Movement of vector elements inside a de-coupled vector processing unit for high-performance memory operations

Final degree Thesis

Computer engineering specialization

AUTHOR:
Albert Aguilera Dangla

SUPERVISOR:
Miquel Moretó Planas

CO-SUPERVISOR:
Abraham Josafat Ruíz Ramírez

GEP TUTOR:
Paola Lorenza Pinto

January 17, 2023

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Abstract

This thesis is part of the eProcessor project. Within it, the BSC is developing a RISC-V based decoupled vector accelerator. This accelerator must support the execution of vector memory instructions. More specifically, I have worked on the development of a set of modules oriented to the displacement of data between the vector registers and the memory hierarchy, as well as their correct mapping. For this task, it is essential to elaborate a design that is able to meet the requirements faced by the project. This is a first implementation that may receive updates in the future.

Resum

Aquesta tesi forma part del projecte eProcessor. Dins d'ell, el BSC està desenvolupant un accelerador vectorial desacoblat basat en RISC-V. Aquest accelerador ha de suportar l'execució d'instruccions de memòria vectorial. Més concretament, he treballat en el desenvolupament d'un conjunt de mòduls orientats al desplaçament de dades entre els registres vectorials i la jerarquia de memòria, així com el seu correcte mapeig. Per a aquesta tasca, és fonamental elaborar un disseny que sigui capaç de satisfer els requisits que afronta el projecte. Aquesta és una primera implementació que pot rebre actualitzacions en el futur.

Contents

List of Tables

List of Figures

Chapter 1

Contextualization and scope of the project

1.1 Context

This is a final degree thesis, framed within an extensive project, which is composed of many sections and implemented with different technologies that are far beyond the scope of this document and will not be discussed here.
Therefore, it will focused on a subset of the work carried out by the BSC, representative of my internship in the company.

1.1.1 The whole frame

In recent years, HPC in Europe has become a field with strategic interest. [1] describes how the development of this technology, which can be achieved thanks to the common efforts of the members of the union and other private entities, would lead to greater competitiveness in this industry and many other industries that can benefit from this technology.
Flowing this current of thought, in 2018, The European High Performance Computing Joint Undertaking (EuroHPC JU) was created [2]. This is an entity which allows the European Union and participating countries to coordinate their efforts and pool their resources to develop top-of-the-range exascale supercomputers.
Later, in 2021, eProcessor started as a project co-funded by the European Union under the EuroHPC JU initiative [3]. eProcessor is led by BSC which contributes to several work packages (WPs), in providing its knowledge of hardware design, FPGA emulation, benchmarking, and system simulation tools.
BSC leads WP5 where the target eProcessor RTL design is developed. BSC will design the RISC-V vector accelerator for HPC workloads for the case studies scoped by eProcessor. Scoped case studies are: HPC, AI and Bioinformatics. An important requisite is the exploration of different implementations in order to obtain an optimized design in terms of energy efficiency.

1.1.2 My perspective

As stated previously, BSC is working on the development of a vector processor unit (VPU) in eProcessor. This hardware unit works as a co-processor for the out-of-order (OoO) scalar core of the system, the vector instructions dispatched by the core are executed by this unit and the result is sent back to the core.
I have been working in this unit for more than a year now and, for this thesis, my scope is to work on the datapath from the memory accesses of the VPU. More specifically, I have worked in a set of modules that move and shifts the data (vector elements and indexes)

between the vector register file (VRF) and the Request Queue (rqueue) which is connected to Memory Hierarchy through a Network on Chip (NoC).

1.2 Stakeholders

The direct stakeholders of the thesis are the partners of eProcessor. They will get further steps towards the finalization of the whole project, specially the BSC, which will obtain an initial implementation of a part of the hardware needed to execute a memory instruction. Holistically, eProcessor will provide the basis for the development of new technologies; the software, hardware and other produced material will be reused to reduce the costs in future projects.
The indirect stakeholders of my thesis are the members of the scientific community and organizations that will use the future IPs (intellectual properties) developed in eProcessor, for the shake of solving a wide range of computational problems. My work involves one small step in the development of this technology.
Finally, I also add myself to the stakeholders list due to the skills that I will learn and improve during the course of the thesis.

1.3 Justification

1.3.1 Background

"In 1974 Robert Dennard observed that power density was constant for a given area of silicon even as you increased the number of transistors because of smaller dimensions of each transistor. Remarkably, transistors could go faster but use less power. *Dennard scalling* ended around 2004 because current and voltage could not keep dropping and still maintain the dependability of integrated circuits. This change forced the microprocessor industry to use multiple efficient processors or cores instead of a single inefficient processor. Indeed, in 2004 Intel canceled its high-performance uniprocessor projects and joined others in declaring that the road to higher performance would be via multiple processors per chip rather than via faster uniprocessors. This milestone signaled a historic switch from relying solely on instruction-level parallelism (ILP), to data-level parallelism (DLP) and thread-level parallelism (TLP)" [4, p. 4].

Under this paradigm, Single instruction stream, multiple data streams (SIMD) and Multiple instruction streams, multiple data streams (MIMD) become popular.
"SIMD—The same instruction is executed using different data streams" [4, p. 11].
SIMD kind architectures make possible a big exploitation of DLP. [4, p. 282].
"MIMD—Each processor fetches its own instructions and operates its own data" [4, p. 282].
MIMD kind architectures make possible a big exploitation of TLP, and also DLP [4, p. 11].
"Since a MIMD architecture needs to fetch one instruction per data operation, SIMD is potentially more energy-efficient since a single instruction can launch many data operations ... over the next decade the potential speedup from SIMD parallelism is twice that of MIMD parallelism." [4, p. 283]. These are important aspects in favour of the use of the VPU, SIMD based, in eProcessor as the desired characteristics and restrictions about performance and energy can be fulfilled. However, MIMD will also be very important in further steps on eProcessor, as the plan is to add more processors to the system.

Also, it is required to take a look at the methods used to obtain even a better performance of this vpu accelerator.
I want to focus on one optimization applied to this VPU, which consists in a fully decoupled vector architecture, this means that the arithmetic and memory(vstore and vload) datapaths have different physical structures and no interaction at all (except for chaining). The idea of having a fully de-coupled architecture is to allow complete independence between arithmetic and memory operations, reducing the complexity and over-kill control for datapaths which do not require such restriction/control, e.g. the Finite State Machine arbiter inside the Vector Lane and the accesses to the Vector Register File. As we can see in the study [5, p. 9], this can be a good cost/performance optimization for any memory latency because this technique helps hiding that latency.
This unit is also being implemented with some other optimizations like register renaming and some capabilities of OoO, such as partial ordering in the execution of vector loads and stores (out-of-order loads, in-order stores), and will be updated with new ones in the future, like an upgrade in the chaining mechanism.

1.3.2 Available resources

A big plus for this thesis is that, not only do I have access to material that explains and justifies the theoretical concepts that underpin the current VPU implementation and will guide the scoped implementations of this thesis, but also that BSC has useful material to reuse.
BSC is working on another project called EPI, in which a different VPU is being developed. I am taking advantage of EPIs IPs in order to carry out the implementation of eProcessor. For the development of the eProcessor's base vector processing unit, we took the EPI's "Vitruvius" vector processing unit as a baseline, then adapted it to the new architecture requirements, except for the "vector load store unit" (along some other modules) which was developed from scratch.
In addition, the BSC has more than enough economical and material resources to take care of this thesis.

1.4 Scope

1.4.1 Objectives

The main objective of this project is to develop an RTL implementation of the following modules:

1. Store management unit.
2. Store buffer.
3. Index management unit.
4. Index buffer.

The implementation is done in System Verilog (SV). RTL is an implementation approach that can be emulated on a field programmable gate array (FPGA) because it takes into consideration the characteristics of the available hardware. The RTL is also the basis for obtaining an integrated circuit (IC).
The development process includes design, implementation, testing and documentation. Material produced may be updated for following versions of the VPU.

The sub-objectives that will be achieved in the process are:

1. Understand the use of the EDA tools needed in the project. This includes simulation and linting tools.

2. Understand RISC-V ISA and eProcessor scoped RISC-V extensions.

3. Understand the architecture and functionalities of the current VPU design. This includes understanding the process of execution of a memory instruction, making special focus on the modules connected to the focused ones, and its interfaces.

4. Understand differences between EPI architecture and eProcessor, as well as what can be reused.

1.4.2 Requirements

In order to achieve the specified objectives and obtain a satisfying result, first, it is compulsory to specify the requirements of the interfaces and functionality of those modules. In order to achieve this, I will proceed according to the specifications that Abraham, the team lead, is specifying for the current version of the VPU.
What I can really decide is within the modules' implementation, because the interface is given to me. Also, I can help to contribute to decision-making about the details of the memory instruction pipeline.

1.4.3 Obstacles and risks

There are plenty of situations that can affect the progress of a computer science project, and more specifically, there are setbacks that will probably appear due to the nature of hardware development. In this section I will present the main ones and suggest a possible solution or, in its absence, a way to reduce their effect.

- Design errors, this means that a theoretical design is not correct. The implementation will result incorrect. A deeper look has to be taking in design stage.

- Implementation errors, due to the nature of the hardware, it is possible that a design that intended to support certain functionality needs a more sophisticated implementation than what would be done in software. It has to be taken into account that this is an RTL design.

- Modifications in the VPU during the development, the requisites of these modules can be modified during the project due to the modification of other related modules. A good communication between members of the team is key.

- Testing time, the time allocated to testing the modules makes up a significant part of the total time, and the tests must be sufficiently extensive so as not to lengthen the future verification stages too much. A brief description of corner cases is interesting in order to reduce testing time and increase its reliability.

- Lack of resources, I refer to this situation as the lack of the tools, information and material to proceed to the next iteration of the project. This can happen as a result of, for example, not having enough licenses, a delay in other members of the team work, or a power failure. In this case, if a task is not available at a certain moment, another task of the list will be performed.

1.5 Methodology and rigour

1.5.1 Project management methodology

Hardware development presents some fundamental differences with respect to software. Hardware is less flexible and requires a stricter initial design stage. This causes that it requires a different planning and that it will be more costly to adapt to the specification changes that may happen [6].
The methodology used will be an agile process, which is based on the principles of scrum [7]. In this thesis case each week a meeting will be handled, in which the advances on the current sprint will be reported, which tasks have been done and whose are to do. Also, I will receive feedback on my reports to know if I can continue advancing or something has to be modified. Later, the team leaders will plan the next sprint. The sprint will specify which tasks must be done, the requisites and the time limitation.
To make the design accomplish an acceptable level of quality, each module will move through the stages of: specification (logical functionality and requisites), design (through the elaboration of schemes and tables and text), implementation (writing code and linting) and testing (elaborating a sufficient set of inputs and checking the correctness of outputs for those entries).

1.5.2 Validation

The following list contains the mechanisms used to verify the correctness of the material developed and, therefore, the appropriate progress of the project.

- Gitlab. This platform offers an effective way to manage, control and share the code and joint documents of the project.
- Synchronization meetings. As stated previously, each week there will be a team meeting in which I will receive feedback about my latest advances. Also, there will be other individual meetings with Abraham or other members of the team in case it is necessary.
- Documentation development. Supports used are: schemes, tables and text. The fact of having a visual or conceptual support of what is being done is really important in order to obtain a clearer idea of the details of the thesis.
- Testing. Phases of testing will be necessarily executed together with the implementation process in an iterative and cyclic manner. Testing results are a metric used to check the advances of the thesis.

Chapter 2

Time planning

2.1 Time resume

The completion of this work covers the time between 1st of July 2022, when the planning of the project begins, and January 17, 2023, the deadline for the thesis delivery. The hours defined below have been drawn up taking into account the time spent during the internship and at home autonomously. Values in hours have been rounded to the nearest tenth.
The total time dedicated will be around 595 hours from July to the end of December.

2.2 Tasks definition

In order to be able to prepare an adequate approximation of the time and resources used, the work will be divided into tasks so that it is possible to define their characteristics individually.

2.2.1 Project management tasks

I define a set of tasks intended to specify the different chapters related to the organization and management of the project.

- PM.1: Context and scope. In this chapter, I set the context in which the thesis relies on, indicate the most relevant objectives and justify choices in the decision-making. Time spent will be 1 week $\times$ 16 h/week = 16h.
- PM.2: Project planning. In this chapter, steps to execute the project are set, aiming to track the progress and accomplish the deadlines. Time spent will be 1 week $\times$ 16 h/week = 16 h.
- PM.3: Budget and sustainability. In this chapter, monetary costs are accounted for. Also, a sustainability report is made. Time spent will be 1 week $\times$ 16 h/week = 16 h.
- PM.4: Final organization and management deliverable. An update of previously defined chapters is performed. Time spent will be 1 week $\times$ 16 h/week = 16 h.
- PM.5: Final Thesis memory deliverable. PM.4 is updated and the whole content elaborated is added and expanded in the thesis body. Time spent will be 10 week $\times$ 5 h/week = 50 h.

- PM.6: Meetings. Each week there are meetings scheduled in order to report the advances, get feedback and guide and plan the next steps. Time spent will be 6 month × 4 weeks/month × 2 h/week = 48 h.

2.2.2 Development tasks

I define a set of tasks intended to specify the processes related to the development of the project. This group of tasks is divided into theoretical and practical.

Theoretical study:

- THE.1: Understand vector architectures.

  i. Understand basic vector architecture.

  ii. Understand decoupled vector architectures and other optimized implementations used in design.

  Time spent will be 3 weeks × 7 days/week × 1.5 h/day = 32 h.

- THE.2: Understand ISA.

  i. Understand RISC-V ISA basis.

  ii. Understand RISC-V extension "V" ISA basis and take a brief look to other extensions used in design.

  iii. Understand in more detail instructions and parameters involved in memory instructions in RISC-V "V" extension.

  Time spent will be 3 weeks × 7 days/week × 1.5 h/day = 32 h.

- THE.3: Understand eProcessor's VPU state.

  i. Understand eProcessor's VPU state.

  ii. Understand eProcessor's VPU pipeline.

  iii. Understand eProcessor's VPU memory access pipeline in detail.

  Time spent will be 3 weeks × 7 days/week × 1.5 h/day = 32 h.

- THE.4: Understand EPI's VPU state.
  It isn't as important as understanding VPU eProcessor architecture, so it will be done in less detail.

  i. Understand EPI's VPU state and differences with eProcessor's one.

  ii. Understand EPI's VPU memory access pipeline in detail.

  Time spent will be 2 weeks × 7 days/week × 1.5 h/day = 21 h.

Practical work:

Next tasks are closely related to each one of the target modules, each task implies all modules.

- SPEC: Specification. All details regarding what the design has to accomplish are defined here. Time spent will be 3 weeks × 7 days/week × 2.5 h/day = 53 h.

