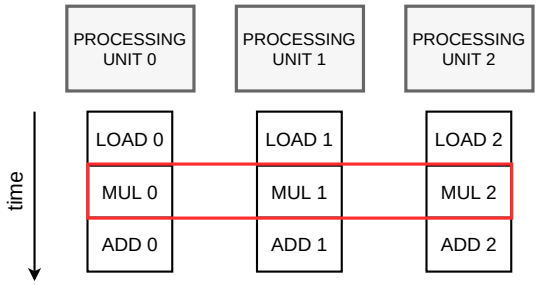

Figure 2.2: Packed-SIMD architecture block scheme. The red box highlights the operations that can be executed simultaneously.

2.3 Related works

In order to understand the context where the thesis project can be placed, in this section different existing solutions of processor architectures implementing the vector extension are presented.

- ARM developed two vector extensions: SVE [15] and SVE2 [16], introduced respectively in Armv8.2-A and Armv9-A architectures [17].
  They do not define the size of the vector registers, but constrains it to a range of possible values, from a minimum of 128 bits up to a maximum of 2048, with the constraint of being a power of two and a multiple of 128 bits. Therefore,

by choosing the vector register size, these extensions can be implemented in architectures belonging to different application domains. A 512-bit SVE variant has already been implemented on the world's fastest supercomputer of 2020 Fugaku using the Fujitsu A64FX ARM processor, born from the collaboration of Fujitsu with ARM.
A second key point is that it is also possible to configure the length of the vector, provided it is a multiple of 128 bits and does not exceed the architectural maximum of 2048 bits. The vector length can be configured dynamically by writing some vector control registers and each vector can hold 64, 32, 16, and 8-bit elements. The main advantage is that SVE/SVE2 extensions guarantees that the same code can run on different implementations that support SVE/SVE2, without the need to recompile the code.
The difference between SVE and SVE2 is that SVE is designed for high-performace computing and machine learning applications; SVE2 extends the SVE instruction set to accelerate the common algorithms that are used in applications like computer vision, multimedia, genomics and general-purpose software.

- RISC-V features a vector extension providing that the parallelism of vector registers can be chosen from a minimum value of 32 bits to a maximum value of 2^{16}, with the constraint of being a power of 2. Then, depending on the application domain in which the processor will work, a different register size can be chosen. Since the range of the possible parallelism is very large, this makes this extension versatile for very different purposes.
  An important aspect of the RISC-V extension is that the vector and element width can be dynamically configured by writing some control registers. The elements can have four different sizes: 8 bits, 16 bits, 32 bits or 64 bits; the vector length is very flexible since it can *exceed the vector register size*, thus leading to consider groups of registers as a single vector.
  The RISC-V vector extension has been implemented on Ara [18], a 64-bit vector processor based on the version 0.5 draft of RISC-V 's vector extension (V). It works as a co-processor for Ariane, an open-source, in-order, single-issue, 64-bit application-class processor implementing RV64GC. It is designed for high-performance computing, aiming to reach high clock frequencies, maximize the use of functional units while maintaining good energy efficiency.
  Ara is a scalable architecture, since it is made up by a set of identical lanes, each hosting part of the vector register file and functional units. In this way, each lane processes different elements of a vector independently from the others. There are other units (like the load/store unit) that communicate with all lanes, since they execute instructions needing all elements of vectors. These units represent the weak points when it comes to scalability, because

they get wider when the vector length increases and therefore as the number of lanes increases.
Talking about performance, Ara achieves up to 97% FPU utilization when running a 256 × 256 double precision matrix multiplication on sixteen lanes. It runs at more than 1GHz, achieving a performance up to 33DP-GFLOPS. In terms of energy efficiency, Ara achieves up to 41DP-GFLOPS/W under the same conditions, which is slightly superior to other vector processors.

Chapter 3

Design and Implementation

In this chapter, the design of the *LEN5* vector processing unit is presented.
It is a scalable and configurable co-processor implementing a subset of the RISC-V vector extension (RV64V); the different categories of supported instructions are reported in section 3.1.2.
The hardware description language used to describe the processor architecture is SystemVerilog: it is suitable to model and synthesize digital systems, allowing a higher level of abstraction and consequently a more compact code. In addition, SystemVerilog supports critical features for design verification, including SystemVerilog Assertions (SVA), thus making this language suitable for both design and verification (further details about the testing methodology are given in section 4.1).

3.1 Top level architecture

The vector processor is characterized by a scalable architecture: most of its components are made up of a set of identical *lanes*, each processing different elements of a vector independently from the others. In particular, both the *vector register file* and the *processing element* are divided into lanes together with the structures to support them. In this way, each lane in the *vector register file* contains a portion of a vector and the latter will be processed by the corresponding lane inside the *processing element*. The advantage of this structure is that, even for long vectors, high performance can be achieved since a lot of elements can be simultaneously processed in parallel.
For example, if a vector is made up of 16 elements and there are four lanes, lane 0 in the *vector register file* contains the first four elements of the vector; these are processed by lane 0 of the *processing element*. At the same time, lane 1 will process the second four elements of the vector which are stored in lane 1 of the *vector register file* and so on for the other lanes.

On the other side, there are other units (like the *load-store unit* or the *reduction unit*) that execute instructions needing all elements of vectors so they require data coming from all lanes of the *vector register file*. Surely, since they cannot perform parallel operations, their speed is lower than the one of the components exploiting the parallelism of lanes.

Another important aspect is the configurability: many implementation parameters can be set, from the number of lanes and physical vector register size to the number of processing elements and arithmetic operators.
Moreover, the vector and element width can be dynamically configured by writing some control registers. Elements can have four different sizes: 8 bits, 16 bits, 32 bits or 64 bits, while the vector length can be set to be lower, equal or even greater than the physical size of a vector register.
A detailed explanation of the parameters configuration is in the section 3.1.1.

The vector processing unit uses the same handshake protocol of the scalar core (*LEN5*). It is an AXI-like protocol (without being AXI-compliant), based on *valid* and *ready* signals. This protocol allows having a modular design supporting variable latency components. Further details are reported in section 3.1.2.

In this paragraph, a brief overview of all components present in the vector processing unit is exposed. Then, from section 3.2, a deeper description along with some block schemes is reported for each module.

- **Instruction Decoder**: It receives the vector instruction from the scalar core along with the scalar operands coming from the scalar register file (if needed by the vector instruction); it decomposes the instruction in different fields depending on its type and sends it to the *instruction status table*.

- **Instruction Status Table**: It is a FIFO that keeps track of the status of instructions under execution. It receives the instructions from the *instruction decoder* and it sends them to the *hazard detection unit* in order to verify if they can be executed without hazards; if this happens, then the *instruction status table* sends them to the *scheduling logic* for the operands fetch.
  When an instruction is completed, it is removed from the *instruction status table*.

- **Scheduling Logic**: It is made up of two units, the *hazard detection unit* and the *operand requester*. The first one is in charge of checking for data and structural hazards related to the instruction needing to be executed; the second one performs the operands fetch from the *vector register file* for instructions that passed the hazard check. The *operand requester* is divided into lanes, such that each lane makes a request for the data stored in the

corresponding lane of the *vector register file*. Once the operands are ready, if the instruction is arithmetic, each lane of the *operand requester* sends the data to the corresponding *source table* and it allocates one entry in the corresponding *destination table*; for the load-store instructions, the *operand requester* sends the data directly to the *load-store unit*.

- **Vector Register File**: It contains 32 registers (v_0 - v_{31}). The parallelism of the registers is parametric and it can be chosen from a minimum value of 64 bits to a maximum value of 2^{16}, with the constraint of being a power of 2. Each vector register is divided into lanes: each of them is 64-bit wide (`LANE_WIDTH`) and it contains different elements of a vector. The total width of a vector register corresponds to the maximum amount of data that can be processed at the same time by a *processing element*.

- **Source Table**: It is a FIFO that stores arithmetic instructions with their operands waiting to be sent to the *processing element* for execution.
  There is one *source table* for each lane, such that each one receives the instructions from the corresponding lane of the *operand requester* and sends data to the corresponding lane inside the *processing element*.

- **Processing Element**: It performs the real arithmetic instruction execution; it is divided into lanes, such that each one contains one *arithmetic unit* made up of multiple ALUs, multipliers and dividers. Each *source table* sends data to the corresponding *arithmetic unit*; once the results are ready, each *arithmetic unit* sends them to its *destination table*.
  In addition to the *arithmetic units*, there is also a single *reduction unit* performing the reduction operations. Since this type of instruction needs all elements of a vector, the *reduction unit* receives the data from all *source tables*.

- **Destination Table**: It is a FIFO which contains the results of arithmetic instructions waiting to be sent to the *vector register file* for the write-back operation. There is one *destination table* for each lane, which receives the instruction results from the corresponding *arithmetic unit* inside the *processing element*; each *destination table* sends the results to the corresponding lane inside the *vector register file*.

- **Load-Store Unit**: It is in charge of performing load-store instructions. Once it receives the data from the *operand requester*, the *load-store unit* executes the instruction; for the load operations, it directly performs the write-back operation in the *vector register file* without passing through the *destination tables*.

The top-level architecture of the vector processing unit is reported in figure 3.1.

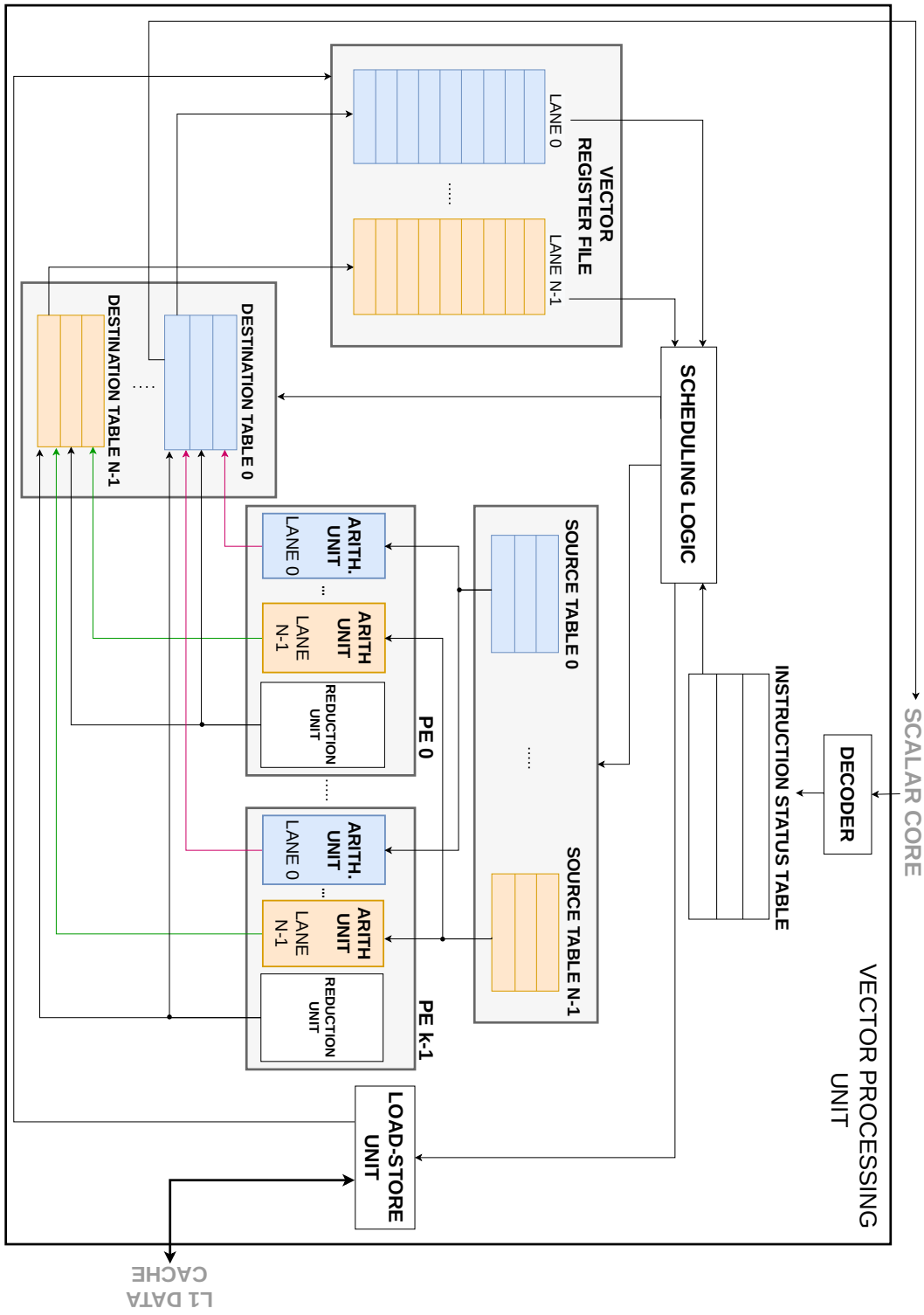

Figure 3.1: Top-level view of the vector processing unit

3.1.1 Design parameters

Before entering into details of the single modules inside the vector processing unit, it is important to see the main parameters that can be set and that characterize the processor.

The main implementation-defined parameters that remain constant at run-time are (the parameters name is the same of the one used in the SystemVerilog code):

- `VLEN`: number of bits in a single vector register. It can be chosen from a minimum value of 64 bits to a maximum value of 2^{16}, with the constraint of being a power of 2.
- `NUM_PE` : number of *processing elements*.
- `NUM_LANE`: number of lanes.
- `NUM_ALU_PER_LANE` : number of ALUs inside each *arithmetic unit* (there is one arithunit for each lane).
- `NUM_MUL_PER_LANE` : number of multipliers inside each *arithmetic unit*.
- `NUM_DIV_PER_LANE` : number of dividers inside each *arithmetic unit*.
- `IST_DEPTH` : depth of the *instruction status table*.
- `ST_DT_DEPTH` : depth of each *source table* and *destination table*.
- `NUM_VRF_PORTS`: number of read/write ports of each *vector register file*'s lane.

Parameters that can change at run-time can be set through a set of configuration instructions, which will modify some Control and Status Registers (CSRs) inside the scalar *LEN5* core. These parameters are:

- `SEW`: It is the selected element width and it can assume four values: 8 bits, 16 bits, 32 bits or 64 bits. By default, a vector register is divided into `VLEN/SEW` elements.
- `LMUL`: A single vector instruction can operate on multiple vector registers. So a vector register group refers to one or more vector registers used as a single operand to a vector instruction. The vector length multiplier, `LMUL` represents the default number of vector registers that are combined to form a vector register group. It can be any value among 1, 2, 4 and 8.

- `VL`: It represents the number of elements to be updated with results of a vector instruction.
  One of the common approaches to manage a large number of elements is "stripmining": considering an application, each iteration of a loop handles some number of elements; iterations continue until all elements have been processed. The application specifies the total number of elements to be processed (the application vector length or AVL) as a candidate value for `VL`, and the hardware responds via a general-purpose register with the (frequently smaller) number of elements that the hardware will handle per iteration, based on the architectural structure. Considering `VLMAX` as the maximum number of elements that can be processed with a single vector instruction and depending on the current iteration, `VL` can be smaller or equal to `VLMAX` = `LMUL`·`VLEN`/`SEW`.

In addition to the parameters that can be set dynamically, it is important to consider that each vector operand has an effective element width (`EEW`) and an effective `LMUL` (`EMUL`) that are used to determine the size and location of all the elements within a vector register group. By default, most of the instructions are **single-width**, meaning that `EEW` = `SEW` and `EMUL` = `LMUL`. However, some vector instructions have source and destination vector operands with the same number of elements but with different widths, so `EEW` and `EMUL` differ from `SEW` and `LMUL` respectively, but `EEW`/`EMUL` = `SEW`/`LMUL`.
There are two types of arithmetic instructions where it is possible to see this feature:

- **Widening** arithmetic instructions: they have a source group with `EEW` = `SEW` and `EMUL` = `LMUL` but they have a destination group with `EEW` = $2 \cdot$ `SEW` and `EMUL` = $2 \cdot$ `LMUL`.
  There is a second variant in which they have the source group of the first operand with `EEW` = $2 \cdot$ `SEW` and the one of the second operand with `EEW` = `SEW`; they have a destination group with `EEW` = $2 \cdot$ `SEW` and `EMUL` = $2 \cdot$ `LMUL`.

- **Narrowing** arithmetic instructions: they have a source group with `EEW` = $2 \cdot$ `SEW` and `EMUL` = $2 \cdot$ `LMUL` but they have a destination group with `EEW` = `SEW` and `EMUL` = `LMUL`.

Considering the load-store operations, they have `EEW` encoded directly in the instruction; then there is the `nf` parameter which identifies one less than the number of registers groups (each one made up of `EMUL` registers) involved in one instruction.

The important thing is that the maximum number of registers involved in one instruction can be equal to 8. From this, it follows that for arithmetic operations

$$\texttt{EMUL} \leq 8 \tag{3.1}$$

and for load-store operations

$$\texttt{EMUL} \cdot (\texttt{nf} + 1) \leq 8 \tag{3.2}$$

The configuration instructions used to set the dynamic parameters (`SEW`, `LMUL` and `VL`) are not executed by the vector processing unit. Since the vector processor is a co-processor thought to be integrated inside a scalar core, all CSRs (both vector and scalar) are included in the scalar core.
It is expected that all instructions (both vector and scalar) are sent to the scalar core; the latter performs a first decoding in order to identify the type of instruction. If it is a configuration instruction for the vector processing unit, the scalar core will execute it modifying the CSRs and sending to the vector processing unit the updated parameters. Unfortunately, the integration of the vector processing unit in the scalar core has not been done yet, so the design was tested by directly sending the CSR values to the vector processor through the testbench.
Since this type of instructions changes the configuration of the processor, it is important that, when there is a change of the dynamic parameters, the instructions with the new configuration do not start their execution until all instructions with the previous configuration are completed. This allows to correctly complete the older instruction with the previous configuration while the newer ones are temporarily stalled. This behaviour is inherently enforced by the microarchitecture of the scalar *LEN5* core.

3.1.2 Design choices

In this section, some design choices are discussed before entering into details of the single units present in the vector processing unit.
In particular, it is reported the types of supported instructions and how the load-store unit has been implemented; then some common features of most of the units inside the vector processor are presented, like the control model of the data structures and the handshake protocol.

Supported instruction subset

The subset of supported instructions from the RISC-V vector extension (V) includes:

- **Integer arithmetic instructions**:
  - *Addition, Subtraction, Multiplication, Multiply-Accumulate, Division* (For the first four types it is supported the single-width and the widening variant).
  - *Logic* operations: AND, OR, XOR.
  - *Shift* operations: logical shift left, logical and arithmetic shift right, narrowing shift right.
  - *Comparison* operations: equal, not equal, lower, lower or equal, greater.
  - *Extension* operations: it executes a zero or sign extension of a source vector integer operand with `EEW` less than `SEW` to fill `SEW`-sized elements in the destination. The `EEW` of the source is 1/2, 1/4, or 1/8 of `SEW`, while the destination has `EEW` equal to `SEW` and `EMUL` equal to `LMUL`.
  - *Move* operations: vector integer move instructions copy a vector source operand to a vector register group, while the integer scalar read/write instructions transfer a single value between a scalar register and element 0 of a vector register.
  - *Min/Max* operations: they find the minimum or maximum between two signed/unsigned integer operands.
- **Integer reduction instructions**: Vector reduction operations take a vector register group and a scalar held in element 0 of a vector register; they perform a reduction using some binary operators to produce a scalar result in element 0 of the destination vector register.
  The supported reduction operations are: *addition* (both single-width and widening variant), *logic operations* (AND, OR, XOR), *minimum* and *maximum*.
- **Strided load-store instructions**: it is supported the constant-stride and unit-stride addressing mode. Vector unit-stride operations access elements

stored contiguously in memory starting from the base effective address. Vector constant-strided operations access the first memory element at the base effective address, and then access subsequent elements at address increments given by a byte offset (the following paragraph 3.1.2 is dedicated on the choice of the Load-store unit implementation).

Vector instructions expect these possible type of source operands:

- VECTOR: the operand is a vector coming form the *vector register file*. It has `EEW` and `EMUL` used to determine the size and location of all the elements within a vector register group.
- SCALAR: the operand is a scalar value coming from the scalar register file.
- IMMEDIATE: the operand an immediate encoded in the vector instruction.

Vector instructions expect these possible type of destinations:

- VECTOR: the destination is a vector so, once the instruction result is available, it is saved in the *vector register file*.
- SCALAR: the destination is a scalar value so, once the instruction result is available, it is sent to the scalar core.

Among the unsupported instructions, there are:

- Vector *mask* operations: vector mask-register operations work on mask registers. The latter are vector registers where each element is a single bit, so these instructions all operate on single vector registers regardless of the setting of `SEW` and `LMUL`.
- *Addition with carry* and *Subtraction with borrow*: they are addition (subtractions) which consider also the carry (borrow) input and the carry (borrow) output. The carry inputs and outputs are represented using the mask register layout, so each carry fills one bit of a mask register.
- *Merge* operation: they combine two source operands based on a mask. For elements where the mask value is zero, the first operand is copied to the destination element, otherwise the second operand is copied to the destination element.
- *Permutation* operations: they are used to move elements around within the vector registers (from this category, only the integer scalar move is supported).

- *Indexed load-store* operations: each element stored in memory is at the effective address given by the sum of a base address to an offset stored in each element of a vector source operand.

- *Floating-point* operations.

The way in which the vector processing unit is structured would lead to not executing these types of instructions efficiently. The processor architecture is thought to be divided into lanes, each of them processing different elements of a vector. So, lanes work in parallel and each of them does not interact with the others.
The unsupported instructions would involve data exchange between different lanes, leading to implement a complex connection logic in order to support different element widths. For example, considering a situation in which `LMUL` $= 1$ and `SEW` $= 16$ bits, each lane contains four elements of a vector (`LANE_WIDTH`/`SEW` $= 64/16 = 4$). If we want to execute instructions involving masks like addition with carry, each mask element is one bit so each lane contains 64 mask elements. In order to process four elements for each lane, we need to spread the mask elements such that only four mask elements remain in each lane. Obviously, this reasoning must be extended to all possible element widths (`SEW`). So, instructions involving massive interaction between lanes (like masking operations or permutations) are not implemented; the best solution is to implement a separate unit that handles this type of instructions in a more optimized way.
There are only two exceptions which include operations involving interaction between different lanes and they are the widening or narrowing variant of some instructions and the extension operations. They work with operands with two times (widening, narrowing, extension by two), four times (extension by four) or eight times (extension by eight) the element width, so it is needed to extend the operands and spread them in different lanes. However, this can be done with a quite regular structure since the possible movements between lanes are limited having few extension factors.

A second reason behind the exclusion of some instruction types relies on the possible applications that can take advantage of a vector processor. For example, considering deep learning applications using neural networks, the operations most commonly executed by the algorithms concern sums, multiplications and above all MACs. This is further explained in [2], which considers also the possibility of enabling efficient processing of deep neural networks by reducing the precision of operations and operands. This can be achieved by moving from floating-point to fixed-point or even integer operations, which reduces the energy and area cost and it does not have a significant impact on accuracy if managed in the right way.
In the face of these considerations, attention has been focused on the most common integer operations for this type of applications.

Load-store unit implementation

In the vector processor, the *load-store unit* has been implemented with an emulator which cannot be synthesized. It executes strided load-store instructions, interacting with a memory emulated with a SystemVerilog class and a file.
This choice comes from the fact that, usually, the way in which the load-store unit is implemented strongly depends on the final application in which the processor will have to work and the host memory architecture. In this way, depending on the target of the processor, the load-store unit can be optimized in a suitable way. Since this is a project not aimed at a specific application yet, it was decided to implement an emulator for the load-store unit in order to describe at a higher level the functions that this unit must perform and focus more attention on the implementation of other components.
This is a preliminary discussion about the *load-store unit* implementation, but a detailed explanation of this module is reported in the section 3.8.

Handshake protocol

In order to be compliant and synchronized with *LEN5* (and possibly with other modular systems), the vector processing unit follows its same handshake protocol both for the communication between the internal units and for the interface with the scalar core itself. This protocol allows to have a modular design with latency-independent components: different sub-units can have a variable latency and, thanks to the handshake signals, all communications remain synchronized.
So, the vector processing unit uses an AXI-like handshake protocol (without being AXI-compliant), based on two main signals:

- **valid:** the sender sends this signal to inform the receiver that the data on the bus is valid in this clock cycle.
- **ready:** the receiver sends this signal to inform the sender that it can accept the data.

The data is considered transferred only when both the *valid* and the *ready* signals are asserted in the same clock cycle.
For further details, the complete explanation of this handshake protocol is reported in *LEN5* execution pipeline design [19].

Control model

As for the handshake protocol, also for the control model of most components inside the vector processing unit, the same methodology of *LEN5* is followed.
The idea is that Moore FSMs is not the best solution in order to control different

data structures efficiently. In a Moore FSM, the input signals of the control unit (i.e. status signals of the data structure) are used to update the state of the FSM through the next state generation combinational network. On the next clock cycle, the output evaluation network generates the control signals for the data structure. This mechanism requires two cycles to update the data structure and this leads to increase the latency and consequently to performance degradation.
For most of the data structures inside the vector processing unit, the adopted solution considers that the operation to be performed is decided based on the status of the data structure itself: there is a combinational network that produces the control signals based on the status signals of the data structure and the requests of other components. So, modules can be seen as FSMs where the status is encoded in the data structure itself, without being explicitly defined. Since the data structure is a sequential block, its status is always updated synchronously so this separates the input signals of the control combinational network from its output signals, as it happens in a Moore FSM. Then, the updated status and values will be available from the next cycle, thus reducing the latency from two clock cycles of the FSMs to one clock cycle.
For further details, always refer to *LEN5* execution pipeline design [19].

Even though for most of the modules the approach described above is used, in some cases it has been decided to use explicit Moore FSMs. This is because in some components the control part is very complex and/or more delicate, and a Moore FSM is easier to describe and validate than a combinational control logic.
An example is given by the control unit of the *operand requester* for the arithmetic instructions inside the *scheduling logic*. This is used to fetch the operands from the *vector register file* in case of arithmetic operations. It iterates with the requests until all needed registers belonging to a register group are read. Since it is necessary to manage different types of instructions (widening, narrowing, extensions or other categories of arithmetic operations) with different operand types (vector, scalar or immediate), this FSM is divided into many branches, one for each type of instruction "category". In this way, each type of instruction can perform operands fetch according to its needs.
Obviously, this leads to a FSM with a lot of states (170) and an average latency of about 7 clock cycles. Certainly, this represents a bottleneck of the design and it will have to be subject to future optimizations, but for a first implementation, the use of a Moore FSM has allowed to easily describe the branches with all possible instruction types making it much easier to debug this structure, which would have been very difficult using combinational control logic.

From here on, all components of the vector processing unit are described with further details.

3.2 Instruction Decoder

The *instruction decoder* is the first module of the vector processing unit, positioned at the interface with the scalar core.
Since the vector processor is a co-processor thought to be integrated inside a scalar core, it is expected that all instructions (both vector and scalar) are first sent to the scalar core; the latter performs a first decoding in order to identify the type of instruction. If it is a vector instruction, the scalar core understands if some scalar operands are needed (`rs1` and/or `rs2`); if this is the case, it executes the scalar operands fetch from the scalar register file, or forwards them from the other scalar units as soon as they are available. Another operation that it performs is to allocate one entry in the Reorder Buffer (ROB) for the vector instruction: this ensures the correct synchronization between the execution of scalar and vector instructions, since, while the co-processor is executing vector instructions, the scalar core can continue to execute scalar instructions and the ROB guarantees that all instructions are committed in program order.
Once these operations are performed, the instruction is sent to the reservation station dedicated to the vector instructions. Here, once all the scalar operands are available (if needed), a *valid* signal is sent to the vector processing unit to notify it that a new instruction is available; the *valid* is received by the *instruction decoder*, which propagates it to the *instruction status table* (the *instruction decoder* is purely combinational, so handshake signals are simply forwarded to the upstream and downstream hardware). When the *instruction status table* asserts a *ready* signal, the instruction is sent to the vector processing unit along with scalar operands (if needed) and the index of the ROB entry where the vector instruction is stored. At this point, the instruction arrives at the *instruction decoder*, which decomposes it in different fields depending on its type; after this operation, the *instruction decoder* sends the instruction to the *instruction status table*.
A block scheme of the *instruction decoder* is shown in figure 3.2, where the handshake signals *valid* and *ready* are shown in light blue.

Before describing how the *instruction decoder* works, some basic information about the instructions format is presented.
RISC-V uses instructions with a fixed length of 32 bits; depending on its type, each instruction is divided into different fields, holding distinct information. The different formats to represent vector instructions can be seen in figure 3.3.

