POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Design of an Advanced Register Renaming Architecture in a RISC-V Vector Processing Unit

Supervisors

Prof. Guido MASERA

Francesco MINERVINI

Candidate

Lorenzo DELTETTO

July 2025

Summary

Modeling and simulation are essential tools in science, enabling researchers to analyze complex systems and predict their behavior. High Performance Computing (HPC) plays a crucial role by providing the computational power necessary to run large-scale simulations, driving breakthroughs across various scientific fields. For example, an ongoing challenge is climate modeling, which relies on HPC to predict and understand the effects of climate change, helping to guide global mitigation efforts. In addition, recent years have seen machine learning take on a growing role in technological development, enabling progress in areas such as audio and image processing, natural language processing, and autonomous driving. High-performance, efficient computing systems are essential for both training and real-time inference of these models.

To address these challenges, computer architects have leveraged Data-Level Parallelism (DLP), a processing approach in which a single instruction operates on multiple data elements simultaneously. This results in improved performance and reduced demands on instruction and memory bandwidth, contributing to lower power consumption. A key implementation of DLP is found in vector processors, which feature two essential components: a vector register file (VRF), capable of holding a large number of elements, and multiple deeply pipelined functional units (FUs). To further enhance the performance of vector processors, design concepts from superscalar architectures can be applied. This is exemplified by the analyzed RISC-V-V 1.0 Vector Processing Unit (VPU), which features register renaming and lightweight out-of-order execution. This work details the complete design process, from conceptualization to RTL implementation, of an optimized register renaming mechanism. The mechanism introduces a new scheme for vector registers utilization in specific ISA instructions, leading to improvements in both performance and power efficiency. Its effectiveness is evaluated using state-of-the-art benchmarks from high performance computing and machine learning domains, analyzing trade-offs in power consumption and area.

Acknowledgements

I would like to thank Professor Guido Masera for agreeing to supervise this thesis and for preparing me, through his courses, to face the challenges of this work.

My gratitude goes to Francesco Minervini for giving me the opportunity to collaborate with him on this fascinating project. This experience deepened my knowledge and introduced me to both challenging and enriching concepts.

I am also grateful to all my colleagues in the Hardware Design Group at the Barcelona Supercomputing Center for making me feel part of the team from day one. In particular, I would like to thank the entire *Grupo de Apoyo* for welcoming me so warmly to a new city and work environment.

A special thank you goes to all the friends and loved ones who, in one way or another, helped me during my university years, making this journey lighter, richer, and more meaningful.

Finally, I would like to sincerely thank my family, who have been a constant source of support, encouragement, and profound inspiration at every step along the way.

Grazie di cuore.

Table of Contents

List of Tables

List of Figures

Acronyms

ISA
Instruction Set Architecture

VPU
Vector Processing Unit

RAT
Register Alias Table

FRL
Free Register List

FSRL
Free Squeezed Register List

ROB
REORDER BUFFER

VRF
Vector Register File

SVRF
Squeezed Vector Register File

VCU
Vector Control Unit

FSM
Finite State Machine

Chapter 1

Introduction and Motivations

This chapter introduces several key technical concepts, including RISC-V, the vector extension, and register renaming. It also discusses the motivations behind the design choices made in this thesis.

1.1 Introduction

1.1.1 Vector processing

Vector processors implement an instruction set where the instructions are designed to operate on large one-dimensional arrays of data called vectors. This is in contrast to classical General Purpouse processors, whose instructions operate on single data items.

The Cray-1 [1], designed by Seymour Cray in 1975, was the first supercomputer to successfully implement the vector processor architecture and SIMD instructions. Due to the success of this design, vector machines quickly became the preferred choice for supercomputing.

However, advancements in microprocessor integration led, during the 90s, to their replacement by systems featuring multiple out-of-order superscalar cores [2].

The slowdown of Moore's Law [3] is making engineers reconsider SIMD architectures. The primary advantage of these systems lies in their efficiency, as many scientific models inherently exhibit data-parallel characteristics. Furthermore, leveraging data parallelism reduces code size, memory bandwidth requirements and lowers power consumption [4], reducing the need for repetitive instruction fetching and decoding.

1.1.2 Vectorization example

The following snippet shows how vectorization can be used to operate on entire arrays, rather than processing elements one at a time inside a loop.

Starting from a simple vector sum written in C:

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

The following is its equivalent implementation in scalar assembly:

```
li    t0, 64          # Initialize loop counter to 64
loop:
lw    t1, 0(a0)       # Load A[i]
lw    t2, 0(a1)       # Load B[i]
add   t3, t1, t2      # Compute A[i] + B[i]
sw    t3, 0(a2)       # Store result in Sum[i]
addi  a0, a0, 4       # Increment pointer A by 4 bytes (integer)
addi  a1, a1, 4       # Increment pointer B by 4 bytes
addi  a2, a2, 4       # Increment pointer Sum by 4 bytes
addi  t0, t0, -1      # Decrement loop counter
bnez  t0, loop        # Loop if counter not zero
```

The following is its equivalent implementation in vector assembly:

```
li    t1, 64          # Set vector length to 64
setvl t1              # Configure vector length register
vle.v v8, (a0)        # Load vector from A
vle.v v9, (a1)        # Load vector from B
vadd.v v10, v8, v9    # Vector addition of A and B
vse.v v10, (a2)       # Store vector result in Sum
```

Vectorization brings several benefits:

- The number of instructions is reduced, and the program becomes more concise and readable. A smaller instruction count lowers the required *instruction fetch bandwidth*, which is the rate at which a processor retrieves instructions from memory and delivers them to the decoding stage.

- The loop is eliminated, avoiding the need for branch prediction and the latency caused by mispredicted branches. This results in more efficient and faster execution.

1.1.3 RISC-V

RISC-V is an open-source, customizable, and free Instruction Set Architecture (ISA). Started at UC Berkeley in 2010 as an academic project, the standard is now managed by RISC-V International and supported by a wide range of members, from academia to industry [5].

The relevance of RISC-V lies not only in its royalty-free nature and simplicity, but also in its modular architecture, which allows for the inclusion of various extensions [6].

In particular, the Vector Processing Unit studied in this work implements the RISC-V Vector Extension (RVV) Version 1.0, ratified by RISC-V International in 2021 [7]. A key defining feature of the RISC-V Vector ISA is that it is vector length agnostic. Software written for any RISC-V vector-compliant processor will work on any other RISC-V vector processor. This is valuable to the customer from a software reuse perspective.

1.1.4 RVV 1.0: the RISC-V vector extension

The Vector Extension includes 32 vector registers, each with a width of *VLEN* bits. These registers are capable of holding elements up to a maximum size of *ELEN* bits. An important parameter in this context is *VLMAX*, which denotes the maximum number of elements that can be stored in a single vector register. It is defined as the ratio of *VLEN* to *SEW* (Single Element Width), i.e.

$$\text{VLMAX} = \frac{\text{VLEN}}{\text{SEW}}$$

This extension has been designed to allow the same binary code to work across variations in VLEN. In addition to the fundamental instructions typically found in a Vector Architecture (such as move, add, xor, etc.), there are also operations specifically designed to leverage the characteristics of vector computations. Here is a summary of the most important ones:

- **Vector load/store:** These are used to move data between vector registers and memory. These instructions can be *strided* or *indexed.*
  - Strided loads/stores index the vector elements by referring to a starting element and then adding (or subtracting) a certain stride to the base

address. These are particularly fast, especially in cases like unit-strided or optimized powers of 2.

  – Indexed loads/stores use a vector of indexes, added to a base address, to directly select elements. While more flexible, this method is generally slower.

- **Widening/narrowing:** These operations are used to increase or decrease the size of the vector's contents. For example, a multiplication between two 32-bit integers results in a 64-bit output to avoid loss of information. Some operations require the inverse resizing (narrowing).

- **Vrgather:** These are specialized operations that allow indexing a vector using another vector as the index, enabling complex data access patterns. The index values in the second vector are treated as unsigned integers The pseudocode for this operation is:

```
for (i = 0; i < N; ++i)
    x[i] = y[idx[i]];

```

  where `y` is the source vector, `idx` is the index vector, and `x` is the result. Vector-scalar and vector-immediate forms of the vrgather also exists. These special cases allow a scalar or immediate value to be used as the index. A single element is read from the source vector at this index and broadcast to the active elements of the destination vector. Note that the same element is written to all active elements of the destination vector.

  `vrgather.vx vd, vs2, rs1, vm` uses the value in the scalar register `rs1` as the index.

  The pseudocode for this operation is:

```
for (int i = 0; i < VL; ++i)
    if (vmask[i])
        vd[i] = (rs1 >= VLMAX) ? 0 : vs2[rs1];

```

  `vrgather.vi vd, vs2, uimm, vm` uses an unsigned immediate value as the index.

  The pseudocode for this operation is:

```
for (int i = 0; i < VL; ++i)
    if (vmask[i])
        vd[i] = (uimm >= VLMAX) ? 0 : vs2[uimm];

```

- **Reductions:** Apply a binary operator across a group of elements in a vector register, combining them with an initial scalar value stored in element 0. The final reduced result (scalar) is written back to element 0 of the same or another vector register. Example: vredsum.vs vd, vs2, vs1, vm

```
int result = vs1[0];
for (int i = 0; i < VL; ++i) {
    if (vmask[i]) {
        result += vs2[i];
    }
}
vd[0] = result;
```

- **Vector moves:** The vector integer move instructions copy a source operand to a vector register.
  - `vmv.v.v` copies one vector register group to another.
  - `vmv.v.x` and `vmv.v.i` splat a scalar register or immediate to all active elements of the destination vector register.
- **Integer scalar moves:** These instructions transfer a single value between a scalar `x` register and element 0 of a vector register:
  - `vmv.x.s rd, vs2                    # x[rd] = vs2[0] (vs1=0)`
  - `vmv.s.x vd, rs1                    # vd[0] = x[rs1] (vs2=0)`

  The `vmv.x.s` instruction copies a single SEW-wide element from index 0 of the source vector register to a destination integer register. Given that *XLEN* denotes the number of bits in a scalar register
  - If SEW > XLEN, only the least-significant XLEN bits are transferred and the upper SEW - XLEN bits are ignored.
  - If SEW < XLEN, the value is sign-extended to XLEN bits.
- **Floating-Point scalar moves:** Similar to the integer scalar moves, but for floating-point values. These instructions transfer a single value between a scalar floating-point register and element 0 of a vector register.

1.1.5 Register renaming

Register renaming is a technique used in out-of-order processors to eliminate WAR (Write After Read) and WAW (Write After Write) hazards. These dependencies are considered name dependencies and exist only due to the scarcity of architectural registers. To eliminate false dependencies between instructions, enabling more parallelism, the number of available registers can be increased. In particular, the Vector Register File of the VPU contains 40 physical registers, eight more than the 32 logical, ISA-defined registers.

When an instruction reaches the renaming unit, its destination register is mapped to a free physical register. The source registers, previously renamed, are also mapped to their corresponding physical registers.

Multiple implementations of register renaming exist, but the simplest approach is supported by two data structures:

- **Register Alias Table (RAT):** A lookup table with one entry per logical register. Each entry stores the physical register currently associated with that logical register.
- **Free Register List (FRL):** A FIFO queue containing all physical registers that have not yet been allocated. When a new destination register is needed, a register is popped from this list.

Implementation details of the RAT and FRL will be provided in the next chapter.

1.2 Motivations and proposed idea

As highlighted in Section 1.4.1, the RVV 1.0 ISA features vector instructions that produce scalar values as their result:

- **Scalar moves**: both integer and floating point: `vmv.s.x`, `vmv.s.f`
- **Vector moves with scalar source** : `vmv.v.x`, `vmv.v.i`
- **Scalar vrgather**: `vrgather.vx`, `vrgather.vi`
- **Reductions**: e.g., `vredsum.vs`

To optimize the use of register resources, a small additional scalar register file, referred to as the **Squeezed Vector Register File (SVRF)**, is introduced. This register file is not be confused with the register file of the scalar core. The purpose of the SVRF is to store the results of these instructions within the Vector Processing Unit.

With register renaming, an instruction can only be issued if physical registers are available. This optimization prevents the inefficient allocation of an entire vector register, such as one capable of holding 256 double-precision elements, when only a single scalar value needs to be stored. Instead, the scalar result is written directly to the SVRF, providing a more compact and efficient storage solution.

In a standard vector processing unit, scalar move instructions (e.g., `vmv.v.i`, `vmv.v.x`, `vmv.s.x`, `vmv.s.f`) are handled similarly to other arithmetic operations: a physical register is allocated through renaming, the instruction is issued to the lanes and the value is written to the Vector Register File. With this optimization, however, the scalar result is written directly to the Squeezed Register File during the renaming stage, completing the operation in a single clock cycle. This frees the previously allocated vector register (the old destination) and improves overall performance by reducing execution time, since these instructions no longer enter the lanes pipeline, saving multiple clock cycles.

This approach not only optimizes register usage but also reduces power consumption. In instructions like `vmv.v.i` and `vmv.v.x`, all active elements of the destination vector register are written. This leads to unnecessary power consumption, as the same scalar value may be written up to 256 times, in contrast to a single write operation to the SVRF enabled by this optimization. In addition, all the scalar moves are executed directly at the renaming stage, bypassing the lanes pipeline and avoiding the activation of internal buffers and control logic. This results in additional dynamic power savings.

Chapter 2

Context

2.1 VPU microarchitecture

This section describes the microarchitecture of the VPU.

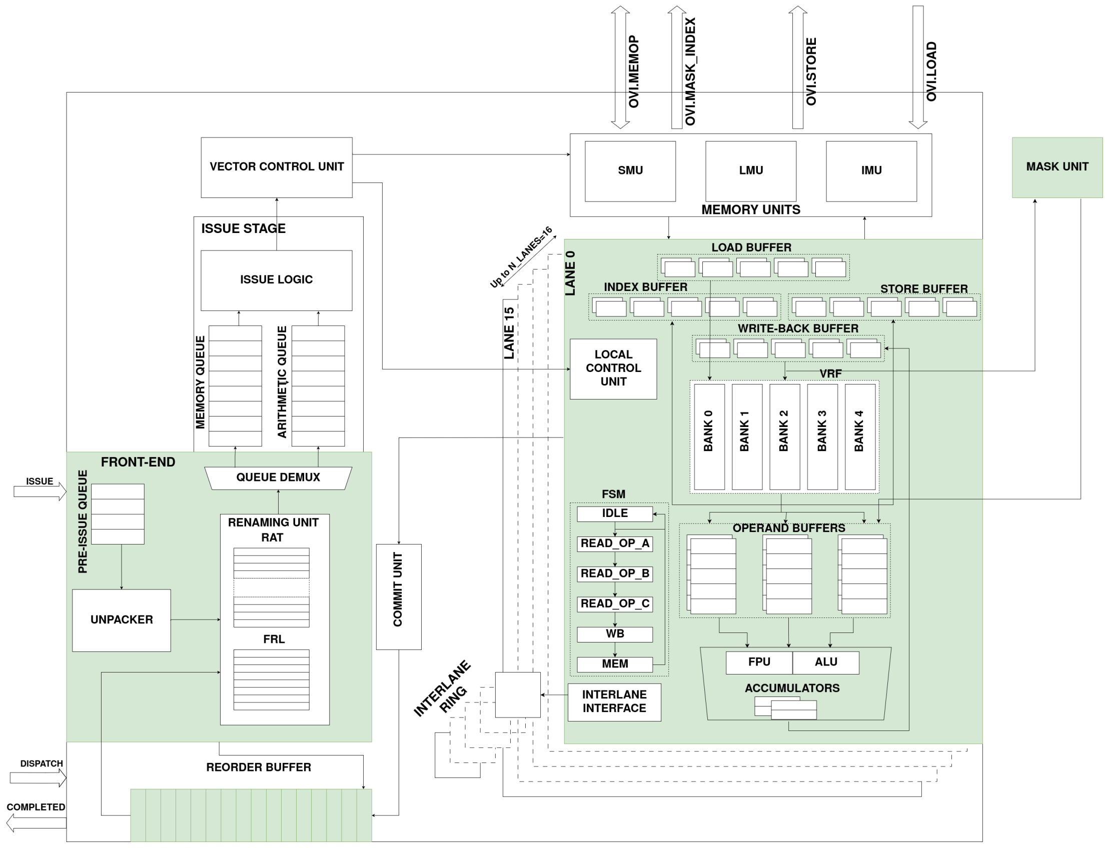

Figure 2.1: VPU architecture

The VPU presented in this work is fully designed at BSC and it is designed to operate as a co-processor to a scalar core also developed at BSC [8].