- DES: Design. Specification must be traduced into a format that represents the hardware that will perform specified functionalities and agreed requirements. This format includes explanations, diagrams, tables and lists of logic signals. Time spent will be 4 weeks × 7 days/week × 2.5 h/day = 70 h.
- IMP: Implementation. The design will be used to program a SV code. In order to get a correct code, linting will be used in this stage. Time spent will be 2 weeks × 7 days/week × 2.5 h/day = 35 h.
- TEST: Testing. The resulting design will be tested with a set of representative inputs. Time spent will be 4 weeks × 7 days/week × 2.5 h/day = 70 h.
- DOC: Documentation. In order to define the specifications, implementation and covered tests, a written memory has to be made up for each module. It is very important at the time of using this code, making updates and teaching other members of the team about the status of these modules. Time spent will be 5 weeks × 7 days/week × 2.5 h/day = 88 h.

2.3 Resources definition

- ST: Staff. The content of this project has been carried out by the following professionals.

  i. ER: Established researcher, team supervisor.

  ii. PSL: Phd student, team lead.

  iii. PS: Phd student.

  iv. US: Undergraduate student.

  It is also worth mentioning that the entire VPU team consists of 3 other members, and that this number may vary during the different stages of the eProcessor. However, these details are not relevant for the desired calculation. Also, I have decided to not taking into account other costs like the place location, water, electricity, etc, in order to simplify the calculation.

- HDW: Hardware equipment. What is needed is a specific computer equipment. One remarkable point here is that having at least 8 GB of RAM is a must to get a good work throughput cause there will be lots of programs working concurrently. Also, it will be needed access to EPI internal server in order to run some EDA tools with a good performance.
- SW: Software equipment. This includes:

  Programming tools: VS code, vim, bash.

  Testing tools: Questasim license.

  Linting tools: Verilator.
- DOC: Documentation. Access to EPI and eProcessor documents and code.

2.4 Task resume table

ID	Name	Time(h)	Dependencies	Resources
PM.1	Context and scope	16		ST,HDW,DOC
PM.2	Project planning	16	PM.1	ST,HDW,DOC
PM.3	Budget and sustainability	16	PM.1, PM.2	ST,HDW,DOC
PM.4	Final organization and management deliverable	16	PM.1, PM.2, PM.3	ST,HDW,DOC
PM.5	Final thesis deliverable	50	PM.1, PM.2, PM.3, PM.4	ST,HDW,DOC
PM.6	Meetings	48		ST,HDW,DOC
PM	Project management	162		
THE.1	Understand vector architectures	32		HDW,DOC
THE.2	Understand ISA	32		ST,HDW,DOC
THE.3	Understand eProcessor's VPU state	32	THE.1, THE.2	ST,HDW,DOC
THE.4	Understand EPI's VPU state	21	THE.1, THE.2	ST,HDW,DOC
THE	Theoretical work	117		
SPEC	Specification	53	THE	ST,HDW,DOC
DES	Design	70	SPEC	ST,HDW,SW,DOC
IMP	Implementation	35	DES	HDW,SW
TEST	Testing	70	IMP	HDW,SW
DOC	documentation	88	TEST	ST,HDW
PRA	Practical work	316		
Total	Total time	595		

Table 2.1: Summary of tasks information. [Own creation].

The gray rows show the group of tasks that contains the ones shown above inside the same group.

2.5 Gantt diagram

This graphic representation makes it easier to visualize the time planning and sets the workload for each week.

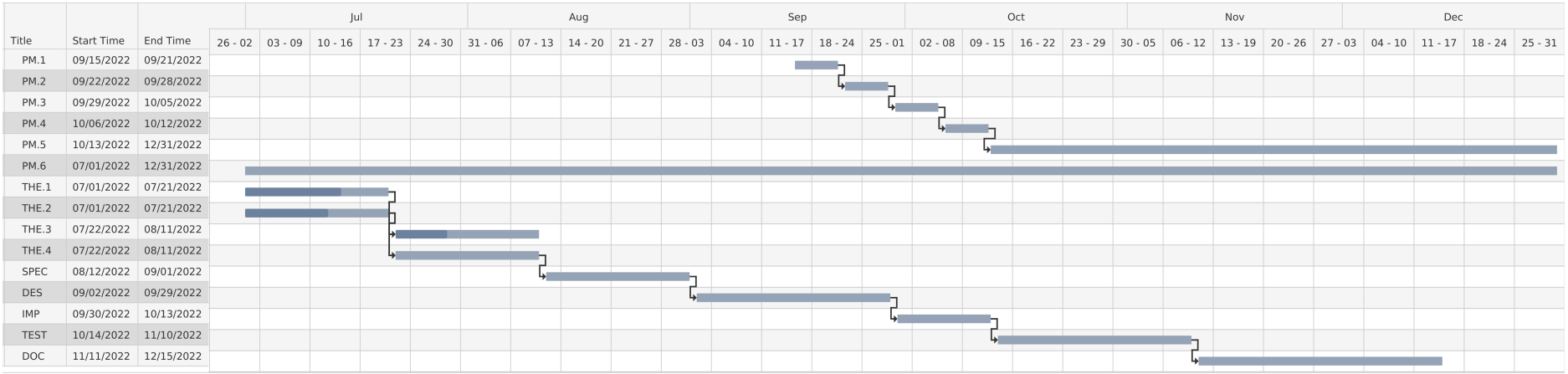

Figure 2.1: Gantt diagram showing. [Own creation]

2.6 Flexibility

In order to adapt to the dynamism of the project, I must take into account that difficulties may arise, modifications with respect to the original plan. So, it will be necessary to review the actions to be carried out.

2.6.1 Risk Management

Given that the planning used is based on an agile methodology, the different stages of the project progress in such a way that certain unexpected events may occur. In those cases, measures have to be adopted.

- Inability to access BSC files. This isn't a usual problem. But, if this happens, documentation files will be stored in local folders to be able to continue with documentation and memory writing while other tasks are delayed.
- Long dependence in work. If I have to wait for another team member to continue with the development of the modules, because my work depends on theirs, I will have no choice but to focus on other tasks.
- Deadlock in design. If I reach a point where I cannot move forward, I can turn to my colleagues and superiors, even, if necessary, I could turn to people from other related projects in the company.
- Deadlock in implementation. If I reach a point where I cannot move forward, I'll have to look for more information about SV, and in the case of RTL implementation, I should take a look into computer architecture structures.
- Deadlock in testing. If I reach a point where I cannot move forward, I will have to take a deeper look into the design to obtain corner cases and to check the aimed functionality. Also, I'll probably have to look for more information about testing tools and how to build proper scripts.
- Deadlock in documentation writing. If I reach a point where I cannot move forward, I will have to ask for more information to the other members of the team, to get more ideas or to solve doubts that are slowing me down.

Chapter 3

Budget and sustainability

3.1 Budget

In this section, a budget calculation is performed, tasks and resources identified in chapter 2 will be analyzed in this estimation. The total estimated cost of the project is 15904.01 €. Values in euros have been rounded up.

3.1.1 Tasks cost

The cost of the tasks is the cost of the salaries of the people who perform them. The developed roles and salaries are the following:

- ER: Manager, team supervisor. Salary = 21.25 × 1.3 = 27.63 €/h.
- PSL: Phd student, team lead. Salary = 12.93 × 1.3 = 16.81 €/h.
- PS: Phd student. Salary = 12.93 × 1.3 = 16.81 €/h.
- US: Undergraduate student. Salary = 9.83 × 1.3 = 12.78 €/h.

Salary costs are extracted from [8]. Taking into account the roles for salary calculation: US will be considered as newly qualified, PSL and PS junior, and ER senior.

Values have been multiplied by 1.3 to add the costs of social security and other taxes.

ID	Name	ER (h)	PSL (h)	PS (h)	US (h)	Cost (€)
PM.1	Context and scope	0	2	0	16	238.1
PM.2	Project planning	3	3	3	16	388.23
PM.3	Budget and sustainability	0	0	0	16	204.48
PM.4	Final organization and management deliverable	0	0	0	16	204.48
PM.5	Final thesis deliverable	8	8	8	50	1129
PM.6	Meetings	48	48	48	48	3553.44
PM	Project management	59	61	59	162	5717.73
THE.1	Understand vector architectures	1	4	4	32	571.07
THE.2	Understand ISA	1	2	2	32	503.83
THE.3	Understand eProcessor's VPU state	1	4	4	32	571.07
THE.4	Understand EPI's VPU state	1	1	1	21	122.83
THE	Theoretical work	4	11	11	117	1768.27
SPEC	Specification	1	4	2	53	118.97
DES	Design	1	2	8	70	121.88
IMP	Implementation	1	1	1	35	508.55
TEST	Testing	1	1	2	70	949.75
DOC	Documentation	1	4	8	88	1353.99
PRA	Practical work	5	12	21	316	3053.14
TC	Tasks cost	68	84	91	595	10539.14

Table 3.1: Tasks cost. [Own creation]

The gray rows show the group of tasks that contains the ones shown above inside the same group.

3.1.2 Resources cost

- HDW: Hardware equipment. The computer is valued around 1200 €and the server maintenance is around 200 €per month. What is accounted for here is the computer plus the access to the server.
- SW: Software equipment. What is accounted for here is the Questasim license it is 997.51 €[9].

- DOC: Documentation. This is reused material, so it doesn't add another cost.

ID	Name	Cost (€)
HDW	Hardware equipment	1200 + 200 × 6 months = 2400
SW	Software equipment	997.51
DOC	Documentation	0
RC	Resources cost	2703.51

Table 3.2: Resources cost. [Own creation]

The gray rows show the group of tasks that contains the ones shown above inside the same group.

3.1.3 Other costs

- Contingencies
  Calculated budget is increased to cover unexpected events. A margin of 15% is applied to the tasks and resources cost for these emergency cases.
  Contingencies cost: (10539.14 + 2703.51) × 0.15 = 1986.4 €.

- Incidental
  Another important aspect to take into consideration is the cost of risk events. Those were specified in section 2.5.

  - Inability to access BSC files. This is a low risk incident, the probability is 15% and its cost is 20 h of US, 255.6 €.
  - Long dependence in work. This is a mid risk incident, the probability is 30% and its cost is 20 h of US, 255.6 €.
  - Deadlock in design. This is a high risk incident, the probability is 60% and its cost is 10 h of US and 4 h of PSL, 195.04 €.
  - Deadlock in implementation. This is a mid risk incident, the probability is 30% and its cost is 5 h of US, 63.9 €.
  - Deadlock in testing. This is a high risk incident, the probability is 60% and its cost is 20 h of US, 255.6 €.
  - Deadlock in documentation writing. This is a high risk incident, the probability is 60% and its cost is 30 h of US and 4 h of PSL, 450.64 €.

  Estimated incidental cost = (255.6 × 0.15) + (255.6 × 0.3) + (195.04 × 0.6) + (63.9 × 0.3) + (255.6 × 0.6) + (450.64 × 0.6) = 674.96 €.

3.1.4 Total cost

Total: Tasks costs + resources costs + contingencies + incidental = 10539.14 + 2703.51 + 1986.4 + 674.96 = 15904.01 €.

3.1.5 Management control

In a big project, it is very difficult to make an exact estimation of the total costs. For this reason, it is interesting to define some margins in which these costs will end up moving.

HR Deviation = Cost Per Hour × (Estimated Hours Of Dedication - Real Hours Of Dedication).
Since most of the costs come from salaries, it is important to take into account the actual hours spent. These can vary due to unforeseen events, so it is important to make a good approximation.

Contingencies cost deviation = (Estimated Contingencies Cost - Real Contingencies Cost).
In the contingency cost, a representative percentage of the variability of the project costs due to unexpected factors is chosen. Therefore, this cost can vary greatly.

Incidental costs deviation = (Estimated incidental hours - Real incidental hours) × Total incidental hours.
Because there are multiple risk factors in the project, the costs due to incidents can vary widely due to initially unknown factors. What we try to do is to establish a probability for each type of expected incident in order to make a better approximation.

3.2 Sustainability

This section covers aspects of the project's impact, in economic, environmental and social terms.

3.2.1 Economy

Have you estimated the cost of carrying out the project?
The cost of the project is detailed in section 3.1. The main human and material resources involved in the project have been taken into account. Possible deviations and risks have also been added.

How is the problem you want to address currently being solved? How will your solution economically improve with respect to other existing ones?
The solution I am working on will give a first implementation to a segment of code not yet written, so that it will enable the execution of the first memory instructions.

The hardware development is oriented to perform, in the most optimal way possible and with the available resources, the execution of applications in the target areas.

3.2.2 Environment

Have you estimated the environmental impact that the project will have? Have you considered minimizing the impact, for example, by reusing resources?
The impact of the project is that of the resources used in it, a tiny part of all the resources allocated to eProcessor. These resources are leveraged because segments of EPI code and documentation, as well as other modules, are reused.

How is the problem you want to address currently being solved? How will your solution environmentally improve with respect to other existing ones?

The solution I am working on will give a first implementation to a segment of code not yet written, so that it will enable the execution of the first memory instructions.

Regarding eProcessor, the fruits of this project will form part of the network of IPs intended to offer low-consumption processors.

3.2.3 Society

What do you think the realization of this project will bring you on a personal level?
This project will improve my skills in the field of computer architecture. It will provide me with a theoretical and practical basis on topics such as RISC-V, vector architectures. Furthermore, I will improve other very important skills through the different phases of development, such as the creation of schematics and documentation, team working, and improving my English. In general, I will become a better professional.

How is the problem you want to address currently being solved? How will your solution socially improve with respect to other existing ones?
The solution I am working on will give a first implementation to a segment of code not yet written, so that it will enable the execution of the first memory instructions.

eProcessor is one of the pieces that is being placed in order to develop an European supercomputing infrastructure and to achieve technological independence. Therefore, it is an interesting project for the entire territory and related sectors.

Is there a real need for the project?
This project is part of the eProcessor project and is necessary for its progress, specific hardware is required and tested following the requirements of the project so that it is adequate. eProcessor is receiving European funding and involves multiple companies united by an ambitious project, aimed at developing computing technology in Europe.

Chapter 4

Architecture

4.1 Introduction

To present the content of the project, it will start with the different architectures involved. The architecture is defined by the ISA, whereas the microarchitecture is composed by a specific implementation. The main pieces of the following architecture are RISC-V and the RISC-V extensions used, with particular emphasis on "V", Vector Extension.

4.2 What is the ISA?

"An Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software. The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done.
The ISA provides the only way through which a user is able to interact with the hardware. It can be viewed as a programmer's manual because it's the portion of the machine that's visible to the assembly language programmer, the compiler writer, and the application programmer.
The ISA defines the supported data types, the registers, how the hardware manages main memory, key features (such as virtual memory), which instructions a microprocessor can execute, and the input/output model of multiple ISA implementations. The ISA can be extended by adding instructions or other capabilities, or by adding support for larger addresses and data values." [10].