All instructions, both scalar and vector, have a 7-bit OPCODE (ins[6:0]) field, which allows to distinguish between loads, stores and arithmetic operations.
Going into details of the different fields, the load and store instructions have:

- `nf`: it is used for the load-store segment instructions, which move multiple contiguous fields in memory to and from consecutively vector registers. The

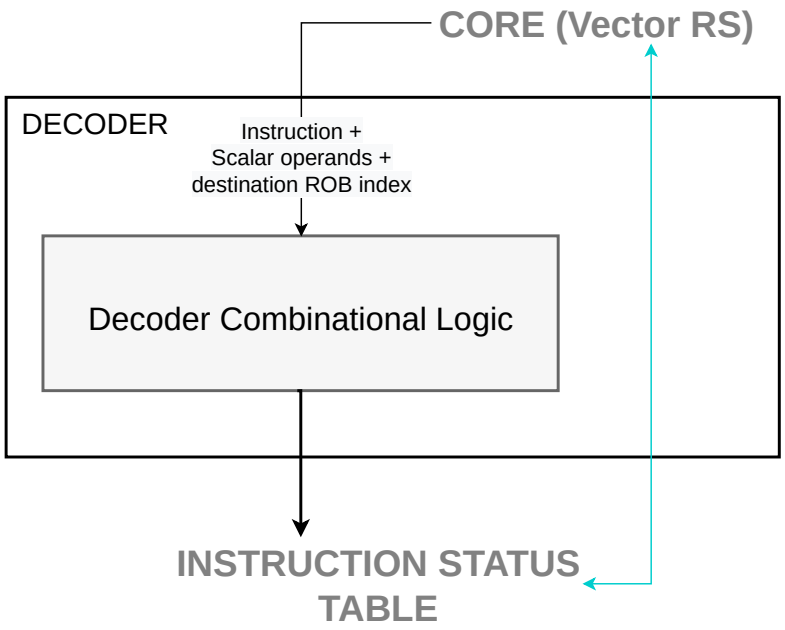

Figure 3.2: *instruction decoder* block diagram

`nf` field is an unsigned integer that contains one less than the number of fields per segment (i.e. one less than the number of registers groups involved in one instruction).

- `mop`: it represents the addressing mode, so it allows to distinguish between strided and indexed load-store operations.
- `vs2`/ `rs2`: for strided load-store operations, the scalar register `rs2` contains the stride, meaning the byte offset between subsequent elements in memory; for indexed load-store operations, each element of the vector register `vs2` contains the offset at which the element is present in memory.
- `rs1`: scalar register `rs1` contains the base address; in order to find the effective address it is needed to sum `rs1` to `rs2` for the strided operations and to each element of `vs2` for the indexed operations.
- `width`: vector loads and stores have `EEW` encoded directly in this field. Strided instructions use `EEW` encoded in the instruction for the data values; indexed instructions use the `EEW` encoded in the instruction for the index values and `SEW`, coming from the CSRs, for the data values.
- `vd`/ `vs3`: for the load instructions, `vd` is the destination vector register; for the store instructions, `vs3` contains the elements to be stored in memory.

Figure 3.3: RISC-V vector extension instruction format (taken from the RISC-V "V" specifications [9])

Looking at the different fields of the arithmetic instructions, we have:

- `vs2`: it is the first vector source operand.
- `vs1`/ `rs1`/ `imm`: it is the second operand, which can be the vector register `vs1`, the scalar register `rs2` or the immediate value.
- `vd`/ `rd`: it is the vector destination register `vd` or the scalar destination register `rd`.
- `funct6`: it specifies the type of arithmetic operation to be executed.

- `funct3`: it specifies the operand types. The possible combinations are: VECTOR-VECTOR, VECTOR-SCALAR, VECTOR-IMMEDIATE.

The *instruction decoder* receives the instruction from the scalar core and, starting from the OPCODE field, it analyses the instruction according to its fields. Then, it generates an output with all the information required to execute the instruction. The data generated by the *instruction decoder* is made up of:

- `vs1_idx` and `vs1_needed`: index of `vs1` register and if it has to be read.
- `vs2_idx` and `vs2_needed`: index of `vs2` register and if it has to be read.
- `vd_vs3_idx` and `vd_vs3_needed`: index of `vd` or `vs3` register and if `vs3` has to be read.
- `operand_type`: for the arithmetic operations it specifies the operand types (`funct3`); for the load-store instructions it specifies the `width`.
- `destination_type`: it can be VECTOR or SCALAR for arithmetic instructions or LS_OP for load-store instructions.
- `rs1_value`: `rs1` scalar value.
- `rs2_value`: `rs2` scalar value.
- `dest_rob_idx`: index of the ROB entry where the vector instruction is stored (in the scalar core).
- `nf`: it specifies `nf` for load-store instructions.
- `ls_add_mode`: it represents the addressing mode of load-store instructions, STRIDED or INDEXED.
- `imm`: immediate value, already sign or zero extended to 64 bits depending on the instruction type;
- `func_unit_type`: it specifies the type of the functional unit in charge of executing the instruction: LOAD, STORE, ALU, MUL, DIV, MAC, RED (*reduction unit*).
- `operation`: it specifies in details the type of arithmetic operation that must be executed;

3.3 Instruction Status Table

The *instruction status table* is a FIFO buffer that keeps track of the status of instructions under execution. If the *instruction status table* is ready to receive new data and there is a valid instruction available from the scalar core in the same clock cycle, then the instruction passes through the *instruction decoder* (where it is analyzed) and is stored in the *instruction status table*.
Then, the hazard check is performed by accessing the *hazard detection unit* inside the *scheduling logic*; if the instruction can be executed without hazards, the *instruction status table* sets it as busy and sends it to the *operand requester* inside the *scheduling logic* for the operands fetch. Each instruction is not removed by the *instruction status table* until it completes its execution.

Going into further details of the data structure, the *instruction status table* is a circular FIFO buffer where one entry contains a single instruction. Each entry contains all information sent by the *instruction decoder* (see the previous section 3.2) with an additional field that encodes the state of the instruction. This status can be:

- `NOT_VALID`: the entry does not contain any valid instruction.
- `READY_HAZ_CHECK`: the entry contains a valid instruction that is ready for the hazard check.
- `READY_EXEC`: the entry contains a valid instruction that passed the hazard check and it is ready to be executed.
- `BUSY`: the entry contains a valid instruction that is under execution.
- `COMPLETED`: the entry contains a valid instruction that is completed.

In addition to the FIFO structure, there are four modulus counters whose output is used to address four entries of the buffer. In particular, there are:

- **Tail counter**: it points to the first free entry of the *instruction status table*.
- **Head counter**: it points to the entry storing the oldest instruction.
- **Hazard counter**: it points to the entry holding the instruction ready for the hazard check.
- **Execution counter**: it points to the entry holding the instruction ready for the execution.

In order to control these components, a combinational control logic is exploited (refer to 3.1.2 for details related to its characterization). It generates the control signals considering the status of the FIFO and the requests of contiguous units. A block scheme of the *instruction status table* can be seen in figure 3.4.

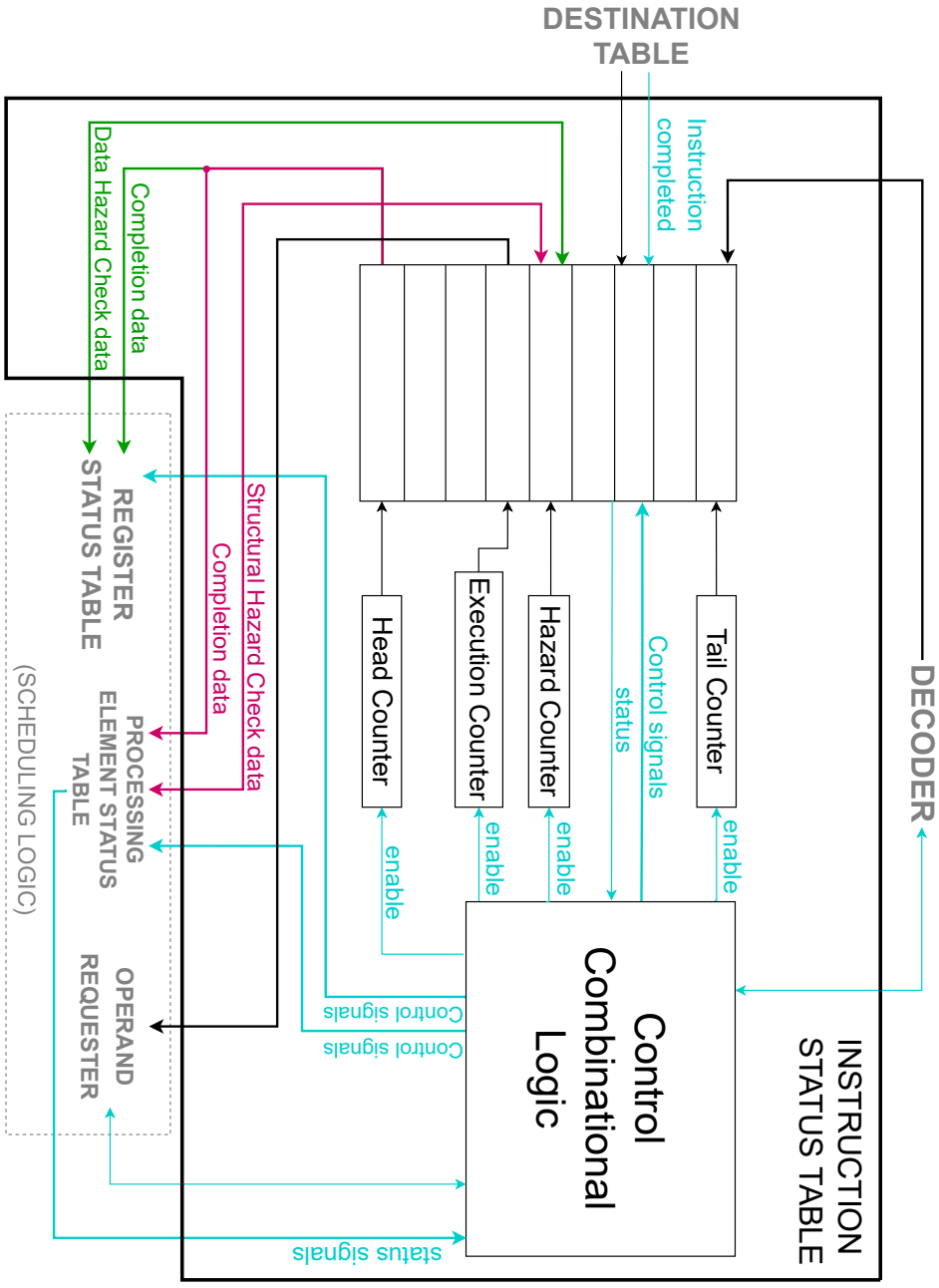

Figure 3.4: *instruction status table* block diagram

In the following paragraphs, the different operations carried out by the control logic are presented.

Push

If the entry pointed by the tail counter is `NOT_VALID`, the *ready* signal is asserted, meaning that the *instruction status table* has a free space to store a new instruction. If the *valid* signal coming from the scalar core and propagated through the *instruction decoder* is asserted in the same clock cycle, a PUSH operation is executed: the instruction is saved in the free entry with the status of `READY_HAZ_CHECK` and the tail counter is incremented.
If the *instruction status table* is full, it does not assert the *ready*; if in the same clock cycle a *valid* is asserted by the scalar core, the *instruction status table* cannot accept the instruction. In this case, no new instruction is pushed into the buffer, and the tail counter is not incremented. In the next clock cycles, the same checks are performed again until the entry pointed by the tail counter reaches the `NOT_VALID` status.

Hazard check

When the entry pointed by the hazard counter is in `READY_HAZ_CHECK` status, it's time to perform both structural and data hazard checks.
For the **data hazards**, the *instruction status table* needs to access the *register status table* inside the *hazard detection unit* in the *scheduling logic*, which keeps track of what vector registers are available in the *vector register file*. If the register groups[1] related to the source operands are available in the *vector register file*, the instruction passes the data hazard check; then, the *register status table* is updated by setting the destination register group as not available in the *vector register file*. In this way, RAW hazards are prevented: if a newer instruction has as source operands the same registers as the destination of an older instruction, the newer one cannot be executed since it sees its source operands as not available in the *vector register file*.
For the **structural hazards**, the *instruction status table* needs to access the *processing element status table* inside the *hazard detection unit* in the *scheduling logic*, which keeps track, for each arithmetic operator inside each *processing element* if it is available for executing the instruction. Then, the first available arithmetic operator and *processing element* are assigned to the instruction pointed by the

[1]Since each vector instruction can involve from 1 to 8 source/destination registers depending on `nf`, `EEW` and `VL` parameters, data hazard checks are performed on all the registers belonging to a group. If one instruction has some scalar source operands, obviously they are not considered.

hazard counter and the *processing element status table* is updated by setting as busy the assigned arithmetic operator. In the case of load-store instructions, these are all directly assigned to the unique *load-store unit* so they do not access the *processing element status table*.
The instruction remains in the `READY_HAZ_CHECK` state until all the hazard checks are passed (the source operands are available in the *vector register file* and one arithmetic operator is assigned to the instruction); once these operations are performed successfully, the entry status of the *instruction status table* is set as `READY_EXEC` and the hazard counter is incremented.

Execution

If the entry pointed by the execution counter is in the `READY_EXEC` status, the *valid* signal is sent to the *operand requester* inside the *scheduling logic*. If the latter asserts a *ready* in the same clock cycle, the instruction is sent to the *scheduling logic* to start the operands fetch. Then, the entry status of the *instruction status table* becomes `BUSY` and the execution counter is incremented.
If the *operand requester* does not assert the *ready*, the entry status of the *instruction status table* remains in the `READY_EXEC` status and the execution counter is not incremented. In the next clock cycles, the same checks are performed again until the *operand requester* becomes ready to accept a new instruction.

Pop

An instruction passes from `BUSY` to `COMPLETED` once it finishes its execution. This event is notified by the *destination table*, which sends a completion signal along with the *instruction status table* entry's index where the completed instruction is stored.
If the entry pointed by the head counter is in the `COMPLETED` status, then its status is set as `NOT_VALID` and the head counter is incremented. At the same time, the *register status table* is updated by setting the destination register group of the instruction as available in the *vector register file*; the *processing element status table* is updated by setting the arithmetic operator used by the instruction as not busy. In this way, the pop operation is performed, by definitely removing the instruction from the *instruction status table*.

3.4 Scheduling Logic

The *scheduling logic* is a quite complex component, made up of two units: the *hazard detection unit* contains the structures used for the hazard checks and the *operand requester* contains the logic used for the operands fetch.
Considering the *hazard detection unit*, this is made up of the *register status table* and the *processing element status table*: the first one is used for the data hazard check, while the second one is used for the structural hazard check.
The *operand requester* performs the operands fetch from the *vector register file* for instructions that passed the hazard check. The *operand requester* is divided into lanes, such that each lane makes a request for the data stored in the corresponding lane of the *vector register file*. Once the operands are ready, if the instruction is arithmetic, each lane of the *operand requester* sends the data to the corresponding *source table* and it allocates one entry in the corresponding *destination table*; for the load-store instructions, the *operand requester* sends the data directly to the *load-store unit*.
A block scheme of the *scheduling logic* can be seen in figure 3.5.

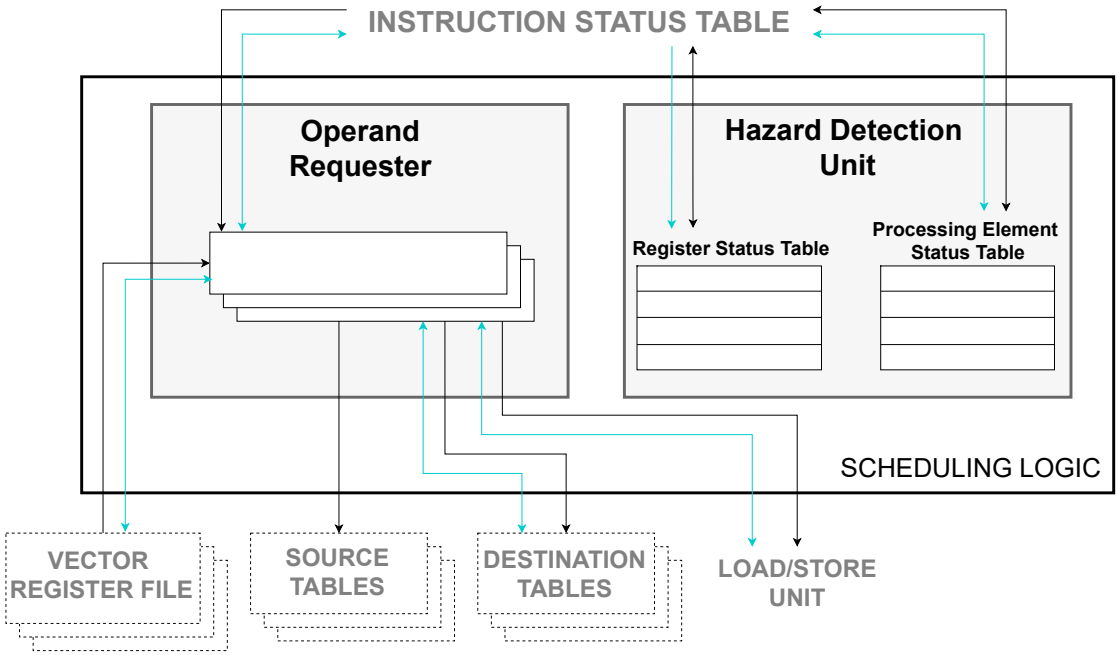

Figure 3.5: *scheduling logic* block diagram

3.4.1 Hazard Detection Unit

The *hazard detection unit* contains the structures to support the data and structural hazard checks.
The *register status table*, used for the data hazard check, is a buffer holding the information about the availability of registers in the *vector register file*.
The *processing element status table*, used for the structural hazard check, is a table holding the information about the availability of the functional units and *processing element*.
They are controlled by the control logic of the *instruction status table* and they exchange data with it.
In the figure 3.6 it is possible to see a block diagram of the *hazard detection unit* and in the following paragraphs the structure and the behavior of the two components are presented.

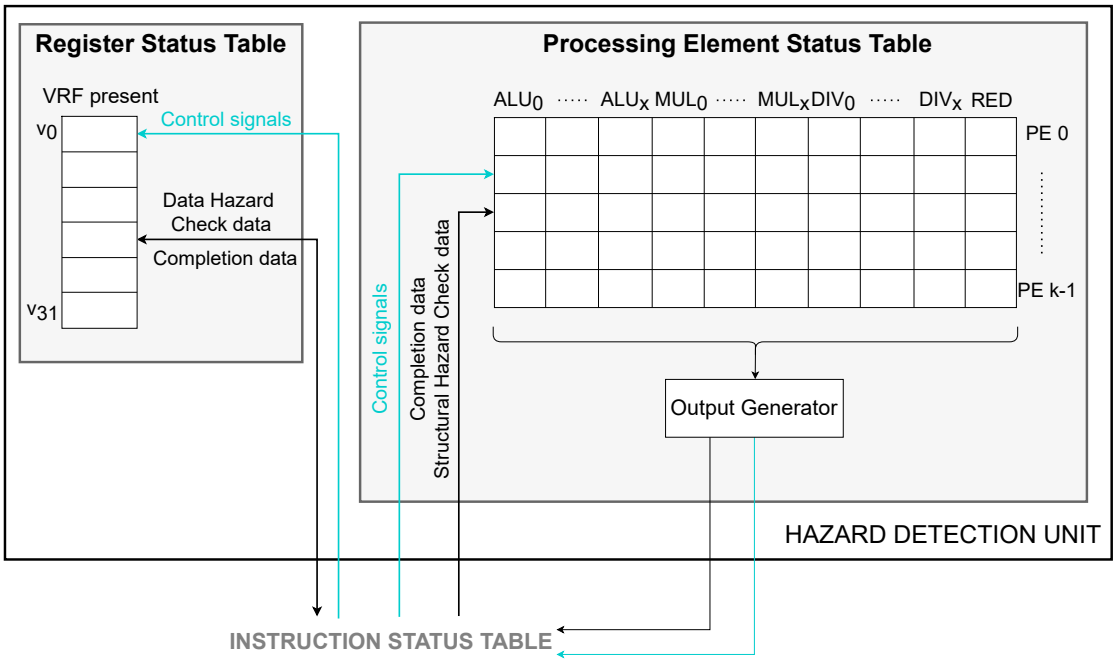

Figure 3.6: *hazard detection unit* block diagram

Register Status Table

The *register status table* is a buffer that stores, for each vector register, a 1-bit information about a vector operand availability in the *vector register file*. In particular, the bit is set to 1 if the register's value is available in the register file; 0 if the value is being computed by an in-flight instruction.

Since each vector instruction can involve from 1 to 8 source/destination registers depending on `nf`, `EEW` and `VL` parameters, data hazard checks are performed on all the registers belonging to a group. In particular, arithmetic instructions considers a number of registers equal to

$$\left\lceil \frac{\texttt{VL} \cdot \texttt{EEW}}{\texttt{VLEN}} \right\rceil \tag{3.3}$$

while load-store instructions consider a number of registers equal to

$$\left\lceil \frac{\texttt{VL} \cdot \texttt{EEW}}{\texttt{VLEN}} \cdot \texttt{nf} \right\rceil \tag{3.4}$$

Possible scalar operands are obviously not considered as they are handled by the scalar pipeline.

In order to perform the data hazard check, the instruction pointed by the hazard counter in the *instruction status table* sends its data to the *register status table*. Depending on the type of instruction, the *register status table* checks if the source register groups are available in the *vector register file*: if the bits corresponding to all registers of the source groups are set to 1, then the hazard check is passed and the instruction can be executed.
Once the data hazard check is passed, the *register status table* is updated by setting the destination register group as not available in the *vector register file*, clearing the corresponding bits. In this way, RAW hazards are prevented: if a newer instruction has as source operand any register inside the destination register group of an older instruction, the former cannot be executed until the latter commits its result(s) in the *vector register file*.

Each time the instruction pointed by the head counter in the *instruction status table* is completed, the *register status table* is updated by setting its destination register group as present in the *vector register file* (the corresponding bits are set to 1). In this way, if one instruction has the same registers as source operands, it can continue with its execution, as explained in the previous paragraph.

Processing Element Status Table

The *processing element status table* is a memory structure modeled as a bi-dimensional array that stores, for each *processing element*, what functional units are available for executing one instruction. In case of load-store instructions, these are all directly assigned to the unique *load-store unit* so they do not access to the *processing element status table* and the latter contains only information related to functional units for arithmetic instructions.
Each cell of the table holds a 1-bit information: the bit is equal 1 if the arithmetic

operator can execute one instruction; the bit is equal 0 if the arithmetic operator is busy (i.e., assigned to a previous in-flight instruction). The size of the table is parametric: the number of the rows is equal to the number of *processing elements*; the number of the columns is equal to the number of all the functional units present in each lane of the *processing element*.

The output generator network, shown in figure 3.7, selects the first available arithmetic operator for the first available *processing element*. In particular:

- For ALU (MUL, DIV) instruction types, there is a first priority encoder which selects the first *processing element* with at least one ALU (MUL, DIV) available along with a valid output. Once the *processing element* is chosen, it is necessary to select which ALU (MUL, DIV) will execute the instruction. So, the output of the encoder is used as a selection signal for a multiplexer that outputs all the ALUs (MULs, DIVs) for the chosen *processing element*. They are sent to a second priority encoder which chooses the first ALU (MUL, DIV) available.
  Narrowing operations, instead, can be executed only by the first ALU. For this reason, in the *processing element status table* there is one priority encoder that selects the first *processing element* having the first ALU available. This allows having, for each *processing element*, only one *narrowing compressor* which halves the size of the output results from the ALU 0 (refer to the *processing element* section 3.7 for further details).

- Multiply–accumulate (MAC) operations need both one multiplier (for the multiplication) and one ALU (for the accumulation). In order to not have a complex connection network between all ALUs and all MULs in the *processing element* to allow all combinations of use, only the last ALU and the last MUL can execute MAC instructions. Then, in the *processing element status table* a priority encoder selects the first *processing element* which has both the last ALU and the last MUL available.

- Reduction (RED) operations are executed by a single *reduction unit* inside one *processing element*. For this reason, in the *processing element status table* there is one priority encoder that selects the first *processing element* having the *reduction unit* available.

In order to pass the structural hazard check, the *instruction status table* needs to monitor the valid signals produced by the priority encoders. In particular, looking at the type of instruction pointed by the hazard counter, it checks the *valid* signal of the encoder processing the corresponding instruction type. If there is a valid *processing element*, then this is assigned to the instruction along with the index of the selected functional unit. Then, the *processing element status table* is updated

by setting as busy that functional unit (the bit is cleared).
When the instruction pointed by the head counter in the *instruction status table* is completed, the *processing element status table* is updated by setting as available the corresponding functional unit (the bit is set to 1).

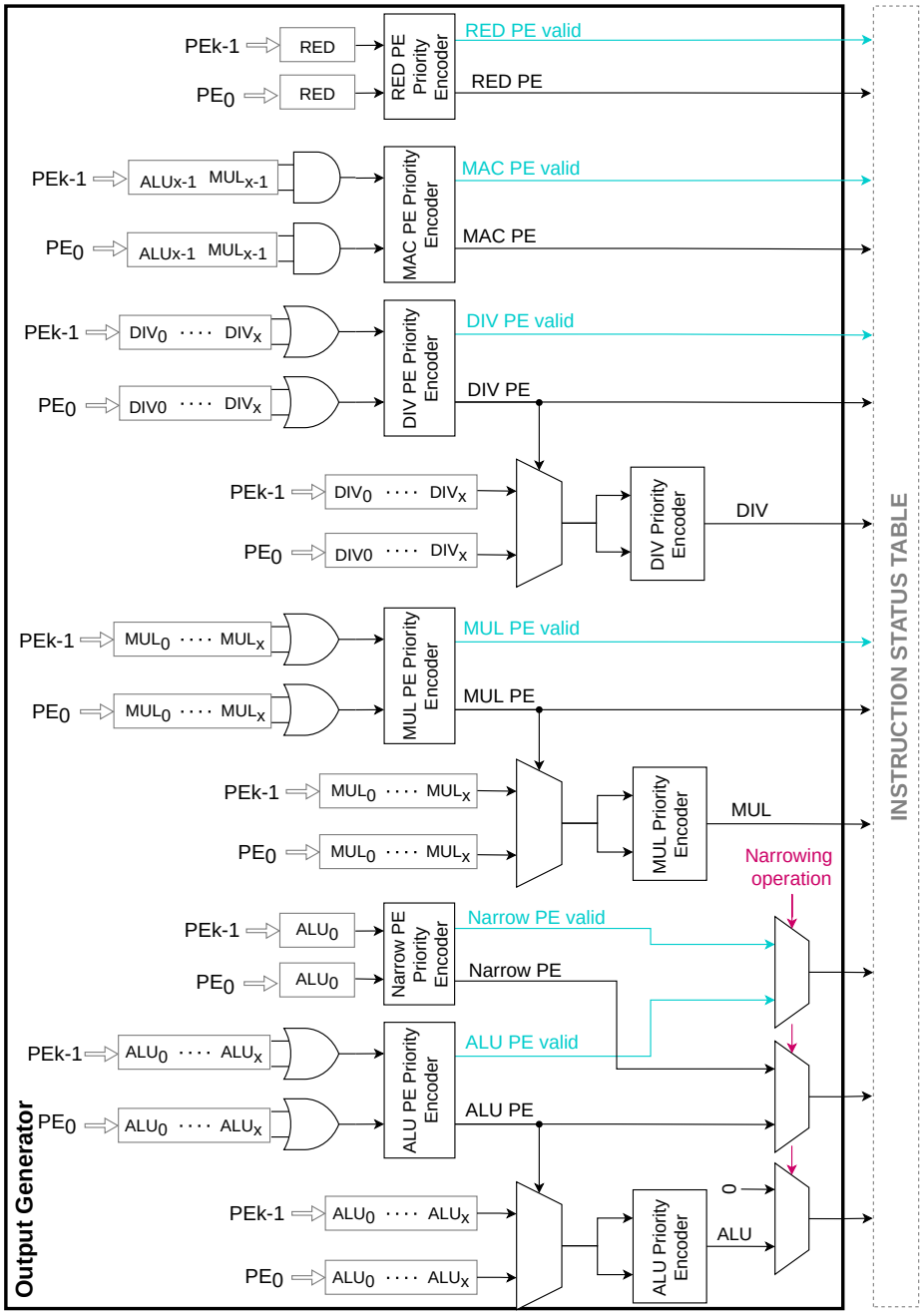

Figure 3.7: Output generator of the *processing element status table*

3.4.2 Operand Requester

The *operand requester*, which is shown in figure 3.8, is responsible for executing the operands fetch from the *vector register file*. It has to manage both arithmetic and load-store instructions: the arithmetic operations are handled by the *arithmetic instruction buffer*, while the load-store operations are handled by the *load-store instruction buffer*.
When an instruction arrives at the *operand requester*, the latter sends to the *position index table* two information: the index of the ROB entry where the vector instruction is stored (in the scalar core) and the index of the *instruction status table* entry where the vector is stored. This allows to address information in the correct positions of the data structures. Further details related to the *position index table* are reported in section 3.6.3.