2.1.1 Front End

The front end acts as the interface responsible for receiving and decoding instructions from the scalar core. It is composed of two main modules:

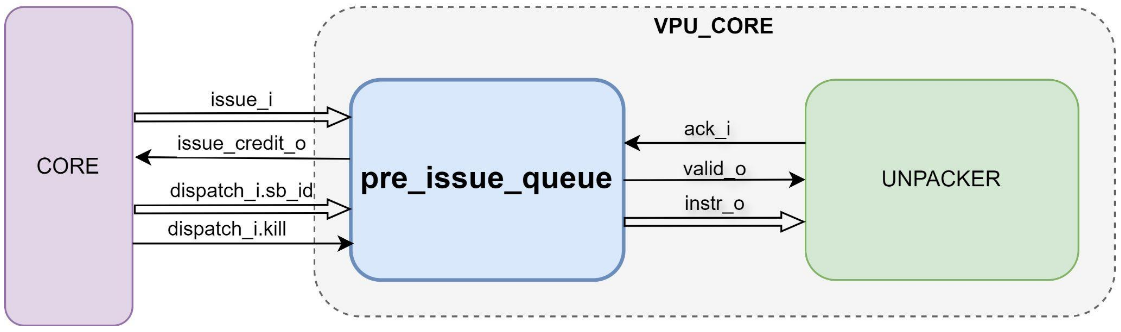

Figure 2.2: Front End stage

- **Pre-Issue Queue**: handles communication with the interface of the scalar core.
- **Unpacker**: decodes instructions using the `OPCODE`, `FUNCTION6`, and `FUNCTION3` fields. The decoded information is packed into appropriate structures and forwarded to the subsequent VPU modules. Instruction flags are set based on the operation type, such as widening, narrowing, slide-up, or slide-down. Most importantly, this component classifies each instruction as either arithmetic or memory-related, allowing it to be routed to the appropriate queue.
- **Renaming Unit**: further detailed in the next chapter, this unit resolves false dependencies such as write-after-read (WAR) and write-after-write (WAW) through register renaming. It contains a Register Alias Table (RAT) that maps logical registers to physical ones, and a Free Register List (FRL) that tracks available physical registers. By eliminating naming conflicts, it prevents data hazards and ensures correct instruction execution.
- **Queue Demultiplexer**: simple combinational module that separates memory and arithmetic instruction streams and dispatches them to different queues, enabling decoupled execution.

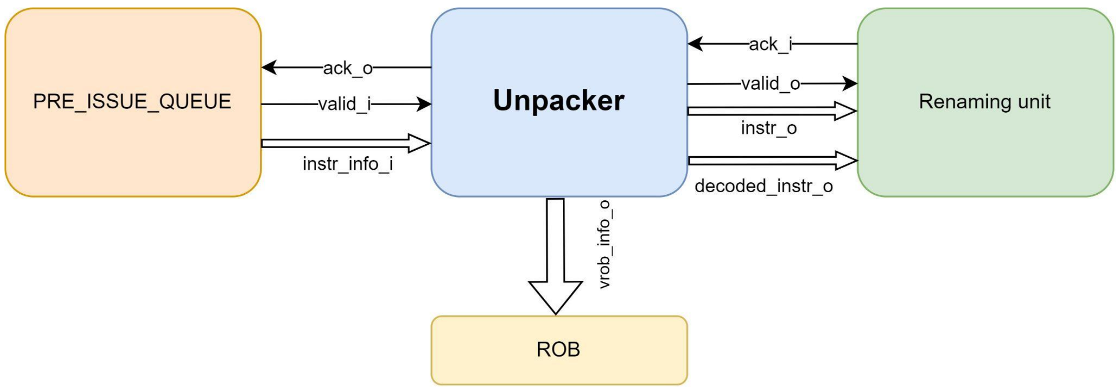

Figure 2.3: Unpacker

2.1.2 Issue Stage

The issue stage includes the arithmetic queue, the memory queue, the Valid Bits module, and the Mask Valid Bits module.

After leaving the front end, instructions are routed to either the arithmetic or memory queue. Each instruction at the head of the FIFO is issued to the lanes as soon as its requirements are met, which depend on whether the instruction is overlappable. Overlapping means back-to-back execution, allowing a new instruction to start before the previous one finished, maximizing vector pipeline utilization. This behavior is controlled by the issue stage logic. Overlappable instructions are sent directly, in order, to the lanes, while non-overlappable instructions send a request signal and are issued only after receiving a grant.

The Valid Bits module monitors register availability. The arithmetic queue sends enable signals and physical addresses to this module, which may activate one, two, or three source operands depending on the instruction.

When an instruction is renamed, the valid bit of its destination physical register is cleared, and set again only after the instruction writes to that register. Since source operands become ready at different times, the Valid Bits module tracks groups of physical registers to finely control operand readiness.

The Mask Valid Bits module functions similarly but focuses on masked instructions, tracking mask register valid bits. It uses a two-dimensional structure to handle instructions composed of two micro-instructions.

2.1.3 Memory Units

The VPU does not have direct access to the memory hierarchy; instead, the scalar core handles memory accesses for vector memory operations. The memory system consists of the Load Management Unit and the Store Management Unit, along with their associated buffers.

Load Path:

- *Load Management Unit (LMU)*: responsible for managing load requests from the VPU and performing the necessary data processing. When a full memory line arrives at the LMU, the offset must be removed, and the correct data selected and compacted based on the stride. The data must then be aligned for distribution to the appropriate lanes.
- *Load Buffer*: receives data from the *LMU* prepares it for writing into the Vector Register File.

Store Path:

- *Store Buffer*: reads data from the Vector Register File (VRF) during a store operation and sends it to the*Store Management Unit (SMU)*.
- *Store Management Unit (SMU)*: collects information from the store buffers and transmits the data to the scalar core for memory write.

2.1.4 Vector Lanes

The lanes are the central components responsible for computation within the VPU. This architecture exploits parallelism by allowing each lane to work concurrently on different chunks of vector data. This version of the VPU features 16 lanes, ecah one contains several key submodules:

- **Vector Register File (VRF)**: the Vector Register File consists of 40 physical registers, each capable of storing up to 256 double-precision elements. These registers are organized into slices distributed across 16 lanes, with each lane storing a 5 KB slice. Each lane's slice of the VRF is implemented using five 1 kB 1-read/1-write SRAM banks. This design choice balances area, timing, and power efficiency while ensuring the VRF can sustain the required throughput. It supports interleaved access to maximize data throughput during vector loads, avoiding stalls in the pipeline.
- **Finite State Machine (FSM)**: each lane features a 5-state FSM (plus idle) that manages read/write operations to the VRF. The FSM coordinates access to the VRF banks and buffers data for the functional units, ensuring efficient pipeline utilization by issuing 64-bit results every cycle after startup.

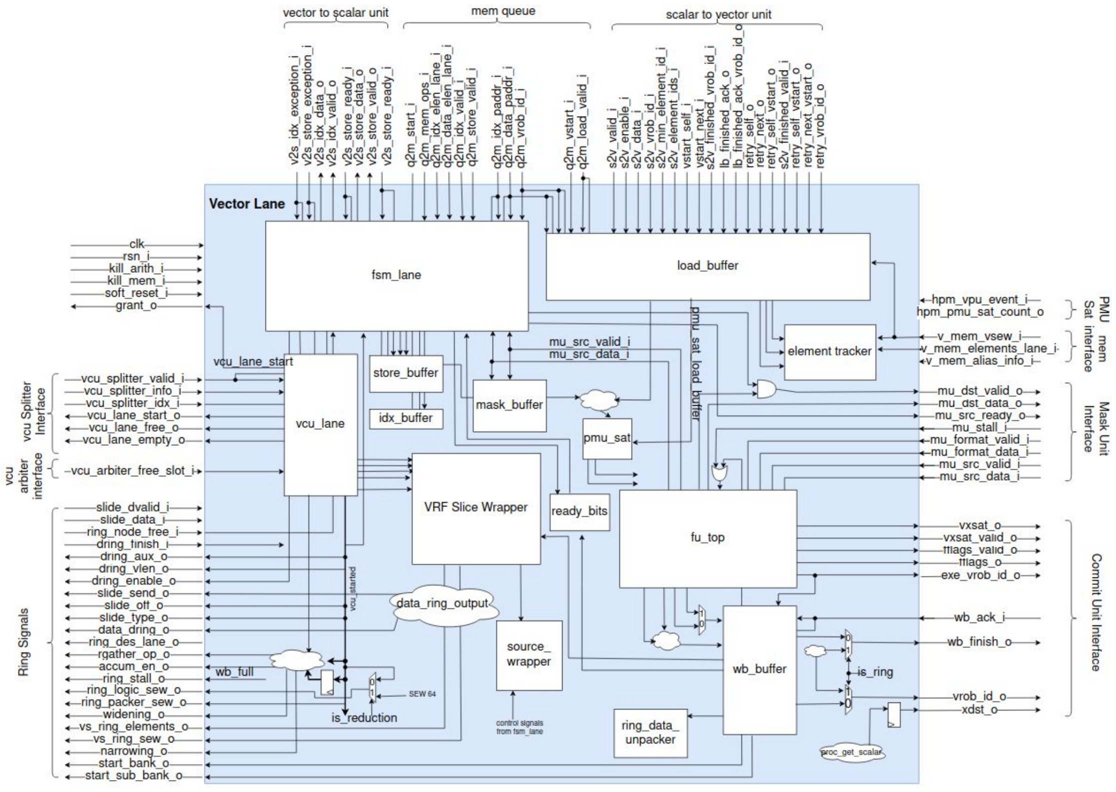

Figure 2.4: Vector Lane

- **Execution Unit Wrapper**: this includes the lane's Floating Point Unit (FPU) and Arithmetic Logic Unit (ALU). Both units are fully pipelined and support SIMD operations, handling floating-point and integer/fixed-point computations respectively. They support a wide range of operations including FMA, division, narrowing, widening, and reductions according to the RVV specification.

- **Write-Back Buffer (WB), Load Buffer (LB), and Store Buffer (SB)**: these buffers temporarily store data during write-back, load, and store operations, ensuring smooth data flow and avoiding pipeline bubbles.

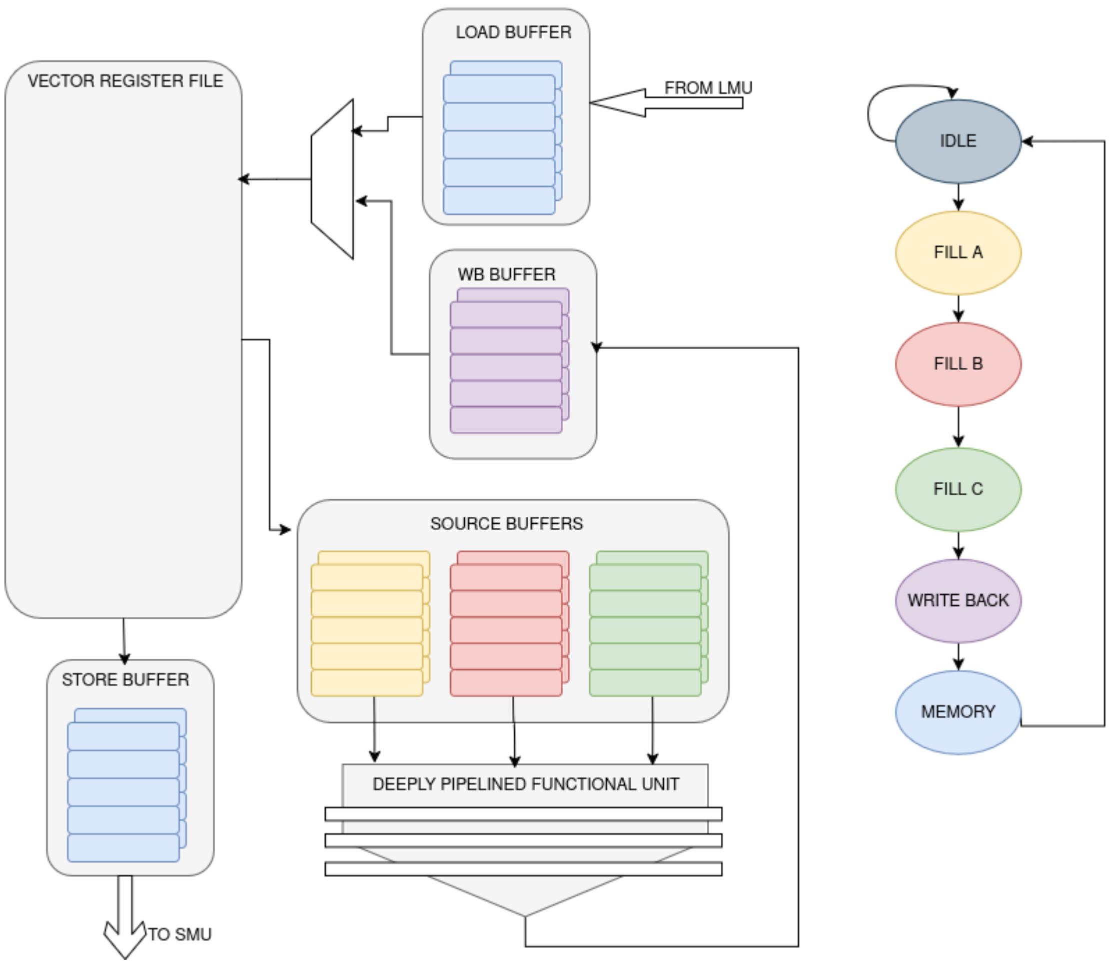

Figure 2.5: Finite state machine and lane buffers

2.1.5 Reorder Buffer

As the execution of arithmetic and memory instructions in this vector processing unit can complete in an out-of-order fashion, a mechanism is needed to ensure instructions still commit in the correct order. A Reorder Buffer (ROB) is used for this purpose, maintaining program order at the commit state, even though instructions may finish execution out of order. The ROB operates as a First-In, First-Out (FIFO) buffer, tracking the order of issued instructions and ensuring that only one instruction is committed per cycle.