4.3 RISC-V

4.3.1 RISC-V History

Pronounced "risk-five", this is the fifth reduced instruction set computer (RISC) architecture developed at the University of California, Berkeley since 1981.
In fact, RISC was the name coined by Berkeley to refer to its first project aiming to develop an architecture, based on the idea of using simpler instructions than the ones widely used by the industry at that moment, complex instruction set computer (CISC), with the objective of obtaining a more efficient architecture in terms of power, consumption and area.
RISC-V was developed to cover Berkeley needs in research and education. It was interested in the development of new hardware architecture implementations for industry and

to provide students with real implementations to explore in classes.
The current proprietary ISAs suffer from a number of limitations to meet the needs exposed: The lack of transparency in their implementations, the lack of detail in documentation and their reliance on outdated concepts, were barriers in favor of developing a new ISA. [11, chapter 28].
In 2010 RISC-V instruction set was started. Later, in 2015 RISC-V Foundation was founded to build an open and collaborative community of software and hardware innovators based on RISC-V ISA [12].

4.3.2 Why RISC-V?

There are some interesting facts about this ISA which made it suitable for this project:

- Free use license in academy and industry.
- Modular design. It counts with standard extensions that can be added to the design to extend the functionalities. Furthermore, personalized extensions can be added.
- Support. RISC-V International is a nonprofit corporation made up of members within the sector, such as the BSC, which work together to update its features and encourage its use. The community can also interact with RISC-V members via public discussion lists, in order to share their concerns and advises about the development in order to assist in different development fields which include ISA update and extension, software tools development and hardware development.
- Flexibility and suitability. An ISA that avoids "over-architecting" for a particular microarchitecture style (e.g., mi-crocoded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these.
- Renewal. This ISA is based on the existing alternatives on the market. It has learned from their mistakes as it isn't dependable on any obsolete elements due to retro-compatibility with binaries for old instructions, that existing architectures must comply with.

Cortus RVOOO (RISC-V Out Of Order) will be a 64 bit RISC-V processor which integrates the RV64GCV ISA.

4.3.3 RISC-V Unprivileged

For the purposes of this thesis, only the unprivileged ISA volume will be discussed. Unprivileged instructions are those that are generally usable in all privilege modes in all privileged architectures.

RV64GCV is composed of a combination of a base 64 bit ISA (RV64I) plus standard extensions IMAFD, Zicsr, Zifencei, "C" and vectorial.

4.3.4 Scalar Extensions

RV64I

RV64I is the 64 bit base integer instruction set. Some characteristic features are:

- XLEN = 64

- The address space is indexed with XLEN bits and is byte addressed. Memory system can be little-endian or big-endian.
- Large number of Integer logical registers: 32, XLEN bits wide.
  From those, x0 is hardcoded to 0 and x1-x31 are general purpose.
  Also there's one extra register, pc, which holds the address of the current instruction.
- Reduced number of instructions: 40, 32 bit wide.

  Decoding register specifiers, the source (rs1 and rs2) and destination (rd), is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats. Another design decision was that immediates (imm) are sign extended. The sign bit for all imm is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.
- The base set of instructions includes: computational instructions(e.g. ADDI, SLLI, NOP), control transfer instructions (e.g. JAL, BEQ), memory instructions(e.g. LHU, SW) memory ordering instructions(e.g. FENCE), environment call and breakpoints(e.g. ECALL and EBREAK).

IMAFD

This extension is a combination of: Integer multiplication and division (M), Atomic instructions (A), Single-Precision Floating-Point (F) and Double-Precision Floating-Point (D).

Zicsr

Control and Status Register (CSR) Instructions. There's a separate address space of 4096 Control and Status registers associated with each hart. This extension defines the full set of CSR instructions that operate on these CSRs.

Zifencei

Instruction-Fetch Fence. This extension includes the FENCE.I instruction that provides explicit synchronization between writes to instruction memory and instruction fetches on the same hart, only. Currently, this instruction is the only standard mechanism to ensure that stores visible to a hart will also be visible to its instruction fetches.

"C" extension

Compressed Instructions. RVC instructions includes a set of short 16-bit instruction encodings for common operations. Typically, 50%–60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%–30% code-size reduction.

4.3.5 Vector extension

RISC-V base vector extension release v0.7 [13]. Since this is a base expansion, it can be expanded with more specialized vector instructions such as cryptography or machine learning.
The most relevant parts of the documentation will be explored:

Implementation-defined Constant Parameters

Each hart defines the following parameters.

- ELEN. The maximum size of a single vector element, in bits, must be a power of 2.
- VLEN. The number of bits in a vector register, must satisfy $VLEN \geq ELEN$ and to be a power of 2.

Miscellany

32 vectorial registers VLEN bit wide are added.

Control and Status Registers (CSRs)

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start position
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register

Table 4.1: Vector CSRs subset. [13, p. 7]

The vstart CSR specifies the index of the first element to be executed by a vector instruction. All vector instructions begin its execution with the element number given in vstart, leaving earlier elements in the destination vector undisturbed, and to reset vstart to zero at the end of execution.

Normally, vstart is only written by hardware on a trap on a vector instruction, with the vstart value representing the element on which the trap was taken (either a synchronous exception or an asynchronous interrupt), and at which execution should resume after a resumable trap is handled. Furthermore, the value of vstart can also be modified by the instructions defined in the extension Zicsr. However, for some instructions there are vstart values that may raise an illegal instruction exception.

The vl CSR holds an unsigned integer specifying the number of elements to be updated by a vector instruction. Elements in any destination vector register group with indices $\geq$ vl are zeroed during execution of a vector instruction. The vl CSR can only be updated by vsetvli and vsetvl instructions.

```
vsetvli rd, rs1, vtypei # rd = new vl, rs1 = AVL, vtypei = new vtype setting
                        # if rs1 = x0, then use maximum vector length
vsetvl  rd, rs1, rs2    # rd = new vl, rs1 = AVL, rs2 = new vtype value
                        # if rs1 = x0, then use maximum vector length
```

Figure 4.1: Vector vsetvl{i} Instruction Set. [13, p. 25]

In 4.1 we can see how vtype value is set, but not only that, this function also serves to obtain the size of the vector and save the previous configuration. AVL stands for application vector length. One possible vsetvl configuration will be explored later.

The vtype CSR provides the default type used to interpret the contents of the vector register file. The vtype CSR can only be updated by vsetvli and vsetvl instructions. The following table describes its fields:

Bits	Name	Description
XLEN-1	vill	Illegal value if set
XLEN-2:7		Reserved (write 0)
6:5	vediv[1:0]	Used by EDIV extension
4:2	vsew[2:0]	Standard element width (SEW) setting
1:0	vlmul[1:0]	Vector register group multiplier (LMUL) setting

Table 4.2: vtype register layout. [13, p. 25]

The vector type illegal (vill) is used to encode that a previous vsetvli instruction attempted to write an unsupported value to vtype. If the vill bit is set, then any attempt to execute a vector instruction (other than a vector configuration instruction) will raise an illegal instruction exception. When the vill bit is set, the other XLEN-1 bits in vtype shall be zero.

EDIV extension is not used. So, vediv value has no purpose and is not implemented.

The vector standard element with (vsew) sets dynamically the SEW of the vector instruction. The elements of a vector register are calculated as VLEN / SEW. In current eProcessor VPU implementation, MAX SEW is equal XLEN.

The vector register grouping (vlmul) is used to execute one vector instruction on multiple vector registers. In current eProcessor VPU implementation, vlmul is always set to 1.

Vector Loads and Stores

Vector loads and stores move values between vector registers and memory. Vector loads and stores are masked and do not raise exceptions on inactive elements. Masked vector loads do not update inactive elements in the destination vector register group. Masked vector stores only update active memory elements.

Vector loads and stores are encoded within the scalar floating-point load and store major opcodes (LOAD-FP/STORE-FP).

```
Format for Vector Load Instructions under LOAD-FP major opcode
31 29 28 26  25  24      20 19       15 14   12 11      7 6     0
 nf  | mop | vm |  lumop   |    rs1    | width |    vd   |0000111| VL*  unit-stride
 nf  | mop | vm |   rs2    |    rs1    | width |    vd   |0000111| VLS* strided
 nf  | mop | vm |   vs2    |    rs1    | width |    vd   |0000111| VLX* indexed
  3     3     1      5           5         3         5       7

Format for Vector Store Instructions under STORE-FP major opcode
31 29 28 26  25  24      20 19       15 14   12 11      7 6     0
 nf  | mop | vm |  sumop   |    rs1    | width |   vs3   |0100111| VS*  unit-stride
 nf  | mop | vm |   rs2    |    rs1    | width |   vs3   |0100111| VSS* strided
 nf  | mop | vm |   vs2    |    rs1    | width |   vs3   |0100111| VSX* indexed
  3     3     1      5           5         3         5       7
```

Figure 4.2: vector memory instructions format. [13, p. 29]

Field	**Description**
rs1[4:0]	specifies x register holding base address
rs2[4:0]	specifies x register holding stride
vs2[4:0]	specifies v register holding address offsets
vs3[4:0]	specifies v register holding store data
vd[4:0]	specifies v register destination of load
vm	specifies vector mask
width[2:0]	specifies size of memory elements, and distinguishes from FP scalar
mop[2:0]	specifies memory addressing mode
nf[2:0]	specifies the number of fields in each segment, for segment load/stores
lumop[4:0]/ sumop[4:0]	are additional fields encoding variants of unit-stride instructions

Table 4.3: vector memory instructions format fields. [13, p. 29]

As we will see below, there are memory accesses at byte, half, word and SEW levels depending on width value. For a memory instruction, in the register banks part, the number of bits accessed is vl $\times$ SEW, while in the memory part, the number of bits accessed is vl $\times$ width. So, in a load, if SEW $>$ width, the elements are extended before being written in the register. Depending on mop value, the space will be filled with 0s or the bit sign value.
Also, in a memory instruction, if SEW $<$ width, an illegal instruction exception is raised.

Another interesting point here is that stride is given in bytes and can be a 0 or unaligned value. For now, those are not supported in eProcessor.

Vector Load/Store Addressing Modes

The base vector extension supports unit-stride, strided, and indexed (scatter/gather) addressing modes. The base effective address for all vector accesses is given by the contents of the x register named in rs1.
Vector unit-stride operations access elements stored contiguously in memory starting from

the base effective address.
Vector strided operations access the first memory element at the base effective address, and then access subsequent elements at address increments given by the byte offset contained in the x register specified by rs2.
Vector indexed operations add the contents of each element of the vector offset operand specified by vs2 to the base effective address to give the effective address of each element. The vector offset operand is treated as a vector of byte offsets, those offsets have the same size as data elements, SEW defined in vsew CSR field. If the vector offset elements are narrower than XLEN, they are sign-extended to XLEN before adding to the base effective address. If the vector offset elements are wider than XLEN, the least-significant XLEN bits are used in the address calculation.

Vector Unit-Stride Instructions

```
# Vector unit-stride loads and stores

# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
vlb.v  vd, (rs1), vm # 8b signed
vlh.v  vd, (rs1), vm # 16b signed
vlw.v  vd, (rs1), vm # 32b signed

vlbu.v vd, (rs1), vm # 8b unsigned
vlhu.v vd, (rs1), vm # 16b unsigned
vlwu.v vd, (rs1), vm # 32b unsigned

vle.v  vd, (rs1), vm # SEW

# vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
vsb.v  vs3, (rs1), vm  # 8b store
vsh.v  vs3, (rs1), vm  # 16b store
vsw.v  vs3, (rs1), vm  # 32b store
vse.v  vs3, (rs1), vm  # SEW store
```

Figure 4.3: Vector Unit-Stride Instruction Set. [13, p. 32]

Vector Strided Instructions

```
# Vector strided loads and stores

# vd destination, rs1 base address, rs2 byte stride
vlsb.v  vd, (rs1), rs2, vm # 8b
vlsh.v  vd, (rs1), rs2, vm # 16b
vlsw.v  vd, (rs1), rs2, vm # 32b

vlsbu.v vd, (rs1), rs2, vm # unsigned 8b
vlshu.v vd, (rs1), rs2, vm # unsigned 16b
vlswu.v vd, (rs1), rs2, vm # unsigned 32b

vlse.v  vd, (rs1), rs2, vm  # SEW

# vs3 store data, rs1 base address, rs2 byte stride
vssb.v vs3, (rs1), rs2, vm  # 8b
vssh.v vs3, (rs1), rs2, vm  # 16b
vssw.v vs3, (rs1), rs2, vm  # 32b
vsse.v vs3, (rs1), rs2, vm  # SEW
```

Figure 4.4: Vector Strided Instruction Set. [13, p. 32]

Vector Indexed Instructions

```
# Vector indexed loads and stores

# vd destination, rs1 base address, vs2 indices
vlxb.v  vd, (rs1), vs2, vm  # 8b
vlxh.v  vd, (rs1), vs2, vm  # 16b
vlxw.v  vd, (rs1), vs2, vm  # 32b

vlxbu.v vd, (rs1), vs2, vm  # 8b unsigned
vlxhu.v vd, (rs1), vs2, vm  # 16b unsigned
vlxwu.v vd, (rs1), vs2, vm  # 32b unsigned

vlxe.v  vd, (rs1), vs2, vm  # SEW

# Vector ordered-indexed store instructions
# vs3 store data, rs1 base address, vs2 indices
vsxb.v vs3, (rs1), vs2, vm  # 8b
vsxh.v vs3, (rs1), vs2, vm  # 16b
vsxw.v vs3, (rs1), vs2, vm  # 32b
vsxe.v vs3, (rs1), vs2, vm  # SEW

# Vector unordered-indexed store instructions
vsuxb.v vs3, (rs1), vs2, vm  # 8b
vsuxh.v vs3, (rs1), vs2, vm  # 16b
vsuxw.v vs3, (rs1), vs2, vm  # 32b
vsuxe.v vs3, (rs1), vs2, vm  # SEW
```

Figure 4.5: Vector Indexed Instruction Set. [13, p. 33]

Graphical representation

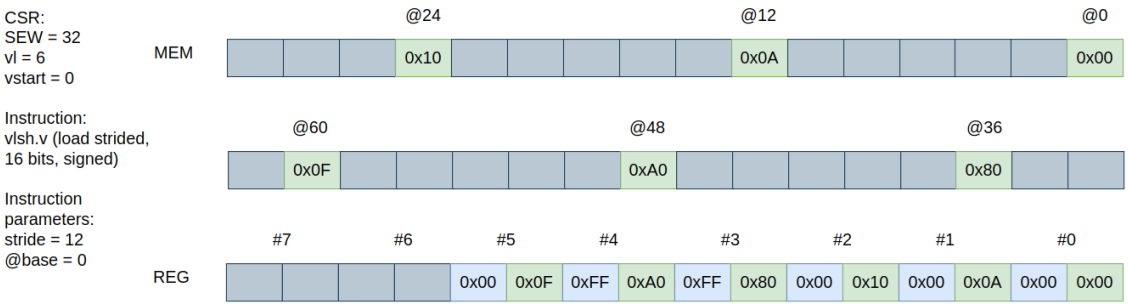

Figure 4.6: ISA graphical example 1, vload strided. [Own creation]

In 4.6, in green we have valid element values, in blue element extend, in gray invalid elements. Each block is 16 bits width, for this reason, addresses shown in memory grown by 2 for each block as memory is indexed by bytes.
As this is a vload, values are moved from memory to the registers. As we can see, elements can be mapped in memory depending on base address and stride, whereas SEW width elements are placed contiguously in destination register file.