In the following paragraphs, each component of the *operand requester* is discussed in further detail.

Activation Instruction Buffer CU

It is a Moore FSM, which waits for a valid instruction coming from the *instruction status table*. Once the latter sends a *valid* signal, the CU sends a start signal to the *arithmetic instruction buffer* or to the *load-store instruction buffer*, depending on the type of instruction. Then it waits until the instruction is processed and successively it returns to wait for new instructions.
A flow chart of this CU is reported in figure 3.9. Given that reporting CUs in full would lead to illegible schemes, groups of states have been merged into macro operations represented by the red boxes. In order give an idea of the latency, the number of states in each group is reported at the bottom right of each macro state.

From how this CU is designed, it is clear that arithmetic and load-store instructions cannot be processed in parallel (since the *activation instruction buffer CU* always waits that both the instruction buffers are free before accepting a new instruction, independently from its type). This is not an optimal solution but it is necessary in order to keep the indispensable in-order instructions commit: the *operand requester* is used to allocate the entries of the *destination table*, which is responsible for the write-back operation in the *vector register file*. So, it is fundamental to compile the *destination table* following the instructions order to consequently have an in-order commit. For this reason, only one instruction at a time is processed by the *operand requester*.

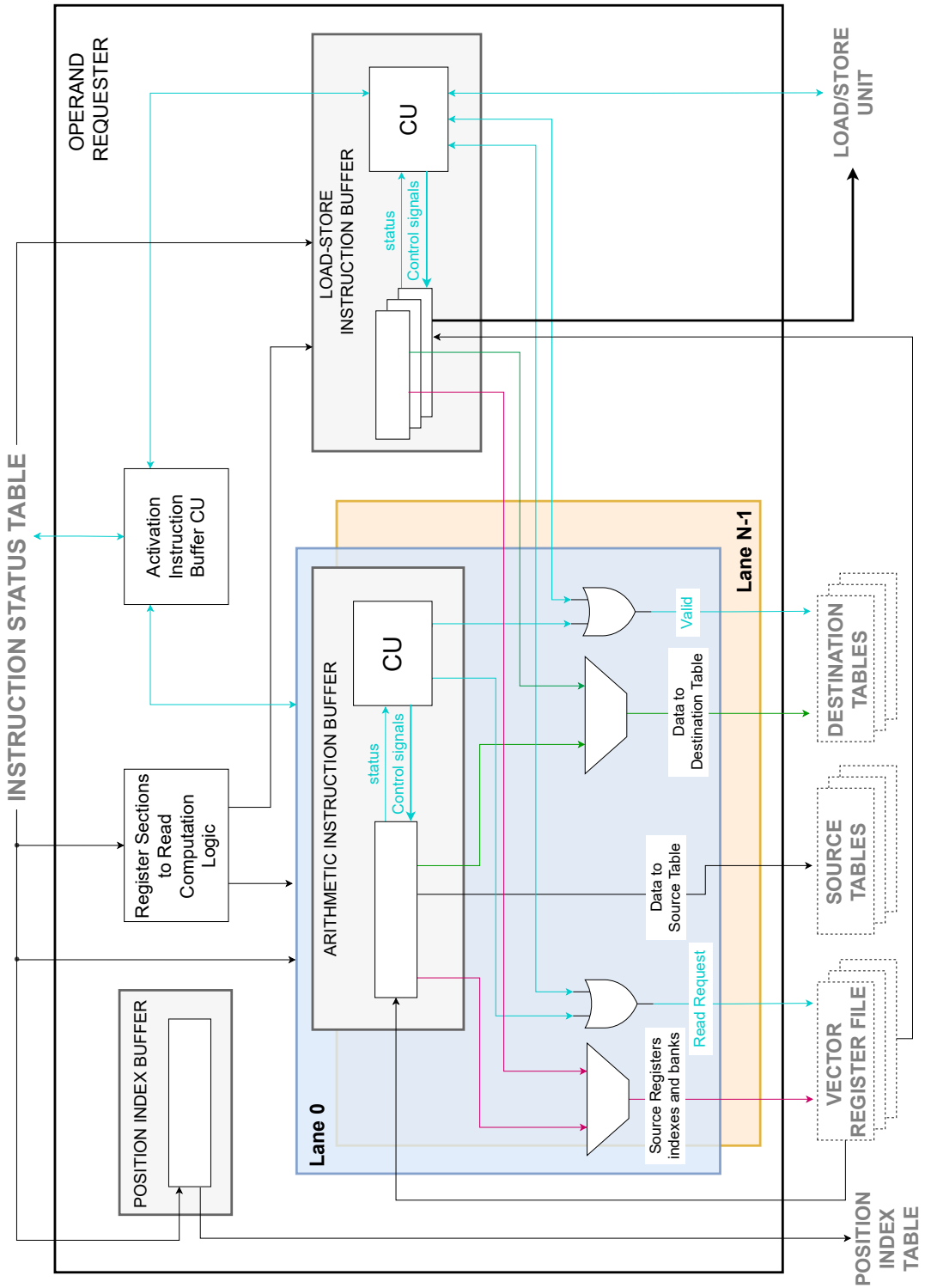

Figure 3.8: *operand requester* block diagram

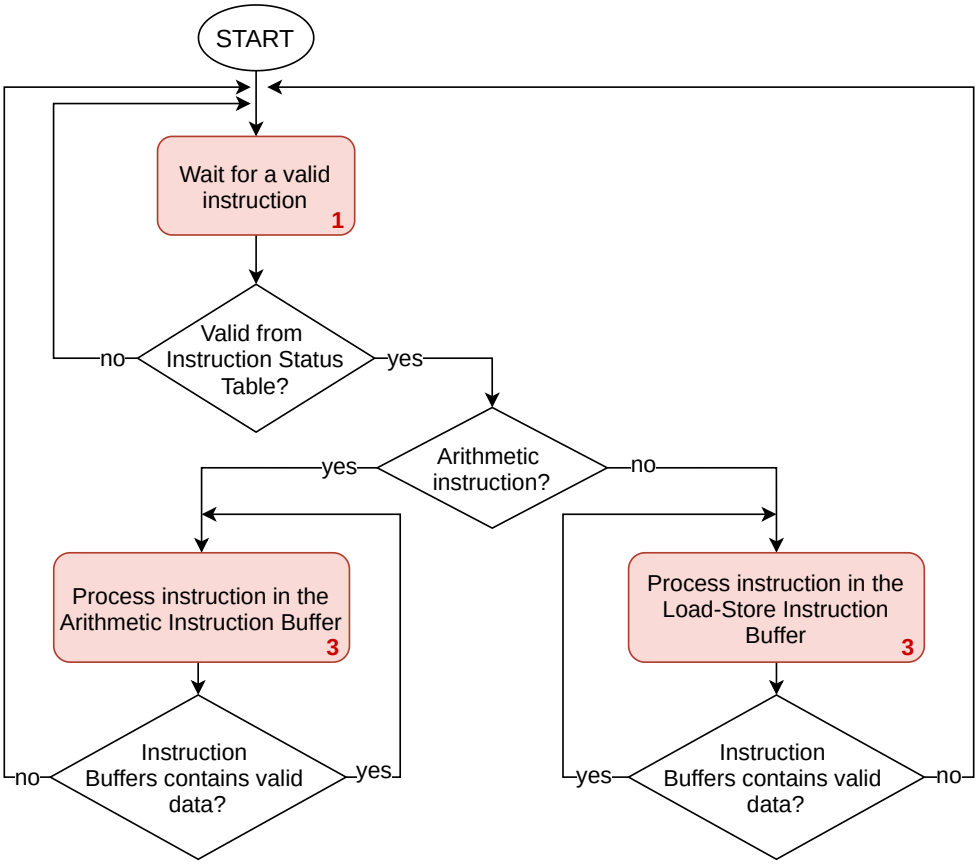

Figure 3.9: *activation instruction buffer CU* flow chart

Register Sections to Read Computation Logic

In order to perform the operands fetch, it is necessary to know what registers need to be read from the *vector register file*. As already mentioned in the previous sections, one instruction can involve not only a single vector register, but a group of maximum eight registers. Moreover, the *vector register file* is divided into lanes, each of them being 64-bit wide (i.e. `LANE_WIDTH`) and containing different elements of a vector.
An additional information concerns how a vector register is represented in the lanes of *vector register file* and, to simplify the explanation, an example is described. Let's consider a parameters configuration with `VLEN` = 512 bits and `NUM_LANE` = 4, then we should have a `LANE_WIDTH` = 512/4 =128 bits to fit the register in the lanes. But this is not possible since the lane width is fixed to 64 bits. In order to solve the situation in which a physical vector register does not fit lanes with `LANE_WIDTH` = 64 bits, it is thought to design each lane as a group of banks: each

bank in one lane contains different elements of the same vector register. Considering the previous example, we will have VLEN = 512 bits, NUM_LANE = 4 and two banks for each lane in order to have a LANE_WIDTH = $512/(4 \cdot 2) = 64$ bits. The complete explanation of how the *vector register file* is structured is in the section 3.5.

The *register sections to read computation logic* first computes how many LANE_WIDTH sections contained in each source register group must be read and this is equal to

$$Total\ reg.\ sec.\ to\ read = \left\lceil \frac{\texttt{VL} \cdot \texttt{EEW}}{\texttt{LANE\_WIDTH}} \right\rceil \tag{3.5}$$

Then, this number is divided between all lanes in order to find the number of registers sections to read for each lane.
The figure 3.10 reports an example that can clarify the explanation. Let's consider a small register file with four registers, each with a size of VLEN = 256 bits. By setting the number of lanes to two, we can derive that the number of banks for each lane is also equal to two. Each vector has EEW = 16 bits so it contains 16 elements. If we want to read 26 elements from the register file, first of all it is necessary to consider a group of two registers (v_0 and v_1); then, we need to read a total of seven LANE_WIDTH sections, four of which from lane 0 and three from lane 1.

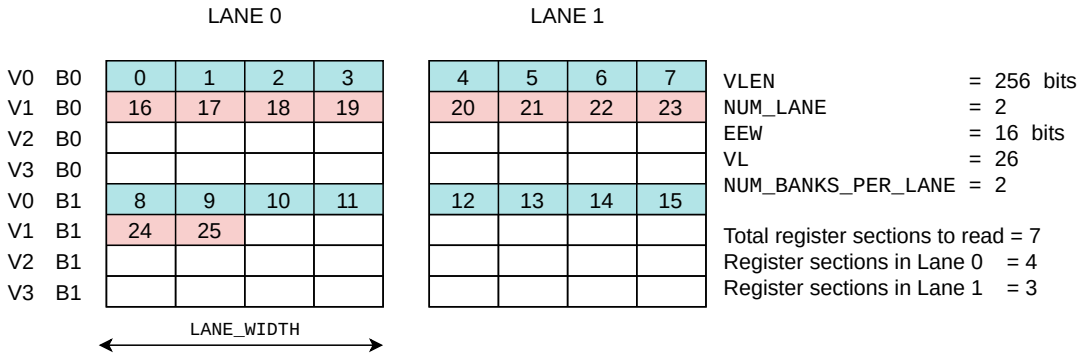

Figure 3.10: Example of how the register sections to read are distributed in the lanes

This module sends the number of LANE_WIDTH sections to read for each lane to the *arithmetic instruction buffer* and *load-store instruction buffer* so they can correctly iterate with the requests to the *vector register file* until all elements are read.

In this module it is also computed the number of bytes inside each LANE_WIDTH section which must be considered as valid data for the future instruction execution. Considering the example 3.10, from bank 1 of lane 0 of vector v_1 it is necessary to consider only two elements, so only the first four bytes must be processed by

the *processing element*. In the same way, the destination register will receive the results, which will be written in the first four bytes of the `LANE_WIDTH` section, leaving the others unchanged.

Arithmetic Instruction Buffer

The *arithmetic instruction buffer* is responsible for executing the operands fetch for the arithmetic instructions.
This data structure is replicated for each lane so that each *arithmetic instruction buffer* performs a requests for source operands[2] to the corresponding lane of the *vector register file*. Once the fetch operation is executed, each *arithmetic instruction buffer* sends the data to the corresponding *source table* and it allocates one entry in the corresponding *destination table*.
From here to the end of the paragraph, the *arithmetic instruction buffer* of a single lane is described and its block scheme is in figure 3.11.

The module is made up of:

- One buffer storing the instruction and the source operands values read from the *vector register file*. An important notice is that each time the *arithmetic instruction buffer* makes a request to the *vector register file*, the latter, if it is ready, outputs a data value that is `LANE_WIDTH` wide. So this data, which is saved in the buffer, can contain one 64-bit element, two 32-bit elements, four 16-bit elements or eight 8-bit elements depending on the element width.

- Four counters:

  - `vs1_banks_counter`: it iterates on the banks of a single register belonging to `vs1` register group. The output of the counter is sent to the *vector register file* in order to read the correct bank of `vs1`.
  - `vs2_banks_counter`: it iterates on the banks of a single register belonging to `vs2` register group. The output of the counter is sent to the *vector register file* in order to read the correct bank of `vs2`.
  - `vd_banks_counter`: it iterates on the banks of a single register belonging to `vd` register group. The output of the counter is sent to both the *vector register file* and *destination table*: if `vd` is a source operand, it is used to read the correct bank inside the *vector register file*; if `vd` is a destination, it is used to know the correct bank in which the result has to be stored.

[2]For arithmetic operations, the first source operand is `vs2` and the second can be `vs1`, `rs1` or `imm`; it is possible to have a third operand for the MAC instructions and it is represented by `vd`.

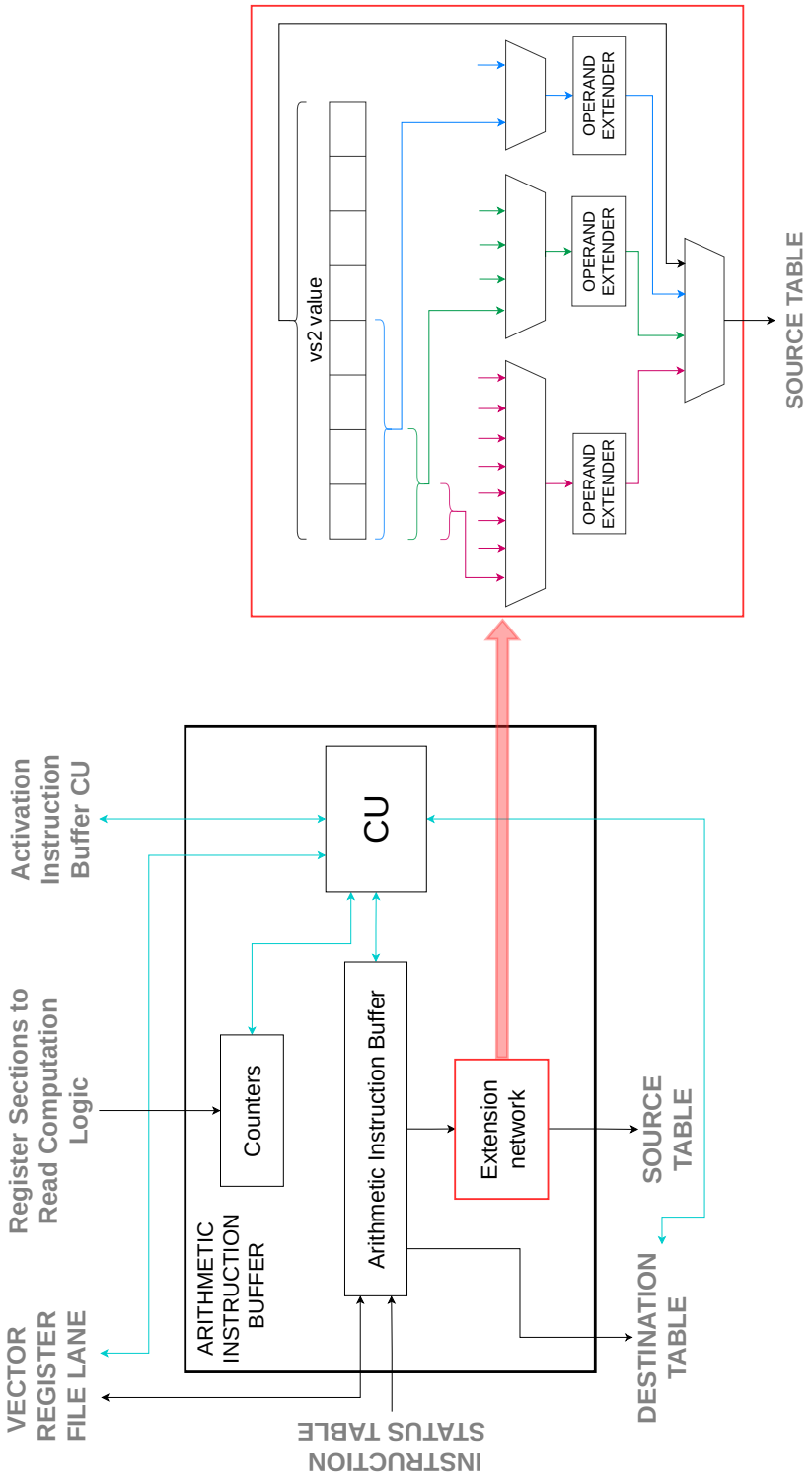

Figure 3.11: *arithmetic instruction buffer* block diagram

  - `reg_sec_counter`: it iterates on the total number of `LANE_WIDTH` register sections inside a register group. This counter is loaded with the value computed by the *register sections to read computation logic*.

- An extension network at the output of the buffer, used to possibly sign- or zero- extend the values of the source operands before they are sent to the *source table*.

- A Moore FSM used for the control of the data structure.

Considering the **extension network**, this is used for widening, narrowing and extension operations where the element width of the result is different from the element width of the operands. The necessity of extending the data has the consequence of possibly transferring them between different lanes and therefore between different *arithmetic instruction buffers*.
An example, which is reported in figure 3.12, considers `NUM_LANE` = 2 and `SEW` = 32 bits, so in each `LANE_WIDTH` section there are two elements, for a total of four elements. If we have to double them for a widening operation, we should have two elements with `EEW` = 64 bits in each *arithmetic instruction buffer*. But this is not possible (always because it would exceed the maximum size of 64 bits), so the solution assumes that two iterations are needed in order to correctly manage the data. The first iteration considers the elements in lane 0: the element 0 remains in *arithmetic instruction buffer 0* where it is extended to 64 bits, while the element 1 is sent to *arithmetic instruction buffer 1* where it is extended. The second iteration considers the elements in lane 1: the element 2 is sent to the *arithmetic instruction buffer 0*, while the element 3 remains in the *arithmetic instruction buffer 1*. The consequence is that it takes twice as long to manage the widening operations. Then, the different *arithmetic instruction buffers* are connected to each other so that they can exchange data before extending them.
To support this mechanism, a 2-way multiplexer, a 4-way multiplexer and an 8-way multiplexer are present in each *arithmetic instruction buffer* in order to select the correct part of the `LANE_WIDTH` section coming from either the same *arithmetic instruction buffer* or from other *arithmetic instructions buffers*. The 2-way multiplexer handles widening, narrowing[3] and extension by 2 operations, so it manages 32-bit values (which can be seen as one 32-bit element, two 16-bit elements or four 8-bit elements). The 4-way and the 8-way multiplexers handle the

[3]Narrowing operations have the first operand with $\texttt{EEW} = 2 \cdot \texttt{SEW}$ and the second operand with $\texttt{EEW} = \texttt{SEW}$, while the result must have $\texttt{EEW} = \texttt{SEW}$. During the operands fetch, they are managed as widening operations so the second operand is extended; then, the *processing element* executes the instruction with a double parallelism and, at the output, the result is compressed by the *narrowing compressor*, which halves its size (refer to section 3.7 for further details).

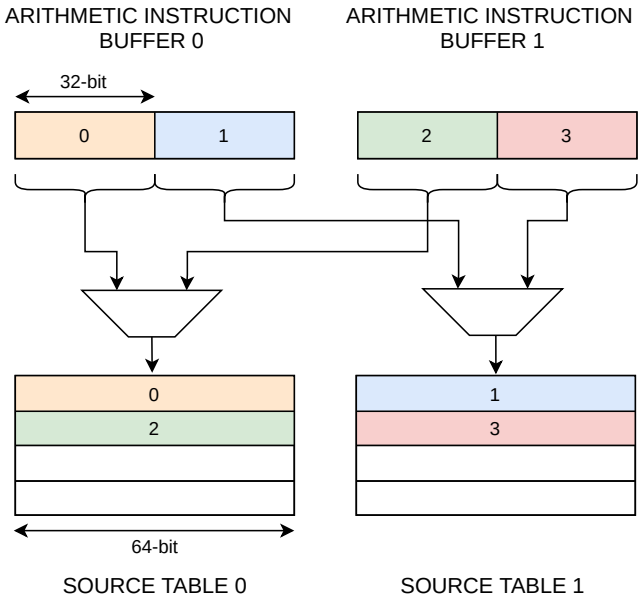

Figure 3.12: Data exchange between *arithmetic instructions buffers* for widening operations

operations of extension by 4 and extension by 8 respectively, so the first one manages 16-bit values (which can be seen as one 16-bit element or two 8-bit elements) and the second one 8-bit values. The output of multiplexers is then extended to 64 bits maintaining the same number of elements. After the extension, an additional multiplexer selects the correct 64-bit data from either the extended ones or from the `LANE_WIDTH` section directly read from the *vector register file*.

The *arithmetic instruction buffer* **control unit** is a Moore FSM, whose state diagram is summarized in the flow chart 3.13 (as in the previous flow chart, it does not show all the possible states, but only the macro states).
Once it receives the start signal from the *activation instruction buffer CU*, a new instruction is saved in the buffer and the `reg_sec_counter` is loaded with the value computed by the *register sections to read computation logic*. The output of the latter is used as the iteration count of the control unit.
Then, the control unit sends a read request to the *vector register file* with the source registers' indexes and banks and it waits until it asserts a *ready* signal. When this happens, the operands' values are stored in the buffer.
At this point, data has to be sent to the *source table*. Since different types of operations need to be managed in different ways, the control unit is divided into multiple branches, each dedicated to a specific "category" of instructions. Each

branch has a different latency[4] but its average is about 4 clock cycles. So, in each branch, the control unit sends a *valid* to the *destination table*. If the latter is ready, the instruction and source operand values are sent to the *source table* and one entry is allocated in the *destination table* with the information for the write-back operation (like the destination register's index and bank). In this way, the allocation of the *source table* and *destination table* provides that each entry contains information about a specific bank of a vector register inside the group and that each entry of the *source table* stores the operands for the instruction which produces the result for the corresponding entry of the *destination table*. A note to highlight concerns the fact that the *valid* signal is sent only to the *destination table* and not also to the *source table*: this is discussed in the section 3.6, but the point is that *source table* and *destination table* are FIFO sharing the same tail counter so if there is a free entry in one table, the same happens for the other. So it is enough to send the *valid* only to one of them.
At this point the control unit checks the value of the `reg_sec_counter`: if other `LANE_WIDTH` register sections must be read, it updates the all counters (it can increment the bank counters to read the next bank or it can clear them, while incrementing the source register's index, in order to read the first bank of the following register inside the source register group; it decrements the `reg_sec_counter` for the next iteration), otherwise the operands fetch is terminated and the control unit can return to wait for a new valid instruction.

Since it is necessary to manage different types of instructions (widening, narrowing, extensions or other categories of arithmetic operations) with different operand types (vector, scalar or immediate), there is a quite large number of branches. This leads to a FSM with a lot of states (170) and an average latency of about 7 clock cycles. Certainly, this represents a bottleneck of the design and it will need to be optimized but, for a first implementation, it makes debugging easier.

Another point that can limit the performance concerns the allocation of the *source tables*. As already mentioned, they have one common tail counter and the problem occurs when not all *arithmetic instruction buffers* are ready to send new data to the *source tables* because some lanes of the *vector register file* are not ready for the fetch operation. In this case, the ready *arithmetic instruction buffers* are stalled until the fetch operation is completed for all the others before sending the instruction to the *source tables*. At this point, all *source tables* can receive synchronously the instruction and the tail counter is incremented.

[4]For example, if we consider the widening instructions, they will have a latency equal to twice the single-width instruction, as explained in the previous paragraph.
In the flow chart 3.13 the different latency of the branches is reported through different letters.

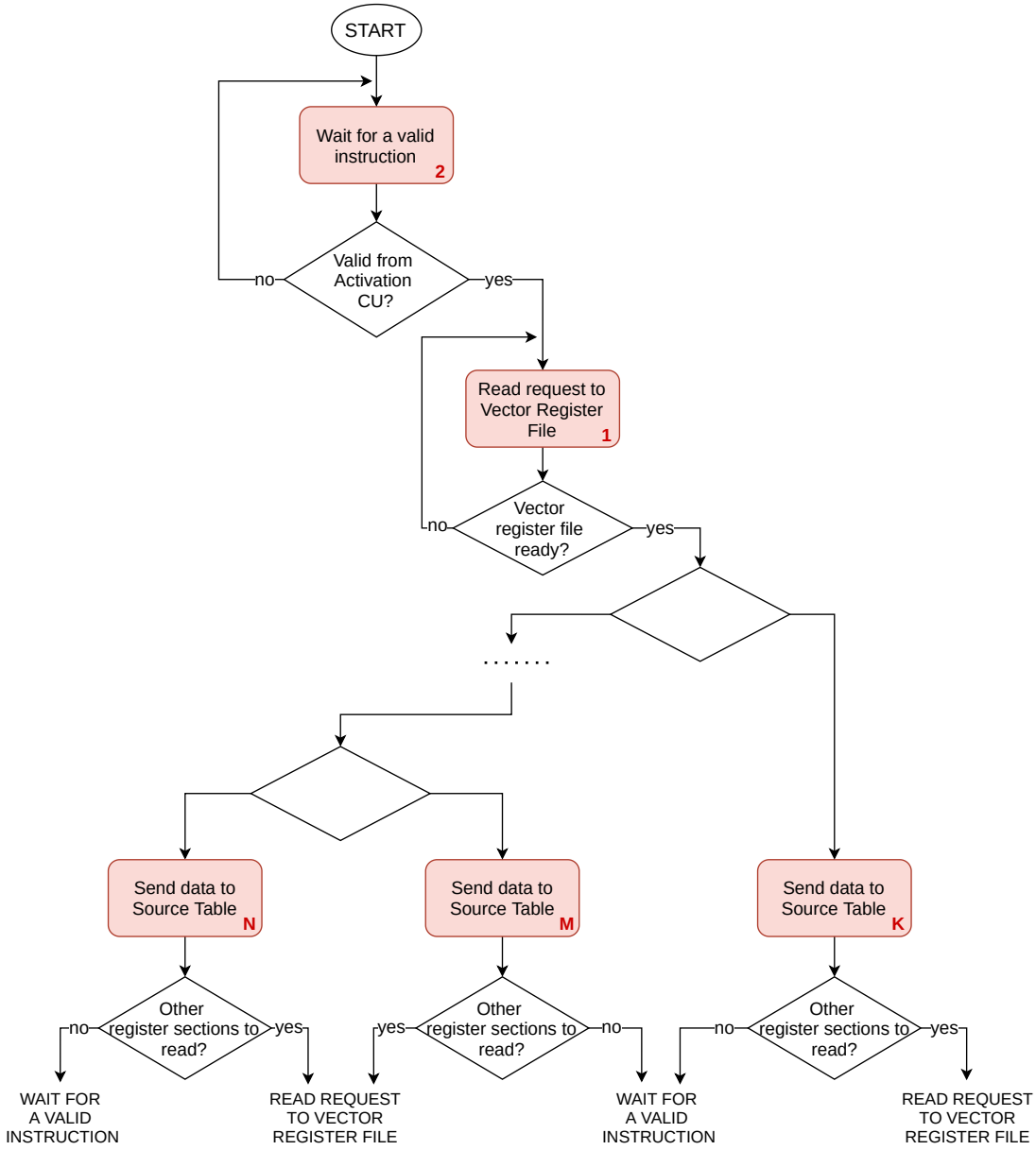

Figure 3.13: *arithmetic instruction buffer* CU flow chart

Load-store Instruction Buffer

The *load-store instruction buffer* is responsible for executing the operands fetch for the load-store instructions.
Also in this case, the structure is divided into lanes such that each lane performs a requests for source operands[5] to the corresponding lane of the *vector register file*. Once the fetch operation is executed, data are sent directly to the *load-store unit*. For each instruction, the *load-store instruction buffer* allocates only one entry in the *destination table*: since the write-back operation in the *vector register file* is executed by the *load-store unit*, it is not necessary to pass through the *destination table*; in any case, it is important to allocate one entry for each load-store operation to maintain the order of instructions in the *destination table* for the indispensable in-order commit.