The Control Unit (CU) signals the ROB whenever an instruction has finished executing. Once notified, the ROB checks if the associated scoreboard ID is ready to be committed back to the scalar core, and sets the appropriate interface signals to carry out the commit. When the scalar core issues a vector instruction, the

ROB logs the corresponding scoreboard ID for tracking.

If an exception occurs during execution, the ROB triggers an internal rollback process to restore the vector unit to a previously known good state. During this phase, the ROB blocks any new instructions from entering the front-end, ensuring the rollback completes safely and system consistency is maintained.

2.1.6 Interlane Crossbar

The interlane crossbar is used for permutations that require communication and data movement between vector elements, such as slides, gathers, reductions, narrowing, and widening operations. The current interlane interconnect employs a full crossbar network topology.

It takes two clock cycles to transmit an item, whether data or an index, from one lane to another. The datapath consists of a series of configurable switches that establish a connection between any source lane and any destination lane.

2.2 Methodology

In this section the experimental setup and the tools used are described

2.2.1 Design and Simulation

The Vector Processing Unit is fully designed at the Register Transfer Level (RTL) using SystemVerilog. SystemVerilog is a hardware description and verification language standardized by the IEEE. Originally developed as an extension of Verilog [9], it enhances the capabilities of the language for the design, simulation, and description of digital and mixed-signal systems. It introduces features such as parameterizable modules, interfaces, and advanced data types, which enable designers to create scalable, modular, and reusable hardware components. The design is simulated using QuestaSim, a commercial simulation and verification tool for digital designs developed by Mentor Graphics (now part of Siemens EDA). QuestaSim is widely used for hardware description languages such as SystemVerilog, Verilog, and VHDL, as well as for mixed-language designs.

2.2.2 Verification

The design is verified using an environment built in SystemVerilog. When used for verification, SystemVerilog follows object-oriented programming (OOP) paradigms, offering a wide range of classes and libraries that enable the development of reusable and powerful tests.

The verification structure follows the Universal Verification Methodology (UVM), a standardized framework that defines best practices for building reusable, extensible, and scalable testbenches. The key components of a UVM testbench are:

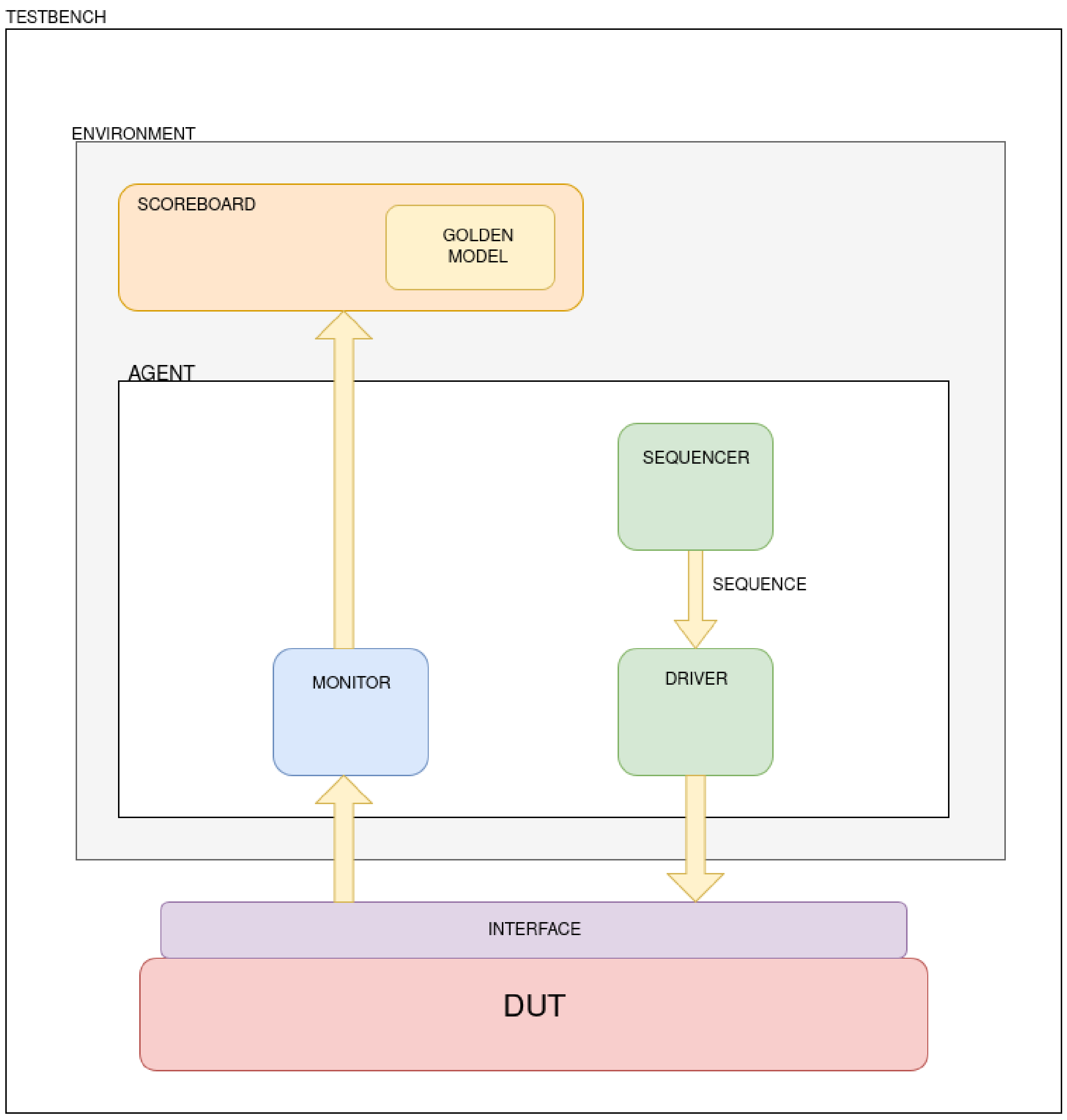

Figure 2.6: UVM environment architecture

- **Test:** The top-level UVM component that instantiates the environment and configures it by overriding parameters or objects as needed.

- **Environment:** A container for all the verification components targeting the DUT. It defines the reusable component topology of the UVM tests It contains the **Scoreboard:** the component that compares DUT outputs against a reference model to check for correctness.

- **Agent:** The core building block of the UVM testbench, containing the following sub-components:
  - **Sequencer:** Generates and sends transactions (sequence items) to the driver.
  - **Driver:** Receives transactions from the sequencer and drives them to the DUT through a virtual interface.
  - **Monitor:** Observes signals from the DUT via the virtual interface and forwards them to the scoreboard.
  - **Virtual Interface:** Facilitates communication between the testbench and the DUT, connecting all agents and enabling signal-level interactions.

2.2.3 Synthesis

In digital design, register-transfer level (RTL) descriptions are transformed into gate-level implementations through the synthesis process. This process explores different ways of implementing a logic function in order to optimize it according to specific design constraints.

Synthesis is typically divided into two main phases:

- **Logic Synthesis:** This phase focuses on optimizing the logic structure to improve efficiency and meet performance requirements. The goal is to generate a functional netlist that is independent of any specific technology.

- **Technology Mapping:** This critical step in technology-dependent optimization maps the optimized netlist onto available logic gates and cells from a specific standard cell library, composed of pre-designed and pre-characterized components. At this stage, initial estimates of power consumption and chip area are also obtained.

For this study, the TSMC 7 nm technology library was used, as it is the designated platform for the European projects in which the Barcelona Supercomputing Center is involved. The synthesis process was carried out using **Cadence Genus**, a widely adopted industry-standard tool.

2.2.4 Power analysis

For the power analysis, the *Joules RTL Power Solution* has been used. This is a commercial tool developed by Cadence, widely adopted in both industry and academia for performing power measurements at both the register-transfer-level and gate level.

Chapter 3

Design

After a brief introduction to the baseline architecture of the renaming unit and an explanation of the fast-move feature, this chapter explains how the optimization works and provides the microarchitectural details of the modules designed to support it.

3.1 Baseline renaming architecture

This section provides a detailed explanation of the data structures used by the renaming mechanism in the baseline architecture.

3.1.1 Register Alias Table

The Register Alias Table (RAT) is a structure that stores the mapping between logical and physical registers. In this architecture, the table contains 32 entries, one for each logical vector register, as specified by the RVV 1.0 ISA.

To support the rollback mechanism required for lightweight out-of-order execution, there are as many copies of the RAT as there are entries in the Reorder Buffer, 32 in this case. If an instruction needs to be killed, the correct state is restored by using the RAT associated with the $\mathrm{vrob}_{\mathrm{id}}$ of the last valid instruction.

When a new instruction is renamed (i.e., when a new vector or squeezed register is assigned as the physical destination), the RAT corresponding to the current $\mathrm{vrob}_{\mathrm{id}}$ inherits all the information from the RAT of the previous $\mathrm{vrob}_{\mathrm{id}}$, with the addition of the new mapping for the current logical destination register. At the same time, the RAT is accessed using the indices of the logical source registers to retrieve their corresponding physical registers This information is then packaged into a parameterized struct, which is passed to subsequent modules, such as the queue demultiplexer and the instruction queues. The same logic is used for the mask register alias table.

3.1.2 Free Register List

The Free Register List (FRL) is a data structure that tracks the availability of free physical registers. It is implemented as a FIFO queue. When an instruction is renamed, a new physical register is allocated by reading the queue entry pointed to by the `frl_read_ptr`. Logic is included to detect when the queue is empty, which triggers the `rename_stall` signal. This signal halts the issue of new instructions until a previous instruction commits and frees a physical register.

The commit of a register back to the FRL occurs when an instruction completes execution and is controlled by a signal from the Reorder Buffer. The register that is committed back is the old physical destination of the instruction that has just completed execution. Subsequent instructions will now require and access the new value stored in the newly assigned register. Since the old physical register is no longer needed, it can be safely returned to the Free Register List (FRL).

3.2 Fast Moves feature

The RISC-V-V 1.0 ISA includes the vector-vector move instruction, encoded as vmv.v.v vd, vs1, which copies the contents of the source register vs1 into the destination register vd, up to the current vector length vl. The baseline architecture implements an optimized version of these vector-to-vector moves, known as Fast Moves [10]. This optimization is discussed in this section because the logic and modules required are closely related to the designed optimization.

In particular, the *fast move* instruction is fully resolved during the renaming stage. To support this optimization, two additional structures are introduced:

- **Element Table**: For each logical register, it stores the number of vector elements assigned to the destination register.
- **Alias Counters**: A set of 40 counters, one for each physical register, tracks how many logical registers are currently mapped to the same physical register. The counter is incremented when a physical register is assigned to a logical one and decremented when that logical register is renamed again to a different one.

When a *fast move* is executed, no new physical register is allocated. Instead, the RAT is accessed using the index of the source register, and the physical register value retrieved is directly written into the RAT entry of the destination logical register. The instruction is completed by updating the Element Table with the current value of `vl`.

This approach allows a single physical register to be associated with multiple logical registers. The *alias counter* tracks how many logical registers are mapped to the same physical register, increasing with each *fast move*.

The alias counters are essential for the proper management of the Free Register List (FRL). As discussed in the previous section, a physical register can only be returned to the FRL when it is no longer needed, that is, when the value it holds is no longer required by any logical register. With *fast moves*, a physical register is not immediately freed when a logical register is renamed; instead, it is only returned to the FRL when the alias counter reaches zero, indicating that all logical registers previously mapped to that physical register have been renamed to new physical registers.

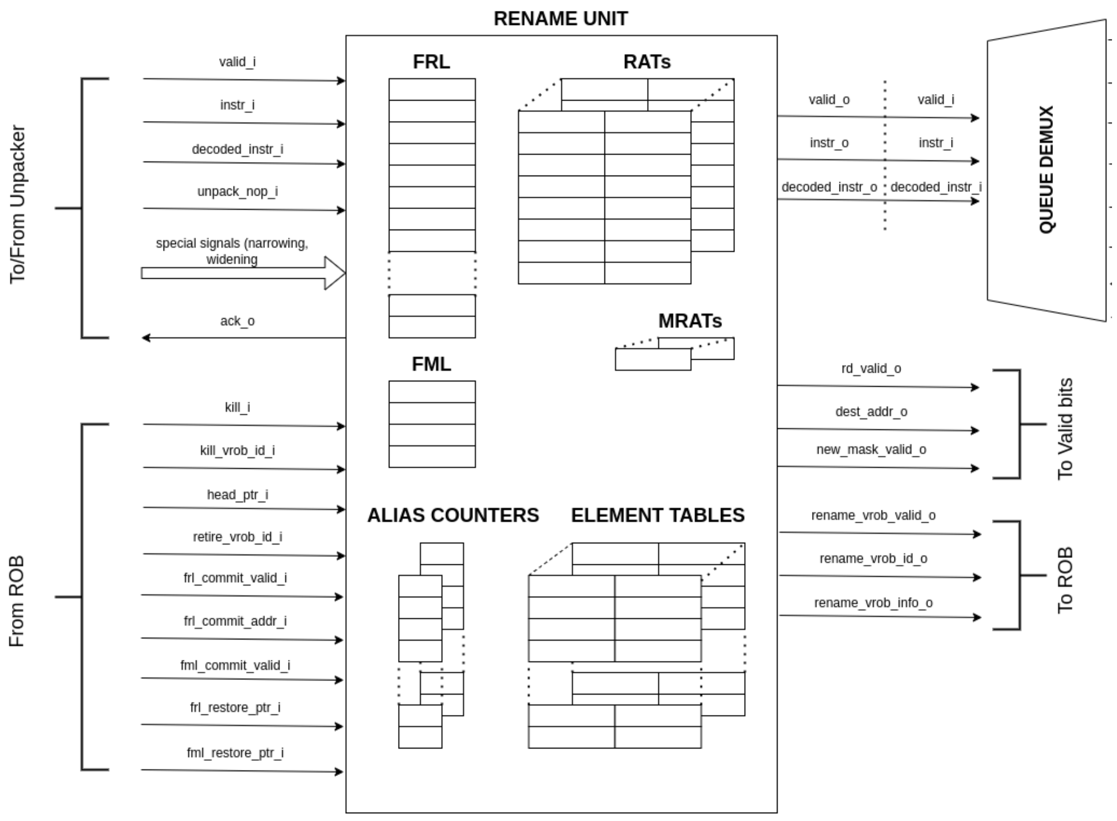

Figure 3.1: Baseline renaming unit architecture

3.3 Implementation

In this section the modules required to support the optimization are described

3.3.1 Register Renaming Module

Squeezed Free Register List The Free Squeezed Register List (FSRL) serves the same purpose as the standard FRL, but it tracks the availability of Squeezed registers instead. Its length matches the total number of entries in the SVRF (which is parameterized, in this thesis both the version whti 16 and 8 SVREGS have been studied). Initially, at the start of the program, all vector logical registers are mapped to physical vector registers, meaning all squeezed registers are free.