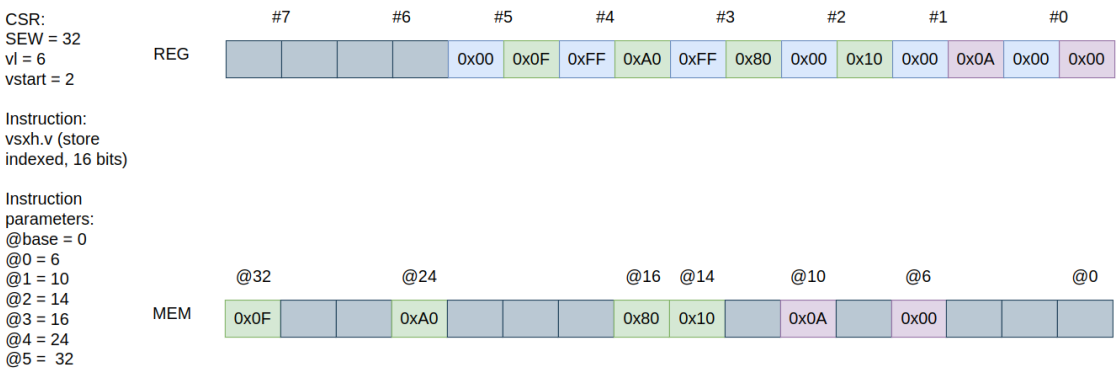

Figure 4.7: ISA graphical example 2, vstore indexed. [Own creation]

In 4.7, in green we have valid element values, in blue element extend, in gray invalid elements and in purple we have the elements which were computed before the trap that has modified vstart value.
As this is a vstore, values are moved from registers to memory. As we can see, elements can be mapped in memory depending on base address and index vector.

Consistency Model

A memory consistency model is a set of rules specifying the values that can be returned by loads of memory.
RISC-V uses a memory model called "RVWMO" (RISC-V Weak Memory Ordering) which is designed to provide flexibility for architects.
Like other architectures(x86, ARM, MIPS), code running on a single hart appears to execute in order from the perspective of other memory instructions in the same hart. But, as it is using a weak model and as a consequence of the implementation functionality, memory instructions from another hart may observe the memory instructions from the first hart being executed in a different order in many cases. Therefore, multithreaded code may require explicit synchronization to guarantee ordering between memory instructions from different harts when this is needed.
Unfortunately, those rules are only for scalar instructions. For now there's no support such as a Vector extension Consistency Model, one of the causes is that topics related such as memory overlapping caused by memory accesses of different type, are still under research by academic community.

Code example

This is a simple code example, a memcpy implementation, where we can see CSR configuration and vectorial memory operations.

```
# void *memcpy(void* dest, const void* src, size_t n)
# a0=dest, a1=src, a2=n
#
memcpy:
mv a3, a0 # Copy destination
loop:
vsetvli t0, a2, e8,m8 # Vectors of 8b
vlb.v v0, (a1)
 # Load bytes
```

```
add a1, a1, t0
 # Bump pointer
sub a2, a2, t0
 # Decrement count
vsb.v v0, (a3)
 # Store bytes
add a3, a3, t0
 # Bump pointer
bnez a2, loop
 # Any more?
ret
 # Return
```

As we can see in this example, each iteration of the loop works in the following way: vsetvl sets SEW 8(e8) and LMUL 8(m8), then executes a vload and updates the vload pointer and byte counter, later it executes a store to previously copied destination address and checks counter. As vsetvli sets vl according to the remaining n, this code covers all possible values of n.

Chapter 5

eProcessor

5.1 eProcessor VPU

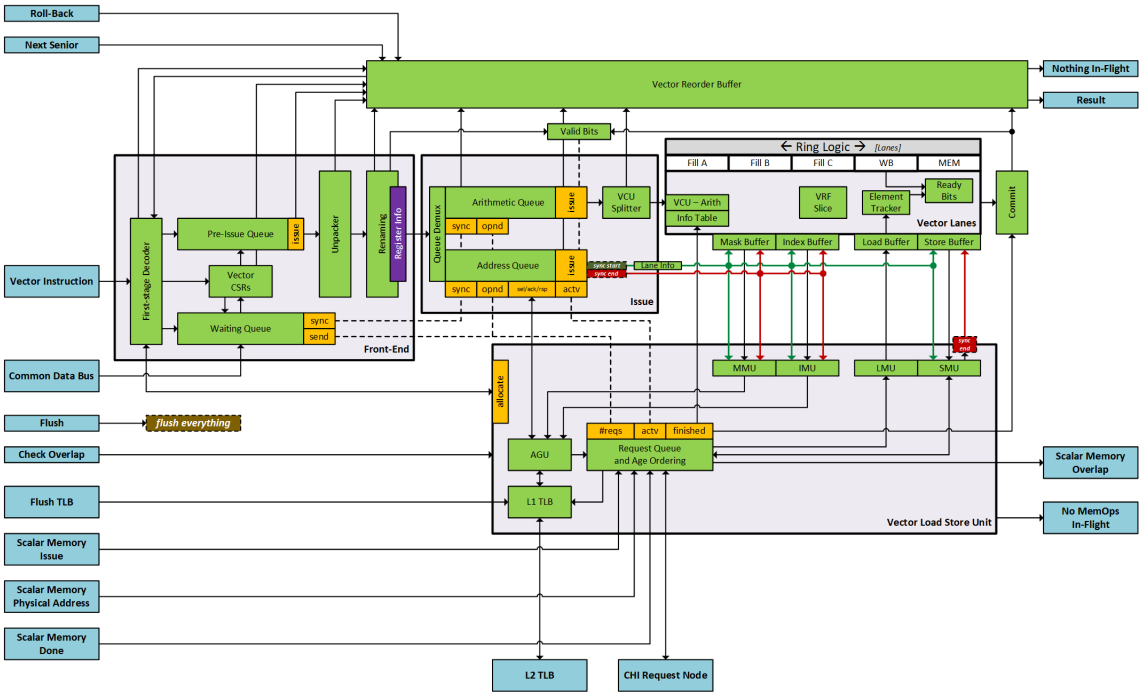

Figure 5.1: eProcessor VPU scheme. [Abraham's creation]

All starts inside the core: When the vector instruction is dispatched by the core, the instruction reaches the First Stage Decode through Vector Instruction Interface. Inside this module the fields of the instruction are decoded: The operands are extracted and the kind of vector instruction is identified.

The instruction is then bypassed to the Pre-Issue Queue. There, it will remain until it is issued to the Unpacker. Scalar operands are received from the Scalar Core in the Waiting Queue and sent to the corresponding slot of the Arithmetic or Address Queue.

If the Unpacker is not stalled, the instruction is sent from the Pre-Issue Queue The Unpacker is in charge of the fine-grained decoding of the instructions. The Unpacker accepts the request coming from the Pre-Issue Queue, decodes this instruction figuring out the opcode, type of instruction, if it's masked, function6, function3, etc. but more importantly, since some instructions require the generation of auxiliary operations to be

RISC-V compliant (according to the current microarchitecture), e.g. masked loads, the Unpacker is in charge of generating these auxiliary instructions. The Unpacker is also in charge of detecting illegal instructions, overall regarding illegal configurations (e.g. widening operations when LMUL > 4).

Instruction joins then the Renaming Unit. The goal of the renaming is to resolve the false dependencies in order to exploit the available ILP. 40 physical registers are implemented for renaming. In this process, the architectural vector registers defined by the RISC-V V-extension are mapped, according to the state of the vector engine, to a certain physical register. Furthermore, the VPU makes use of physical Mask Registers, and for that reason, extending the renaming support for the mask registers will lead to obtain more ILP. The vector engine implements 4 renamed mask registers for renaming purposes.

Queue Demux is a simple demultiplexer that sends the output of the Renaming Unit to one of the issue queues.

From now on, focus will be on memory path. Therefore, Address Queue and following modules will be explored. Address Queue is in charge of receiving the vector memory instructions from the Front-End and issuing the required instruction information to the Vector Load Store Unit. SMU, Store Buffer, MMU, Mask Buffer, IMU, Index Buffer units require synchronization signals as input requests.

Inside the VLSU, instructions are divided in pieces called memory requests and executed individually. The granularity or size of a request is the size of every individual cache line that will be read or written to memory hierarchy in that memory instruction, depending on either it is anstore or load instruction.

The AGU is in charge of generating the necessary memory requests to satisfy any load or store vector instruction coming from the Address Queue, whichever the addressing mode (unit-stride, strided and indexed) and if it happens to be a masked or unmasked operation. In case it is a Masked Instruction, it also converts the masks coming from the MMU into the masks required for each request and, in case it is a Indexed instruction, it uses the Indexes from the IMU into the indexes required in each request. After computing this information, the request is sent to the Request Queue.

Request Queue is in charge of allocating and managing the memory access requests to the CHI Request Node. A distinction must be made: In a load data is sent from the Request Node to the LMU, in a store, data is received from the SMU to the Request Node. Disambiguation is also performed in this module, it checks if there's a dependency between accessed area of a vector request and a scalar memory operation, the mechanism checks the elements masked in each vector request.

With regard to memory management units, those are in charge of rearranging the elements in order to be correctly placed in its new location. And in respect of memory buffers are used to move elements between registers and and memory management units.

For a store instruction, once the last request transaction is finished by the CHI Request Node, a commit message is sent to the Reorder Buffer in order to clean its structures and communicate the finalization of the instruction to the OoO Core. For a load, the same happens when all values get finally stored inside VRF.

Each Vector Lane contains a group of relevant modules in the execution of a Vector Memory Instruction: Finite State Machine (FSM), Mask register file (MRF), Ready Bits(RB) and Element Tracker. Vector Lane is connected to the other lanes via the ring index and data interfaces
VRF: each lane contains a subset of a complete vector register (vector slice), allowing that all the lanes works simultaneously on the same vector. The VRF is implemented on five memory banks to let the VRF work as it was a 5-port memory, by using a group of buffers. VRF Arbiter: This module arbitrates the access to the VRF ports, setting access priorities. MRF: Mask registers are kept in a separated structure to avoid requiring an additional read to the VRF for Masked Instructions. RB: tracks the availability of groups of values of the physical registers and has to be checked in order to access to a VRF element. Element Tracker: The Element Tracker provides support for chaining. The main idea of chaining is to start executing an arithmetic instruction that depends on a load while this load is still receiving data. This element modifies RB value.

The Vector ReOrder Buffer (VROB) is in charge of keeping the ordering between the in-flight vector instructions that are in the VPU's pipeline. It is implemented a lightweight in-order/out-of-order execution mechanism by splitting CSR, arithmetic, and memory operations into different queues. This feature allows parallel execution of these different types of instructions in the corresponding functional units to sustain high performance.
To keep track of the instruction, each instruction is assigned an entry in the VROB upon being decoded, and as long as it flows through the VPU pipeline, it will add information on its status accessing the corresponding VROB entry. On instruction commit, the VROB will notify the scalar core by driving the corresponding ports of the interface.
In case an instruction kill has to be performed, VROB contains the information needed to perform the rollback of that instruction and younger ones. This module stores the state of the instructions in its different stages of the pipeline in order to recover it if it is necessary.

5.2 EPI differences

The main differences to take into account between eProcessor and EPI are:

- eProcessor supports any stride multiple of SEW. This functionality has been added in eProcessor because this is required for a HPC aimed implementation, which isn't needed in EPI, where strides supported are one SEW $\times$ 1, 2 and 4, because this design would consume more area than available.

- Direct memory access. As it can be seen in 5.1, eProcessor has direct access to memory hierarchy via a request node, compliant to AMBA5-CHI protocol (RNI). In EPI, accesses to memory are managed by the core so it has an extra interface with the VPU for these purposes.

- Modified FSM in lanes. In eProcessor the access to VRFs in lanes is managed by the buffers whereas. So, inside the lanes, the FSM has been replaced by a VRF arbiter. EPI, uses FSM inside the lanes, which manages the accesses to the VRF resources in all kinds of instruction execution.

- Modified exception handle. In EPI exceptions are handled by the core, whereas in eProcessor the Request queue is the main agent in charge of managing traps related to memory accesses, this module reports this events to the Reorder buffer in order to use instruction stages information to recover previous stable VPU state.

Chapter 6

Modules

6.1 Introduction

For the interface subsection, signals named x_i are input signals, which are the ones that receive a value outside the module and are used inside. At the same time, signals named y_o are output signals, which are the ones whose value is set inside the module and is used outside. This code convention can be found in the System Verilog Guidelines used in eProcessor VPU [14].

These modules have been tested with a testbench [15]. The testbench architecture 6.1, consists of a buffer for each one of the input channels. In parallel, the output channels are connected to the logic that control input buffers in order to synchronize the data flow of the testbench. Module state (i.e. internal signal values and its evolution through time) is checked at the end of the simulation.
For each one of the modules, the circumstances to be tested will be shown, as well as the reason for them. However, it is important to find different sets of value inputs that generate these states within the modules.
Furthermore, the modules have been tested individually and in pairs, IMU + Index Buffers, SMU + Store Buffers.