The module, whose block scheme is in figure 3.14, is made up of:

- a sub-unit for each lane containing:
  - A register data buffer with the information related to the source registers' indexes (`vs2` and `vs3`).
  - Two banks buffers (one for `vs2` and the other for `vs3`) containing the data read from the *vector register file* related to all banks of a single register inside its source register group.
  - Five counters:
    - `vs2_banks_counter` and `vs3_banks_counter`: they iterates on the banks of a single register belonging to respectively `vs2` and `vs3` register group.
    - `iter_vs2_counter` and `iter_vs3_counter`: it iterates on the total number of `LANE_WIDTH` register sections inside respectively `vs2` and `vs3` register groups. This counter is loaded with the values computed by the *register sections to read computation logic*. In this case, the two values can be different because strided instructions use `EEW` encoded in the instruction for the data values and indexed instructions use `EEW` encoded in the instruction for the index values and `SEW` for the data values, so the number of `LANE_WIDTH` register sections of `vs2` can be different from the one of `vs3`.

[5]For load-store operations the source operands are `rs1` for the base address, `rs2` for the stride offset of strided operations, `vs2` for the indexes of indexed operations and `vs3` for the data to be stored in memory.

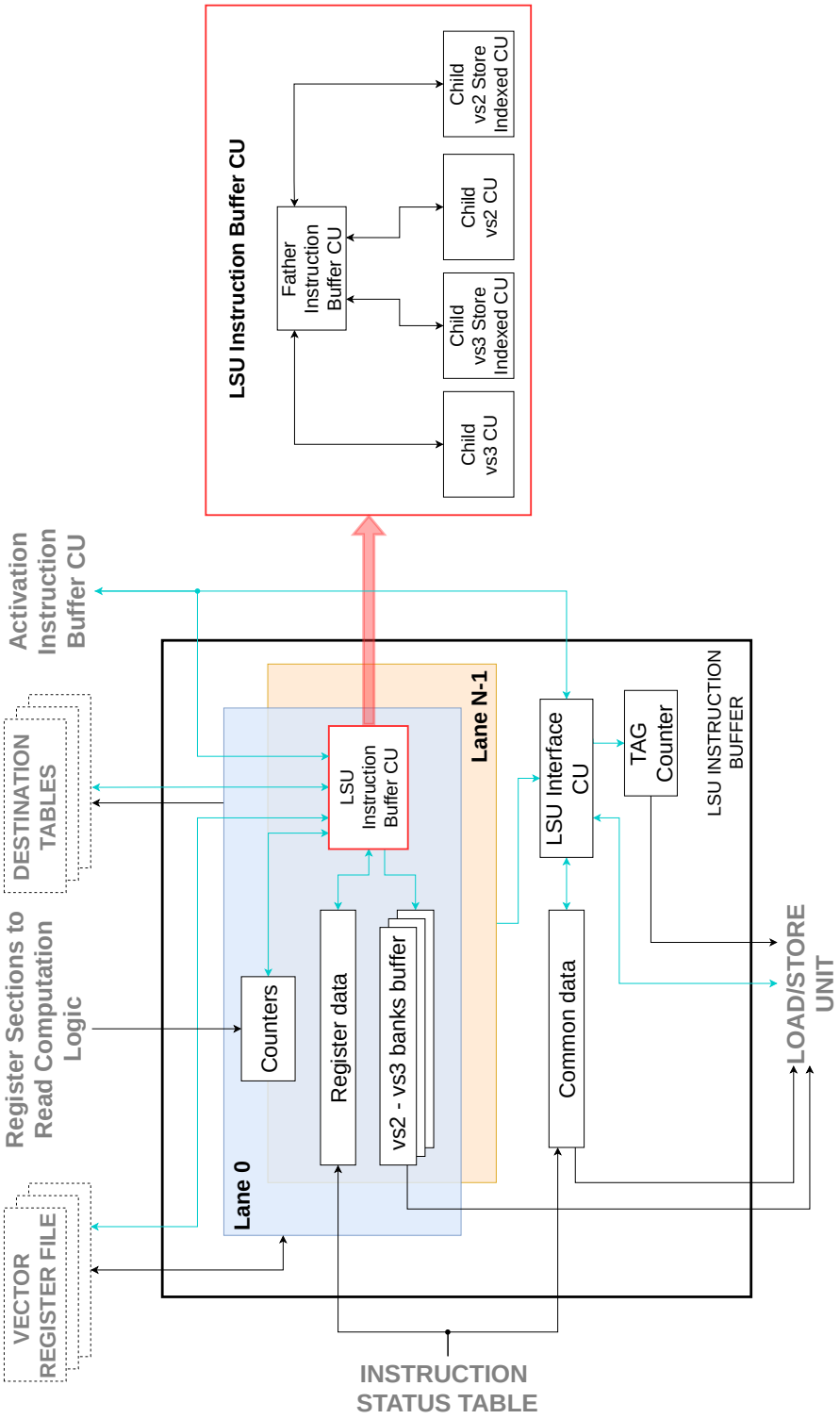

Figure 3.14: *load-store instruction buffer* block diagram

    - ∗ `nf_counter`: it iterates of `nf` parameter used for the segment load-store instructions.
  - *LSU instruction buffer CU*, which is a Moore FSM used for the control of the data structures described above, interacting with the *vector register file* and with the *destination table*.

- A buffer containing the data in common with all lanes, like `rs1`, `rs2`, `nf` and other information needed to execute the instruction.

- A counter (`tag_counter`) used to mark each instruction with a tag (i.e. the output value of the counter). It should be used in the *load-store unit* to maintain a correct order of memory access for different instructions (at the moment, the load-store emulator, described in section 3.8.1, does not use it since it executes one instruction at a time, so there is no problem of memory access conflicts; the role of the counter is discussed in the section 3.8.2 about the preliminary design of the *load-store unit*).

- The *LSU interface CU* used to control the two components described above, interacting with the *load-store unit*.

Let's consider the *LSU instruction buffer CU*. It is made up of a "father" CU which can activate one or more "children" CUs depending on the type of instruction to be processed.
Once the father CU receives the start signal from the *activation instruction buffer CU*, a new instruction is saved in the register data buffer and the counters are loaded. Then it sends a *valid* to the *destination table* and, when the latter asserts a *ready*, it allocates one entry in both the *source table* and *destination table*.
Next, if the instruction requires vector operands, the father CU sends a read request to the *vector register file* and waits until it asserts a *ready* signal. When this happens, the operands' values are stored in the two bank buffers.
At this point, the father CU branches into four parts:

- the first one is dedicated to the load-strided instructions. In this case, no vector operand is required so the father CU waits that the instruction is sent to the *load-store unit* and then it returns to the waiting state for a new start signal.

- the second one is dedicated to the store-strided instructions, which need `vs3` as vector operand. In this branch, the father CU activates the child `vs3` CU. The latter checks if all banks of a single register have been read: if this is the case, it waits until they are sent to the *load-store unit*. Then, it updates the counters related to `vs3` and `nf` and it returns in idle.

The father CU, who was waiting for the term of child `vs3` CU, returns to the operand request state if not all the registers sections have been read; otherwise, it returns to the waiting state for a new start signal.

- the third one is dedicated to the load-indexed instructions, which need `vs2` as vector operand. In this branch, the father CU activates the child `vs2` CU and then the same operations of the previous branch are executed.
- the fourth one is dedicated to the store-indexed instructions, which need `vs2` and `vs3` as vector operands. In this branch, the father CU activates the children `vs2` Store-Indexed CU and `vs3` Store-Indexed CU. These two last CUs have the same behavior as the child `vs2` CU and child `vs3` CU but they do not update the `nf_counter`. The latter is updated by the father CU once the children have finished. This happens because indexed instructions use `EEW` encoded in the instruction for the index values and `SEW` for the data values so the number of `LANE_WIDTH` register sections of `vs2` can be different from the one of `vs3` and so the number of iterations of the children CUs can be different. In this way, for segment instructions which require a number of register groups equal to `nf`, it is ensured that one group related to `vs2` and one related to `vs3` have been fetched and sent to the *load-store unit* before starting to fetch a new group.

Finally, the last control unit is the *LSU interface CU*. Once it receives the start signal from the *activation instruction buffer CU*, information related to the new instruction is saved in the buffer containing the data in common with all lanes. Then, also this control unit branches into four parts: they perform the same operations but they consider either `vs2` or `vs3` depending on the type of instruction. Basically, each branch waits until all banks of a single register have been fetched from the *vector register file*. Once this happens, it sends a *valid* to the *load-store unit*. If the latter asserts a *ready*, then the data are sent to the *load-store unit* and the *LSU interface CU* returns to wait for new data to be sent. Once all source register groups have been sent to the *load-store unit*, it enables the `tag_counter` and it returns to the waiting state for a new start signal.

3.5 Vector Register File

The *vector register file* is made up of 32 registers (v_0 - v_{31}). The registers' size is parametric and it can be set through the `VLEN` parameter. The *vector register file* is divided into lanes: each of them is 64-bit wide (`LANE_WIDTH`) and it contains different elements of a vector.
Once the parameters `VLEN` and `NUM_LANE` are fixed, then it is necessary to represent the physical vector register inside the lanes. For example, if we want to have `VLEN` = 512 bits and `NUM_LANE` = 4, we should have `LANE_WIDTH` = 512/4 = 128 bits in order to represent the register in all lanes. But this is not possible since the lane width is fixed to 64 bits. In other words, in order to manage a 512-bit register with `LANE_WIDTH` lanes, we should have `NUM_LANE` = 8.
In order to solve the situation in which the size of a physical register is too large to fit lanes with `LANE_WIDTH` = 64 bits, it is thought to design each lane as a group of banks: each bank in one lane contains different elements of the same vector register. In this way, it's like the excess part of the register that should require additional lanes is moved into the available ones.
So, the number of banks for each lane can be computed as

$$\texttt{NUM\_BANKS\_PER\_LANE} = \frac{\texttt{VLEN}}{\texttt{LANE\_WIDTH} \cdot \texttt{NUM\_LANE}} \tag{3.6}$$

Considering the previous example, we will have `VLEN` = 512 bits, `NUM_LANE` = 4 and `NUM_BANKS_PER_LANE` = 2. The figure 3.15 represents the *vector register file* structure for the current example.

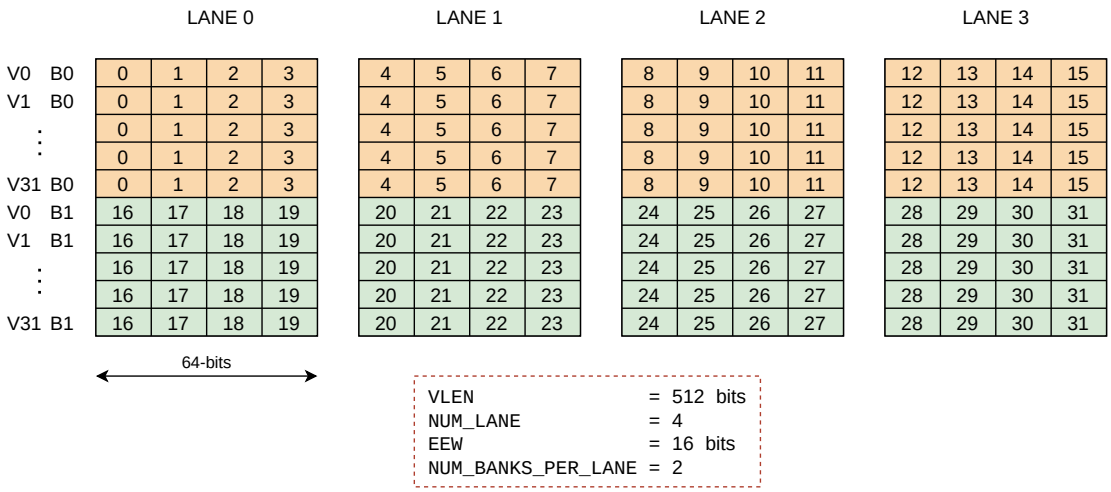

Figure 3.15: Example of *vector register file* configuration with two banks for each lane

Then, since the element width can be dynamically configured, each lane can contain eight 8-bit elements, four 16-bit elements, two 32-bit elements or one 64-bit element. In the example of figure 3.15, we have EEW = 16 bits, so each lane contains four 16-bit elements and the total number of elements in each physical vector register is equal to 32: 16 elements are in the first bank, while the others are in the second bank.

At this point, it is necessary to consider also that a single vector instruction can operate on multiple vector registers. So, a vector register group refers to one or more vector registers used as a single operand in a vector instruction. The parameter LMUL represents the default number of vector registers that are combined to form a vector register group and it can assume four values: 1, 2, 4 and 8.
In order to understand this point, another example is presented. Let's consider LMUL = 2, SEW = 16 bits and VLEN = 512 bits. In this case, considering LMUL, two physical vector registers are grouped in order to form a single vector operand.
Then, it is necessary to consider the effective number of registers inside a group[6], which is

$$\texttt{EMUL} = \texttt{LMUL} \cdot \frac{\texttt{EEW}}{\texttt{SEW}} \tag{3.7}$$

If we have EEW = 32 bits, it means that EMUL = 4, then each register group doubles its size, since it is made by four physical registers.
However, the maximum number of elements inside the group is always equal to

$$\texttt{VLMAX} = \texttt{LMUL} \cdot \frac{\texttt{VLEN}}{\texttt{SEW}} = \texttt{EMUL} \cdot \frac{\texttt{VLEN}}{\texttt{EEW}} = 64 \tag{3.8}$$

So, this means that the number of elements involved in the instruction is always the same but, since each element doubles its size, the number of physical registers needed to store them must double.
The change of *vector register file* configuration, when considering EEW, can be seen in figure 3.16.

[6]In any case, the total number of physical vector registers involved in each instruction cannot be greater than 8.

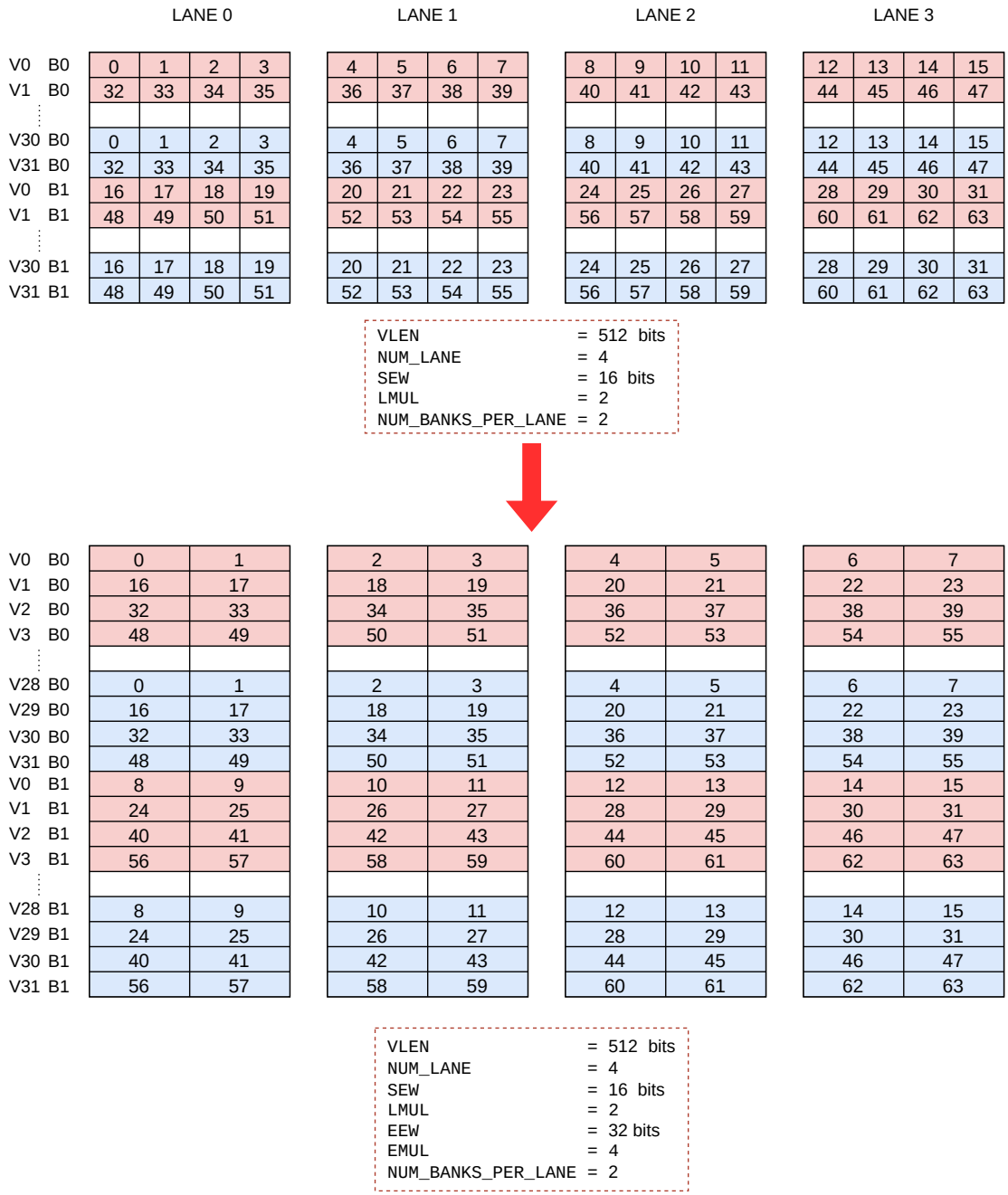

Figure 3.16: Example of *vector register file* configuration, starting from `EEW` = `SEW` and arriving to `EEW` = 2 · `SEW`.

Once the organization of the *vector register file* has been clarified, it's time to go into the details of how the data structure is implemented. From here to the end of the paragraph, the structure of a single lane of the *vector register file* is described, given that all lanes are implemented in the same way.

In order to arrive at the structure of each lane, it is necessary to start with a consideration. Each lane is possibly made up of several banks, each of them containing 32 `LANE_WIDTH` register sections (one for each vector register). Accessing all the banks of all the registers simultaneously would require very large access ports since each would be connected to $32 \cdot$ `NUM_BANKS_PER_LANE` inputs, resulting in a significant impact on area and power consumption.
This problem is solved by providing each lane with a limited number of read/write ports, such that each port is connected to a subset of registers[7]. In this way, not all registers and banks can be accessed at the same time since on a single port only one bank of one register can be read or written in a certain clock cycle. This solution has the advantage of reducing the overall complexity of the network; the limitation of having multiple registers mapped on the same port is partially bypassed by mapping on the same port registers that are not expected to be used at the same time. For example, a compiler may try to map an instruction's source operands on consecutive registers, like $v_2 = v_1 + v_0$. Following this assumption, it is preferable to map consecutive registers on different access ports, so all the operands can be fetched in a single cycle. The proposed architecture follows this assumption to map the 32 vector registers to the available access ports. Thus, each lane is divided into a number of slices equal to `NUM_VRF_PORTS`. Each slice contains a port and a portion of the register file made up of all banks of a vector registers' subset. The number of registers inside each slice is equal to

$$\texttt{NUM\_REG\_PER\_SLICE} = \frac{32}{\texttt{NUM\_VRF\_PORTS}} \tag{3.9}$$

while the number of entries in each slice is equal to

$$\texttt{SLICE\_DEPTH} = \texttt{NUM\_REG\_PER\_SLICE} \cdot \texttt{NUM\_BANKS\_PER\_LANE} \tag{3.10}$$

To clarify the explanation, let's consider an example, which is shown in figure 3.17. If `NUM_VRF_PORTS` $= 8$ and `NUM_BANKS_PER_LANE` $= 2$, the lane is made up of 8 slices (one for each port), each of them containing all banks of four registers (so its depth is equal to 8). Then, the first slice holds the banks of registers v_0, v_8, v_{16} and v_{24}; the second one holds the banks of registers v_1, v_9, v_{17} and v_{25} and so on for the others.

[7]All the banks of a single register are mapped to the same access port.

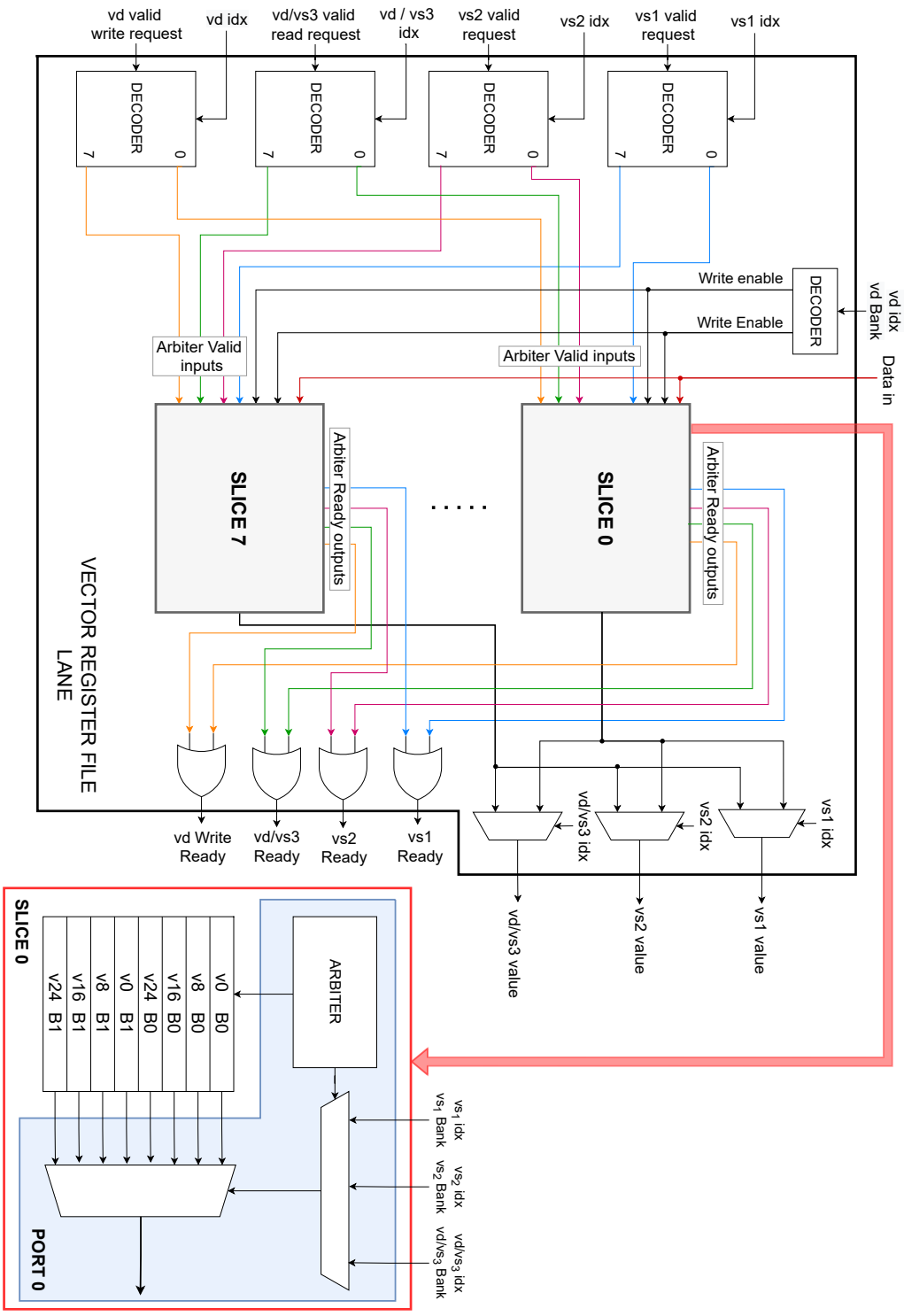

Figure 3.17: *vector register file* lane block diagram

Read operation

In order to perform the read operation, there are three decoders (respectively for `vs1`, `vs2`, `vs3`/`vd`) that receive the read requests (*valid* signals) and the registers' indexes from the *operand requester*. Based on the indexes, each of them propagates the request to the correct slice.
Each port receives the requests, which are processed by an arbiter. The latter selects what operation must be executed on the slice if multiple requests arrive on the same port. In the case of multiple read requests, `vs1` has higher priority with respect to `vs2` and `vs2` has higher priority with respect to `vs3`. In case there is a read request and a write request at the same time, write priority is higher than read priority. So, the arbiter outputs the ready signals related to the input requests, such that only the one of the selected operation will be at 1. Then, two multiplexers inside each port are used to select the correct entry of the slice containing the data to be read, based on the selected source register index and bank.
Then, data at the output of the ports are sent to other three multiplexers (respectively for `vs1`, `vs2`, `vs3`/`vd`), which select data coming from the correct ports. Finally, there are three logic OR ports that take the ready signals generated by the arbiters in all ports and they generate as outputs the *ready* related to the requests. In this way, data with their relative *ready* signals are sent to the *operand requester*.

Write operation

In order to perform the write operation, there is a first decoder which receive the write request (*valid* signal) and the index of `vd` from the *destination table*. Based on the index, it propagates the request to the correct slice. Then a second decoder, based on the register's index and bank, selects the correct entry of the slice where to write the data.
Each port receives the request from the first decoder, which is processed by an arbiter in the same way described above. When one arbiter selects the write operation, it sends to the slice a *valid* signal that, together with the output of the second decoder, allows to write the data coming from the *destination table* in the correct entry and slice.
Finally, there is a logic OR port that takes the ready signals of the write operation generated by the arbiters of all ports. It generates as output the *ready* related to the request, which is sent to the *destination table*.
One note to highlight is that, together with the data, the *destination table* sends also the number of bytes to be written in the `LANE_WIDTH` register section. This is already explained in the section 3.4.2: if $\texttt{VL} \cdot \texttt{EEW}$ is not multiple of `LANE_WIDTH`, the last elements of a vector fill only a fraction of a `LANE_WIDTH` register section. Then, it is necessary to modify only that portion of the register section with the results of the instruction, leaving the other part unchanged.

3.6 Source and Destination Tables

This module is made up of four components: *source tables*, *destination tables*, *position index table* and *source and destination table counters*.
There is one *source table* and *destination table* for each lane and, together with the *position index table*, they are all FIFO buffers. The counters used to point to their entries are all in the *source and destination table counters*, taking into account that all FIFOs share the same tail counter; then there is a single head counter for all *source tables* and a single head counter and write-back counter for all *destination tables* (the first one is used also by the *position index table*).

This structure allows the allocation of the FIFOs such that each entry of the *source table* stores the operands for the instruction which produces the result for the corresponding entry of the *destination table*.
In order to be more clear, an example is shown in figure 3.18. Let's consider the operation $v_2 = v_0\ op\ v_1$ and a configuration of the parameters such that `VL` $= 10$, `VLEN` $= 256$ bits, `NUM_LANE` $= 2$ and `EEW` $= 16$ bits. Then, entry 0 of *source table* 0 stores the first four source elements coming from bank 0 of v_0 and v_1; entry 0 of *destination table* 0 will store the first four destination elements with the result of the operation, which will be written in bank 0 of v_2. The same happens for entry 0 of *source table* 1 and *destination table* 1, which consider other four elements.
Regarding the bank 1, only three elements must be processed. So, entry 1 of *source table* 0 stores the three operands and entry 1 of *destination table* 0 will receive the results; then, since the elements are finished, entry 1 of *source table* 1 and *destination table* 1 will not receive anything so their status remains not valid.

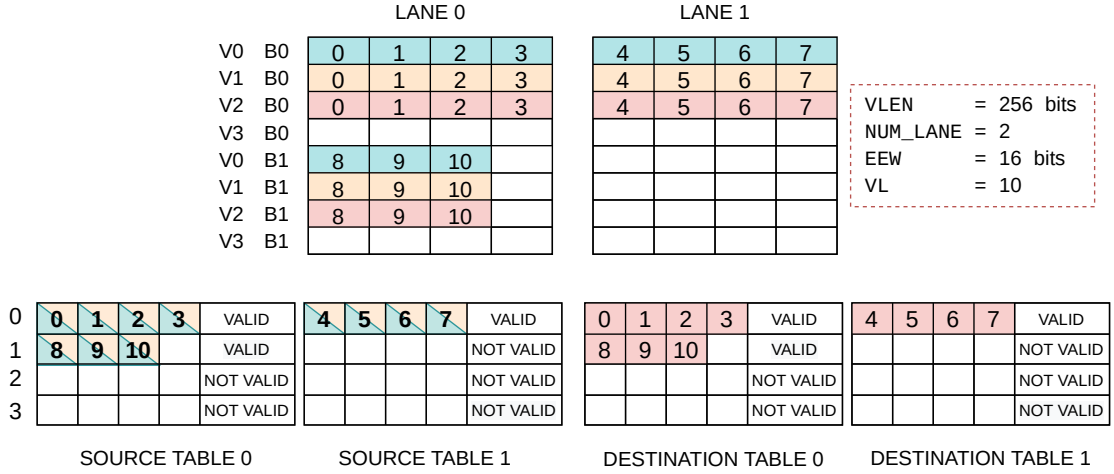

Figure 3.18: *source table* and *destination table* allocation

The block diagram of the module is represented in figure 3.19.