The FSRL receives requests for registers through signals from the unpacker. These signals are asserted (set to 1) when an instruction is marked for optimized execution.

There are situations where many vmv.v.x or vmv.v.i instructions arrive at the VPU simultaneously, typically during the initialization of counters (e.g., for accumulation registers). This can lead to a rapid consumption of squeezed registers. In these cases, if no squeezed register is available, the instruction is renamed to a standard vector register to avoid stalling the issue stage, allowing execution to proceed as normal.

When a instruction enters the renaming stage and the FSRL grants a new squeezed register, the squeezed register's ID is written into the corresponding RAT entry.

A commit to the FSRL occurs when a destination register, previously mapped to a squeezed register, is renamed and completes execution, meaning the old value is no longer needed.

Register Map Table The **register map table** can be considered an extension of the RAT. It has the same number of entries as the number of logical registers. For each logical register, it indicates whether the register is currently mapped to a squeezed or a standard vector register. This distinction is necessary because, during instruction issue, the information in the RAT alone is not sufficient; it is also essential to know whether the logical register is mapped to a squeezed or a vector register. A single bit per entry is enough to represent this distinction:

- The bit is set to 1 if the destination is renamed to a squeezed register, or if a fast move instruction (e.g., `vmv.v.v` with a squeezed register source is issued.
- The bit is set to 0 if the destination is mapped to a standard vector register, or if a fast move instruction with a vector register source is issued.

For rollback support, we need as many copies of the register map table as there are entries in the ROB (as explained in the previous chapter regarding the RAT).

Squeezed Alias Counters It is a table with as many entries as the number of registers in the SVRF. Each entry is a counter that tracks how many logical registers are currently mapped to the same (squeezed) physical register. Counts greater than zero are the result of fast moves. All counters are initialized to zero, since at the beginning, all logical registers are mapped to standard vector registers. The counter, for a squeezed register and for the current `vrobID`, is incremented when that squeezed register is piacked from the FSRL and assigned to a new destination. At the same time, the counter corresponding to the previous destination must be decremented:

- If the previous destination was a squeezed register, the corresponding Squeezed Alias Counter entry (indexed by `old_dest_address_d`) is decremented.
- Otherwise (if the previous destination was a standard vector register), the corresponding `alias_cnt` is decremented instead (see Section 3.2).

The check to determine whether a register was mapped to a squeezed or a standard vector register is done by accessing the Register Map Table using the logical register ID (`rat_waddr`), before the table is updated in the next clock cycle (via `vregs_map_q`).

3.3.2 Squeezed vector register file

The **Squeezed Vector Register File (SVRF)** is instantiated in the top module of the VPU (`vpu_core.sv`). It is implemented using the `multiport_regfile_ff` module, which is a parameterized register file based on flip-flops.

The SVRF has two write ports, necessary because the register file interacts with both the renaming unit and the lanes. Specifically, two types of instructions can write into a squeezed register:

- `vmv.v.x`, `vmv.v.i`, `vmv.s.x`, `vmv.s.f`, which are resolved during renaming.
- Reductions, , `vrgather.vx`, and `vrgather.vi`, which enter the pipeline and are executed in the lanes.

Each write port has its own enable signal, one coming from the renaming unit, and the other from the lanes.

Regarding the read ports:

- Three read ports for arithmetic instructions. This is because the RISC-V Vector ISA allows for vector instructions with up to three source operands, and each source may be mapped to a squeezed register.
- Two read ports for memory instructions, to support stores: one to read the base address and another to read the data to be stored in memory.

Each entry in the SVRF is 64 bits wide, which is necessary to store a full scalar value when the architectural parameters are such that `VSEW` is equal to `ELEN` $= 64$ (i.e., the maximum size of a vector element in the RISC-V Vector extension).

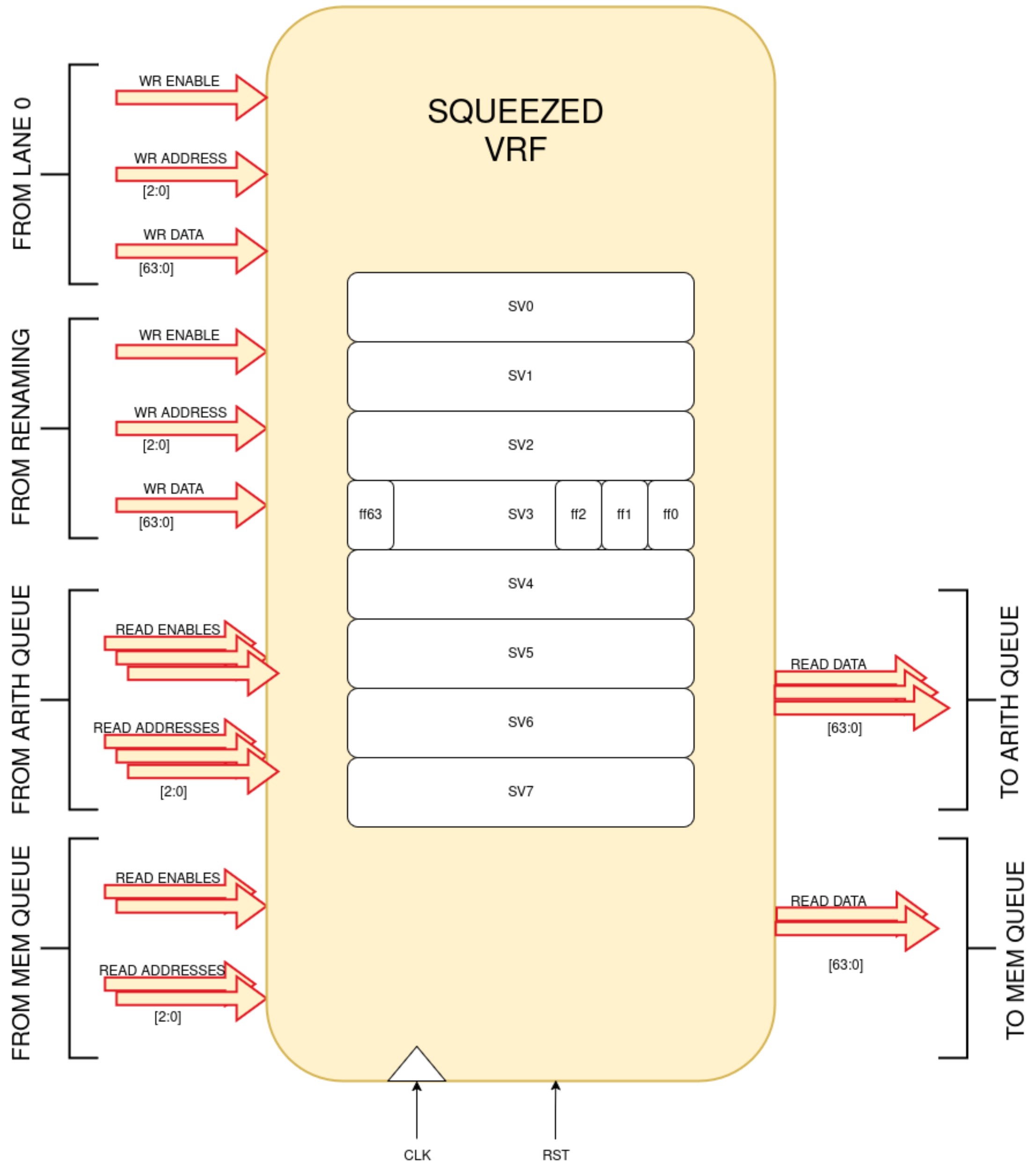

Figure 3.3: Squeezed Vector Register File

3.3.3 Impact on other architectural modules

This section describes the design changes made to various modules to support the optimization.

Unpacker The unpacker sets specific bits when an instruction produces scalar data, making it a candidate for optimized execution. In particular, two distinct cases are identified using separate flags (`squeezed_valid` and `squeezed_valid_vmv`): reductions and gathers executed in the lanes, and moves handled during renaming. These flags, generated through combinational logic from other flags obtained during the decoding, are passed to the renaming unit.

Instruction Queues An arithmetic instruction may use a source operand stored in a squeezed register. In such cases, indicated by specific bits set in the `decoded_instruction` signal from the renaming, the arithmetic queue issues a request to the `valid_bits` module to determine the availability of the operands. Once the data becomes available, the value is retrieved from the Squeezed Register File and encapsulated into a structure suitable for execution by the vector lanes.

For store instructions, the memory queue also interacts with the Squeezed Vector Register File (SVRF) to obtain the data that needs to be written to memory.

Valid Bits Another `valid_bits` module, `svregs_valid_bit_i`, is instantiated to track the availability of squeezed registers. It interacts with the renaming logic to set the corresponding bits once instructions such as `vmv.v.x`, `vmv.v.i` complete. It also communicates with vector lane 0, since its the lane responsible for performing the last step in reduction operations. The output of this module consists of valid bits, which inform the queues about the availability of operands.

Vector Lane The execution of instructions in the lanes is managed by a finite state machine, implemented in the `fsm_lane` module. The execution process is divided into several states. In the `FSM_FILL_A`, `FSM_FILL_B`, and `FSM_FILL_C` states, the `squeezed_operand` signal is asserted. This signal prevents reading from the vector register file when the operand is squeezed, and while it is active, the squeezed data is directly forwarded to the lanes. In the `WRITE_BACK`state, the write-enable signal for the squeezed vector register file (SVRF) is asserted if the destination register is squeezed, as determined by the FSM.

Reorder Buffer The Reorder Buffer (ROB) is the key module responsible for instruction commit. Instructions arriving from the renaming unit also carry information indicating whether the old destination register was a squeezed register.

In such cases, when the commit is performed, the old destination register is returned to the free squeezed register list instead of the standard free register list.

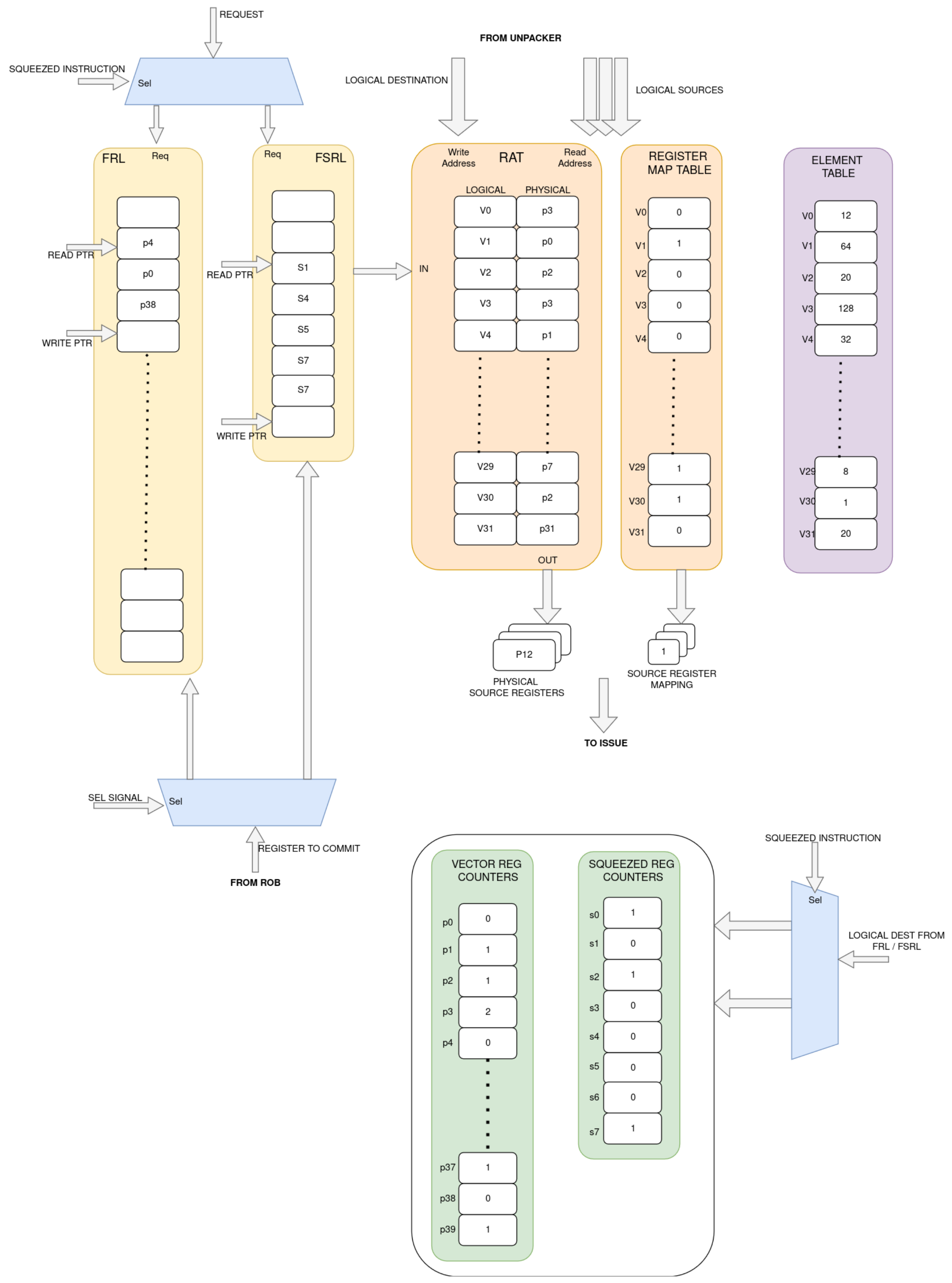

Figure 3.2: Optimized renaming architecture

Chapter 4

Code execution example

In this section, an example of code execution is provided, showing how various data structures are updated.

```
vadd.vv   v2, v1, v0    VL=35
vredsum.v v3, v2, v1    VL=30
vmv.v.x   v4, rs1       VL=25
vmv.v.v   v6, v4        VL=30
vadd.vv   v4, v1, v2    VL=30
```

4.1 Initial state of modules

Note: There are 32 copies of the following structures, one per ROB entry, to support the rollback mechanism in case an instruction is killed. For simplicity, only a single instance is shown and discussed here. The version discussed corresponds to a system with 16 physical squeezed registers.

- **Register Alias Table (RAT):** 32 entries, one per logical register. Each logical register is initially mapped to the physical register with the same index (i.e., logical register `Rn` → physical register `pn`).
- **Free Register List (FRL):** A first-in, first-out queue with a total capacity of 40 physical registers. Initially, 8 registers (physical registers 32 to 39) are free. The number of physical registers in the list can increase dynamically if fast move instructions are executed.
- **Free Squeezed Register List (FSRL):** First-In, First-Out queue with a total capacity of 16 entries, equal to the total number of squeezed vector registers (SVRF). Initially, all 16 registers are present in the list.