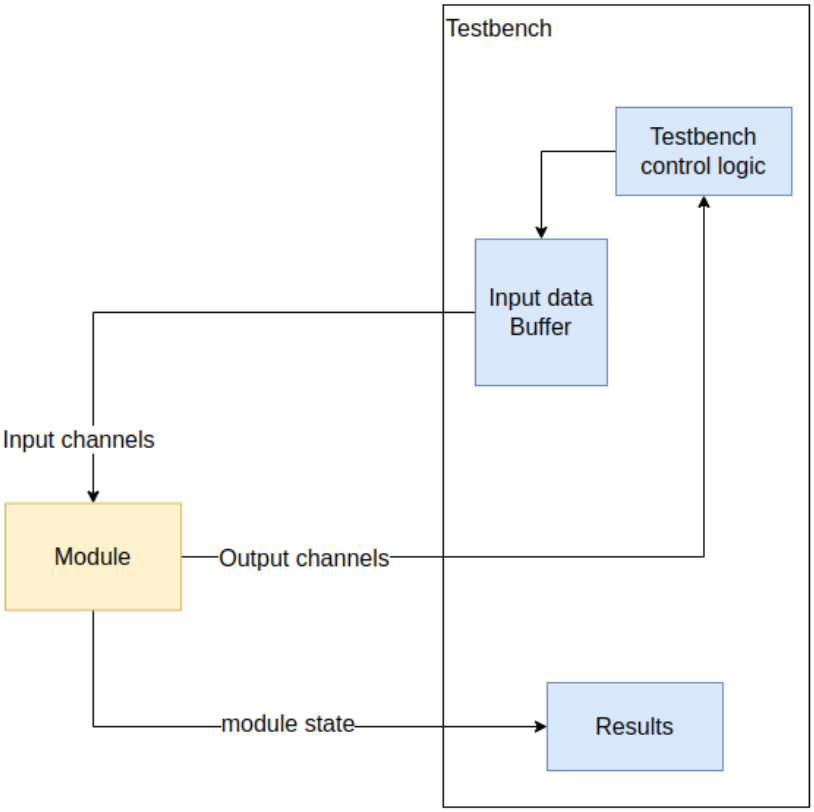

Figure 6.1: Testbench Architecture. [own creation]

6.2 Store Management Unit

This module provides support to vstore instructions. Its main functionality consists of to generate the l2 cache lines that will be stored in the Memory Hierarchy in vstore requests. This process involves the rearranging of elements coming from the Store Buffers, according to 2 kinds of data related to the vstore operation being executed: vstore instruction and vstore request information.

6.2.1 Interface

SMU is instantiated inside Vector Load Store Unit and is connected to Address Queue, Request Queue and Store Buffer.

Top: Vector Load Store Unit (vlane)

- clock and reset signals.

Signal	Type	Description
clk_i	logic	clock signal
rsn_i	logic	reset signal (active low)

Table 6.1: SMU VLSU interface. [Own creation]

Address Queue (mqueue)

- vstore instruction information.
  Synchronization is implicit, so mqueue_sync_start_i is only asserted when SMU is available.

Signal	Type	Description
mqueue_sync_start_i	logic	start store instruction
mqueue_info_i	mqueue2smu_t	store instruction info

Table 6.2: SMU Address queue interface. [Own creation]

Request Queue (rqueue)

- vstore request information.
  Synchronization is explicit via a handshake protocol.

Signal	Type	Description
rqueue_req_valid_i	logic	store request valid
rqueue_req_ack_o	logic	store request acknowledged
rqueue_req_info_i	rqueue_smu_info_t	store request info

Table 6.3: SMU request-queue request interface. [Own creation]

- vstore response information. This channel includes the Cache Line assembled.
  Synchronization is explicit via a handshake protocol.

Signal	Type	Description
rqueue_resp_valid_o	logic	store response valid
rqueue_resp_ack_i	logic	store response acknowledged
rqueue_resp_info_o	smu_rqueue_resp_t	store response info

Table 6.4: SMU request-queue response interface. [Own creation]

Store Buffer

- Buffer data elements coming in order.
  Synchronization is explicit via a handshake protocol.

Signal	Type	Description
stbf_valid_i	logic	store buffer data package valid
stbf_ack_o	logic	store buffer data package acknowledged
stbf_data_i	buffers_data_t	store buffer data package

Table 6.5: SMU store-buffer data interface. [Own creation]

- End vstore instruction signal.
  Synchronization is implicit, so stbf_sync_end_o is only asserted when SMU has finished computing a vstore instruction but it is still in flight.

Signal	Type	Description
stbf_sync_end_o	logic	reset signal for store buffers (active high)

Table 6.6: SMU store-buffer reset interface. [Own creation]

6.2.2 Data Types

Signal	Type	Description
row	logic [$clog2(N_LANES × N_BANKS × ELEN / MIN_SEW)]	row offset

Table 6.7: mqueue2smu_t elements. [Own creation]

row value consists on the row offset of a vector instruction extracted from vstart value. Indicates which is the first valid element in the first Buffer Data line for the first request in a certain vstore instruction.
Buffer data treatment will be shown below. So, this signal usage will be clarified.

Signal	Type	Description
opmode	mem_op_mode_e	Instruction valid signal
stride	xlen_t	Request stride value
vsew	vsew_t	Request vsew value
last	logic	Instruction's last Request signal
elem_id	elem_id_t	Request first element
elem_cnt	elem_count_t	Request amount of elements
elem_offset	elem_offset_t	Request first element offset
vmot_id_t	vmot_id_t	Vector load store unit id
kill	logic	Indicates if the request must be killed
tag	rqueue_id_t	Request tag
line_mask	l2_cache_line_mask_t	l2 cache line mask

Table 6.8: rqueue_smu_info_t elements. [Own creation]

Signal	Type	Description
vmot_id	vmot_id_t	vector load store unit id
kill	logic	Indicates if the request has been killed
tag	rqueue_id_t	Request tag
line_mask	l2_cache_line_mask_t	l2 cache line mask
data	l2_cache_line_t	l2 cache line data

Table 6.9: smu_rqueue_resp_t elements. [Own creation]

buffers_data_t is a logic [N_LANES × ELEN].

6.2.3 Architecture

The SMU consists of a design which consists of 5 components: Instruction and Request info logic, an internal buffer, a combinatorial logic path, store data response and reset logic.

Instruction and Request info logic

This information indicates whether instruction and request information has to be used in that cycle on other parts of the architecture or not.

- Instruction info logic

Signal	Type	Description
op_valid_q	logic	Instruction valid signal
mqueue_op_info_q	mqueue2smu_t	Instruction info

Table 6.10: SMU Instruction info logic signals. [Own creation]

mqueue_op_info_q.row is used for 3 purposes:
Discard invalid Store buffer data packages.
Discard invalid elements from a valid Store buffer data package.
Initialize sb_wr_pt_offset_q and sb_rd_pt_offset_q.

- Request info logic

Signal	Type	Description
req_valid_q	logic	Request valid signal
rqueue_req_info_q	rqueue_smu_info_t	Request info

Table 6.11: SMU Request info logic signals. [Own creation]

rqueue_req_info_q is mainly used for 2 reasons:
Checking whether the Instruction must be finished in this request or not.
Setting the parameters to construct request's l2 cache line.

Internal Buffer

- Buffer State Logic Buffer size is the same as a l2 cache line. This is the case because rearranging logic uses the Buffer Data state as its source data elements.
  Furthermore, Buffer is composed of 2 entries of 256 bits. This is the case because data packages are 256 bits long (ELEN × N_LANES).

Signal	Type	Description
sb_full	logic	Buffer full occupation
sb_depth_q	elem_count_t	Buffer occupation counting in bytes
sb_free	elem_count_t	Buffer emptiness counting in bytes

Table 6.12: SMU Buffer state signals. [Own creation]

- Write Buffer Operation

Signal	Type	Description
sb_wr	logic	Buffer writing
sb_wr_pt_q	stage_offset_t	Buffer entry writing pointer
sb_wr_pt_offset_q	all_lanes_bytes_idx_t	Buffer offset entry writing pointer in bytes
sb_wr_remaining_cnt_q	all_lanes_bytes_num_t	Current data package remaining writing bytes
sb_wr_current_cnt	all_lanes_bytes_num_t	Current writing bytes
sb_wr_mask	all_lanes_bytes_t	Writing mask, byte granularity

Table 6.13: SMU Buffer write signals. [Own creation]

Only valid data packages are written to the Buffer.

When a whole invalid data package has to be discarded, stbf_ack_o is asserted without asserting sb_wr, this happens when mqueue_op_info_q.row $\geq$ data package. mqueue_op_info_q.row is reduced by data package size in Bytes when a package is discarded.
In a write operation, sb_wr_pt_q indicates the correct entry of the buffer, sb_wr_pt_offset_q indicates the pointer inside the chosen vector entry, sb_free and sb_wr_remaining_cnt_q are used to set both sb_wr_mask and sb_wr_current_cnt. Finally, stbf_ack_o is **only** asserted when writing last elements of a data package.

Example 1:

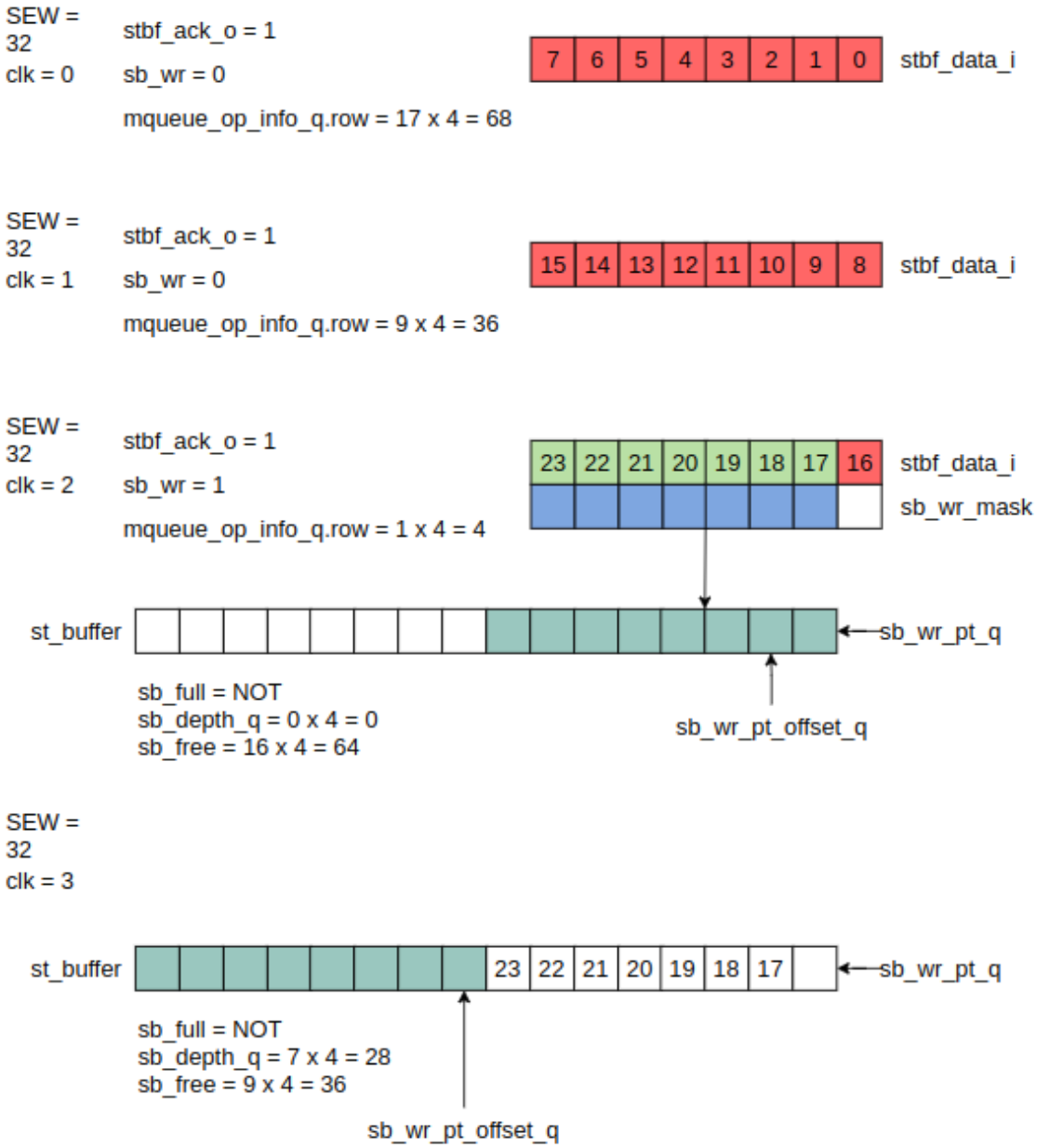

Figure 6.2: SMU write operation example 1. [own creation]

In the data package stbf_data_i we can see invalid elements in red and valid in green. Other colours are, in green sb_wr_pt_q entry and in blue sb_wr_mask valid elements.

Example 2:

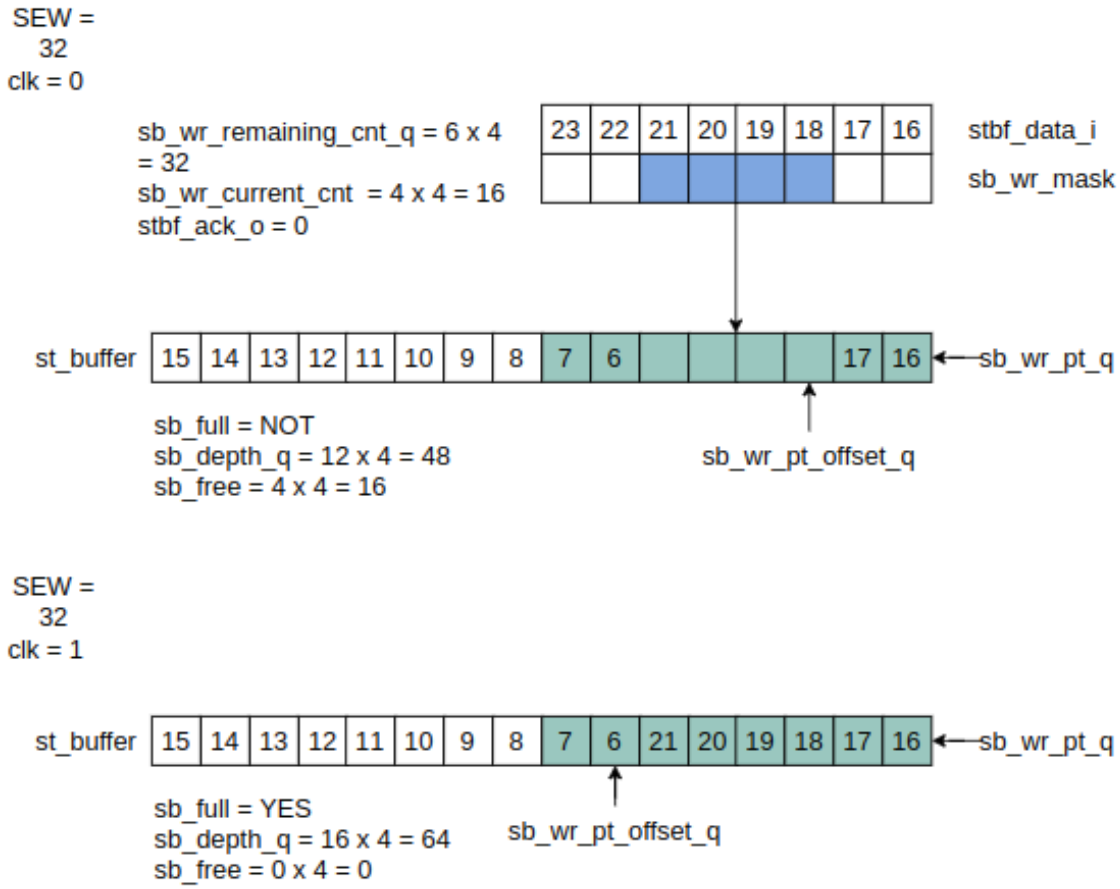

Figure 6.3: SMU write operation example 2. [own creation]

In green sb_wr_pt_q entry, in blue sb_wr_mask valid elements. As sb_free < sb_wr_remaining_cnt_q, sb_wr_current_cnt is set equals sb_free.