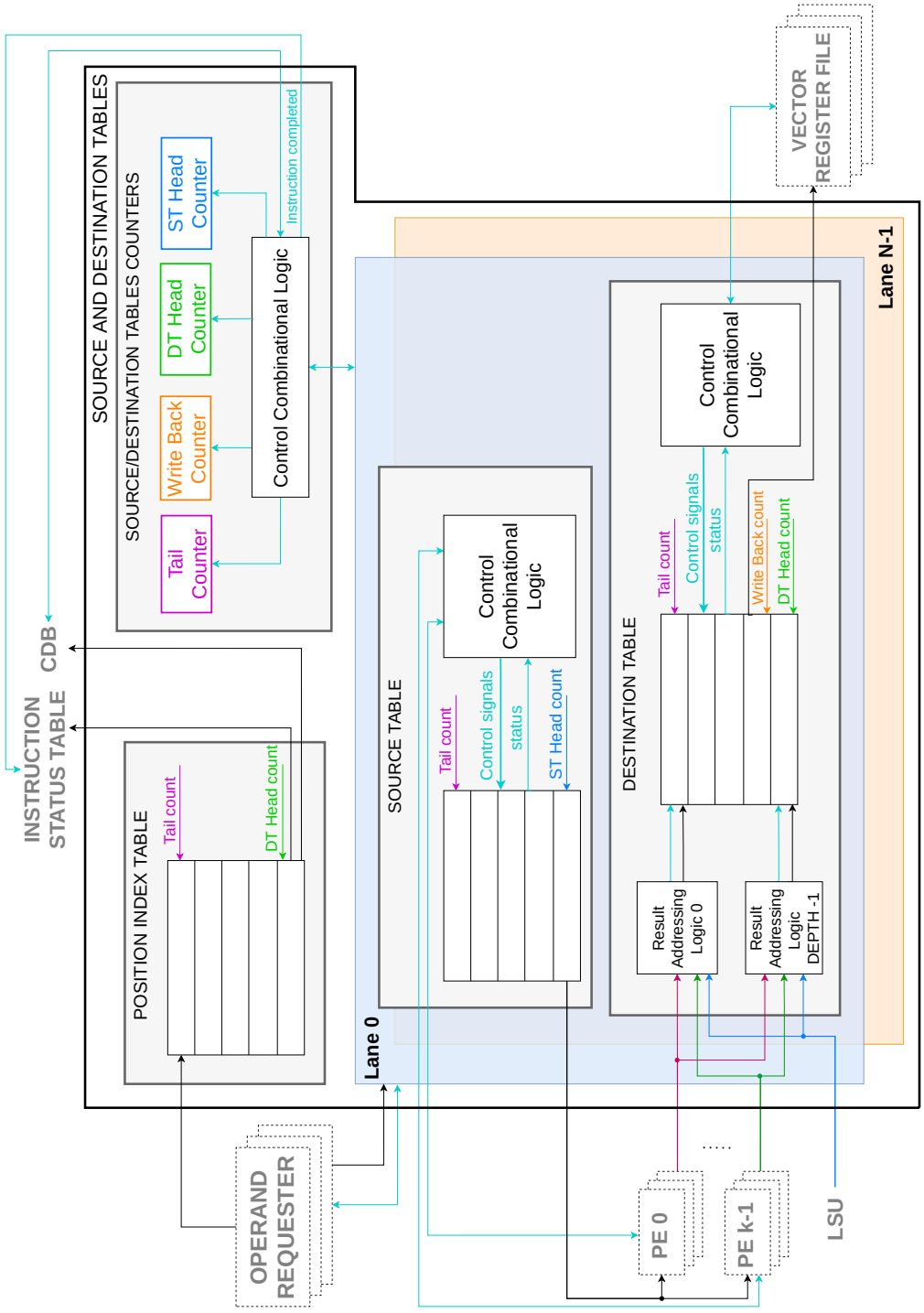

Figure 3.19: *source table* and *destination table* block diagram

3.6.1 Source Table

It is a FIFO controlled by a combinational control logic, which stores arithmetic instructions with their operands waiting to be sent to the *processing element* for execution. There is one *source table* for each lane, such that each one receives instructions from the corresponding lane of the *operand requester* and it sends data to the corresponding lane (*arithmetic unit*) in the correct *processing element*.
Considering the data structure, each *source table* is a circular FIFO buffer where each entry stores the instruction to be executed. In particular, the main information concern:

- source operand values required by the instruction (each vector operand is a `LANE_WIDTH` section belonging to a specific bank of a vector register, which can contain from one 64-bit element to eight 8-bit elements).
- *processing element* and functional unit where to execute the instruction.
- type of operation to be performed.
- *source table* status: it can be valid (`ST_VALID`) or not valid (`ST_NOT_VALID`).

Push

The push operation is not controlled by the combinational control logic of the *source table*: it is the *destination table* that sends a push control signal to the *source table* in order to store a new instruction in the entry pointed by the tail counter. Once the instruction is stored, the entry is marked as `ST_VALID`.

Pop

If the entry pointed by the *source table* head counter has a `ST_VALID` status, then the control logic checks the type of the instruction: if it is an arithmetic operation, it sends a *valid* signal to the *arithmetic unit* inside the correct *processing element* and, if the latter asserts a *ready* in the same clock cycle, the instruction is sent to it for the execution and the status of the entry is set as `ST_NOT_VALID`; if it is a load-store operation, the entry is directly set as `ST_NOT_VALID`, since these instructions must be executed by the *load-store unit* and not by the *processing element*.

3.6.2 Destination Table

It is a FIFO controlled by a combinational control logic, which is firstly allocated by the *operand requester* and then it receives the results of arithmetic instructions waiting to be sent to the *vector register file* for the write-back operation. There is

one *destination table* for each lane, such that each one receives results from the corresponding *arithmetic unit* inside the *processing element*. Each *destination table* sends the results to the corresponding lane inside the *vector register file*.
Considering the data structure, each *destination table* is a circular FIFO buffer where each entry stores the following main information:

- destination register's index and bank for instructions with a vector destination.
- `LANE_WIDTH` data for the result of the instruction.
- *destination table* status: it can be
  - `DT_NOT_VALID` : the entry is not valid.
  - `DT_VALID`: the entry contains a valid content after it is allocated by the *operand requester*, so it waits for the result of the instruction.
  - `DT_READY_WB`: the entry has received the result of the instruction and it waits for the write-back operation (if it has a vector destination).
  - `DT_WB_COMPLETED`: the commit of the instruction stored in the entry has been executed.

Push

If the entry pointed by the tail counter has a `DT_NOT_VALID` status, the *ready* signal is asserted. If the *valid* signal coming from the *operand requester*'s lane is asserted in the same clock cycle, a new instruction is saved in the free entry with the status `DT_VALID`.
At the same time, a push control signal is sent to the *source table* and *position index table*, such that they can store the new instruction too.
The *valid* signal from the *operand requester*'s lane is sent only to the *destination table* and not also to the *source table* because they share the same tail counter so, if there is a free entry in one buffer, the same happens for the other. So the *valid* is sent to the *destination table* and the latter propagates the push control signal to the *source table* and *position index table*.

Write back

When one entry is in the `DT_VALID` status, three possibilities open up:

- if the instruction has a vector destination type, once the entry receives the result of the instruction, its status becomes `DT_READY_WB` so it is ready for the write-back. A note to highlight is that the *destination table* is always ready to receive the results of instructions and a *result address logic* is used to select the correct entry of the *destination table* where to store the results.

- if the instruction has a scalar destination type, once the entry receives the result of the instruction, its status becomes `DT_WB_COMPLETED` since it does not have to write the result in the *vector register file* but it has to send it to the scalar core.

- if the instruction is a load-store, once the entry receives a signal notifying that the *load-store unit* finishes performing the operation, the status is set as `DT_WB_COMPLETED`. This happens because the *load-store unit* directly performs the write-back operation.

At this point, if the entry pointed by the write-back counter falls into the first category having a `DT_READY_WB` status, a *valid* is sent to the *vector register file*'s lane and, if the latter asserts a *ready*, the write-back is executed and the entry status is set to `DT_WB_COMPLETED`.

Pop

The pop operation is not directly controlled by the combinational control logic of the *destination table*. If the entry pointed by the *destination table* head counter is in `DT_WB_COMPLETED` status, there are three possibilities:

- if the instruction has a vector destination type and the entry stored the result related to the last `LANE_WIDTH` register section involved in the instruction, a signal of instruction completion is sent to *source and destination table counters*, since all elements of the destination register have been written in the *vector register file*.

- if the instruction has a vector destination type and the entry stored the result not related to the last `LANE_WIDTH` register section involved in the instruction, a signal of write-back completion is sent to *source and destination table counters* since other elements of the destination register must be written in the *vector register file* (they are in other entries of the *destination table*).

- if the instruction has not a vector destination type (either it has a scalar destination or it is a load-store instruction), a signal of instruction completion is sent to *source and destination table counters*. A note to highlight is that, for instructions with a scalar destination type, the scalar result is sent to the output of the *destination table* so that it can be sent to the CDB at the interface with the scalar core.

Then, the *source and destination table counters* analyzes these signals and can send a pop control signal to the *destination table*: when it is asserted, the status of the entry pointed by the *destination table* head counter is set as `DT_NOT_VALID`.

3.6.3 Position Index Table

It is a single FIFO buffer that stores information in common to all lanes and it has the same number of entries of the *source table* and *destination table*. In particular, it contains the index of the ROB entry and the index of the *instruction status table* entry where the instruction is stored.
It uses the tail counter of the *source table* and *destination table* to point to the first free entry and the *destination table* head counter to point to the oldest one. The data of the entry pointed by the *destination table* head counter is sent at the output of this buffer.
It does not have a control logic, since the push control signal arrives from the *destination table* and the pop operation coincides with the one of the *destination table*.
When the *position index table* receives the push control signal, the two information sent by the *operand requester* is saved in the entry pointed by the tail counter. When the instruction pointed by the *destination table* head counter is completed, meaning that the *destination table* asserts the instruction completion signal, the *position index table* sends to the CDB the index of the ROB entry and it sends to the *instruction status table* the index of the entry where the completed instruction is stored. At the same time, the *source and destination table counters* exchanges handshake signals with the CDB and it sends a completion signal to the *instruction status table*.

3.6.4 Source and Destination Tables Counters

This module contains all counters needed to point to the entries of the *source tables*, *destination tables* and *position index table*. These are controlled by a combinational control logic, which enables them depending on the requests of contiguous units and on the status of *source tables* and *destination tables*.
The counters present inside this module are:

- **Tail counter**: it is in common with all *source tables*, *destination tables* and *position index table*. It is used to point to the first free entry of the FIFOs.
- **Source table head counter**: it is in common with *source tables* and it points to the entry holding the oldest instruction.
- **Write-back counter**: it is in common with all *destination tables* and it points to the entry holding the instruction that must execute the write-back operation.
- **Destination table head counter**: it is in common with all *destination tables* and it points to the oldest instruction.

The control combinational logic drives the counters in this way:

- For the tail counter enable, the control logic monitors the handshake between the *destination tables* and the *operand requester*. When there is at least one *valid* signal from the lanes of the *operand requester* and all *destination tables* assert a *ready*, the tail counter is enabled.

- For the *source table* head counter enable, the control logic monitors the handshake between the *source tables* and the *processing element*. When there is at least one *valid* signal from a *source table* and all the *arithmetic units* inside the *processing element* assert a *ready*, the *source table* head counter is enabled.
  For load-store instructions, the control logic checks if at least one entry pointed by the *source table* head counter has a valid content and if the operation is a load-store; in this case, the counter is enabled.

- For the write-back counter enable, the control logic monitors the handshake between the *destination tables* and the *vector register file*. When there is at least one *valid* signal from a *destination table* and all the lanes of *vector register file* which received the *valid* assert a *ready*, the write-back counter is enabled.
  For load-store instructions, the control logic waits for a signal coming from the *load-store unit* which states the write-back completion for the instruction pointed by the write-back counter. Once this happens, the counter is enabled. This ensures a correct in-order commit, avoiding WAW hazards between arithmetic instructions and load-store instructions.

- For the *destination table* head counter enable, the control logic monitors the status signals sent by the *destination tables*, described in paragraph 3.6.2. If the *destination tables* with a valid content send a write-back completion signal, the counter is enabled and a pop control signal is sent to them. Otherwise, if they send an instruction completion signal, the control logic sends a *valid* to the Common Data Bus (CDB) at the interface with the scalar core in order to inform it that the vector instruction has been completed. When the CDB asserts a *ready*, the counter is enabled and a pop control signal is sent to the *destination tables*. At the same time, a signal of instruction completion is sent to the *instruction status table* so that the instruction can be removed from the latter.

3.7 Processing Element

The *processing element* is the module responsible for the arithmetic instruction execution. The number of PEs is parametric and it can be set through the `NUM_PE` parameter.
The *processing element* is divided into lanes, such that each one contains an *arithmetic unit* which processes in parallel different elements of a vector. Each *arithmetic unit* is made up of multiple ALUs, multipliers and dividers.
In addition to the *arithmetic units*, the *processing element* contains also a single *reduction unit* performing the reduction operations.

The *processing element* receives a *valid* signal from each *source table* storing a valid data. This signal arrives at a decoder that, depending on the type of the functional unit in charge of executing the instruction, propagates it either to the *reduction unit* or to the set of ALUs, multipliers or dividers inside the *arithmetic unit*. If the functional unit is ready, the instruction can start its execution.
An important note is that reduction operations need all elements of a vector, so the *reduction unit* receives the *valid* signal and the data from *all* the source tables. Once the instruction is completed, each *arithmetic unit* sends a *valid* signal and the result to the corresponding *destination table*.
Considering the narrowing instruction, the *arithmetic units* execute it with double parallelism, so the resulting vector has elements with $\texttt{EEW} = 2 \cdot \texttt{SEW}$, as it is discussed in section 3.4.2. Since narrowing operations require the destination vector to have elements with `EEW` = `SEW`, the results coming from the ALU_0 of all *arithmetic units* are sent to a *narrowing compressor* which halves the size of the elements. Then, only for the ALU_0, there is a multiplexer in order to select the results to be sent to the *destination tables* from either the compressed data or from the direct output of ALU_0. The results of the other ALUs are directly sent to the *destination tables* without passing through the *narrowing compressor* since they cannot execute narrowing operations. This allows reducing the area and power consumption by having only one *narrowing compressor* for the ALU_0 and not one for each ALU.
For the reduction operations, the *reduction unit* sends both the *valid* and the result to all *destination tables*. In the section 3.7.2, related to the latter unit, this point is discussed with further details.

A block scheme of the *processing element* is shown in the figure 3.20; in the following sections, the *arithmetic unit* and the *reduction unit* are described.

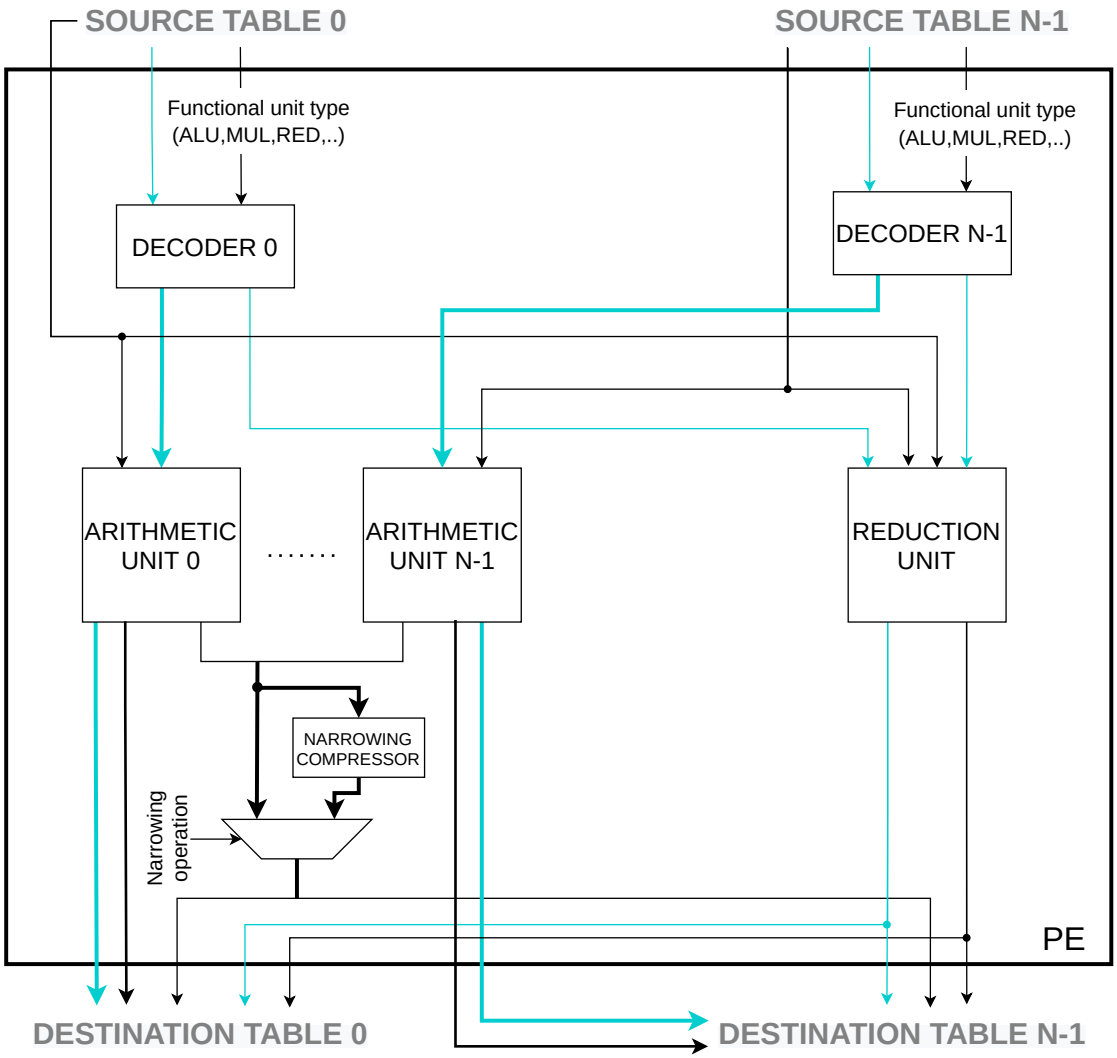

Figure 3.20: *processing element* block diagram

3.7.1 Arithmetic unit

This module is replicated for all lanes, such that each *arithmetic unit* processes a subset of vector elements.
Each *arithmetic unit* is made up of a set of ALUs, multipliers and dividers. The number of these operators is parametric and it can be set through the `NUM_ALU_PER_LANE`, `NUM_MUL_PER_LANE` and `NUM_DIV_PER_LANE` parameters.
In addition to ALU operations, multiplications and divisions, it is necessary to execute the MAC instructions. They take the form $result = \pm\ (op1 \cdot op2) + op3$, so they require both one multiplication and one addition. In order to execute them, it is used the last multiplier and the last ALU.

If there is a new instruction to be processed, the *arithmetic unit* receives three *valid* signals, one for each functional unit type (ALU, MUL or DIV). Depending on the operation to be executed, only one of them will be asserted at a time.
Each *valid* signal is sent to a decoder that, looking at the index of the functional unit assigned to the instruction, propagates it to the correct operator. If the latter is ready, the instruction coming from the *source table* can start its execution.
Once the operator computes the result, it sends a signal of instruction completion to the *destination table*; the latter, which is always ready, will store the data.
For the MAC instructions, it is necessary to first execute the multiplication, so, among the three valid inputs, the multiplication one is asserted. Then, the decoder propagates it to the last multiplier, which executes the multiplication. At this point, instead of sending a completion signal to the *destination table*, the multiplier sends a *valid* signal to an arbiter connected to the last ALU. This arbiter selects either the data at the output of the last multiplier or the data coming from the *source table*. In the case of MAC instructions, the first one has higher priority so the result of the multiplication is sent to the last ALU, which can finish the instruction by performing the addition. Finally, the ALU can send the result to the *destination table*.
The choice of using only the last multiplier and the last ALU for the MAC operations is made in order to not have a complex connection network between all ALUs and all multipliers allowing all combinations of use. This would require more arbiters and multiplexers, increasing the area and power consumption. Certainly, if the number of MACs to be processed would be considerably greater than other types of instructions, a solution could be to enable other operators for its execution.

A block scheme of the *arithmetic unit* is shown in figure 3.21.

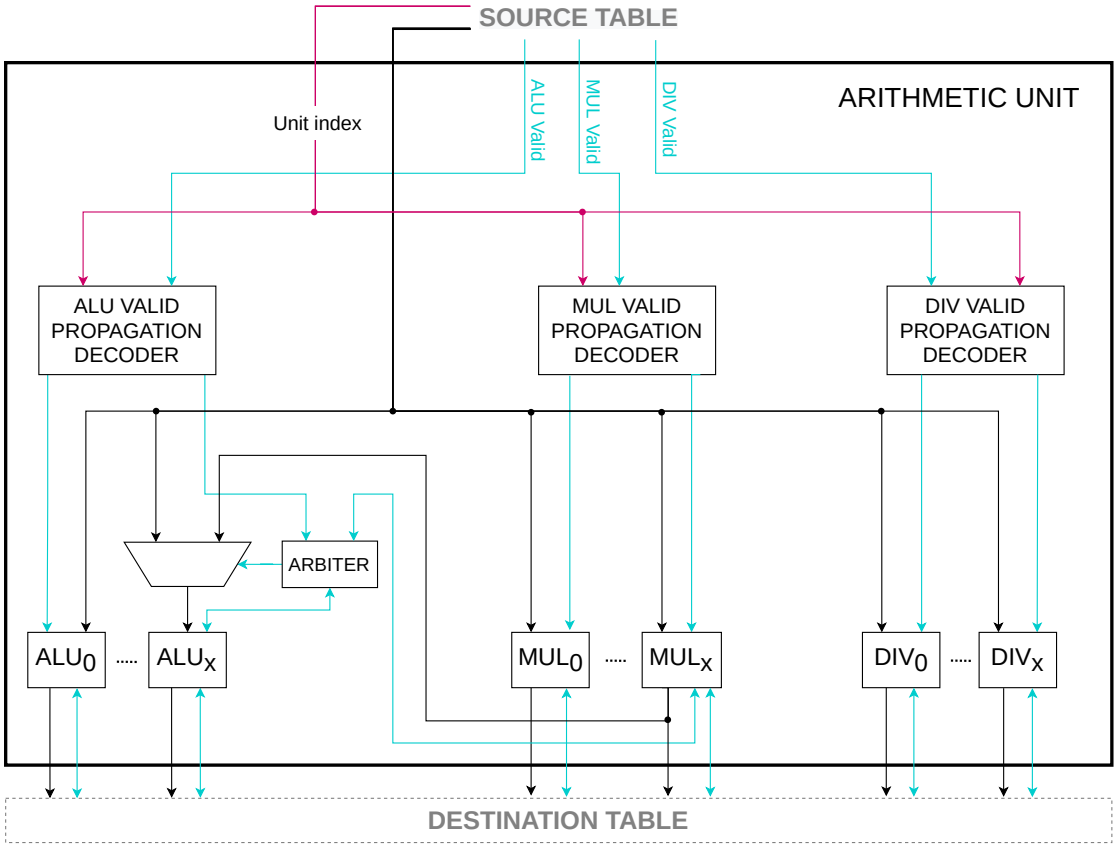

Figure 3.21: *arithmetic unit* block diagram

ALU

The ALU performs arithmetic instructions like additions, subtractions, comparisons, logic, and shift operations (see the section 3.1.2 with the supported instruction subset).
As already explained, the data coming from the *source table* with the source operands[8] is `LANE_WIDTH` wide, so it can contain from one 64-bit element to eight 8-bit elements. The aim is to process these elements in parallel producing a `LANE_WIDTH` wide result with the same number of elements of the input. In order

[8]Considering the operands, instructions have

- `vs2` as the first operand.
- `vs1`, `rs1` or `imm` as the second operand. In the last two cases, the value is replicated to all elements of the input vector operand (`vs2`).

to do that, it is instantiated:

- one 64-bit ALU to process one 64-bit element.
- two 32-bits ALUs to process two 32-bit elements.
- four 16-bits ALUs to process four 16-bit elements.
- eight 8-bits ALUs to process eight 8-bit elements.

Depending on `EEW`, only one set of them is enabled for each instruction. Inside the selected set of ALUs, the correct number of operators is enabled depending on the number of elements to be processed: for example, if we have `EEW` = 8 bits and in one `LANE_WIDTH` section only four elements must be processed, only four ALUs out of eight are enabled inside the set containing eight 8-bit ALUs.
This is not a definitive solution, as the fact of implementing four sets of ALUs of which only one will be active at a time, leads to a great waste of area. A future optimization will be to implement a single run-time reconfigurable ALU able to perform sub-word parallel computations. This means that, giving a fixed parallelism of the ALU, it can process in parallel multiple elements with parallelism that is smaller than the maximum one. An example of such operators is proposed in [14].

At the output of the ALU there is a *spill cell* to store the result of the operation. The spill cell is a module used to interrupt a combinational handshaking path. It contains two registers and one multiplexer to choose which of the two registers' content send to the output, all handled by a control unit. Normally, if the downstream hardware is ready to accept new data, the spill cell sends it a valid signal in the next cycle after it receives some data (i.e., a *valid* signal) from the upstream hardware. If in a certain cycle, the downstream hardware lowers its *ready* signal, the incoming data from the upstream hardware is still stored in the second register of the spill cell and the *ready* output for the upstream hardware is lowered in the next clock cycle and until the downstream hardware becomes ready again. At this point, the oldest data inside the spill cell registers is sent to the downstream hardware, and the spill cell becomes ready again.
Currently, in the case of the ALU, the spill cell is used as a normal output register, since the *destination table* is always ready to accept new data. When there is a new instruction to be executed, the *valid* coming from the *source table* is sent to the spill cell and the *ready* output of the spill cell is sent to the *source table*.
The instruction is processed by one set of ALUs with a latency of one clock cycle. The result at the output of each set of ALUs is sent to a multiplexer, which selects the correct one depending on `EEW`. Then the data arrives at the spill cell; at the next clock cycle, the latter sends it to the *destination table*, along with a completion signal.

The block diagram of the ALU is represented in figure 3.22.

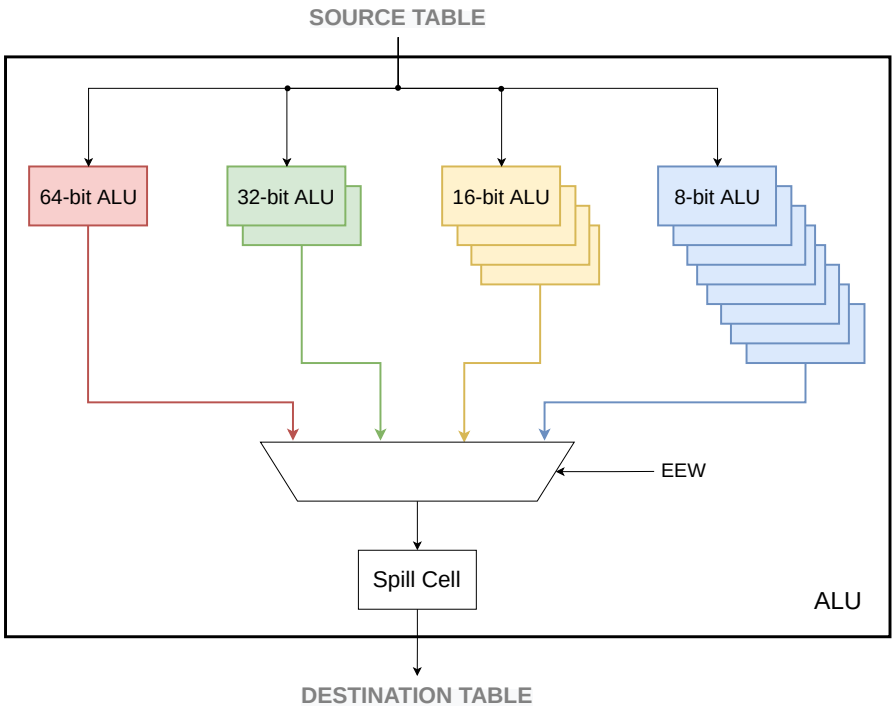

Figure 3.22: ALU block diagram

Multiplier and divider

The multiplier and divider are discussed together since they are managed in the same way, with the only difference being the performed arithmetic operation.
The structure is similar to the one of the ALU, so there are four sets of multipliers (dividers) for the different element widths (refer to the previous paragraph 3.7.1 for the complete description).
The difference with respect to the ALU is that the multiplier and divider have a latency greater than one clock cycle:

- The multiplier, which performs signed and unsigned multiplications, is described in a behavioral way (with the '$*$' operator) and a set of pipeline registers are positioned at the output so that retiming can be performed during the synthesis (the synthesizer can move the position of registers to reduce the critical path of the multiplier). Each set of multipliers has a pipeline depth that depends on the size of the multiplier: it is more likely that the number of registers increases with the increasing multiplier's size. The number of pipeline registers can be set through parameters.