- **Element Table:** 32 entries, one for each logical register.
- **Register Mapping Table (RMT):** 32 entries, one per logical register. All entries are initially set to zero, indicating that logical registers are mapped to the default vector physical registers.
- **Vector Register Usage Counters (VREG Counters):** 40 entries, one per physical vector register. Entries 0–31 are initialized to 1 (each physical register is currently assigned to its corresponding logical register). Entries 32–39 are initialized to 0 (indicating those physical registers are free and listed in the FRL).
- **Squeezed Register Usage Counters (SVREG Counters):** 16 entries, all initialized to 0, meaning all squeezed physical registers are currently free (available in the FSRL).

4.2 Code execution

The execution of the assembly code is presented step by step, with a summary of the changes in the register renaming modules for the optimized version. Tables are included to provide a more schematic overview.

vadd.vv v2, v1, v0 VL=35

Standard vector add instruction. The destination register `v2` is renamed to a new physical register allocated from the Free Register List (e.g., register 32). The RAT entry for `v2` is updated, and the Element Table is written with the VL value (35). The alias counter for the previously assigned physical register (e.g., `vreg2`) is decremented to 0 and the counter for the new physical register (32) is set to 1. `vreg2` can be committed back to the FRL, at the tail of the queue, pointed by frl_commit_ptr.

FRL	**FSRL**

Rdest = FRL[frl_read_ptr] = 32
frl_read_ptr = frl_read_ptr + 1
p2 committed back
	/

RAT	**RMT**	**ELEM_TABLE**
RAT[2] ← 32 logical destination = 2	RMT[2] ← 0 vector register	element table[2] ← 35 logical destination = 2

V-REGS COUNTERS	S-REGS COUNTERS

old destination mapping = 0
new destination mapping = 32
cnt[0] = 0
cnt[32] = 1
	/

vredsum.v v3, v2, v1 VL=30

This is a reduction instruction, so the destination `v3` can be renamed to a squeezed register. The first available squeezed physical register is selected from the Free Squeezed Register List (e.g., register 0). The Register Alias Table, Register Map Table, and Element Counter at entry number 3 are updated accordingly: the RAT is updated with the new destination, the Element Table is set to 1 because it is a reduction, and the Register Map Table is set to 1 to indicate that the logical register is now mapped to a squeezed register. The Squeezed Alias Counter for the selected squeezed register is incremented, and the alias counter for the old destination vector register is decremented accordingly. That register, `p3`, can then be committed back into the FRL.

FRL	FSRL

p3 committed back	Rdest = FSRL[fsrl_read_ptr] = 0
	fsrl_read_ptr = fsrl_read_ptr + 1

RAT	RMT	ELEM_TABLE
RAT[3] ← 0 logical destination = 3	RMT[3] ← 1 squeezed register	element table[3] ← 1 reduction operation

V-REGS COUNTERS	S-REGS COUNTERS
old destination mapping = 3 vreg_cnt[3] = 0	new destination mapping = 0 svreg_cnt[0] = 1

vmv.v.x v4, rs1 VL=25

Scalar-to-vector move. The destination register `v4` is renamed to a new squeezed physical register taken from the Free Register List (e.g., register 1). The RAT and Element Table are updated with the new mapping and VL value (25). The alias counter of the previously assigned physical vector register is decremented, and the counter for the new squeezed physical register is set to 1. The old destination vector register can then be committed back to the FRL.

FRL	**FSRL**

p4 committed back	Rdest = FSRL[fsrl_read_ptr] = 1
	fsrl_read_ptr = fsrl_read_ptr + 1

RAT	**RMT**	**ELEM_TABLE**
RAT[4] ← 1 logical destination = 4	RMT[4] ← 1 squeezed register	element table[4] ← 25 VL elements

V-REGS COUNTERS	**S-REGS COUNTERS**
old destination mapping = 4 vreg_cnt[4] = 0	new destination mapping = 1 svreg_cnt[1] = 1

vmv.s.x v5, rs1 VL=25

Scalar-to-vector element move. It executes similarly to the previous instruction. A new squeezed register is taken from the FSRL, and the RAT and RMT are updated accordingly. The only difference is that, according to the ISA, this instruction writes only to element 0 of the vector. Therefore, the VL stored in the Element Table is set to 1. The counters are updated, and the vector register is committed back to the FRL.

FRL	**FSRL**

p5 committed back	Rdest = FSRL[fsrl_read_ptr] = 2
	fsrl_read_ptr = fsrl_read_ptr + 1

RAT	**RMT**	**ELEM_TABLE**
RAT[5] ← 2 logical destination = 2	RMT[5] ← 1 squeezed register	element table[3] ← 1 scalar to VREG[0]

V-REGS COUNTERS	**S-REGS COUNTERS**
old destination mapping = 5 vreg_cnt[5] = 0	new destination mapping = 2 svreg_cnt[2] = 1

vmv.v.v v6, v4 VL=30

This is a vector move that executes as a fast move; no new physical register is allocated. The RAT entry for `v6` is updated to match that of `v4`. Same for Register Map Table, set to 1 beacuse the source is mapped to a squeezed register. The Element Table is updated with the minimum between the current VL (30) and the VL previously associated with the source register (`v4`, which is 25). Since both `v4` and `v6` now map to the same physical register, its Squeezed Alias Counter is incremented to reflect the additional logical alias.

FRL	**FSRL**
p6 committed back	/

RAT	**RMT**	**ELEM_TABLE**
RAT[6] $\leftarrow$ 0 logical destination = 6	RMT[6] $\leftarrow$ 1 squeezed register	element table[6] $\leftarrow$ 25 $VL_{source} < VL_{instruction}$

V-REGS COUNTERS	**S-REGS COUNTERS**
old destination mapping = 6 vreg_cnt[6] = 0	new destination mapping = 1 svreg_cnt[1] = 2

vadd.vv v4, v1, v2 VL=30

This instruction overwrites `v4` again. A new physical vector register (e.g., 34) is allocated from the Free Register List. The RAT entry for `v4` is updated, and the Element Table is set with VL = 30. The alias counter for the old physical squeezed register is decremented, while the counter for the new physical vector register is set to 1. No commit to the FSRL occurs because, due to a previous fast move, the alias counter for that register is still 1 (i.e., greater than 0).

FRL	**FSRL**

Rdest = FRL[frl_read_ptr] = 33
frl_read_ptr = frl_read_ptr + 1
	/

RAT	**RMT**	**ELEM_TABLE**
RAT[4] $\leftarrow$ 33 logical destination = 4	RMT[4] $\leftarrow$ 0 vector register	element table[4] $\leftarrow$ 30 current VL

V-REGS COUNTERS	**S-REGS COUNTERS**
new destination mapping = 33 vreg_cnt[33] = 1	old destination mapping = 1 svreg_cnt[1] = 1

Chapter 5

Verification

Verification is a critical step to ensure that the Vector Processing Unit still behaves correctly after the microarchitectural optimization and complies with the ISA specifications. To achieve this, a UVM (Universal Verification Methodology) environment has been developed. UVM is a standardized, reusable verification methodology based on SystemVerilog, widely adopted in the industry for verifying complex digital designs. It provides a structured framework for building testbenches, generating stimuli, and checking functional correctness (see Chapter 2.2.2). In this setup, the UVM environment takes as input the RTL-level design,a RISC-V-V golden reference model, and the binary of a test to perform

5.1 VPU simulation wrapper

In the UVM environment, the scoreboard is the component responsible for verifying the functionality of the design. It does this by comparing the output from the Design Under Test (DUT) with that from a golden reference model.

In this implementation, the VPU is encapsulated within the `vpu_sim_wrapper`, an interface module that provides access to internal signals for verification purposes.

With the introduction of two separate register files in the VPU, the Squeezed Vector Register File (SVRF) and the standard Vector Register File (VRF), the `vpu_sim_wrapper` requires several modifications.

In the standard setup, data is retrieved from a mirrored data structure that replicates the contents of the VRF. This mirror is accessed using information provided by the Reorder Buffer (ROB) and the renaming logic. Specifically, the scoreboard uses the logical destination of the instruction (obtained from the ROB) to query the Register Alias Table (RAT) in the renaming unit. The RAT returns the corresponding physical register index, which is then used to access the mirrored VRF.

IN the new version, each time an instruction is renamed, the register mapping table is queried to indicate whether the register has been mapped to a squeezed or a standard vector register. This mapping information is stored in an internal table indexed by the ROB ID, allowing it to be retrieved later when the instruction completes.

When an instruction completes, the Destination Mapping Table (data structure inside the VPU simulation wrapper) is accessed using the associated ROB ID. If the result was mapped to a squeezed register, data is retrieved from the SVRF and forwarded to the scoreboard. Otherwise, the data is taken from the VRF mirror.

5.2 ISA Tests

The ISA tests consist of a variety of tests categorized to cover different types of instructions in the ISA. These tests, written in RVV 1.0 assembly and compiled using the GNU toolchain for RISC-V [11], are used by the sequencer to apply stimuli to the Device Under Test. The main categories include:

- Vector loads and stores
- Integer arithmetic instructions
- Fixed-point arithmetic instructions
- Floating-point instructions
- Reductions
- Mask instructions
- Permutation instructions

Each of these instruction categories includes numerous tests to ensure comprehensive coverage of all instructions. The tests also vary key parameters such as `VSEW` (Vector Standard Element Width) and `VL` (Vector Length) defined in the RISC-V Vector Extension (RISC-V-V).

5.3 Spike

Spike is a RISC-V ISA simulator maintained and updated by the RISC-V Foundation [12]. It includes support for many extensions (V included) and serves as a reference implementation of the RISC-V architecture. In the verification environment, Spike is used as the *golden model* to compare against the results produced by the Vector Processing Unit.

5.4 Simulation workflow and verification

The simulation, along with the test environment and test cases described, is carried out using Questasim. The output is a simulation transcript file that contains detailed information about all executed instructions and the internal state of the scoreboard. If an error occurs, the simulation halts, and the debugging phase is conducted by analyzing the waveforms within Questasim.

Once all ISA tests pass, a first stage of architectural verification is considered complete. At this point, performance evaluation can proceed using more complex and comprehensive kernels.

Chapter 6

Performance evalutaion

Benchmarking a processor involves evaluating its performance across a variety of scenarios using code from different application domains. In this study, the developed Vector Processing Unit (VPU) targets domains such as High Performance Computing (HPC) and Machine Learning (ML), and benchmarks have been selected accordingly.

In the context of this study, the performance evaluation aims to compare the simulation results against the baseline architecture, while varying the number of elements in the Squeezed Vector Register File. The optimized VPU exhibits behavior identical to the baseline VPU when executing instructions not impacted by the optimization. Performance improvements become evident as the number of optimized instructions increases.

6.1 Code vectorization and benchmark suites

A key advantage is that the RISC-V Vector Extension (RVV) version 1.0 is fully supported by modern compilers. However, the compiler's auto-vectorization capabilities are still limited and often unable to fully exploit the vectorization potential of the hardware [13]. To achieve optimal performance, code must be explicitly vectorized by developers. This is typically done using RISC-V intrinsics [14], built-in compiler functions that map directly (one-to-one or many-to-one) to specific assembly instructions, enabling fine-grained control over vector execution. Existing vectorized benchmarks from academic literature were used as an initial reference for this work:

- **RiVEC Benchmark Suite** [15]: The RiVEC suite is a collection of data-parallel applications from diverse domains, developed specifically to address the lack of comprehensive vectorized benchmark suites. It focuses on evaluating vector microarchitectures and is particularly well-suited for this study, as it targets the RISC-V architecture. RiVEC supports the latest RVV 1.0 intrinsics and can be compiled using the up-to-date RISC-V GNU toolchain.

- **Ara2**: Ara is a vector processing unit designed to operate as a coprocessor alongside the CVA6 core, developed as part of the PULP platform at ETH Zurich [16]. In line with the open-source hardware philosophy, all benchmarks and programs used to evaluate the VPU are publicly available. Since Ara supports the RISC-V Vector Extension version 1.0, the benchmarks are written using RVV intrinsics, making it a suitable reference for this project.

6.2 Other challenges

The performance of the VPU is evaluated through RTL simulation using Questasim. While RTL simulation offers cycle-accurate results, it becomes increasingly inefficient for large and complex designs like the VPU. As an event-driven process, the simulation time and computational load grow significantly due to the large number of events the simulator has to manage. This problem is especially pronounced when running long benchmarks, such as those described earlier.

In contrast, FPGA emulation executes the design on actual hardware, providing much faster performance. It is typically preferred when a large number of test cycles are needed. The speed and scalability of hardware emulation make it ideal for evaluating extensive workloads. However, at the time of this work, the FPGA emulation environment was not yet ready. Therefore, the performance evaluation in this project is conducted exclusively through RTL simulation.

Evaluating performance at the RTL level introduces another important limitation: all tests and benchmarks must be compiled for a pure bare-metal environment. This requirement significantly constrains the range of usable benchmarks, as many rely on operating system services or runtime libraries. As a result, the benchmark suites described previously cannot be used directly without adaptation and reworking to remove dependencies and ensure compatibility with a bare-metal setup.

6.3 Microbenchmarks

To address both the long simulation times and the limitations imposed by the bare-metal environment, a set of targeted microbenchmarks has been used for the initial performance evaluation.

Microbenchmarks are small, focused programs designed to test specific functions or segments of code. In this work, different microbenchmarks have been developed starting from the existing benchmark suites mentioned before, compiled using the

GNU toolchain, and refined through manual inspection of the disassembled output to conform with the constraints of the simulation environment.

The selected microbenchmarks include:

- **Matrix Multiplication (matmul)**: based on the matrix multiplication kernel from the Ara2 benchmark suite, this benchmark was tested with various matrix dimensions. Matrix multiplication is a fundamental operation in linear algebra, signal processing, and machine learning. The benchmark features key vector instructions such as `vmv.v.x` and `vmv.v.i`.

- **Dot Product**: also sourced from the Ara2 suite, this benchmark implements a vectorized dot product kernel. Widely used in linear algebra and scientific computing, the kernel highlights the use of vector reductions.

- **Particle Filter**: taken from the Rivec benchmark suite, this kernel implements a powerful methodology for sequential signal processing with a wide range of applications in science and engineering [17]. It is characterized by a high number of scalar move instructions, offering insight into mixed scalar-vector workloads.

- **Layer Normalization (Layer Norm)**: a custom microbenchmark developed specifically for this evaluation. It directly implements the Layer Normalization function using RVV 1.0 assembly. The kernel includes both vector reductions and scalar moves. Layer Normalization is a common operation in deep learning models [18], used to accelerates training by normalizing neuron activations.

6.3.1 Results

Here we present the performance results in simulation, comparing the optimized architecture to the standard one. The speedup is expressed as a percentage and is computed using the following formula:

$$\text{Speedup (\%)} = \left(\frac{C_{\text{baseline}} - C_{\text{optimized}}}{C_{\text{baseline}}} \right) \times 100 \tag{6.1}$$

where C_{baseline} and $C_{\text{optimized}}$ represent the number of clock cycles required to complete the benchmark on the standard and optimized architectures, respectively.