- Read Buffer Operation

Signal	Type	Description
sb_rd	logic	Buffer Reading
sb_rd_pt_offset_q	l2_cache_line_idx_t	Buffer offset reading pointer
rqueue_rd_byte_cnt	elem_count_t	Reading bytes, specified by Store Request

Table 6.14: SMU Buffer read signals. [Own creation]

In a read operation, sb_rd_pt_offset_q indicates the first element to be read. Read condition is rqueue_rd_byte_cnt $\geq$ *sb_depth_q*.

Example:

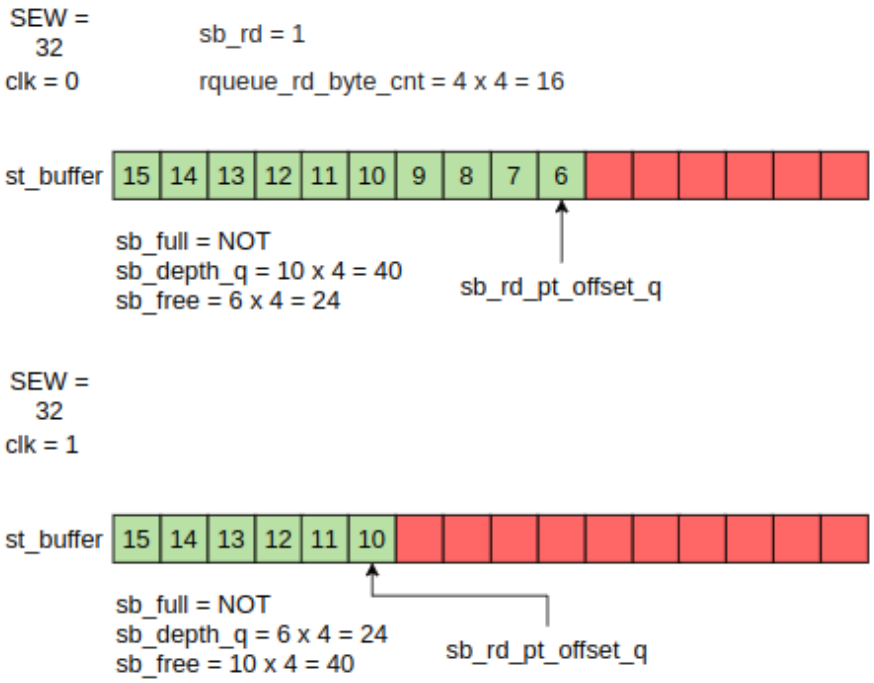

Figure 6.4: SMU read operation example 1. [own creation]

In green valid elements, in red invalid elements.

Combinatorial logic path

This logic conforms a datapath which is combinatorial and have 4 stages, this is the rearranging logic. Each stage generates a l2 cache line size output according to Store Request Parameters.

- Barrel shifter

  The Barrel shifter input is the state of the Internal Buffer. Data elements are circular shifted to the right according to sb_rd_pt_offset_q. This stage makes possible to rearrange the data according to the rqueue_req_info_q.elem_id.

  Example:

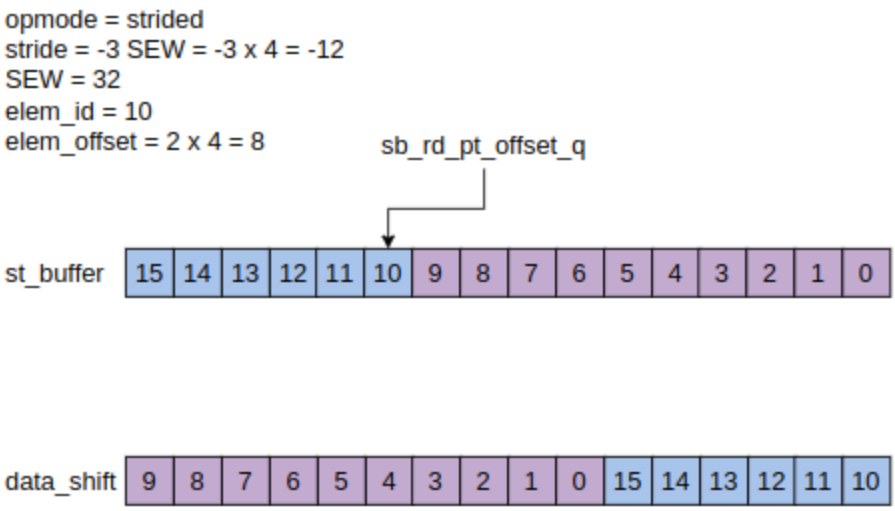

Figure 6.5: SMU Barrel Shifter example. [own creation]

RTL circuit:

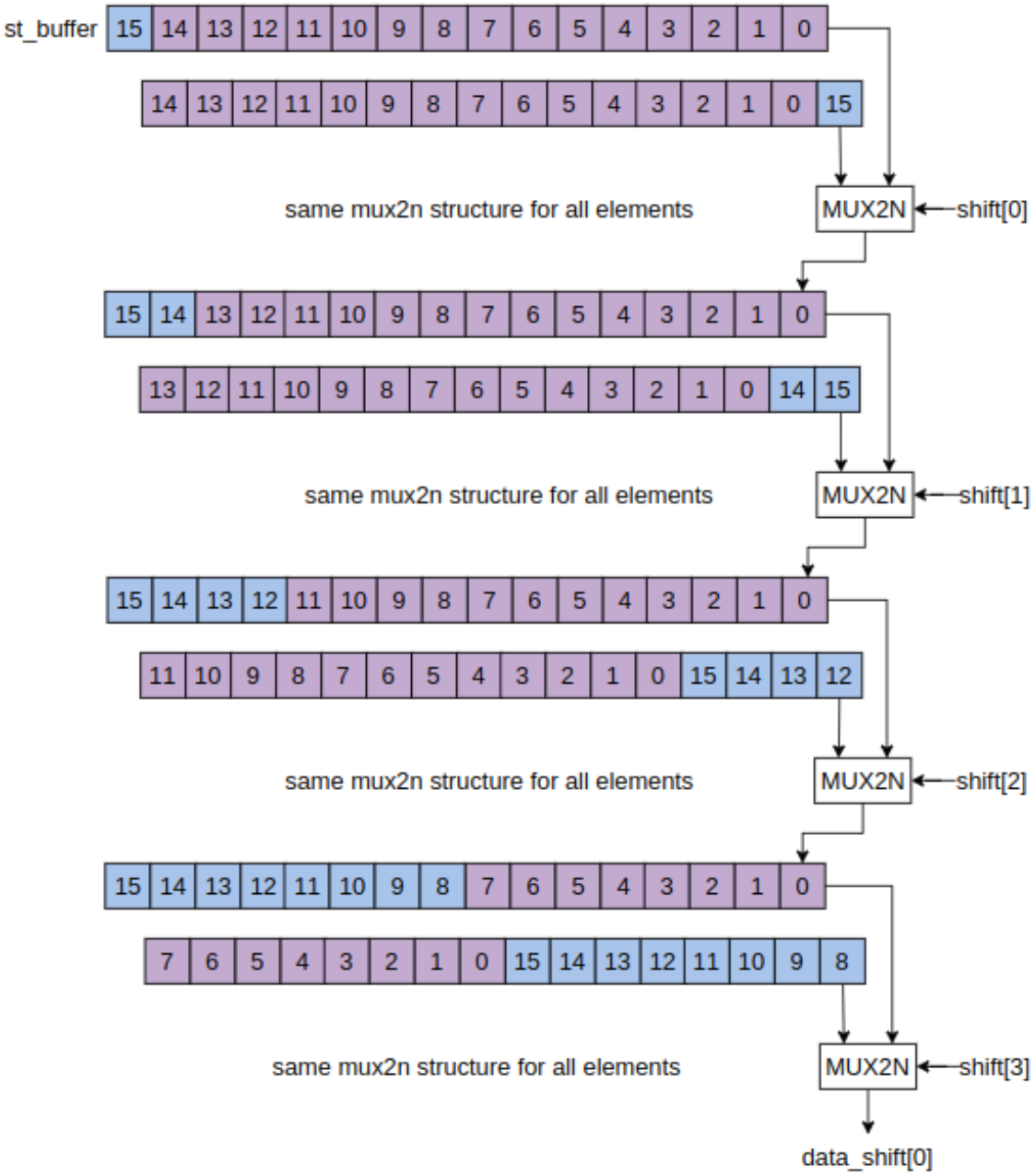

Figure 6.6: SMU Barrel Shifter RTL. [own creation]

This implementation is an example in which the width of the structure is composed of only 16 elements of undefined size. Design can also be found in [16].

- Strider

  Circular shifted elements are strided to the left according to rqueue_req_info_q.vsew and the absolute value of rqueue_req_info_q.stride.

  Example:

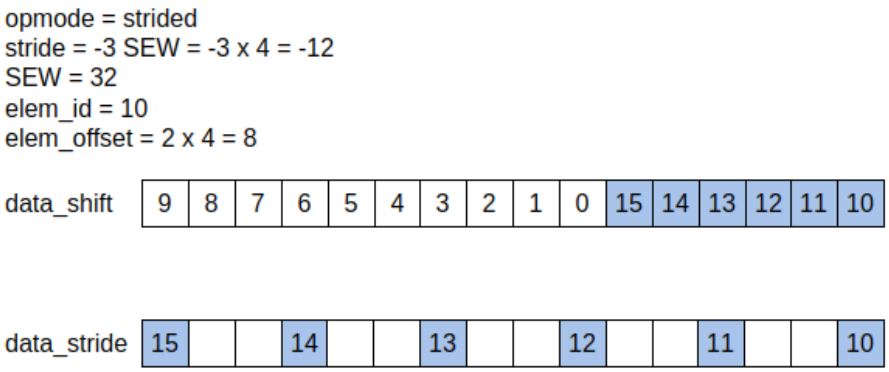

Figure 6.7: SMU Strider example. [own creation]

As it can be seen in 6.7, stride sign is not used in this phase.

RTL circuit:

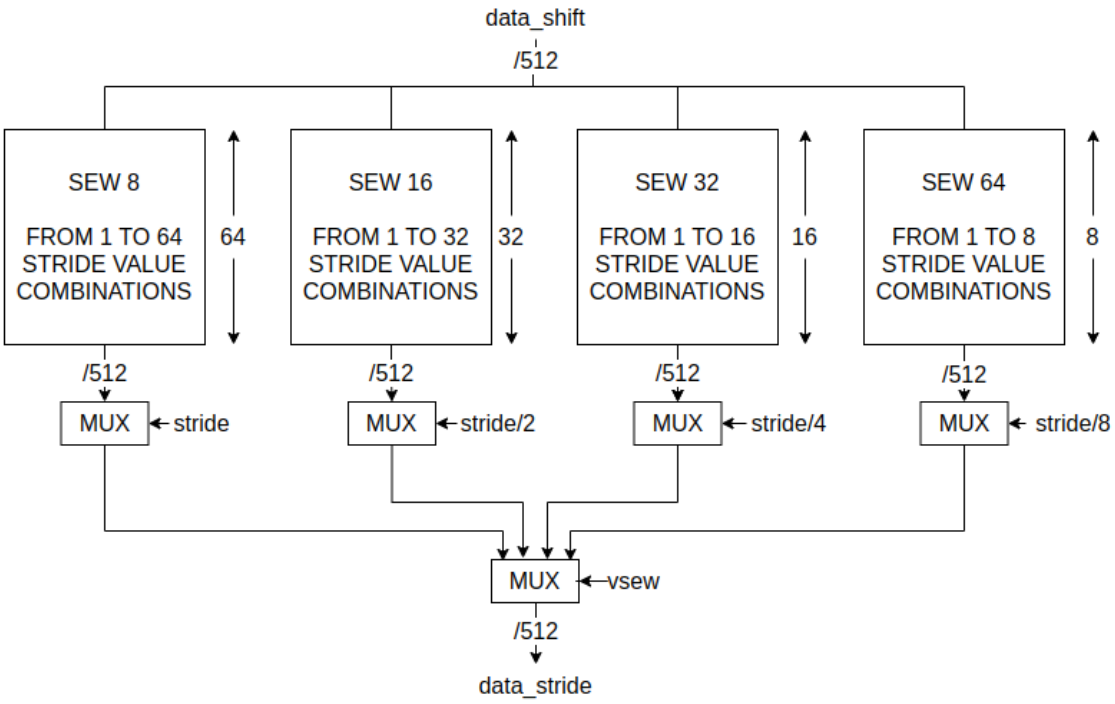

Figure 6.8: SMU Strider RTL. [own creation]

As stated previously, this implementation makes possible to use the scoped stride values, this is a important functionality for HPC applications. When stride $\geq$ 64, the stride index value is set to 64. Also, stride 0 and non-multiple strides of vsew are not supported.