- The divider is taken from [20]. It is a serial divider[9] with a latency equal to the data width of the dividend and divisor. So the four sets of dividers have a latency equal to the four possible element widths.

Each set of multipliers (dividers) has another pipeline with the same latency as the operators. This pipeline propagates information like the index of the *destination table* where the result must be stored and the *valid* signal to be sent to the output spill cell, communicating the instruction completion. In this way, the result of the operation and the data at the output of this pipeline are synchronized: the first one goes at the input of the multiplexer and the second one goes at the input of an encoder. The latter selects the input with the *valid* asserted and it drives the selection signal of the multiplexer. In this way, the data with the correct element width is selected and this is sent to the spill cell.

The block diagram of the multiplier (divider) is represented in figure 3.22.

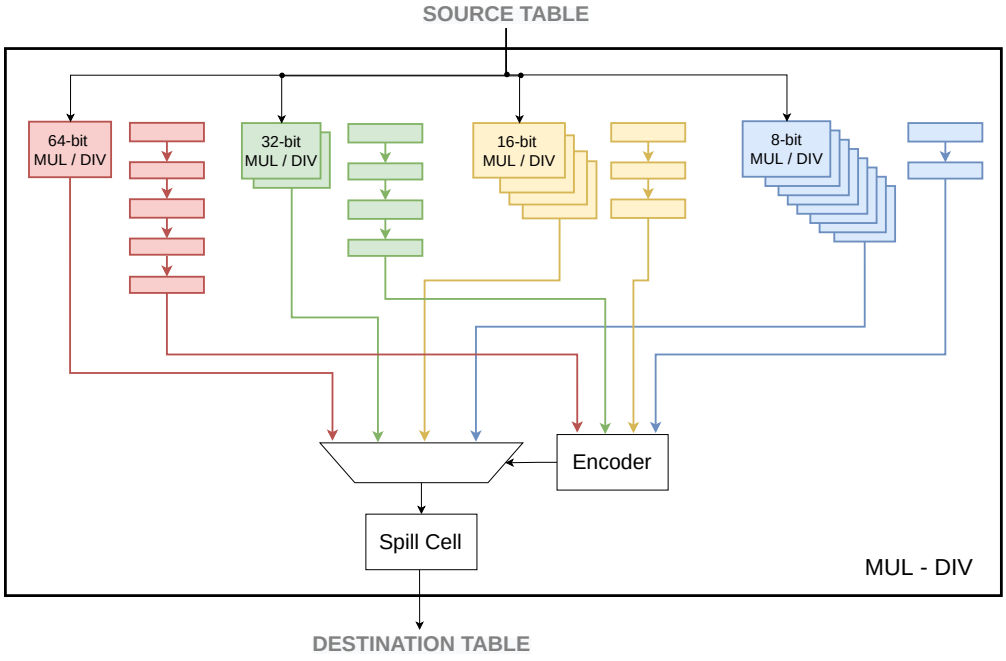

Figure 3.23: Multiplier and divider block diagram

[9]The choice of a smaller, serial divider implementation better suits an edge-oriented microarchitecture, especially in applications where division is seldom used.

3.7.2 Reduction unit

The *reduction unit* is in charge of performing reduction operations.
These instructions take a vector register group and a scalar held in element 0 of a vector register; they perform a reduction using some binary operators to produce a scalar result in element 0 of the destination vector register. So, the type of the instruction is like `vd`[0] = op(`vs1`[0], `vs2`[∗]).
The supported reduction operations are addition (both single-width and widening variants), logic operations (AND, OR, XOR), minimum, and maximum.

Since the instruction requires all the elements of a vector, the *reduction unit* receives the *valid* signal and the data from all *source tables*. A note to highlight is that all the required elements of `vs2` can be stored in multiple entries of the *source tables* if the configuration of the processor considers multiple banks (refer to figure 3.18 for an example).
Once the *reduction unit* finishes processing all the elements, the module sends both the *valid* and the result to all *destination tables* even though the instruction must only update the element 0 of `vd`, which is contained in lane 0. This is not a problem because, for a reduction, *destination table* 0 has the entry related to that operation with a valid status, while the other *destination tables* have the same entries not valid; so, only the first one will effectively store the result.

Going into details of the data structure, there are four single reduction operators to process the four different element widths. Only one of them will be enabled for each instruction, depending on `EEW`.
Each operator has a shift register at the input with a number of registers equal to

$$\texttt{NUM\_LANE} \cdot \frac{\texttt{LANE\_WIDTH}}{\texttt{EEW}} \tag{3.11}$$

where the second factor represents the number of elements in one `LANE_WIDTH` register section.
This is used to store the elements (related to `vs2`) belonging to one bank of all lanes, coming from one entry of all *source tables*. So, the data coming from the *source tables* having the *valid* signals asserted, are pushed in the shift register.
Then, at the beginning, if data come from the first bank, the elements `vs2`[0] and `vs1`[0] are processed; later, at each clock cycle, the other elements of `vs2` are shifted by one position and the one at the output is processed by the operator together with the result of the previous iteration (which is stored in a register).
So, this unit iterates until all elements inside the shift register are processed. Then, if not all banks have been handled, it waits for other *valid* signals from the *source tables*. Certainly, the *reduction unit* has a high latency equal to the number of elements to be processed (`VL`), since it does not exploit the parallelism of operators

and lanes.
In parallel to each set of reduction operators, there is a small pipeline made up of two registers. It is used to propagate information like the index of the *destination table* where the result must be stored and the *valid* signal to be sent to the spill cell. It is loaded when the first bank is processed and it is enabled when all the elements of all banks have been processed except the last one. In this way, in the following cycle, the final result of the reduction and the data at the output of the pipeline are synchronized: the first one goes at the input of the multiplexer and the second one goes at the input of the encoder. From this point, the same behavior of the multiplier and divider is followed (refer to section 3.7.1).

A block diagram of the *reduction unit* is shown in figure 3.24.

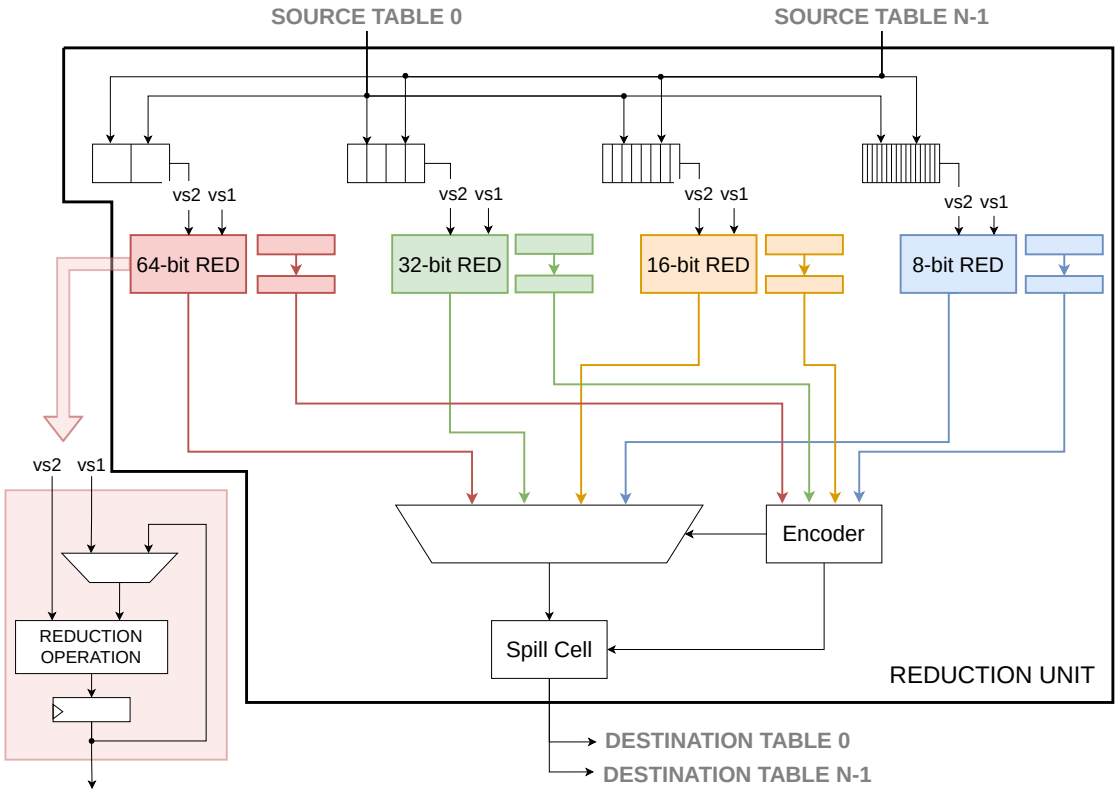

Figure 3.24: *reduction unit* block diagram

3.8 Load-Store Unit

The *load-store unit* is in charge of performing strided load-store operations (indexed operations are not supported yet). This type of instructions provide that the first memory element to be accessed is at the base address specified by `rs1`; then, the subsequent elements are at address increments given by a byte offset specified by `rs2` (stride). The size of the elements to be accessed in memory (`EEW`) is encoded directly in the instruction.
In addition, there is another type of operations, named load-store segment instructions, which move multiple contiguous fields in memory to and from multiple vector register groups. The `nf` parameter contains one less than the number of fields per segment (in other words, one less than the number of register groups involved in one instruction) and the `VL` parameter gives the number of segments to move, which is equal to the number of elements transferred to each vector register group. To be more clear, an example is shown in figure 3.25.

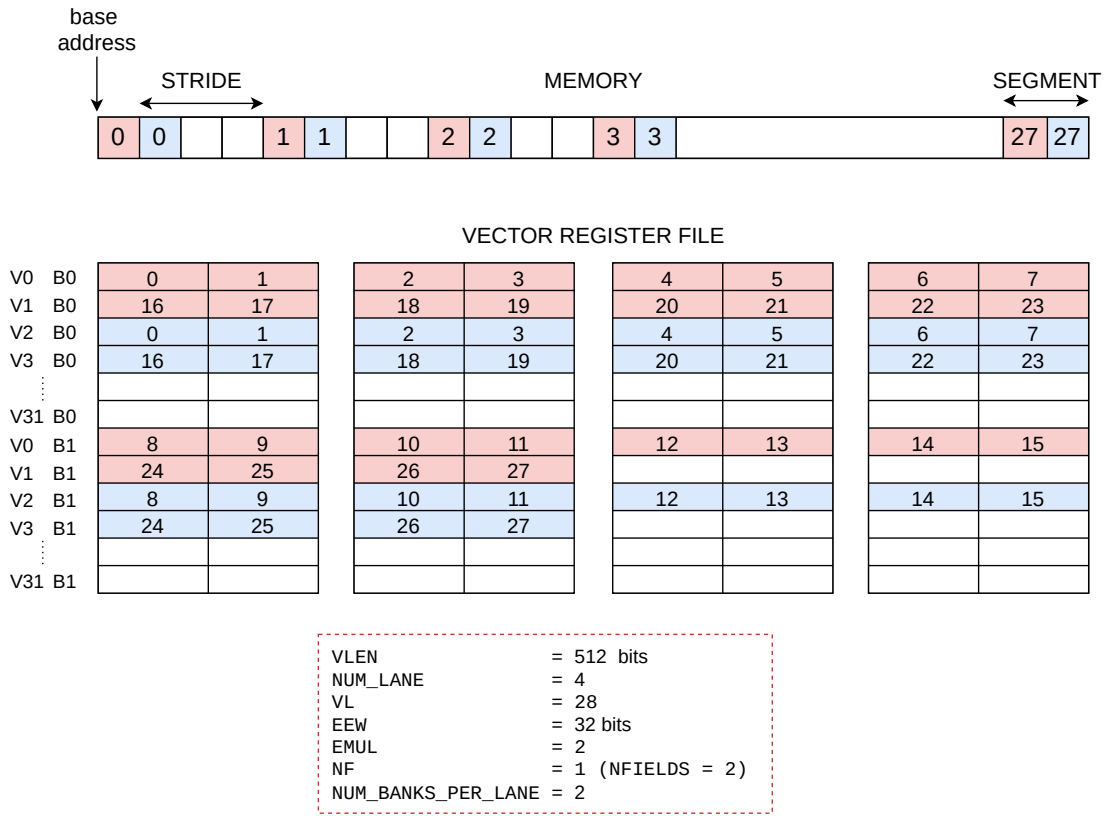

Figure 3.25: Example of strided load-store operation

Here, memory contains the elements equally spaced by the stride value. Then, `nf` parameter is equal to 1, meaning that each segment contains two fields: the first one belongs to the first vector register group (made up of registers v_0 and v_1, since `EMUL` = 2), while the second one belongs to the second register group (made up of registers v_2 and v_3). So, with this segment operation, 28 elements are transferred for each register group, involving a total of 4 registers.
Obviously, if `nf` is equal to 0, this means that a single field is contained in each segment and then it becomes a normal load-store operation that involve a single vector register group.
As already mentioned, one important note is that, considering the possible combinations of `EMUL` and `nf`, the maximum number of registers accessed in one instruction cannot be greater than 8.

3.8.1 Load-store emulator

The *load-store unit* has been implemented with an emulator which cannot be synthesized (refer to the section 3.1.2 for further details about this choice). It executes strided load-store instructions, interacting with a memory emulated with a SystemVerilog class and an ASCII file. The latter contains a fixed number of lines, each of them containing a hexadecimal 64-address and a byte word.
The memory class is made up of attributes and methods used for the memory read and write operations, which involve one bank related to all lanes of a vector register.
The two main methods inside the memory class are:

- `ReadBank()`: it is used for load operations and it accesses memory for reading a number of elements (each at the effective address given by the base address plus the stride) that fits in one bank related to all lanes of a vector register.

- `WriteBank(data)`: it is used for store operations and it accesses memory for writing elements belonging to one bank related to all lanes of a vector register. The elements to be stored are in the `vs3` register group and a note to highlight is that the *operand requester*, at each time, sends all banks of a single register belonging to a group.

Going into further details about the implementation, the emulator receives the instructions from the *load-store instruction buffer* inside the *operand requester*; for the load operations, it directly performs the write-back in the *vector register file*, without passing through the *destination tables*. Once the instruction is completed, the emulator sends a completion signal to the *destination tables*.
The load-store emulator is controlled by a Moore FSM, whose flow chart is reported in figure 3.26.

At the beginning, the control unit is in a sampling state waiting for a valid instruction to be executed, so the *ready* signal towards the *operand requester* is asserted. Once the latter sends a *valid* signal, two branches open up:

- if it is a **load instruction**, the control unit goes into a waiting state and it remains there until the load can be executed. This is necessary to avoid WAW hazards between arithmetic instructions and load-store instructions. In fact, the first ones are committed by the *destination table*, while the second ones by the *load-store unit*: in order to ensure a correct in-order commit, each load instruction cannot start its execution until all the previous arithmetic instructions have already finished the commit in the *vector register file*. So, once the write-back counter of the *destination table* points to the entry containing the load instruction, it means that the load is authorized to access the *vector register file* and then it can start its execution.
  At this point, the control unit enters to a bank request state where the emulator calls the `ReadBank()` method of the memory class, which returns data to fit in one bank related to all lanes of a vector register. Then, the control unit sends a *valid* signal to the *vector register file* and, when the latter asserts a *ready*, the write-back is executed and the remaining number of elements to be loaded is updated (`VL` is decremented by the number of loaded elements).
  Then, some checks are executed in order to know if the instruction is completed:
  - if not all `VL` elements have been loaded (so $\texttt{VL} > 0$), the following parameters used to correctly access the *vector register file* are updated:
    * if not all banks of a single register have already been read from memory, the bank index is incremented
    * if all banks have already been read from memory, the bank index is cleared and the register index is incremented (new register of the group).

    After, the control unit returns to the bank request state.
  - if all elements have been loaded (so $\texttt{VL} = 0$), the control unit checks the `nf` value and, if it is greater than 0, it means that there is a new vector register group waiting for other `VL` elements to be loaded. So, parameters are updated: the bank index is cleared, the register index is incremented and the remaining number of elements is restored to `VL`. At this point, the control unit returns to the bank request state. If `nf` is equal to 0, it means that the instruction is completed, so the control unit sends a signal of instruction completion to the *destination tables* and it returns to sample a new instruction.
- if it is a **store instruction**, the control unit enters into a write bank state where the emulator calls the `WriteBank(data)` method of the memory class,

passing one bank of `vs3` as input parameter. Then, the remaining number of elements to be stored is updated (`VL` is decremented by the number of stored elements) and some checks are executed in order to know if the instruction is completed:

- if not all `VL` elements have been stored (so `VL` > 0):
  - if not all banks of a single register have already been written in memory, the bank index is incremented in order to take the following bank of `vs3` (already present in the *load-store unit*).
  - if all banks have already been written in memory, the bank index is cleared and the control unit waits for a new register of the group coming from the *operand requester*.

  After, the control unit returns into the write bank state.
- if all elements have been stored (so `VL` $= 0$), the control unit checks the `nf` value and, if it is greater than 0, it means that a new vector register group must be written in memory. So, the bank index is cleared, the number of elements is restored to `VL` and the control unit waits for a new register coming from the *operand requester*. After, it returns into the write bank state. If `nf` is equal to 0, it means that the instruction is completed, so the control unit sends a signal of instruction completion to the *destination tables* and it returns to sample a new instruction.

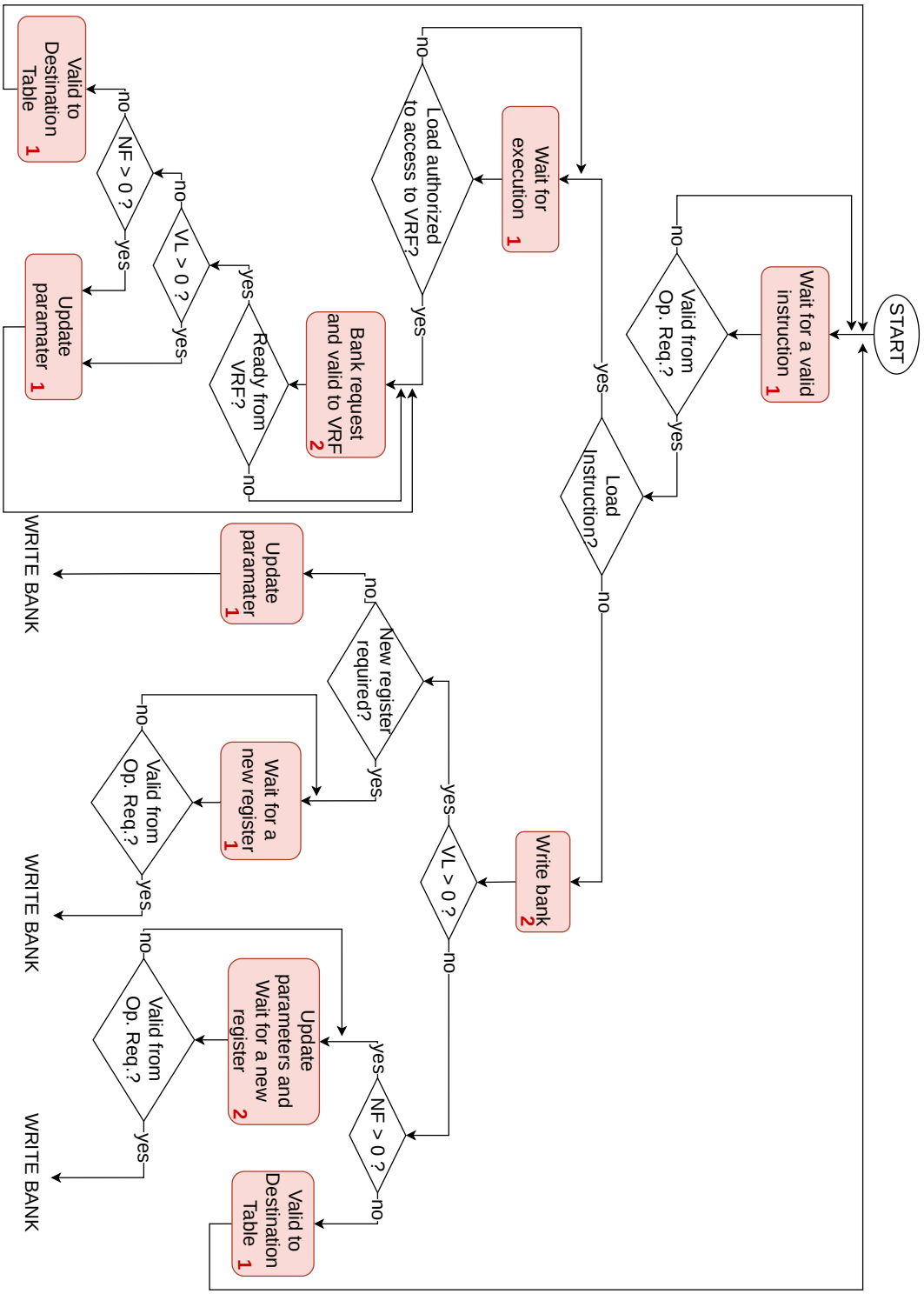

Figure 3.26: Flow chart of the control unit inside the *load-store unit*

3.8.2 Load-store unit preliminary design

In order to test and work on the processor, the load-store emulator has been used. However, a preliminary design for a synthesizable *load-store unit* has been developed and it is discussed in this section. This is a first draft, not described in SystemVerilog yet, which can be used as a starting point for a future, optimized implementation.
Since there are four types of load-store operations which can be executed depending on the addressing mode, the *load-store unit* is made up of four sub-units:

- Load Strided Unit
- Store Strided Unit
- Load Indexed Unit
- Store Indexed Unit

The *valid* signal sent by the *load-store instruction buffer*, notifying a new instruction, arrives at a decoder which propagates it to the correct sub-unit depending on the addressing mode of the incoming instruction. Then, if the *ready* of the sub-unit is asserted, the instruction can be executed.
All four modules interface with a L1 data cache: load units perform requests to read cache lines; store units execute instructions in two phases, so they first read the required cache lines and, after the latter are modified by writing the values required by the instruction, they perform requests to write them in the cache (so each store operation is made up of a first load and then by a store).
Since all sub-units need to send requests to the cache, there is an arbiter that selects which of them is authorized to access by propagating its *valid* signal to the cache; then, the arbiter receives the *ready* signal from the cache stating if the latter is ready to accept the request. On the other hand, the cache will send a *valid* signal to the sub-unit once it is ready to send the cache line.
For load operations, the two load sub-units perform the write-back in the *vector register file*, so also in this case there is an arbiter to select which of them can access the *vector register file*.
Once the instruction is completed, the sub-unit that executed it sends a completion signal to the *destination table* along with the index of the entry in which the instruction is stored.
The block scheme of the *load-store unit* is reported in figure 3.27.

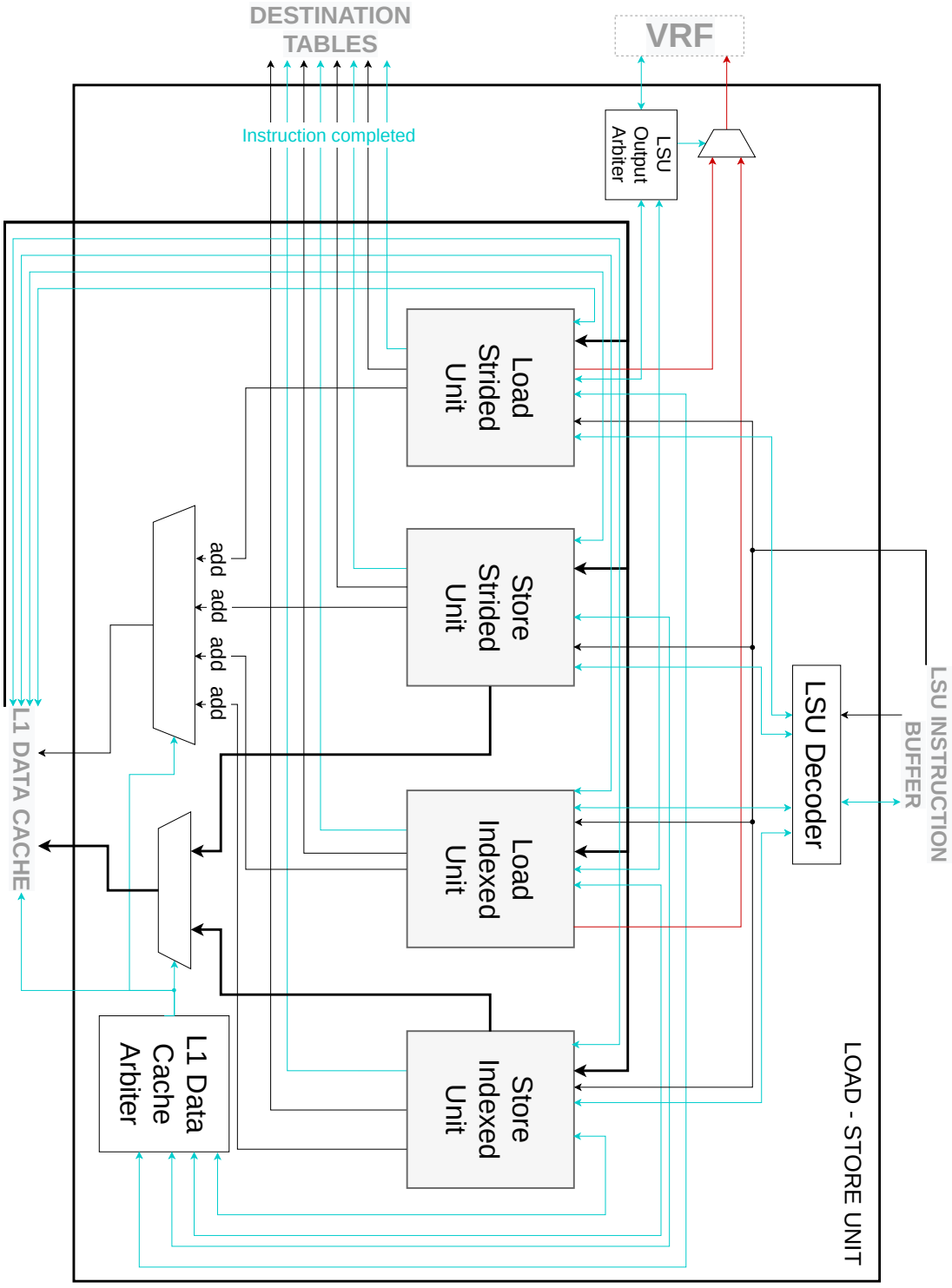

Figure 3.27: *load-store unit* block diagram

Going into further details of the arbiters, let's consider the L1 data cache arbiter (a block scheme is reported in figure 3.28): it has to select which sub-unit can access the cache. In this case, it is important to respect an access order given by the program in order to not have memory hazards (RAW or WAW).
For this purpose, each load-store instruction is processed in order by the *load-store instruction buffer* inside the *operand requester* and here it is marked with a tag value (refer to 3.4.2 to know how this is performed). In the *load-store unit*, there is a counter (head counter), whose output value is equal to the tag value of the oldest instruction to be processed. So, once each sub-unit sends the request to the arbiter, the latter propagates to the cache only the *valid* coming from the unit processing the instruction which has its tag value equal to the head counter value (this means that it is the oldest instruction to be processed). Once the oldest instruction does not have to access the cache anymore, the head counter is incremented, so that another sub-unit with the tag equal to the new value of the head counter can access. In this way, the order of the requests follows the one of the instructions.
A note to highlight is that a single instruction often performs multiple requests to the cache, depending on how many elements are involved and where they are positioned in memory. Then, the head counter must be incremented after the request related to the last cache line needed by the oldest instruction is accepted. So, together with the address of the cache line, each sub-unit outputs a flag stating if that address is the last one needed. When the flag is set and the handshake with the cache completes, the head counter is enabled.
Then a similar behavior must be followed by the arbiter which selects the instruction allowed to perform the write-back in the *vector register file*.