The number of cycles for each configuration was determined by analyzing the Questasim simulation waveforms at the end of each test.

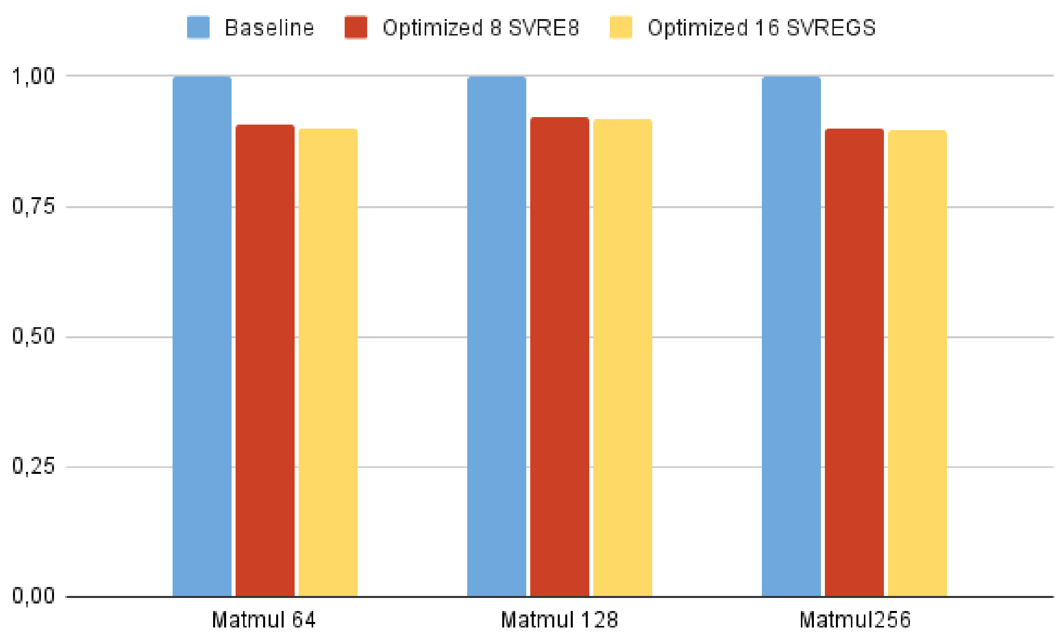

Figure 6.1: Comparison of the execution time of matmul for the Optimized and Baseline VPUs

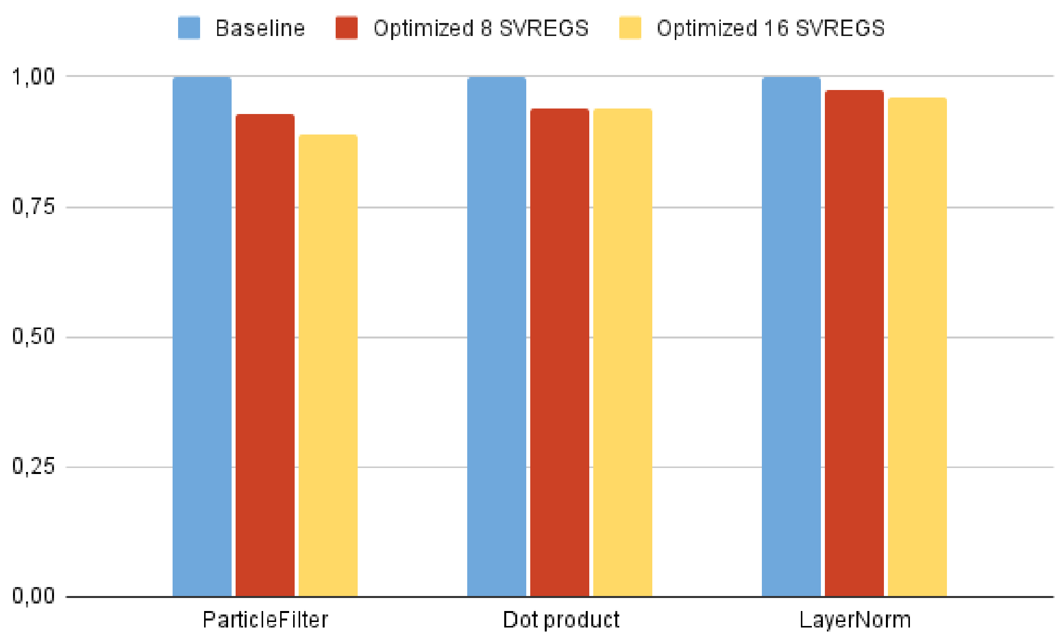

Figure 6.2: Comparison of the execution time of microbenchmakrs for the Optimized and Baseline VPUs

Microbenchmark	Architecture	SVRF size	Result [ns]	Speedup [%]
Matmul 64	Baseline	–	401625	–
	Optimized	8	364525	10.18
	Optimized	16	360985	11.26
Matmul 128	Baseline	–	1040325	–
	Optimized	8	959945	8.37
	Optimized	16	953345	9.12
Matmul 256	Baseline	–	3130025	–
	Optimized	8	2812745	11.28
	Optimized	16	2806785	11.52
Dot Product	Baseline	–	21225	–
	Optimized	8	19935	6.47
	Optimized	16	19935	6.47
Particle Filter	Baseline	–	35755	–
	Optimized	8	33155	7.84
	Optimized	16	31755	12.60
LayerNorm	Baseline	–	3709845	–
	Optimized	8	3620915	2.46
	Optimized	16	3569235	3.94

Table 6.1: Performance results and speedups across Baseline and Optimized architectures.

Comment

As expected, the microbenchmarks that show the most significant performance improvements are those with a higher number of scalar move instructions (`vmv.v.i`, `vmv.v.x`. These instructions benefit from the optimized architecture because the scalar value from the core is directly written into the squeezed register file at the renaming stage. As a result, they bypass the issue stage and are never dispatched to the vector lanes, saving a considerable number of clock cycles.

On the other hand, register reductions operations, do not exhibit a visible speed-up. This is because they still involve instruction dispatch to the vector lanes, and only the destination register is altered.

6.4 Low-Area VPU design exploration

The performance improvements observed in the optimized VPU motivate the design exploration of a low-area version of the unit. A major contributor to the overall VPU area is the vector register file. By leveraging the availability of squeezed registers, the length of the vector register file can be reduced, thereby decreasing the total area. After

6.4.1 Results

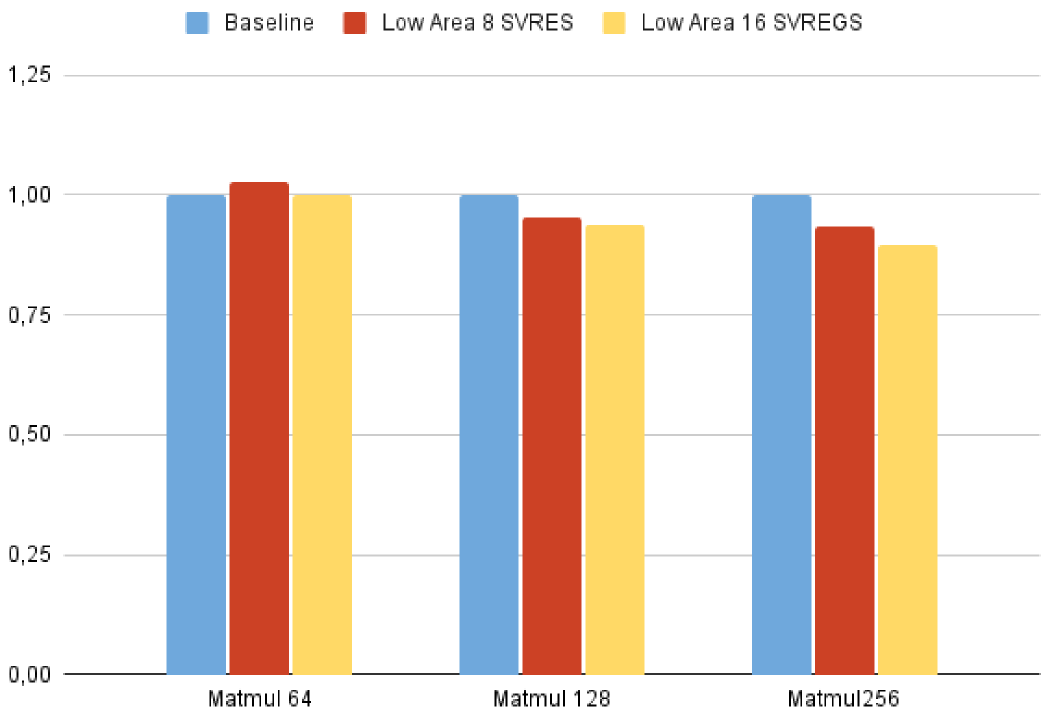

Figure 6.3: Comparison of the execution time of matmul for the Low-Area and Baseline VPUs

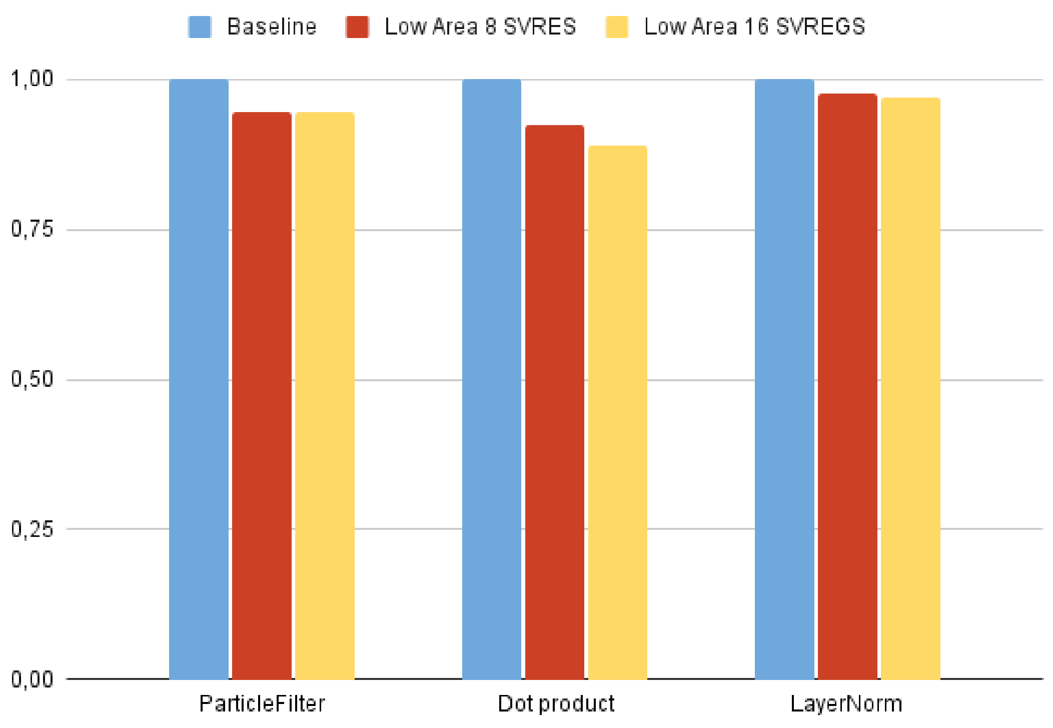

Figure 6.4: Comparison of the execution time of microbenchmakrs for the Low-Area and Baseline VPUs

Microbenchmark	**Architecture**	**SVRF size**	**Result [ns]**	**Speedup [%]**
Matmul 64	Baseline	–	401625	–
	Low-Area	8	411885	-2.49
	Low-Area	16	402005	-0.09
Matmul 128	Baseline	–	1040325	–
	Low-Area	8	990365	5.04
	Low-Area	16	974185	6.79
Matmul 256	Baseline	–	3130025	–
	Low-Area	8	2924985	7.01
	Low-Area	16	2806785	11.52
Dot Product	Baseline	–	21225	–
	Low-Area	8	20045	5.89
	Low-Area	16	20045	5.89
Particle Filter	Baseline	–	35755	–
	Low-Area	8	33155	7.84
	Low-Area	16	31755	12.60
LayerNorm	Baseline	–	3709845	–
	Low-Area	8	3615575	2.61
	Low-Area	16	3594665	3.20

Table 6.2: Performance results and speedups comparing Baseline and Low-Area Optimized architectures.

6.4.2 Comment

Given the promising performance results shown in the table, where speedup is observed across most benchmarks despite a reduced number of vector registers, it is reasonable to proceed with synthesis in order to assess whether the area reduction is significant. All synthesis results are presented in the following chapter.

Chapter 7

Synthesis

7.1 Area

The area results obtained from the synthesis process are presented in this section. Both the **Optimized VPU** and the **Low-Area Optimized VPU** have been synthesized. The first one refers to the optimized VPU configuration featuring 40 physical vector registers, while the area-optimized version reduces this number to 36 in an effort to lower the overall area footprint. Each version has been evaluated with both 16 and 8 squeezed registers. The results are compared against the baseline standard architecture.

Note that there is a significant difference in the area result values between the *optimized version* and the *low-area optimized version* of the VPU. This disparity stems primarily from how the vector register file is implemented in each case.

In the standard optimized version, the vector register file is synthesized using memory instances available in the technology portfolio. These are compiled using foundry-compatible memory compilers. Specifically, the total size of the register file is calculated as:

$$\text{Tot\_Bytes\_VRF} = \text{Num\_VRegs} \times \text{VReg\_Max\_Elements} \times \text{Element\_Bytes}$$

which results in a total of 80 kB. Given that the architecture includes 16 lanes, each lane requires 5 kB of storage. To implement this, the design instantiates five 1 kB 1RW SRAM banks per lane. This approach ensures efficient use of memory, avoiding unused memory space and minimizing area overhead.

However, in the area-optimized version, the design features only 36 physical vector registers. The issue is that the available SRAM memory instances in the technology library are precompiled to support a 40-register configuration. Using these memory blocks directly would result in underutilized memory, leading to the same synthesis results in terms of area.

To enable a meaningful comparison of area efficiency when reducing the number of registers, the vector register file in the low-area optimized version was synthesized entirely using flip-flops. While flip-flops are not a practical choice for large register files due to their significantly higher area and power consumption, they were used here to estimate the potential area savings from register count reduction. The standard baseline VPU was also synthesized using flip-flops to provide a consistent basis for comparison.

It is important to note that this is a preliminary analysis aimed at exploring area reduction strategies. In future implementations, memory instances more closely aligned with the reduced register count can be compiled and used to achieve better area efficiency. The key result is the variation in total area compared to the standard baseline VPU, regardless of the specific technology used for the register file implementation.