- Shifter

  Strided elements are shifted to the left according to rqueue_req_info_q.elem_offset. Example:

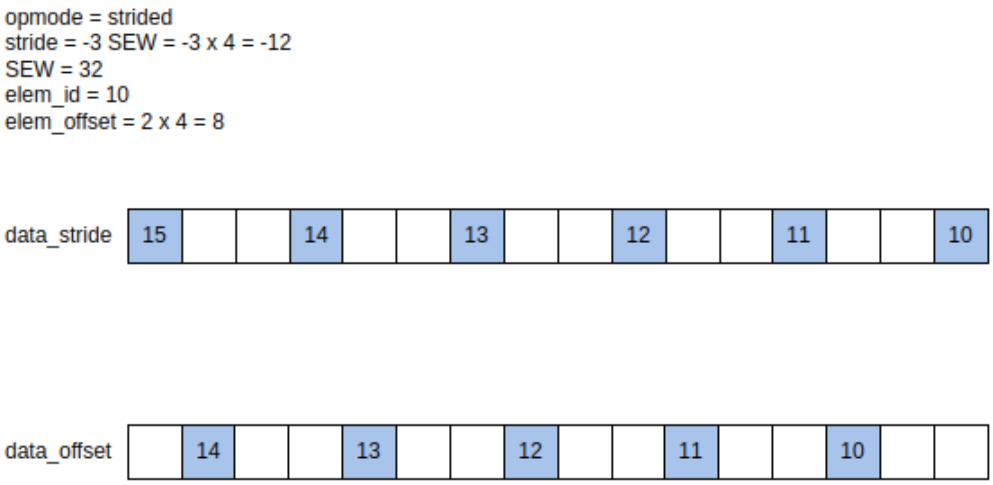

Figure 6.9: SMU Shifter example. [own creation]

- Inverter

  Shifted elements are order inverted if (rqueue_req_info_q.stride < 0), taking into account rqueue_req_info_q.vsew.
  Example:

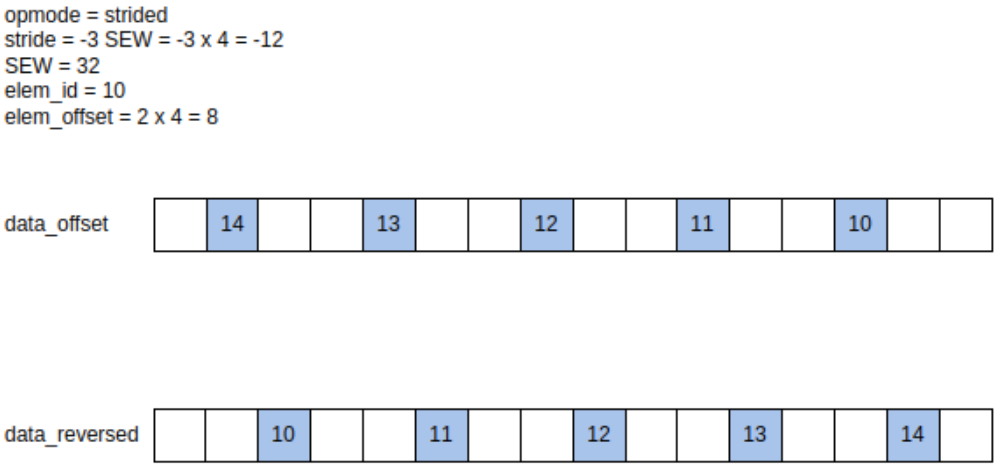

Figure 6.10: SMU Inverter example. [own creation]

Store Data Response

A Store response is served if a Buffer Read Operation is successfully performed or rqueue_req_info_q.kill is asserted (i.e. that request has been computed). In the second case, rqueue_resp_info_o.data is not valid.
When a Store data response is served, rqueue_req_info_q is discarded and if rqueue_req_info_q.kill or rqueue_req_info_q.last is asserted, mqueue_op_info_q.row is also discarded. This is because either of those 2 signals indicates the last request and instruction information has to be cleared as well.
Signal value assignation: rqueue_resp_info_o.vmot_id, rqueue_resp_info_o.kill, rqueue_resp_info_o.tag, rqueue_resp_info_o.line_mask are bypassed from rqueue_req_info_q.

rqueue_resp_info_o.data is obtained from **combinatorial logic path**.

Reset logic

When a Store instruction has been finished (i.e. all requests have been computed) Internal Buffer state signals are cleared and stbf_sync_end_o is asserted. This signal commands store buffers that vstore instruction computation has finished.
The SMU asserts this signal as this is the module has to notify the Store Buffers that the vstore has finished.

6.2.4 Testing

The tests include:

1. vstores that fill the buffer at distinct sb_depth_q values. This allows to check the correctness of the internal buffer.

2. Different requests values: strided and indexed instructions, all supported vsew values, distinct stride values including stride values over 64 bytes(Cache line size), negative stride values. This allows to check the correctness of the combinatorial path.

3. Distinct aqueue_cnt_row_q values. this allows to check the use of masks, the mapping of the combinational path and the mechanisms of the Internal Buffer.

6.3 Store Buffer

This module is in charge of managing the data values flow from the VRF to the SMU in a store operation. Its main function is the serialization of this data flow. This is needed because data values are read from the 5 ports of the VRF inside the VPU Lane.

6.3.1 Interface

Store Buffer is instantiated inside Vector Lane and is connected to Address Queue, Vector Register File and Store Management Unit.

Top: Vector Lane (vlane)

- clock and reset signals.

Signal	Type	Description
clk_i	logic	clock signal
rsn_i	logic	reset signal (active low)

Table 6.15: STBF Vector lane interface. [Own creation]

Address Queue (mqueue)

- vstore instruction information.
  Synchronization is implicit, so mqueue_sync_start_i is only asserted when SMU is available.

Signal	Type	Description
mqueue_sync_start_i	logic	start store instruction
mqueue_info_i	mqueue2stbf_t	store instruction info

Table 6.16: STBF Vector lane interface. [Own creation]

Vector Register File (VRF)

- Vector Register File line data source.
  Synchronization is explicit via a handshake protocol, but data is delivered with 1 cycle of delay.

Signal	Type	Description
vrf_req_o	logic	data request signal
vrf_resp_i	logic	request accepted signal
vrf_info_o	stbf2vrf_t	data request address info
vrf_data_i	logic [N_LANES × N_BANKS]	delayed vector register data

Table 6.17: STBF VRF interface. [Own creation]

Store Management Unit (SMU)

- Data packages send in order.
  Synchronization is explicit via a handshake protocol.

Signal	Type	Description
smu_valid_o	logic	SMU data valid
smu_ack_i	logic	SMU data acknowledgment
smu_data_o	logic [ELEN]	SMU data

Table 6.18: STBF SMU data interface. [Own creation]

- End vstore instruction signal.
  Synchronization is implicit, so smu_sync_end_i is only asserted when Store Buffer has finished computing a vstore instruction but it is still in flight.

Signal	Type	Description
smu_sync_end_i	logic	reset signal (active high)

Table 6.19: STBF SMU reset interface. [Own creation]

6.3.2 Data Types

Signal	Type	Description
line_offset	logic [$clog2(N_LANES × N_BANKS × ELEN / MIN_SEW)]	line offset
preg	preg_t	physical register

Table 6.20: mqueue2stbf_t elements. [Own creation]

line_offset value consists on the line offset of a vector instruction extracted from vstart value. Indicates which is the first Buffer Data line for the first request in a certain vstore instruction.
Buffer data treatment will be shown below. So, this signal usage will be clarified.

Signal	Type	Description
saddr	vrf_addr_t [N_BANKS]	VRF access address
preg	preg_t	physical register

Table 6.21: stbf2vrf_t elements. [Own creation]

6.3.3 Architecture

This module is a pipelined design which consists of 3 stages: Combinatorial, VRF and Buffer.
Besides pipeline path, there are two extra components which are SMU Data set and Reset logic.

Combinatorial stage

This stage holds the data that will be used in following stages.

Signal	Type	Description
comb_valid_q	logic	valid comb stage signal
comb_line_offset_q	logic [VEC_LINES_OFFSET-1:0]	comb stage line offset
comb_phy_reg_q	preg_t	comb stage physical register

Table 6.22: STBF comb stage signals. [Own creation]

Control logic cases:

- invalid: This is the invalid instruction condition, in which comb_valid_q is asserted to 0.
- new: New instruction condition, comb_valid_q is asserted to 1. Also, comb_line_offset_q and comb_phy_reg_q are initilased with mqueue_info_i.line_offset and mqueue_info_i.preg respectively.
- hold: Stall condition.
- next: Next instruction condition, comb_line_offset_q is increased by one, in order to access the consecutive line of the VRF.

The stall condition occurs when comb or VRF stage gets into hold condition. Comb hold condition happens when vrf_req_o is asserted but vrf_resp_i is not.
The invalid condition occurs when there's no valid data in the pipeline.

Vector Register File stage

This stage generates the address used to access the VRF and performs the access.

Signal	Type	Description
vrf_line_offset	logic [VEC_LINES_OFFSET-1:0]	VRF source line offset, equals comb_phy_reg_q
vrf_phy_reg	preg_t	VRF source physical register, equals comb_phy_reg_q
vrf_addr_calc	logic [N_BANKS-1 : 0][VRF_ADDR-1 : 0]	VRF source address

Table 6.23: STBF VRF stage signals 1. [Own creation]

In **previous stage** VRF access was requested (vrf_req_o). As VRF is implemented with RAM, if the access was granted (vrf_resp_i), vrf_data is obtained in **this stage**.

vrf_addr_calc is an address array which contains an independent value for each bank. To calculate vrf_addr_calc, vrf_mapping and vrf_line_offset are added vectorially.

vrf_req_o is set with comb_valid_q, vrf_info_o.saddr is set with vrf_addr_calc and vrf_info_o.preg is set with vrf_phy_reg.

Vector Register File example:

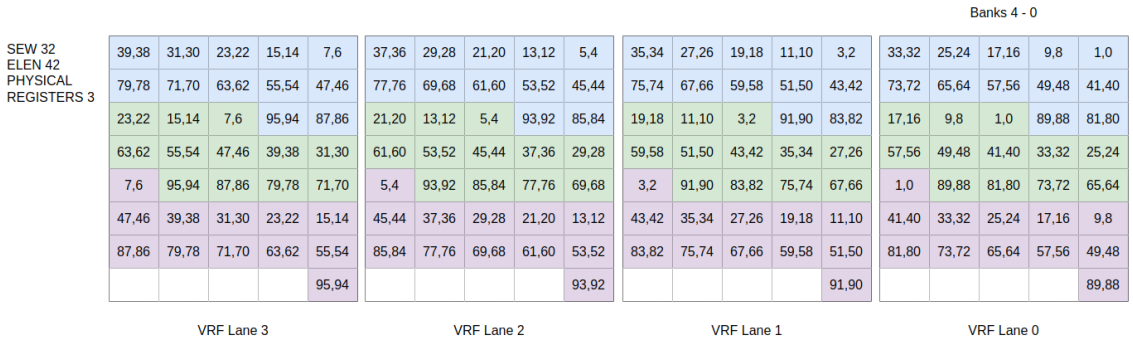

Figure 6.11: STBF VRF SET example. [own creation]

In this representation of a parameterized VRF, we can see how, in registers with an id multiple of N_BANKS, lines are found in a single row, while in those that doesn't, lines are spread in 2 rows.

VRF data shuffle mechanism example: Using previous VRF representation example for Lane 0:

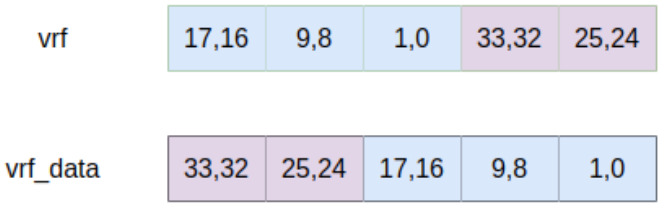

Figure 6.12: STBF VRF shuffle example. [own creation]

It's remarkable that accesses realised to registers with an id that is not multiple of N_BANKS, must be shuffled in order to order them. This functionality is included inside the VRF.

Data access example: Using previous VRF representation example for Lane 0:

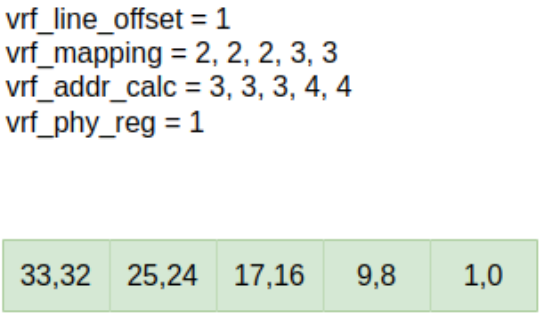

Figure 6.13: STBF VRF access example. [own creation]

6.13 shows vrf_addr_calc value and the accessed values inside the VRF.

Signal	Type	Description
vrf_valid_q	logic	valid VRF stage signal
vrf_data	st_buff_data_t	VRF stage data

Table 6.24: STBF VRF stage signals 2. [Own creation]

Control logic cases:

- invalid: This is the invalid VRF data condition, in which vrf_valid_q is asserted to 0.
- new: Valid VRF data condition, vrf_valid_q is asserted to 1. Also, vrf_data takes value vrf_data_i.
- hold: Stall condition.

The stall condition occurs when VRF stage gets into hold condition. VRF hold condition happens when vrf_valid_q and sb_full are asserted.

The invalid condition occurs when a comb stall condition occurs and there's no VRF stall condition, or there's no valid data in the pipeline.

Buffer stage

This stage stores data inside Internal Buffer.

Buffer entry size is the same as a data packages read from the VRF.
The number of entries is ST_BF_SIZE (2).

Signal	Type	Description
sb_full	logic	Buffer full occupation
sb_empty	logic	Buffer zero occupation
sb_depth_q	st_bf_elen_num_t	Buffer occupation counting value

Table 6.25: STBF buffer state signals. [Own creation]

- Write Buffer Operation

Signal	Type	Description
sb_wr	logic	Buffer writing
sb_wr_pt_q	st_bf_pt_t	Buffer entry writing pointer
buffer_valid	logic	buffer data valid
buffer_data	st_buff_data_t	buffer data

Table 6.26: STBF buffer write signals. [Own creation]

buffer_valid is set with vrf_valid_q and buffer_data with vrf_data.
In a write operation, sb_wr_pt_q indicates the correct entry of the buffer.
Write granularity is st_buff_data_t.
Example:

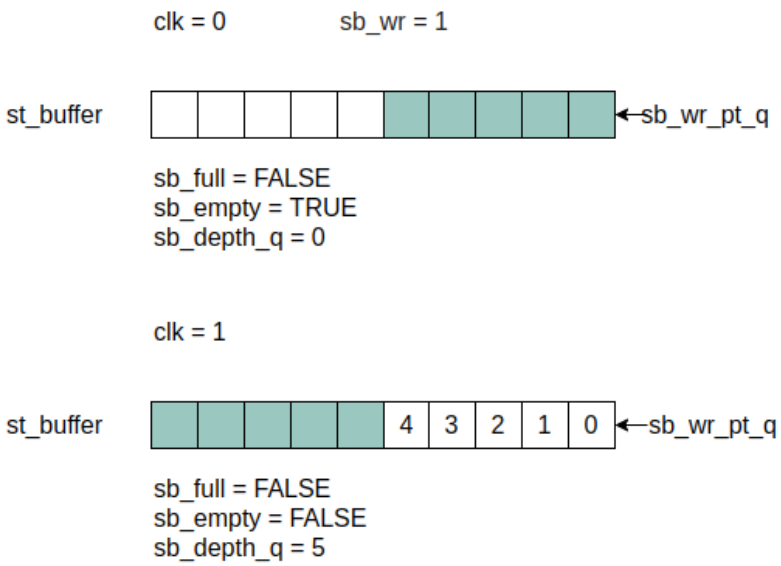

Figure 6.14: STBF write operation example. [own creation]

In green sb_wr_pt_q entry.