In order to give a description of components inside the sub-units, the load strided unit is taken as a reference, considering that the other sub-units have modules which behave similarly (for example, independently from the operation, it is necessary to compute the addresses of the cache lines, so all sub-units will have an address generator that will calculate the addresses in a different way depending on the addressing mode).
So, the load strided unit is made up of the following components:

- **Address generator**: it computes all the addresses of the cache lines required by the instruction. So, considering the base address specified by `rs1` and the stride specified by `rs2`, it computes the addresses of the lines containing all `VL` elements. Most likely, depending on the stride, the elements are spread in multiple cache lines that are not necessarily contiguous (for example, if the stride is very large, there are some cache lines which do not contain any element to be read). Then, the *address generator* outputs only the addresses of the cache lines containing valid elements for the current instruction. In particular, if the stride is larger than the cache line size, the number of required

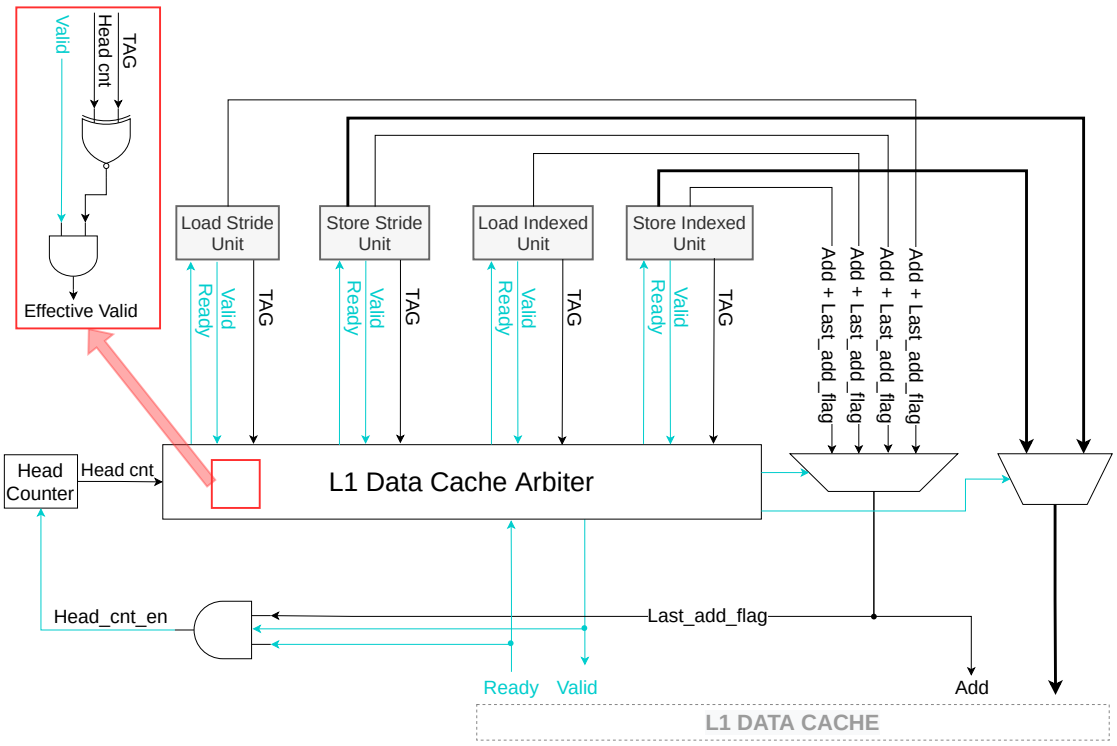

Figure 3.28: L1 Data Cache arbiter

lines is equal to `VL` (each element is stored in a different line); otherwise it is equal to

$$\left\lceil \frac{(\texttt{VL} - 1) \cdot stride + (\texttt{nf} + 1) \cdot \texttt{EEW\_b}}{cache\ line\ size} \right\rceil \tag{3.12}$$

where `EEW_b` is `EEW` expressed in byte.
When there is a new instruction to be executed, the *address generator* does not start its computations until it receives a start signal from the *data generator buffer*. At that point, it generates a new address; then, it sends a *valid* signal to the *address-data table* and, if the latter asserts a *ready*, it can compute the next one.

- **Address-data table**: it is a CAM-like structure which receives the addresses from the *address generator* (it is ready to receive a new address when there it has a free entry); then, it makes the requests to the cache arbiter sending a *valid* in order to read a cache line and, when the cache is ready, it stores the line near its address.
  The *address-data table* has a similar behavior to a CAM: the *data generator* sends to the *address-data table* the address of the required cache line; then,

the incoming address is compared with the ones present in the *address-data table* and if there is a match and the corresponding cache line has already been taken from the cache, the latter is sent to the *data generator* together with a hit signal asserted.

- **Data generator**: it parses a cache line byte-by-byte and, considering the stride, `nf` and `EEW`, it selects the correct bytes of the elements to be read. Then, it sends one byte at a time to the *data generator buffer* in order to store it in the correct vector destination register.
  In particular, the *data generator*, for each byte inside each element (which is `EEW` wide) belonging to each segment (each segment contains `nf` +1 elements and the total number of segments is equal to `VL`), it recomputes the address of the cache line which contains it. Then it sends the address to the *address-data table*: if the latter returns a hit, the *data generator* takes the cache line and selects the correct byte, considering that the last bits of the address contain the offset of the cache line where the byte is positioned. Then, it sends the latter together with a *valid* signal to the *data generator buffer*. Regarding the *valid* signal, the *data generator* has 8 valid output signals, one for each vector destination register that can receive the byte (as already explained, each vector instruction cannot involve more than 8 registers). For each byte, only the valid corresponding to the register that needs to receive it is asserted. When there is a new instruction to be executed, the *data generator* does not start its computations until it receives a start signal from the *data generator buffer*. At that point, it begins the process described above until all `VL` segments are sent to the *data generator buffer*. Certainly, considering that many parameters can change (`nf`, `EEW`, stride and `VL`), this solution is the easiest to handle. On the other hand, parsing iteratively one byte at a time leads to a very long process. For this reason, it was decided to combine this process with a way to speed up the common case of copying an entire cache line within a whole vector destination register (this happens when `EEW_b` = *stride*). So in this situation, when the cache line is taken from the *address-data table*, it is directly addressed to the correct vector register in the *data generator buffer*, without having to send one byte at a time.

- **Data generator buffer**: it is made up of eight buffers, one for each potential destination register. Each buffer has a size equal to the physical vector register size (`VLEN`) and it can receive either one byte at a time or the entire cache line from the *data generator*. Each buffer has its write enable which comes from the *valid* signal sent by the *data generator*: this allows writing the incoming data in the correct buffer and, consequently, in the correct destination register. When there is a new instruction to be executed, the *valid* signal coming from the *operand requester* is sent to the *data generator buffer*. If the latter has all

buffers empty, it asserts the *ready* and sends a start signal to both the *address generator* and *data generator* in order to process the new instruction. Then, it waits for the data coming from the *data generator* and, after receiving the valid signals from the latter, it stores it in the correct buffers. When all `VL` segments have been stored, the *data generator buffer* starts to execute the write-back in the *vector register file*. So, considering one buffer at a time (so one destination register), it divides the content of the buffer in banks (the number of banks in one physical register is equal to `VLEN/LANE_WIDTH`) and it outputs one bank for each lane at a time along with a *valid* signal. The latter is received by the arbiter that selects which unit can access the *vector register file*. If the *data generator buffer* receives a *ready*, then the banks are written in the *vector register file* and a new write request can be performed for the other banks.
When the write-back is completed for all destination registers, a completion signal is sent to the *destination table* and the *data generator buffer* returns to wait for a new instruction.

A block scheme of the load strided unit is reported in figure 3.29.

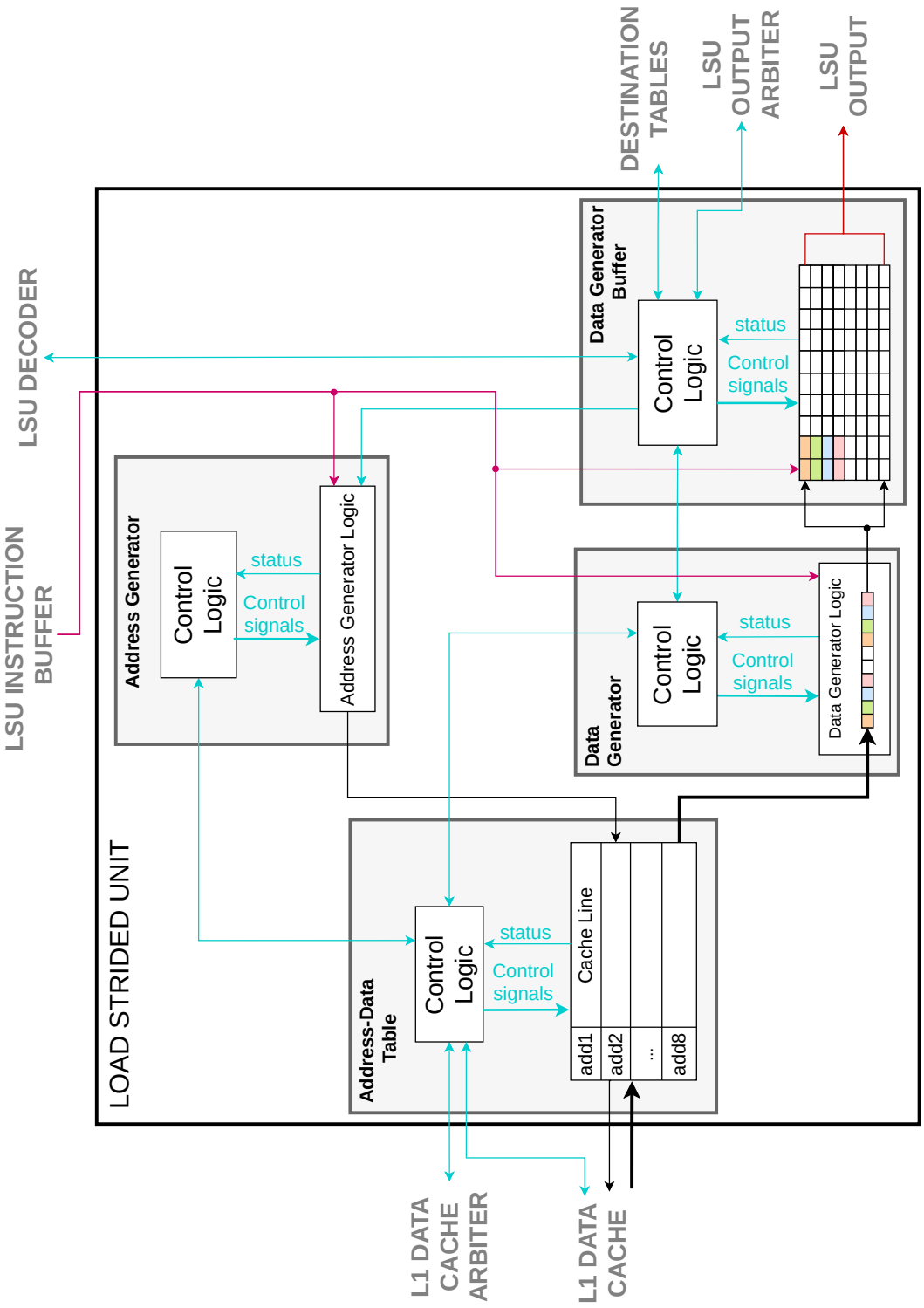

Figure 3.29: Load strided unit block diagram

Chapter 4

Testing and synthesis

4.1 Testing methodology

In order to perform the functional verification of the vector processing unit, each internal component was tested after it was designed and implemented.
So, a testbench that applies input stimuli is dedicated to each unit and to each component within it (refer to sections 3.2 to 3.8 for a description of all the modules inside the vector processing unit). The most relevant output signals are manually verified by printing them on a simulation log; moreover, the waves of the simulator are analyzed in order to verify if the content of the data structures and the internal signals match the expected ones at each clock cycle.
In addition to the testbenches, some SVAs are inserted in the code in order to automatically test specific and potentially critical situations. In fact, assertions allow verifying at run-time some *properties* that a module should satisfy given some boundary conditions. If a property is not met, an output error message may pointing to the failing property is shown to the user, and the simulation can be automatically interrupted.
One example is reported in the listing 4.1, which reports some of the assertions from the *instruction status table* and, in particular, the ones related to the push operation (for the explanation of the *instruction status table* behavior refer to the section 3.3). Considering the first one, the property states that, if in a certain clock cycle the entry of the *instruction status table* pointed by the tail counter has a `NOT_VALID` status and the *valid* from the *instruction decoder* is asserted, then at the next clock cycle the entry assumes a `READY_HAZ_CHECK` status because a push is executed. This property must always be verified so if, during the simulation, it is violated because the state of the entry is not updated, it means that there are some errors in the implementation. The second assertion, instead, states that, if the entry pointed by the tail counter has a status different from `NOT_VALID`,

at the next clock cycle the tail counter value is unchanged, meaning that, if the *instruction status table* is full, no new instructions can be accepted and therefore the tail counter cannot be incremented.
Assertions obviously must not be synthesized since they are needed only to test the functionalities of modules, so the `‘ifndef` directive is inserted to prevent the synthesizer from analyzing the test code.

```
// ----------
// ASSERTIONS
// ----------
`ifndef SYNTHESIS

/* Push operation => a new data is inserted in the entry
   pointed by the tail counter */
property p_push;
    @(posedge clk_i)
    ist_fifo[tail_cnt].ist_status == NOT_VALID &&
       decoder_valid_i |-> ##1
    ist_fifo[$past(tail_cnt)].ist_status == READY_HAZ_CHECK
endproperty
a_push: assert property (p_push);

/* Fifo full => Push cannot be executed */
property p_fifo_full;
    @(posedge clk_i)
    ist_fifo[tail_cnt].ist_status != NOT_VALID |-> ##1
    $past(tail_cnt) == tail_cnt
endproperty
a_fifo_full: assert property (p_fifo_full);

`endif
```

Listing 4.1: Assertion example in the *instruction status table*

Finally, when the vector processing unit has been integrated by connecting all of its modules, a final testbench has been created. Since the processor is not integrated in the scalar core yet, the testbench must perform the same operations that the scalar core would do. In particular, it takes the instructions to be sent to the processor from a binary file and, at each clock cycle, it verifies if the vector processing unit is ready to accept a new instruction: if this is the case, it sends to the processor the instruction along with a *valid* signal; otherwise, it must wait until the vector processing unit asserts a *ready*.
In the same way, the testbench directly provides the dynamic parameters (`SEW` and `VL`) which should be set by the configuration instructions modifying the CSRs (refer

to section 3.1.1 for further details). So, before sending instructions with a new configuration of parameters, the testbench must wait for all running instructions to complete (i.e. when all entries of *instruction status table* are `NOT_VALID`). This allows to correctly complete the older instructions with the previous configuration while the newer ones are temporarily stalled.
At the end of the testbench, the updated memory content is printed on a file in order to verify if it contains the expected values.

4.2 Case study

In order to evaluate the performance of the vector processing unit, the matrix convolution is used as case study. This is a representative workload of machine learning algorithms as it is used, for example, in Convolutional Neural Networks (CNNs), which are widespread in a variety of applications like image and speech recognition, videogames, and robotics.

To give a brief description of the matrix convolution algorithm, let's consider the example in figure 4.1, which will be used for the processor performance evaluation. Considering as a starting point a 16x16 input matrix (containing the data to be elaborated) and an 8x8 filter matrix, the algorithm multiplies, element by element, the filter by an 8x8 submatrix taken from the 16x16 input matrix (colored in light blue in the figure); then, the resulting 64 products are accumulated together with a reduction operation to find one element of the resulting matrix, which is a 9x9 matrix. The same operations are applied to every subsequent submatrix by *sliding* by one position at a time, so every element of the output matrix is computed. In this case, 81 iterations are needed to compute the 81 elements of the 9x9 output matrix.

The vector processing unit is tested with the algorithm described above (the pseudo-code is reported in the appendix A), considering 16-bit elements for all matrices (`SEW` = 16 bits) and performing 81 iterations, each of them operating on 64 input elements (`VL` = 64).
The filter is loaded in a vector register at the beginning of the execution; in each iteration, an 8x8 input submatrix is loaded from the memory and multiplied by the filter to generate the 64 output products that are accumulated with a reduction operation in order to find one resulting element, which is successively stored in memory.
Since the indexed load-store operations are not implemented yet, the memory, instead of containing the original 16x16 matrix, contains all the submatrices already formed, so there's no need to gather only the first elements of each involved row. All matrices contain elements with random values.

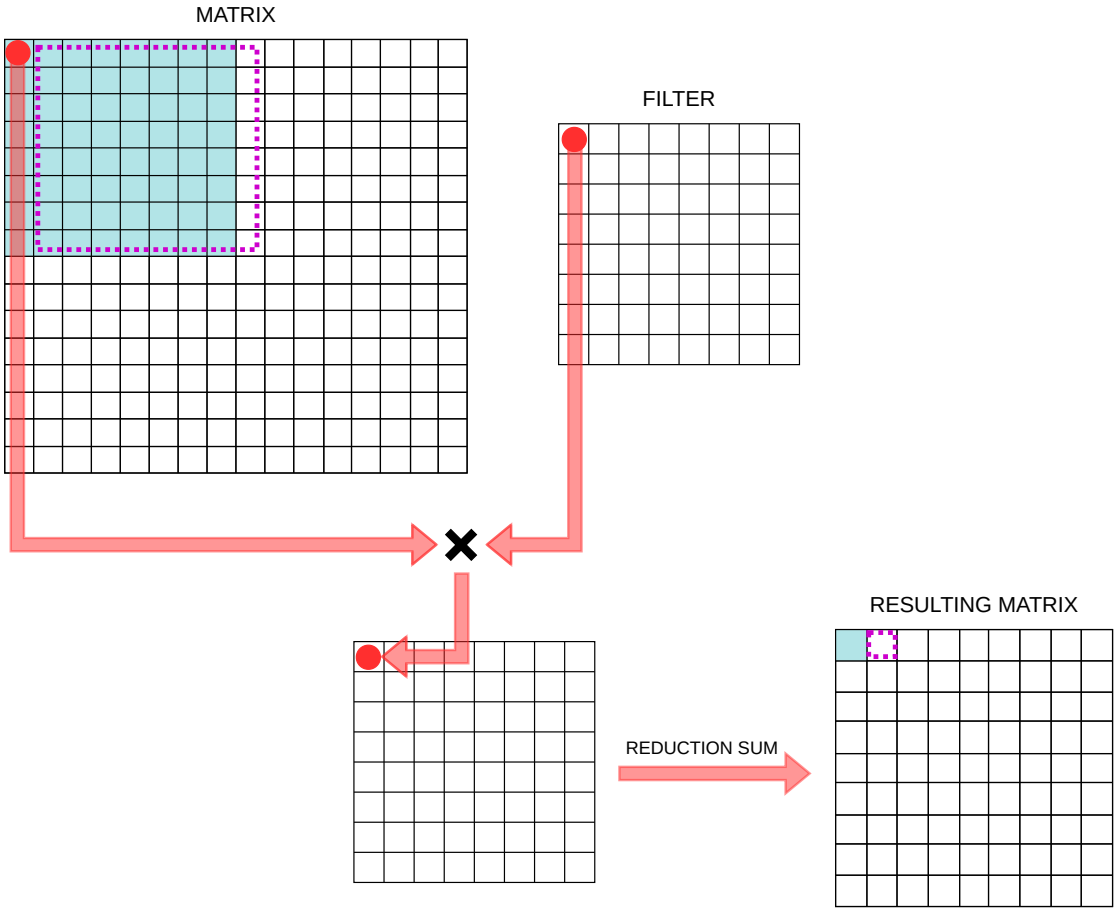

Figure 4.1: Matrix convolution example with 16x16 input matrix, an 8x8 filter and a 9x9 resulting matrix.

4.3 Testing and synthesis results

The performance achieved by the vector processing unit is evaluated considering the results obtained from the simulation and synthesis[1]. In particular, taking into account that the processor is highly configurable, multiple simulations and synthesis can be run with different combinations of parameters. In this section, the attention is mainly focused on two of them: `VLEN` (number of bits in a single vector register) and `NUM_LANE` (number of lanes); so, the different simulations and synthesis are

[1]All results are obtained from the synthesis of the entire vector processing unit without the *load-store unit*, since the latter is not implemented yet and the emulator is used only for simulation purposes without being synthesizable.

performed with changing the values of the latter. The other parameters, which are listed below, are fixed:

- `NUM_PE` = 1
- `NUM_ALU_PER_LANE` = 2
- `NUM_MUL_PER_LANE` = 2
- `NUM_DIV_PER_LANE` = 2
- `IST_DEPTH` = 8
- `ST_DT_DEPTH` = 16
- `NUM_VRF_PORTS` = 8

4.3.1 Baseline

Let's consider as a starting point a configuration with:

- `VLEN` = 64 bits
- `NUM_LANE` = 1

By setting these values, the resulting architecture can be seen as a (quite inefficient) scalar processor with a 64-bit parallelism. The area and cycle time obtained from the synthesis are reported in the table below.

Area [μm^2]	**Cycle time** [ns]
623 529	1.96

Table 4.1: Results with `VLEN` = 64 bits and `NUM_LANE` = 1.

Certainly, the vector processing unit can work in these conditions but it must be considered that, by fixing `VLEN` to such a low value, the total number of elements that can be processed is limited even exploiting groups of registers (with `EMUL` = 8, it can process a maximum of `VLEN` $\cdot$ 8 = 512 bits). Moreover, the execution takes a long time to complete because the processor can process only a maximum of 64 bits at each clock cycle and so it is obliged to perform a lot of iterations to handle all the data. Most importantly, the latency overhead of the complex control is not compensated by an increased throughput.
The vector processing unit is designed to process a large amount of data (which is

the main characteristic of data-driven systems), using multiple lanes to process as many elements as possible in parallel, so this configuration is definitely not suitable for this purpose. Then, to perform calculations like a scalar processor would, a scalar processor itself is obviously more efficient as it is optimized for control flow with low data parallelism.

Still, such minimal configuration is useful to provide a baseline in terms of resources required by each module inside the vector processing unit. Such analysis is reported in figure 4.2, that shows the contribution of each module to the total area. All the values are obtained from the synthesis of the single modules always considering `VLEN` = 64 bits and `NUM_LANE` = 1.

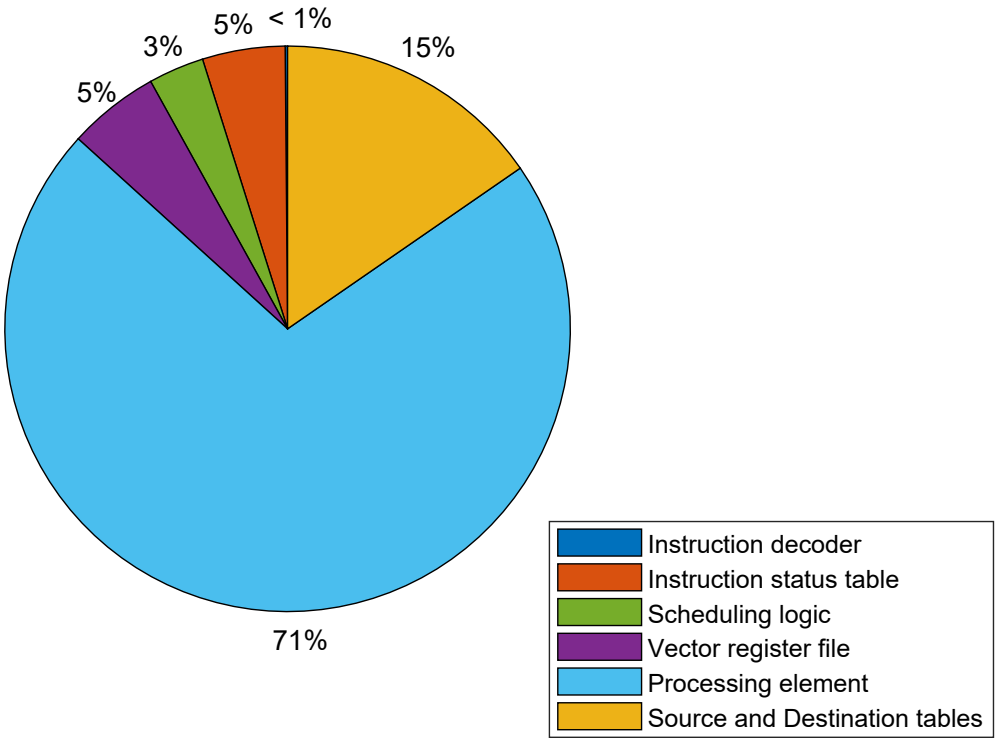

Figure 4.2: Area composition of the vector processing unit considering `NUM_LANE` = 1 and `VLEN` = 64 bits.

As it can be seen from the chart, the module which occupies most of the area is the *processing element*. This is reasonable because the latter contains all the arithmetic operators. However, it must be also considered that, in order to support the four different parallelisms of the elements, four sets of operators are instantiated for each type of arithmetic operator (this is explained in detail in the section 3.7.1), so the area is almost four times greater than the one it should theoretically occupy.

Therefore, the area can be greatly reduced by implementing arithmetic operators able to perform sub-word parallel computations so that a single operator with a fixed parallelism can be used for processing in parallel multiple elements with a parallelism that is smaller than the maximum one.
The area of the single modules is directly proportional to the number of lanes. The only exception is the *instruction decoder* and the *instruction status table*, which instead do not depend on the number of lanes, therefore representing a constant contribution to the total area.
As already mentioned, the *load-store unit* does not appear in the graph since the emulator is not synthesizable.

4.3.2 Results comparison by changing parameters

This section reports the results of sixteen simulations and synthesis, each running with a different parameters configuration. In particular, the same matrix convolution algorithm (see the appendix A for its pseudo-code) is run sixteen times, varying `VLEN` between 128 bits and 1024 bits and `NUM_LANE` between 1 and 8.
Table 4.2 reports the simulation results for all the combinations of `VLEN` and `NUM_LANE`, showing the number of clock cycles necessary to terminate the program and the throughput, expressed as the average number of bits of the input matrix that are processed, on average, in each clock cycle[2].
Table 4.3 reports the results of the synthesis, showing the total area of the vector processing unit and its cycle time (T_{ck}).
For each table, it is reported also the number of banks for each lane of the *vector register file* (`NUM_BANKS_PER_LANE`) and `EMUL`, which are resulting from the choice of `VLEN` and `NUM_LANE`.
In addition, three surface charts are reported below in order to show the trend of data present in the tables 4.2 and 4.3. All of them report a result as a function of `VLEN` and `NUM_LANE`: the first one in figure 4.3 shows the area; the second one in figure 4.4 reports the cycle time and the last one in figure 4.5 shows the throughput.

As expected, increasing `VLEN` and `NUM_LANE` results in a higher **throughput** and thus a lower overall latency, since the possibility of processing many more elements in parallel increases. So, considering the two extremes, we start from a throughput of 0.23 bit/cycle when `NUM_LANE` = 1 and `VLEN` = 128 bits and we arrive to a maximum throughput of 0.45 bit/cycle when `VLEN` = 1024 bits and `NUM_LANE` = 8.
On the other hand, if `VLEN` and `NUM_LANE` increase, the **area** increases as well since more resources are needed in order to process more elements in parallel.

[2]This takes into account all the operations involved in the algorithm (i.e., load, arithmetic, and store instructions).

So, always considering the two extremes, we start from an area of 673 173 µm^2 when `NUM_LANE` = 1 and `VLEN` = 128 bits and reach an area of 4 649 253 µm^2 when `NUM_LANE` = 8 and `VLEN` = 1024 bits.
On the other hand, the **cycle time** remains almost constant for all combinations of `VLEN` and `NUM_LANE`. This is in line with the discussion about the DLP reported in chapter 2: in architectures exploiting DLP, the CPU time can be reduced by maintaining the same clock period while decreasing the number of instructions in the program and adding parallel, independent functional units (lanes). In the case of the designed vector processing unit, the cycle time is on average equal to 2 ns leading to a clock frequency of about 500 MHz.

When `NUM_LANE` = 1, the area is minimized with respect to other cases but the situation that arises is like having a 64-bit scalar processor, which must process long vectors. So, it is certainly not an optimal situation, as the throughput is low (0.23 bit/cycle) and the number of banks in the register file can be very high (since `VLEN` reaches high values), therefore increasing the number of iterations needed to process all elements but also the possibility of conflicts within the register file (since all banks of a single register are mapped on the same access port).