7.1.1 Standard optimized VPU

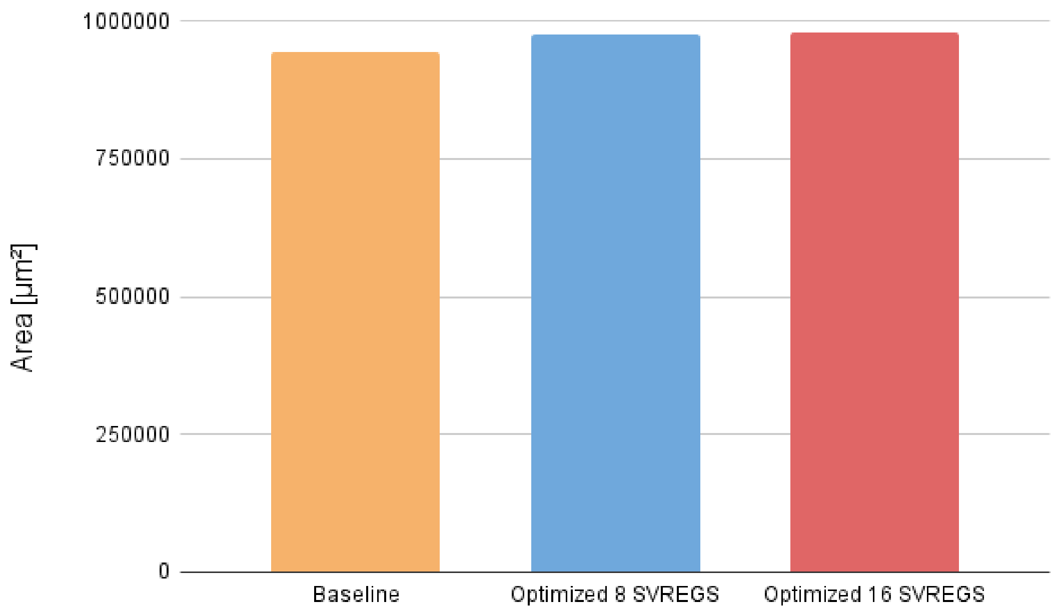

Figure 7.1: Area comparison with optimized VPU

Table 7.1: Area comparison: Standard Optimized VPU

Architecture	**Area [µm^2]**	**Difference (%) with baseline**
Baseline	944079.493	-
Optimized 8 SVREGS	975042.117	+3.28%
Optimized 16 SVREGS	977448.511	+3.57%

7.1.2 Low-Area optimized VPU

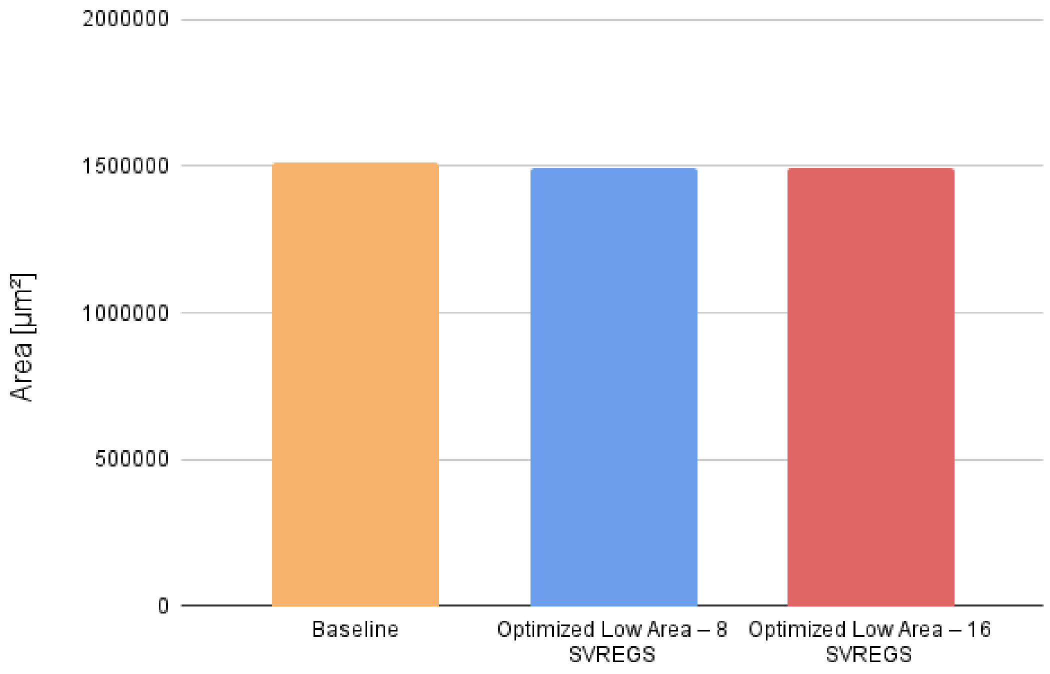

Figure 7.2: Area Comparison with Low-Area Optimized VPU

Table 7.2: Area comparison: Low-Area Optimized VPU

Architecture	**Area [µm^2]**	**Difference (%) with baseline**
Baseline	1513877.345	-
Low-Area 8 SVREGS	1494336.927	-1.29%
Low-Area 16 SVREGS	1495049.635	-1.24%

The observed area reduction is up to 1.29% in the low-area variant of the VPU. Given that this optimization maintains comparable performance levels, it makes the design well-suited for low-area implementations of the VPU without significant performance trade-offs

7.1.3 Detailed area breakdown

As shown in the previous section, the area of the Optimized VPU increases by 3.57% in the version with 16 squeezed registers, and by 3.28% in the version with 8 squeezed registers.

The table below highlights the modules that exhibit the most significant area increases. The version analyzed is the one featuring 8 Squeezed Vector Registers.

Table 7.3: Modules area comparison for the 8 SVRF VPU

Module	**Area [μm^2]**		**Area Increase (%)**
	Baseline	**Optimized**	
Lane	48781.331	49772.382	2.03
Renaming	49466.433	60992.866	23.30
Arithmetic Queue	4698.112	4939.077	5.13
Memory Queue	801.392	803.711	0.29
Reorder Buffer	4966.448	5185.550	4.41
VCU Splitter	2911.715	3254.632	11.78

From the Area breakdown shown in Figure 7.3, it's clear that the major contributions to the area increase come fron the Renaming Unit and the Vector Lanes.

Renaming Unit The Renaming Unit exhibit a 23.30% area growth, which is expected due to the additional data structures required to support the optimization. Specifically, this includes the *Squeezed Alias Counter* and the *Register Map Table*, with 32 copies instantiated to support the rollback mechanism.

Vector Lane The overall area of the lanes increased by 2.03%. More detailed graphs below illustrate the area increase for the different sub-modules of the Vector Lane.

This increase is primarily due to the additional hardware required to execute an instruction that has one or more squeezed registers as sources. In such cases, data is fetched from the *Squeezed Register File* to the *Instruction Queues*, routed through the *VCU Splitter*, and temporarily stored in a buffer inside the *Vector Control Unit.* Once the instruction begins execution, the data is forwarded to the *Finite State Machines*, which distribute it to the internal buffers according to the instruction's execution state.

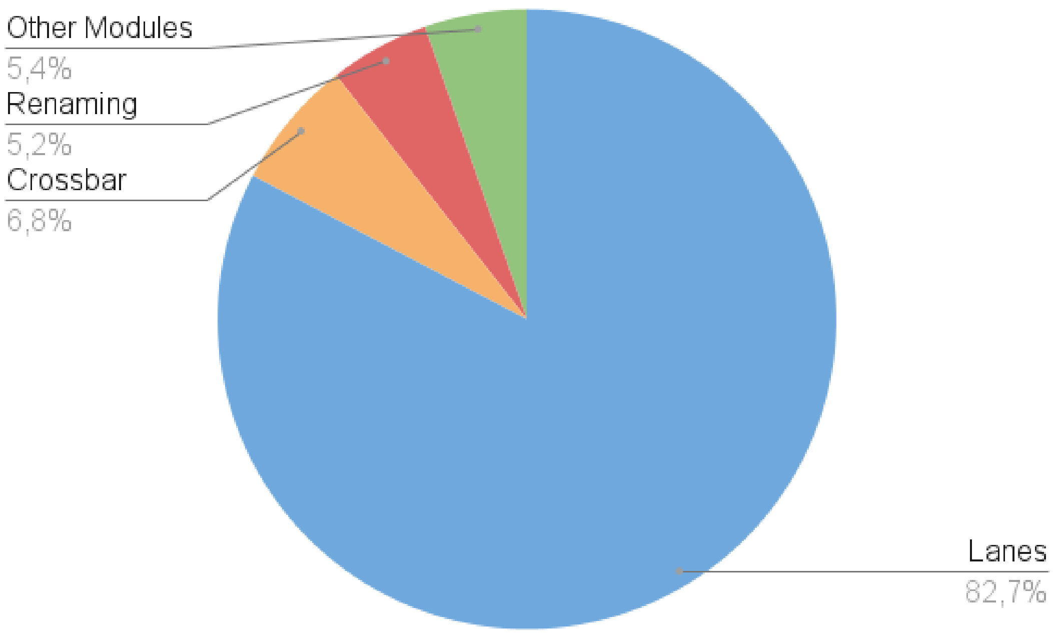

Figure 7.3: VPU area breakdown

Table 7.4: Lanes modules area comparison

Module	**Area** [μm^2]		**Area Increase (%)**
	Baseline	**Optimized**	
VRF Slice	7403.391	7600.776	2.67
FSM	275.553	470.797	70.86
VCU	1163.874	1596.206	37.15

This entire process requires both sequential and combinational logic to be instantiated, resulting in the observed area increase within the Vector Lanes. As shown in the Table 7.4, the modules `vcu_lane` and `fsm_lane` exhibit the largest area increases, with smaller increases also observed in the `vrf_slice_wrapper` and the source wrapper.

In the area breakdown of the Vector Lane (Figure 7.4), the major contribution to the Lane area increase clearly comes from the VCU and the VRF slice wrapper.

Additional Modules The additional modules instantiated in the VPU core: the Squeezed Vector Register Files and the SVREG valid bit, account for only 0.09% and 0.02% of the total area of the optimized VPU, respectively.

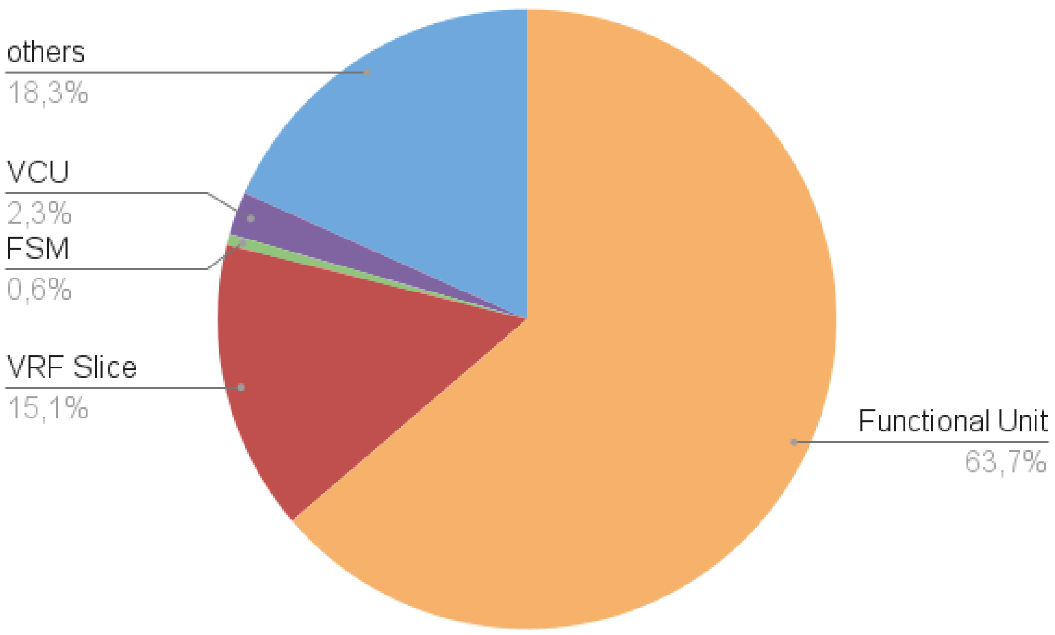

Figure 7.4: Lane area breakdown

7.2 Frequency

The maximum operating frequency of a digital design is limited by the critical path, defined as the longest combinational path in the circuit in terms of signal delay. Additional factors influencing the timing include clock uncertainty, a synthesis-defined parameter that accounts for variations in clock arrival times caused by jitter and skew, and the flip-flop setup time, which represents the minimum interval before the clock edge during which the input signal must remain stable to be correctly sampled.

$$f_{\text{max}} = \frac{1}{t_{\text{critical\_path}} + t_{\text{setup}} + t_{\text{clk\_uncertainty}}}$$

As shown in the table below, the optimization does not affect the existing critical path. Synthesis timing reports indicate that the critical path lies in the routing path of the interlane crossbar. This structure is unrelated to the additional hardware introduced by the optimization, and therefore the critical path remains unchanged. As a result, there is no impact on the maximum operating frequency.

Table 7.5: Timing results and maximum operating frequency

Metric	**Perte SRAM**	**Optimized 40-16**
Setup [ps]	7	7
Uncertainty [ps]	68	68
Data Path [ps]	625	627
f_{max} [GHz]	1.429	1.425

Chapter 8

Power Analysis

8.1 Power Contributions in CMOS

This section provides a brief introduction to the main sources of power consumption in CMOS technology.

- **Static power**: the power dissipated regardless of circuit activity, primarily due to leakage currents.
- **Dynamic power**: the power consumed during the switching of logic gates, resulting from the charging and discharging of load capacitances.

8.1.1 Static power

For technologies below 100 nm [19], transistors exhibit a phenomenon known as subthreshold conduction, in which a leakage current flows even when the transistor is neither active nor switching. This effect results in a static power consumption component that depends on both the circuit configuration and the specific technological parameters of the standard cell.