- Read Buffer Operation

Signal	Type	Description
sb_rd	logic	Buffer Reading
sb_rd_pt_q	st_bf_pt_t	Buffer offset entry reading pointer
sb_rd_pt_offset_q	n_banks_id_t	Buffer entry reading pointer

Table 6.27: STBF buffer read signals. [Own creation]

In a read operation, sb_rd_pt_q indicates the correct entry of the buffer, sb_wr_pt_offset_q indicates the pointer inside the chosen vector entry.
Read granularity is elen_t.

Example:

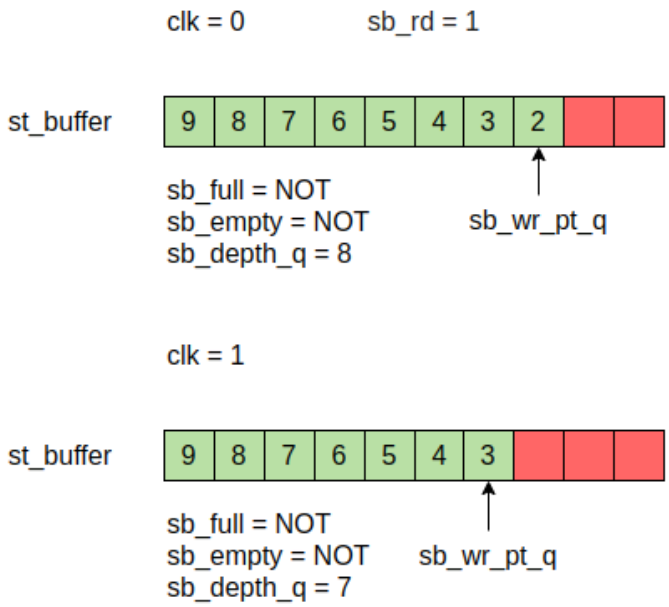

Figure 6.15: STBF read operation example. [own creation]

In green valid elements, in red invalid elements.

SMU Data

Data read from Buffer Stage is used unmodified in SMU Interface.

smu_data_o is set with st_buffer[sb_rd_pt_q][sb_rd_pt_offset_q].
smu_valid_o is set with !sb_empty.

Reset logic

When smu_sync_end_i is asserted, comb and VRF stage control signals are set to invalid and buffer stage signals are cleared.
The SMU asserts this signal as that is the module has to notify the Store Buffers that the vstore has finished.

6.3.4 Testing

The tests include:

1. Vector Memory Indexed instruction that fill the buffer at distinct sb_depth_q values. This allows to check the correctness of the internal buffer.
2. Vector Memory Indexed instruction with different vsew values. This allows to check the correctness of the combinatorial path.
3. Distinct vrf_line_offset and vrf_phy_reg values. this allows to check the correct access to the VRF and address calculation.
4. Cover the possible states of the pipeline.

6.4 Index Management Unit

This module provides support to Vector Memory Indexed instructions. Its main functionality is to move elements from the Index Buffers to the Address Generation Unit using instruction vsew value.

6.4.1 Interface

IMU is instantiated inside Vector Load Store Unit and is connected to Address Queue, Address Generation Unit and Index Buffer.

Top: Vector Load Store Unit

- clock and reset signals.

Signal	Type	Description
clk_i	logic	clock signal
rsn_i	logic	reset signal (active low)

Table 6.28: IMU VLSU interface. [Own creation]

Address Queue

- Vector Memory Indexed instruction information.
  Synchronization is implicit, so mqueue_sync_start_i is only asserted when IMU is available.

Signal	Type	Description
mqueue_sync_start_i	logic	start memory indexed instruction
mqueue_info_i	mqueue2imu_t	memory indexed instruction info

Table 6.29: IMU Address queue information interface. [Own creation]

- End Vector Memory Indexed instruction.
  Synchronization is implicit, so mqueue_sync_end_i is only asserted when IMU has finished computing a Vector Memory Indexed instruction but it is still in flight.

Signal	Type	Description
mqueue_sync_end_i	logic	reset signal (active high)

Table 6.30: IMU Address queue reset interface. [Own creation]

Address Generation Unit

- Vector Memory Indexed information.
  Synchronization is explicit via a handshake protocol.

Signal	Type	Description
agu_index_valid_o	logic	index element valid
agu_index_ack_i	logic	index element acknowledged
agu_index_o	logic[ELEN]	index element info

Table 6.31: IMU Address Generation Unit interface. [Own creation]

Index Buffer

- Buffer index elements coming in order.
  Synchronization is explicit via a handshake protocol.

Signal	Type	Description
idxbf_valid_i	logic	clock signal
idxbf_ack_o	logic	reset signal (active low)
idxbf_data_i	buffers_data_t	reset signal (active low)

Table 6.32: IMU Index Buffer data interface. [Own creation]

6.4.2 Data Types

Signal	Type	Description
row	logic [$clog2(N_LANES × N_BANKS × ELEN / MIN_SEW)]	row offset
vsew	vsew_t	vsew value

Table 6.33: mqueue2imu_t elements. [Own creation]

buffers_data_t is a logic [N_LANES × ELEN].

6.4.3 Architecture

The IMU consists of a design which consists of 5 components: Instruction info logic, an internal buffer, a combinatorial logic path, AGU data and reset logic.

Instruction info logic

Signal	Type	Description
op_valid_q	logic	Instruction valid signal
mqueue_cnt_row_q	logic [ELEMENTS_WIDTH_LANE-1:0]	Auxiliary Instruction info
mqueue_vsew_q	vsew_t	Instruction VSEW

Table 6.34: IMU AGU info logic signals. [Own creation]

mqueue_cnt_row_q is a special signal, used for 3 reasons:

- Discard invalid Index Buffer data packages.
- Discard invalid elements from a valid Index Buffer data package.
- Initialize ib_wr_pt_offset_q and ib_rd_pt_offset_q.

Internal Buffer

This phase is the same as the used in the SMU, with an appropriate change of the names of the signals.

Combinatorial logic path

This logic conforms a datapath which is combinatorial and have 2 stages, this logic selects the correct element and sign extends it to elen_t.

- Shifter
  Internal Buffer elements are shifted to the left according to ib_rd_pt_offset_q.
  Example:

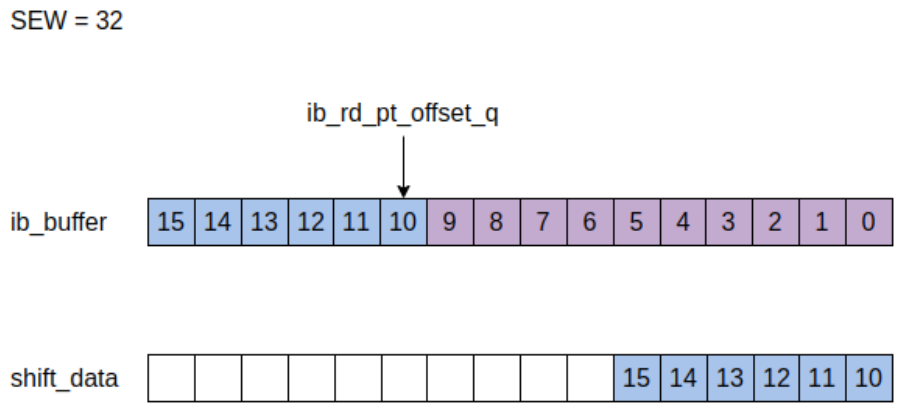

Figure 6.16: IMU shifter example. [own creation]

- Extend Sign
  imu_b_data_out is sign extended depending on SEW, up to 64 bits.
  Example:

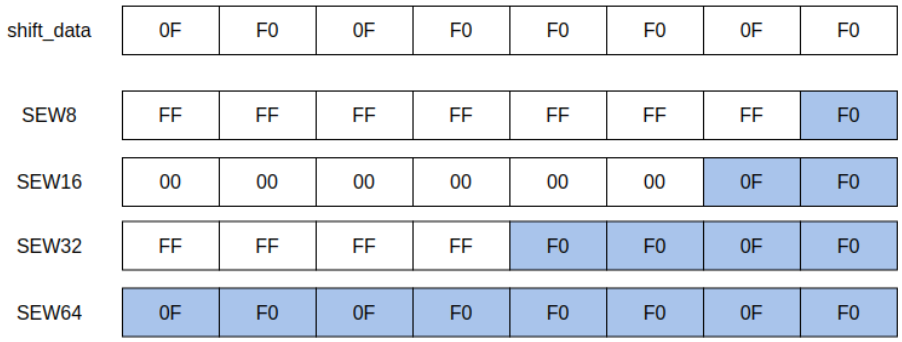

Figure 6.17: IMU extend sign example. [own creation]

Sign extend for all SEW. In blue, valid bytes.

AGU Data

imu_data_o is set with Combinatorial logic path data.
agu_index_valid_o is set with op_valid_q && (ib_depth_q $\geq$ vsew_bytes(mqueue_vsew_q)).

Reset logic

When mqueue_sync_end_i is asserted, comb and VRF stage control signals are set to invalid and buffer stage signals are cleared. The Address Queue asserts this signal as that is the module that communicates with the AGU and knows when the computation of the indexes has finished.

6.4.4 Testing

The tests include:

1. Vector Memory Indexed instruction that fill the buffer at distinct sb_depth_q values. This allows to check the correctness of the internal buffer.
2. Different vsew values. This allows to check the correctness of the combinatorial path.
3. Distinct aqueue_cnt_row_q values. this allows to check the use of masks, the mapping of the combinational path and the mechanisms of the Internal Buffer.

6.5 Index Buffer

This module is in charge of managing the index values flow from the VRF to IMU in a memory index operation.
Its main function is the serialization of this data. This is needed because data values are read from the 5 ports of the VPU lane.

6.5.1 Interface

Index Buffer is instantiated inside Vector Lane and is connected to Address Queue, Vector Register File and Index Management Unit.

Top: Vector Lane

- clock and reset signals.

Signal	Type	Description
clk_i	logic	clock signal
rsn_i	logic	reset signal (active low)

Table 6.35: IDXBF Vector lane interface. [Own creation]

Address Queue

- Vector Memory Indexed instruction information.
  Synchronization is implicit, so mqueue_sync_start_i is only asserted when SMU is available.

Signal	Type	Description
mqueue_sync_start_i	logic	start memory indexed instruction
mqueue_info_i	mqueue2idxbf_t	memory indexed instruction info

Table 6.36: IDXBF Vector lane interface. [Own creation]

- End Vector Memory Indexed instruction.
  Synchronization is implicit, so mqueue_sync_end_i is only asserted when Index Buffer has finished computing a Vector Memory Indexed instruction but it is still in flight.

Signal	Type	Description
mqueue_sync_end_i	logic	reset signal (active high)

Table 6.37: IMU Address queue reset interface. [Own creation]

Vector Register File (VRF)

- Vector Register File line data source.
  Synchronization is explicit via a handshake protocol, but data is delivered with 1 cycle of delay.

Signal	Type	Description
vrf_req_o	logic	data request signal
vrf_resp_i	logic	request accepted signal
vrf_info_o	idxbf2vrf_t	data request address info
vrf_data_i	logic [N_LANES × N_BANKS]	delayed vector register data

Table 6.38: STBF VRF interface. [Own creation]

Index Management Unit (IMU)

- Index elements send in order.
  Synchronization is explicit via a handshake protocol.

Signal	Type	Description
imu_valid_o	logic	IMU data valid
imu_ack_i	logic	IMU data acknowledgment
imu_data_o	elen_t	IMU data

Table 6.39: IDXBF IMU data interface. [Own creation]

6.5.2 Data Types

In this section, the data types used are the same as in the SMU, with an appropriate change of the names of the signals.

6.5.3 Architecture

In this section, the architecture used is the same as in the SMU, with an appropriate change of the names of the signals.

6.5.4 Testing

In this section, the tests used are the same as in the SMU, with an appropriate change of the names of the signals.

Bibliography

[1] SCIENTIFIC COMPUTING WORLD. *How to make HPC happen in Europe*. URL: https://www.scientific-computing.com/feature/how-make-hpc-happen-europe. (accessed: 28.09.2022).

[2] EU. *Discover EuroHPC JU*. URL: https://eurohpc-ju.europa.eu/about/discover-eurohpc-ju_en. (accessed: 29.09.2022).

[3] EuroHPC JU EU. *About eProcessor*. URL: https://eprocessor.eu/about/. (accessed: 29.09.2022).

[4] David A. Patterson John L. Hennessy. *Computer Architecture: A Quantitative Approach (Sixth Edition)*. Elsevier, 2019. ISBN: 9780128119051. (accessed: 2.10.2022).

[5] R. Espasa and M. Valero. "Decoupled vector architectures". In: *Proceedings. Second International Symposium on High-Performance Computer Architecture*. 1996, pp. 281–290. DOI: 10.1109/HPCA.1996.501193. (accessed: 2.10.2022).

[6] Kevin Thompson. *Hardware vs. Software Development: Similarities and Differences*. URL: https://www.cprime.com/resources/blog/hardware-vs-software-development-similarities-and-differences/. (accessed: 5.10.2022).

[7] Kevin Thompson. *What is Agile Hardware Development?* URL: https://www.cprime.com/resources/blog/what-is-agile-hardware-development. (accessed: 5.10.2022).

[8] Universidad Alfonso X El Sabio. *¿Cuánto gana un ingeniero informático? Descubre su sueldo*. URL: https://www.uax.com/blog/ingenieria/cuanto-cobra-un-ingeniero-informatico. (accessed: 11.10.2022).

[9] Intel Programmable Solutions. *Questa*-Intel® FPGA Edition Software*. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/questa-edition.html?wapkw=questa. (accessed: 20.10.2022).

[10] arm glossary. *INSTRUCTION SET ARCHITECTURE (ISA)*. URL: https://www.arm.com/glossary/isa. (accessed: 15.11.2022).

[11] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. English. Version 20191213. RISC-V Foundation. (accessed: 16.11.2022).

[12] RISC-V International. *History of RISC-V*. URL: https://www.riscv.org/about/history. (accessed: 20.11.2022).

[13] *RISC-V "V" Vector Extension*. English. Version 0.7.1-20190610-Workshop-Release. RISC-V Foundation. (accessed: 21.11.2022).

[14] lowRISC. *lowRISC Verilog Coding Style Guide*. URL: https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md. (accessed: 22.11.2022).

[15] chipverify. *SystemVerilog TestBench*. URL: https://www.chipverify.com/systemverilog/systemverilog-simple-testbench. (accessed: 27.11.2022).

[16] Neeta Pandey and Saurabh Gupta. “Design and Implementation of Novel Multiplier using Barrel Shifters”. In: *International Journal of Image, Graphics and Signal Processing* 7 (July 2015), pp. 28–34. DOI: 10.5815/ijigsp.2015.08.03. (accessed: 3.12.2022).