Among the combinations of `VLEN` and `NUM_LANE`, there are some of them which do not make sense to use as they do not bring any advantage either in terms of throughput or in terms of area. This happens in the configurations listed below:

- `VLEN` = 128 bits and `NUM_LANE` = 4
- `VLEN` = 128 bits and `NUM_LANE` = 8
- `VLEN` = 256 bits and `NUM_LANE` = 8

Considering, for example, to fix `VLEN` = 128 bits, the throughput remains constant to 0.32 bit/cycle even increasing `NUM_LANE` from 2 to 4 or 8 since the maximum throughput achievable by the operators is equal to `VLEN`/cycle (meaning that all elements of a vector can be processed in parallel in a single iteration). So, in this case, the disadvantage is that throughput remains constant but the area increases (since `NUM_LANE` increases). This means that, once `VLEN` is fixed, the number of lanes must be set in order to minimize the area while still having the same throughput (in this case the best choice when `VLEN` = 128 bits would be `NUM_LANE` = 2).
Considering the opposite reasoning, if we want to fix `NUM_LANE` = 8, the area remains almost constant even increasing `VLEN` from 128 bits to 256 bits or 512 bits. However, the number of elements that can potentially be processed in parallel is higher than the one contained in a vector then, when `VLEN` =128 bits or 256 bits some lanes are not used. Therefore once `NUM_LANE` is fixed, `VLEN` must be set in order to maximize the throughput while still having the same area (in this case the best choice when `NUM_LANE` = 8 would be `VLEN` = 512 bits).

VLEN [bit]	NUM_LANE	NUM_BANKS_PER_LANE	EMUL	Latency [cycles]	Throughput [bit/cycle]
128	1	2	8	18 013	0.23
256	1	4	4	18 013	0.23
512	1	8	2	18 013	0.23
1024	1	16	1	18 013	0.23
128	2	1	8	12 965	0.32
256	2	2	4	12 965	0.32
512	2	4	2	12 965	0.32
1024	2	8	1	12 965	0.32
128	4	1	8	12 965	0.32
256	4	1	4	10 441	0.39
512	4	2	2	10 441	0.39
1024	4	4	1	10 441	0.39
128	8	1	8	12 965	0.32
256	8	1	4	10 441	0.39
512	8	1	2	9181	0.45
1024	8	2	1	9181	0.45

Table 4.2: Results from the simulations. For each combination, the latency (expressed in clock cycles) and throughput (expressed in bit/cycle) are reported.

VLEN [bit]	NUM_LANE	NUM_BANKS_PER_LANE	EMUL	Area [μm^2]	Cycle time [ns]
128	1	2	8	673 173	1.95
256	1	4	4	736 882	1.87
512	1	8	2	832 838	1.91
1024	1	16	1	1 050 557	1.89
128	2	1	8	1 176 238	1.94
256	2	2	4	1 244 398	2.03
512	2	4	2	1 365 019	1.95
1024	2	8	1	1 584 953	2.04
128	4	1	8	2 255 266	1.96
256	4	1	4	2 247 760	2.01
512	4	2	2	2 374 615	2.08
1024	4	4	1	2 603 050	2.04
128	8	1	8	4 423 913	2.06
256	8	1	4	4 382 766	2.06
512	8	1	2	4 425 330	2.06
1024	8	2	1	4 649 253	2.03

Table 4.3: Results from the synthesis. For each combination, the area and cycle time (T_{ck}) are reported.

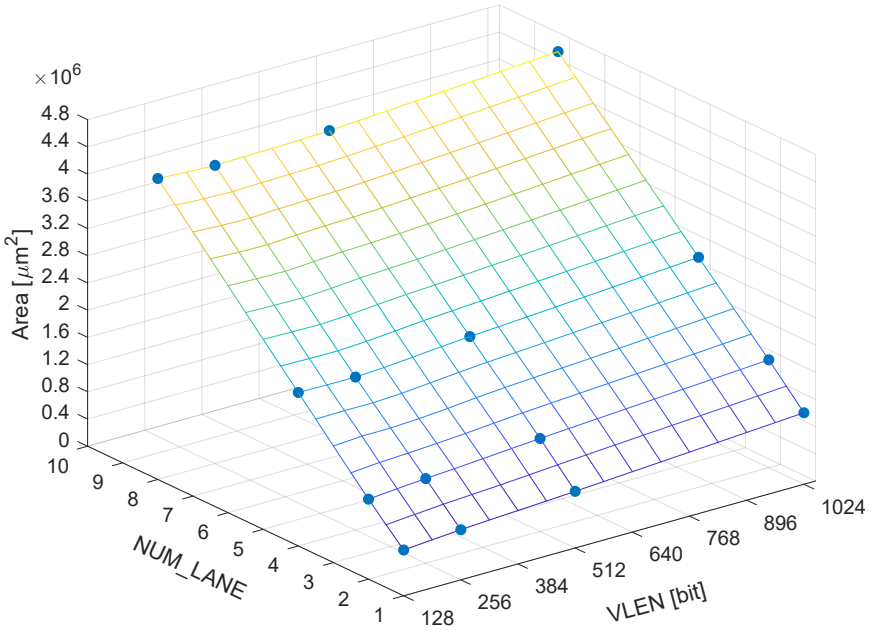

Figure 4.3: Surface chart showing the trend of the area as a function of `VLEN` and `NUM_LANE`.

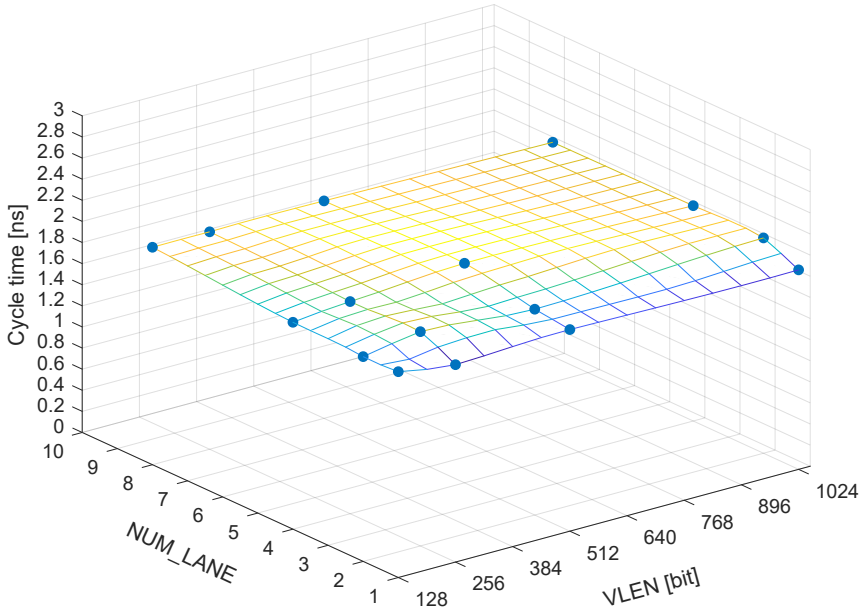

Figure 4.4: Surface chart showing the trend of the cycle time as a function of `VLEN` and `NUM_LANE`.

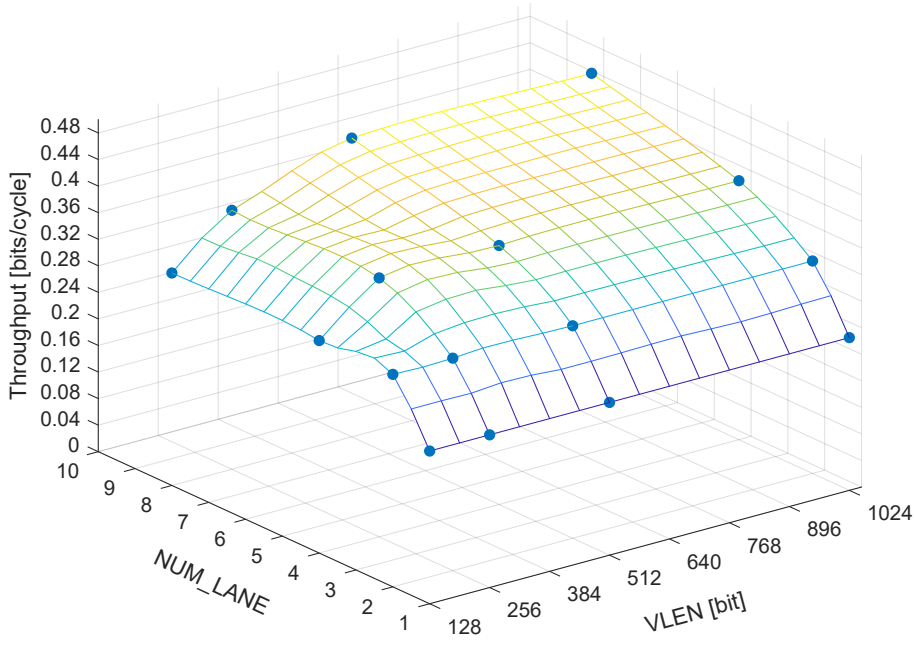

Figure 4.5: Surface chart showing the trend of the throughput as a function of `VLEN` and `NUM_LANE`.

Now, let's consider two specific examples in order to discuss some further details. The first situation considers fixing `VLEN` to 1024 bits: the chart 4.6 shows the trend of the area and throughput by varying `NUM_LANE`. The second situation considers fixing `NUM_LANE` to 2: the chart 4.7 shows the trend of the area and throughput by varying `VLEN`.
Taking into account the first situation, if the number of lanes grows, the area linearly increases (remember that most of the modules inside the processor are made up as many sub-units as the number of lanes). On the other hand, the throughput increases as well since, with a growth of `NUM_LANE`, the number of elements that can be processed in parallel increases. At the same time, the latency to terminate the program decreases since a lower number of iterations is needed to process all elements. This is reflected by the required number of registers banks in each lane of the *vector register file*: increasing the number of lanes, `NUM_BANKS_PER_LANE` decreases and so the time to iterate on all banks decreases as well. Another advantage is that, by having less banks, the possibility of conflicts within the register file is reduced (since fewer register sections are mapped on the same port).

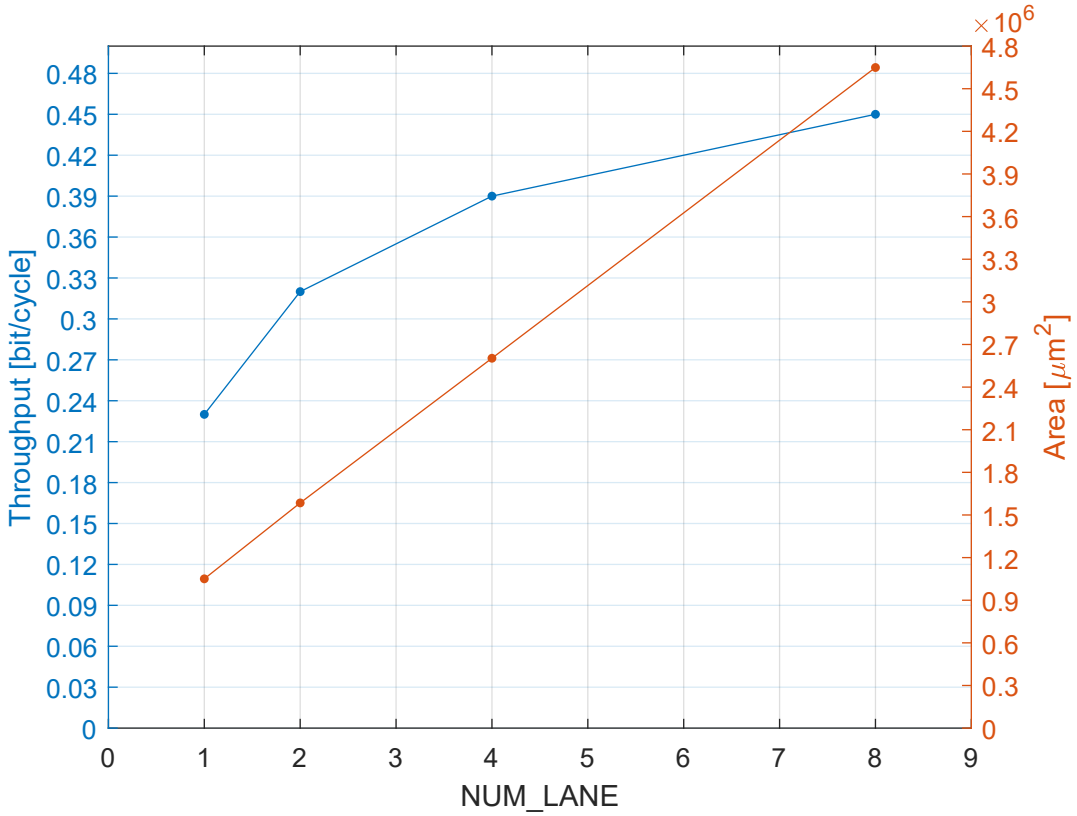

Figure 4.6: Trend of area and throughput by varying `NUM_LANE` while fixing `VLEN` = 1024 bits.

Considering the second situation, if `VLEN` grows, the area increases almost linearly since `NUM_BANKS_PER_LANE` increases (more banks are needed to manage longer vectors). On the other hand, the throughput remains constant at 0.32 bit/cycle, which is fixed by the chosen number of lanes (each lane can process 64 bits at a time). Then, also the latency remains constant and this happens because the number of iterations necessary to process all elements is the same for all values of `VLEN`. To understand this point, it is necessary to look at `NUM_BANKS_PER_LANE` and `EMUL`. For example, a situation considering `VLEN` = 128 bits, `NUM_BANKS_PER_LANE` = 1 and `EMUL` = 8 leads having only one bank for each register, so each register can manage 8 16-bit elements but, in order to process all elements of one vector (64 16-bit elements, as the matrix convolution algorithm asks), it is required to iterate on a group of eight registers (so `EMUL` = 8). On the other hand, when `VLEN` = 1024 bits, `NUM_BANKS_PER_LANE` = 8 and `EMUL` = 1, meaning that `VLEN` value leads having eight banks for each register, so each register can manage 64 16-bit elements

iterating on its banks. Then, a single register is enough to process all elements of one vector (so `EMUL` = 1). In conclusion, the latency is constant because the number of total iterations is the same, either if we iterate on more registers with less banks or if we iterate on less registers with more banks. Certainly having less banks leads to the possibility of reducing conflicts within the register file (since fewer register sections are mapped on the same port), but it is also necessary to consider that `EMUL` cannot be greater than 8 and how many registers are required by the algorithm to be run on the processor.

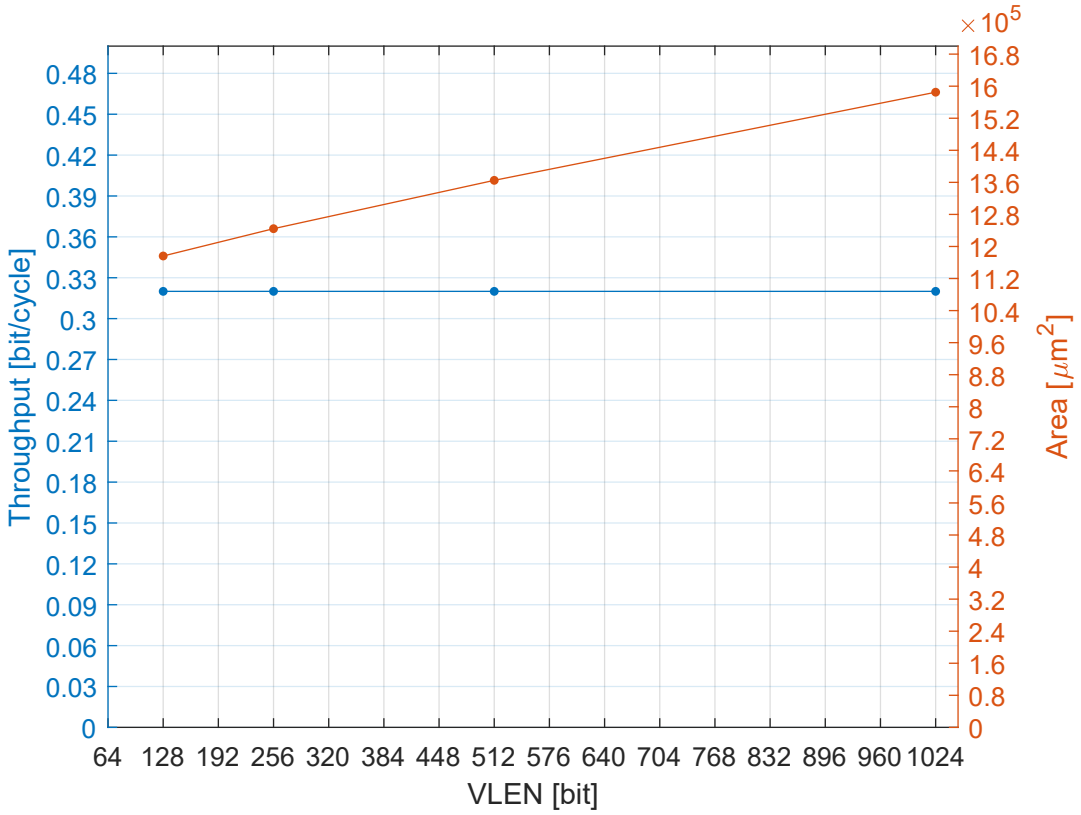

Figure 4.7: Trend of area and throughput by varying `VLEN` while fixing `NUM_LANE` = 2.

Chapter 5

Conclusion

This thesis project led to a first implementation of a vector processor used to accelerate data-parallel applications.
As already discussed in the introduction, a vector processor has the advantage of having high flexibility: it is programmable so it is easy to change functionalities for different application targets. Up to now, attention has been focused on the matrix convolution case study, but other types of algorithms can be surely considered in the future.
The use of the RISC-V vector extension further increases the versatility of the processor: the possibility of setting parameters as the element width (`SEW`) and the number of elements to be processed (`VL`) at run time having a "hardware-agnostic" software, leads to being able to use the processor featuring different vector register sizes (`VLEN`), allowing high performance and great versatility in different application domains.
Moreover, the designed processor is scalable and suitable for a variety of target applications, since many implementation parameters can be set at synthesis time: from the number of lanes and vector register size to the number of processing elements and arithmetic operators. The advantage of this structure is that, depending on the target performance and power consumption constraints, priority can be either given to throughput, using a higher number of lanes and arithmetic operators, or to area and power efficiency, having a less performing yet smaller processor.
This represents a preliminary design and future developments will be necessary to implement the missing parts and to optimize those already present.

5.1 Future improvements

In this section, some suggestions are given for possible future work:

- One missing part that should be implemented is the load-store unit. Until now, the emulator is able to manage strided operations and it is used only for simulation purposes without being synthesizable. So, it is necessary to implement this unit in order to support both strided and indexed memory operations.

- The designed vector processing unit supports only a subset of the RISC-V vector extension so it is useful to design the hardware able to manage the other kind of instructions. The types of non-supported instructions (which are listed in the section 3.1.2) would involve massive interaction between lanes in order to exchange data between them. The best solution is to implement a separate unit that is able to handle these operations (like masking ones or permutations) in an optimized way.

- The vector processing unit needs to be integrated into the scalar core *LEN5*, so that the latter could send the instructions along with the scalar operands to the vector processor; then, it could manage the configuration instructions by modifying the CSRs in order to send to the vector processing unit the updated dynamic parameters (`SEW`, `VL` and `LMUL`). At the moment, all these operations are directly performed by the testbench.

- Some optimizations can be done in order to improve the performance. One point which could be an object of revision is the control unit managing the *arithmetic instruction buffer*, which has 170 states and an average latency of 7 clock cycles. Certainly, this leads the *scheduling logic* to represent a bottleneck of the design considering also the fact that it can manage only one instruction at a time. Then, for this first implementation, the use of this control unit allowed to easily debug the structure but it can be optimized in the future.

- Another point always related to the possible optimizations concerns the allocation of the *source tables*. Now they have one common tail counter and the problem occurs when the *scheduling logic* is not ready to send a new instruction only for some lanes because only some lanes of the *vector register file* are not ready for the fetch operation. In this case, the *scheduling logic* waits until the fetch operation is completed for all lanes before sending the instruction to the *source tables* (even dough some lanes have already the operands ready). At this point, all *source tables* receive synchronously the instruction and the tail counter is incremented. This is certainly easy to manage since all data in the different lanes are synchronized but, in this way

the independence that lanes should theoretically have, is partially lost. So, a solution can be to use one independent counter for each *source table* so that, once the *scheduling logic* has some ready data, it can send it immediately to the correct *source table*. In this way, the independence of lanes would be fully exploited.

- The last point concerns the arithmetic operators. Now, in order to support the four different parallelisms of the elements, four sets of operators are instantiated for each type of arithmetic operator (this is explained in detail in the section 3.7.1). A future work can be the implementation of arithmetic operators able to perform sub-word parallel computations so that a single operator with fixed parallelism can be used for processing in parallel multiple elements with parallelism that is smaller than the maximum one. In this way, the overall area of the vector processor could be greatly reduced.

Appendix A

Matrix convolution pseudo-code

This appendix reports the pseudo-code describing the matrix convolution algorithm which is used for the evaluation of processor performance.
As already discussed in chapter 5, the data to be elaborated is contained in a 16x16 input matrix and the filter is in an 8x8 matrix. Each iteration considers an 8x8 submatrix taken from the input matrix and it computes one element of the resulting 9x9 matrix. All elements have 16-bit parallelism (`SEW` = 16 bits) and then 81 iterations (each of them managing 64 16-bit elements) are needed to compute the 81 elements of the resulting matrix.
The pseudo-code listed in A.1 shows that the filter is first loaded in v_0. Then, at each iteration, a submatrix is loaded in v_9 and the multiplication between v_0 and v_9 is performed (a note to highlight is that, depending on `EMUL`, the registers involved in these operations can be from v_0 to v_7 for the filter and from v_9 to v_{16} for the submatrix). At this point, the multiplication resulting vector (potentially from v_{18} to v_{25}) is subjected to the reduction operation.
The result of the reduction (i.e. the element of the resulting matrix) is not immediately stored in memory: the store operation is executed after the computation of six resulting elements, which are temporally stored from register v_{26} to v_{31}. Then, a single segment store strided can be executed, by involving all six registers. The fact of performing a store instruction for a group of resulting elements and not for one element at a time allows reducing the latency of the program. This happens because store instructions have `VL` = 1 (only one element of the register containing the valid result for one iteration must be stored in memory), which is different from the one of the other instructions (e.i. `VL` = 64). So the processor, before executing a store instruction with a new configuration of dynamic parameters, is stalled waiting that all previous instructions with `VL` = 64 terminate. Then, the overall

time in which the processor is stalled is reduced by grouping store operations.

```
// SEW = 16 bits

// Load filter in v0
Load unit-strided (vd = v0, nf = 0) | vl = 64
//Compute the first 78 elements of the resulting matrix
for(i=0; i<13; i++) {
    // Load matrix in v9
    Load unit-strided (vd = v9, nf = 0) | vl = 64
    // Multiply the filter by the matrix
    MUL               (v18 <- v0 * v9)  | vl = 64
    // Sum all elements of the product matrix
    RED SUM           (v26 <- sum(v18)) | vl = 64
    Load unit-strided (vd = v9, nf = 0) | vl = 64
    MUL               (v18 <- v0 * v9)  | vl = 64
    RED SUM           (v27 <- sum(v18)) | vl = 64
    Load unit-strided (vd = v9, nf = 0) | vl = 64
    MUL               (v18 <- v0 * v9)  | vl = 64
    RED SUM           (v28 <- sum(v18)) | vl = 64
    Load unit-strided (vd = v9, nf = 0) | vl = 64
    MUL               (v18 <- v0 * v9)  | vl = 64
    RED SUM           (v29 <- sum(v18)) | vl = 64
    Load unit-strided (vd = v9, nf = 0) | vl = 64
    MUL               (v18 <- v0 * v9)  | vl = 64
    RED SUM           (v30 <- sum(v18)) | vl = 64
    Load unit-strided (vd = v9, nf = 0) | vl = 64
    MUL               (v18 <- v0 * v9)  | vl = 64
    RED SUM           (v31 <- sum(v18)) | vl = 64
    //Store 6 elements of the resulting matrix
    Store unit-strided(v26,,v31, nf = 5)| vl = 1
}
//Compute the last 3 elements of the resulting matrix
Load unit-strided (vd = v9, nf = 0) | vl = 64
MUL               (v18 <- v0 * v9)  | vl = 64
RED SUM           (v26 <- sum(v18)) | vl = 64
Load unit-strided (vd = v9, nf = 0) | vl = 64
MUL               (v18 <- v0 * v9)  | vl = 64
RED SUM           (v27 <- sum(v18)) | vl = 64
Load unit-strided (vd = v9, nf = 0) | vl = 64
MUL               (v18 <- v0 * v9)  | vl = 64
RED SUM           (v28 <- sum(v18)) | vl = 64
Store unit-strided(v26,,v28, nf = 2)| vl = 1
```

Listing A.1: Pseudo-code describing the matrix convolution algorithm

Acronyms

ALU	Arithmetic Logic Unit
AVX	Advanced Vector Extension
AXI	Advanced eXtensible Interface
CAM	Content Addressable Memory
CDB	Common Data Bus
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CSR	Control and Status Register
DLP	Data Level Parallelism
FIFO	First In First Out
FSM	Finite State Machine
GPU	Graphic Processing Unit
ISA	Instruction Set Architecture
MAC	Multiply - Accumulate
MIMD	Multiple Instruction stream Multiple Data stream
PE	Processing Element
RAW	Read After Write
ROB	Reorder Buffer
SIMD	Single Instruction stream Multiple Data stream
SSE	Streaming SIMD Extensions
SVA	SystemVerilog Assertions
SVE	Scalable Vector Extension
VNB	Von Neumann Bottleneck
WAW	Write After Write

Bibliography

[1] *Len5-vector*. URL: https://git.vlsilab.polito.it/risc-v/len5/len5-vector. (accessed: 19.08.2022) (cit. on p. 1).

[2] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: *Proceedings of the IEEE* 105.12 (Nov. 2017), pp. 2295–2329 (cit. on pp. 1, 22).

[3] Neeraj Magotra and Jim Larimer. «Energy Efficient Digital Signal Processing». In: *2010 53rd IEEE International Midwest Symposium on Circuits and Systems* (2010), pp. 1053–1056 (cit. on p. 1).

[4] Mohammed Alrowaily and Zhuo Lu. «Secure Edge Computing in IoT Systems: Review and Case Studies». In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)* (2018), pp. 440–444 (cit. on p. 2).

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. sixth edition. Morgan Kaufmann Publishers Inc., 2017 (cit. on pp. 2, 6).

[6] John Backus. «Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs». In: *Communications of the ACM* 21.8 (1978), pp. 613–641 (cit. on p. 2).

[7] Daniel Dabbelt, Colin Schmidt, Eric Love, Howard Mao, Sagar Karandikar, and Krste Asanovic. «Vector Processors for Energy-Efficient Embedded Systems». In: *Proceedings of the Third ACM International Workshop on Many-Core Embedded Systems* (2016), pp. 10–16 (cit. on p. 3).

[8] *RISC-V International*. URL: https://riscv.org/. (accessed: 08.08.2022) (cit. on p. 3).

[9] *Vector Extension 1.0, frozen for public review*. URL: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0. (accessed: 08.08.2022) (cit. on pp. 3, 27).

[10] *Intrinsics for Intel Advanced Vector Extensions*. URL: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions.html. (accessed: 17.08.2022) (cit. on p. 7).

[11] *Intrinsics for Intel Advanced Vector Extensions 2 (Intel AVX2)*. URL: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2.html. (accessed: 17.08.2022) (cit. on p. 7).

[12] *Intrinsics for Intel Advanced Vector Extensions 512 (Intel AVX-512) Instructions*. URL: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx-512-instructions.html. (accessed: 17.08.2022) (cit. on p. 7).

[13] *What is NEON?* URL: https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/What-is-NEON-?lang=en. (accessed: 18.08.2022) (cit. on p. 7).

[14] Yunan Xiang, R. Pettibon, and M. Margala. «A versatile computation module for adaptable multimedia processors». In: *2006 IEEE International Symposium on Circuits and Systems (ISCAS)* (2006), pp. 4–60. DOI: 10.1109/ISCAS.2006.1692521 (cit. on pp. 8, 69).

[15] *Introduction to SVE*. URL: https://developer.arm.com/documentation/102476/0100/?lang=en. (accessed: 17.08.2022) (cit. on p. 9).

[16] *Introduction to SVE2*. URL: https://developer.arm.com/documentation/102340/0001/Introducing-SVE2?lang=en. (accessed: 17.08.2022) (cit. on p. 9).

[17] Nigel Stephens et al. «The ARM scalable vector extension». In: *IEEE micro* 37.2 (2017), pp. 26–39 (cit. on p. 9).

[18] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. «Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.2 (2019), pp. 530–543 (cit. on p. 10).

[19] Michele Caon. «Design of the execution pipeline for LEN5, a RISC-V Out-of-Order processor». Master's Thesis. Torino, Italia: Politecnico di Torino, Academic year 2018-2019 (cit. on pp. 23, 24).

[20] *Serial divider*. URL: https://github.com/skmtti/div. (accessed:08.09.2022) (cit. on p. 71).