8.1.2 Dynamic power

Switching power Switching power refers to the power required to charge and discharge the internal nodes of the system. In CMOS logic gates, the load can be modeled as a capacitance C_L, which is driven by a pull-up network of pMOS transistors, responsible for charging the capacitance and pulling the output to a logic high, and a pull-down network of nMOS transistors, which discharges the capacitance and pulls the output to ground. Consider for simplicity an inverter, the simplest CMOS logic gate:

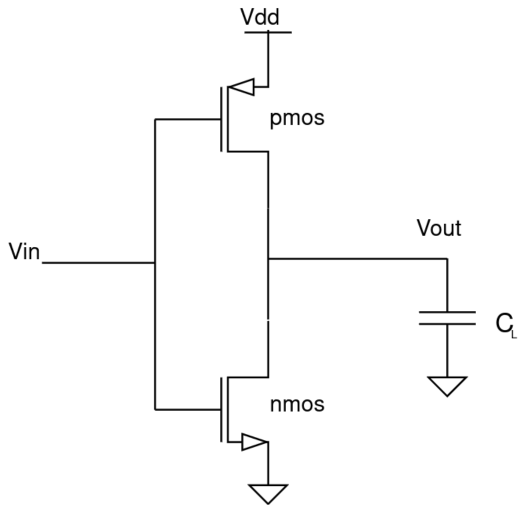

Figure 8.1: CMOS inverter

$$E = \int_0^\infty P(t)\,dt \tag{8.1}$$

Energy for a commutation, during a period of time T:

$$E = \int_0^T I(t) \cdot V(t)\,dt \tag{8.2}$$

Since the energy is provided by the constant source V_{dd} and the current charges a capacitance:

$$E = \int_0^T C \cdot \frac{dV(t)}{dt} \cdot V_{dd}\,dt \tag{8.3}$$

$$E = C \cdot V_{dd} \cdot \int_0^{V_{dd}} dV \tag{8.4}$$

Total energy during the commutation:

$$E_{\text{switch}} = C \cdot V_{dd}^2 \tag{8.5}$$

During the charging phase, when the output goes to a high logical state and the capacitance is charged, half of the energy is dissipated across the on-resistance of the pMOS transistor and half is stored in the capacitance. In the opposite transition, when the output goes to ground, the load capacitance discharges through the nMOS transistor, dissipating the stored energy as heat.

$$E_{\text{Ron}} = E_c = \frac{1}{2} C_L \cdot V_{dd}^2 \tag{8.6}$$

Equation (8.4) represents the energy consumed per transition (i.e., per 0-to-1 or 1-to-0 logic change) in a CMOS gate with load capacitance C_L, assuming ideal charging and discharging through a full voltage swing.

The average dynamic power over time, for a periodic switching signal, is given by:

$$P = \frac{E}{T} = E \cdot f \tag{8.7}$$

However, not all gates switch at every clock cycle. The actual switching activity is modeled by a factor α, representing the probability that a node toggles during a clock period. Incorporating this into the average switching power gives:

$$P = \alpha \cdot C_L \cdot V_{dd}^2 \cdot f \tag{8.8}$$

Short circuit power Another contribution to dynamic power arises from the fact that input signals are not ideal step functions, but instead exhibit finite rise and fall times, and transistors do not behave as perfect switches. During a logic transition in a CMOS gate, there exists a short period in which both the pull-up (pMOS) and pull-down (nMOS) networks are conducting simultaneously. In this interval there is direct current path from V_{dd} to ground. This results in a short-circuit current, denoted as I_{sc}, which flows only during the switching event.

Since this power dissipation occurs only during signal transitions and is proportional to the switching frequency, it is modeled similarly to the switching power with an equivalent capacitance C_{sc}. This capacitance depends on technological parameters as well as the rise and fall times of the input signal. Thies gives the equation for the sshort circuit power:

$$P_{SC} = \alpha \cdot C_{sc} \cdot f \cdot V_{dd}^2 \tag{8.9}$$

The total power consumption of a system is the sum of the contributions from all instantiated standard cells within the design. At the architectural level, various strategies can be employed to reduce the switching activity of specific submodules, thereby decreasing overall dynamic power consumption. This is the goal of the proposed optimization.

8.2 Power implications of the optimization

An increase in the static power of the VPU is expected due to the additional hardware required to support the proposed optimization. However, a reduction in dynamic power is also expected. The optimization enables the direct execution of `vmv.v.x`, `vmv.v.i`, `vmv.s.x`, and `vmv.s.f` instructions during the renaming stage, bypassing the need to issue them to the execution lanes.

In a conventional execution flow, these instructions are first placed into the Arithmetic Queue, then dispatched by the Splitter across multiple lanes. Once received by the VCU lanes, each instruction, along with its associated data and operands, is encapsulated in a structured format, and stored in internal buffers awaiting for execution. In the execution stage, the operand (in this case, scalar data from the scalar core) is written into the Source Buffers and then transferred to the Vector Register File.

This entire process triggers significant switching activity in internal logic, particularly in sequential elements such as queues and buffers, in addition to combinational logic. Moreover, in the standard execution model, `vmv.v.x` and `vmv.v.i` write the same scalar value to all `vl` active elements of the Vector Register File, distributing the data across multiple lanes and banks. This redundant replication leads to substantial switching activity within the VRF. The proposed optimization avoids this unnecessary replication, thereby reducing dynamic power consumption by preventing excessive toggling in both internal logic and memory structures.

8.3 Power estimation steps

To get a precise evaluation of the power impact introduced by the optimization, a workload-aware power estimation must be performed. The process involves the following steps:

Gate-Level Simulation The output of the synthesis stage is a netlist in Verilog format (`.v` file), which contains all the instantiated logic gates of the design. This netlist represents a lower level of abstraction compared to the RTL, as high-level constructs such as registers and processes are translated into logic gates. However, the netlist can still be simulated using tools like Questasim to verify the functional correctness at the logic level. From this simulation, a Value Change Dump (VCD) file is generated. VCD files store simulation data, capturing changes in signal values over time. Since these files are generated during gate-level simulation, they reflect accurate gate-switching activity and can be used for more precise dynamic power calculations.

Power Estimation Joules takes as input the gate-level netlist of the design along with the corresponding technology library. These inputs alone are sufficient for estimating the leakage power. To evaluate dynamic power, Joules additionally requires a VCD file, previously generated from a gate level simulation.

When the VCD is read, Joules correlates each net and gate in the netlist with their corresponding signals in the VCD. This mapping enables the tool to measure actual transition rates across the circuit, which are then used to compute dynamic power consumption based on realistic toggling behavior.

8.4 Results

Due to the timing constraints of the project, the power analysis was conducted focusing on only two workloads. These workloads, matrix multiplication (matmul) and layer normalization (layernorm), were selected because they are highly representative: matmul exhibits the highest number of scalar moves, while layernorm is a more typical workload, involving both scalar moves and reductions, though to a lesser extent. Power consumption was estimated based on synthesis using TSMC's 7 nm technology, a supply voltage (Vdd) of 0.75 V. and an operating frequency during gate-level simulation of 1 GHz.

8.4.1 Matrix multiplication

Table 8.1: Matmul: Baseline vs Optimized VPU power consumption

Power type	**Baseline [W]**	**Optimized [W]**	**% Variation**
Static	$1{,}58 \times 10^{-2}$	$1{,}69 \times 10^{-2}$	7,11%
Dynamic	$4{,}85 \times 10^{-1}$	$4{,}61 \times 10^{-1}$	-4,92%
Total	$5{,}01 \times 10^{-1}$	$4{,}78 \times 10^{-1}$	-4,54%

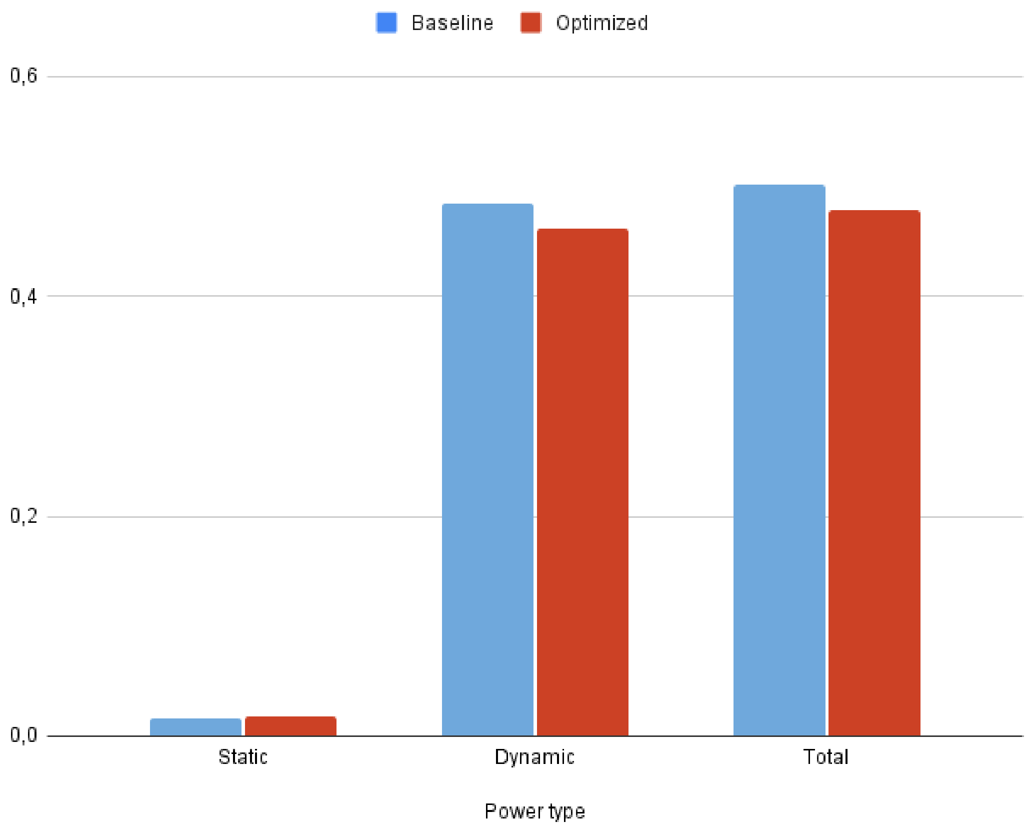

Figure 8.2: Power results Matrix Multiplication workload

8.4.2 Layer Normalization

Table 8.2: LayerNorm: Baseline vs Optimized VPU power consumption

Power type	**Baseline [W]**	**Optimized [W]**	**% Variation**
Static	$1{,}60 \times 10^{-2}$	$1{,}68 \times 10^{-2}$	5,07%
Dynamic	$4{,}95 \times 10^{-1}$	$4{,}78 \times 10^{-1}$	-3,45%
Total	$5{,}11 \times 10^{-1}$	$4{,}94 \times 10^{-1}$	-3,18%

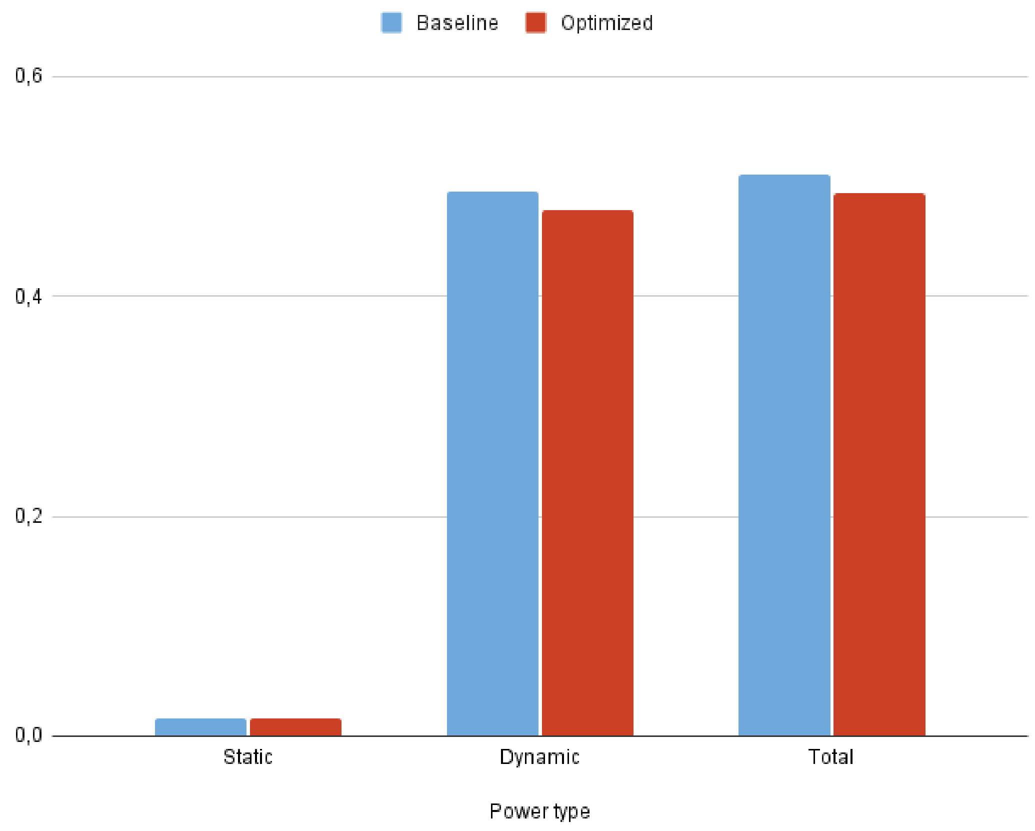

Figure 8.3: Power results Layer Normalization workload

8.5 Comments

As expected, static power consumption increases, reaching up to 7.11%. It is important to note that static power is independent of switching activity, as it depends solely on the netlist and the technology libraries. The observed variation in static power values is due to the inherent margin of error of the analysis tool. The increase in static power is a direct consequence of the additional hardware, which leads to a higher leakage contribution. Nevertheless, static power accounts for only approximately 3% of the total power consumption.

In contrast, dynamic power shows a reduction of 4.92% and 3.44% across the evaluated workloads, resulting in a notable decrease in total power consumption of up to 4.54%. This is a promising result, as it qualifies the proposed optimization as a suitable candidate for low-power VPUs, achieving reduction in power consumption without compromising performance.

Chapter 9

Conclusions

After an in-depth analysis of the Vector Processing Unit microarchitecture, the RISC-V Vector Extension (V 1.0), and the applicability of register renaming techniques to vector processors, this thesis focused on the design and evaluation of an efficient register renaming mechanism aimed at improving register utilization in a RISC-V VPU. The implementation and testing of the proposed solution led to the following key conclusions:

- Performance was evaluated using microbenchmarks from domains such as linear algebra, high-performance computing, and machine learning. The results show performance improvements of up to **12.60%**.
- The area overhead of the optimized architecture after synthesis is minimal, with a maximum increase of only **3.57%**.
- The optimization does **not** impact the operating frequency of the processor.
- A low-area variant of the VPU was explored, leveraging the benefits of the register renaming mechanism and additional scalar registers, while reducing the number of vector registers. This approach achieved an **area reduction of up to 1.29%** and still delivered performance improvements in nearly all benchmarks (best case: **+11.52%**, worst case: **−2.49%**).
- An initial evaluation of **power consumption** was conducted, with preliminary results indicating a reduction up to 4.54%, classifying this optimization a suitable candidate for low-power VPUs.

It is worth noting that the performance gains are closely related to the instruction mix of the evaluated applications, which feature patterns well-suited to the optimized renaming scheme. In any case, three key observations hold:

- Scalar moves and scalar-to-vector moves are consistently frequent across all the analyzed applications, even in those not covered in this thesis.
- The optimized design never degrades performance compared to the baseline architecture, as instructions that are not targeted by the optimization are executed using the standard execution flow.
- The optimization offers minimal area overhead and a reduction in power consumption.

These results demonstrate that the proposed register renaming mechanism is a valuable enhancement for vector architectures, offering significant performance improvements and energy efficiency benefits. As such, it is a strong candidate for inclusion not only in future versions of the VPUs developed at the Barcelona Supercomputing Center, but also in any vector architecture implementing the RISC-V V extension.

The work and the results presented in this thesis serve as a foundation for future work, outlined in the following section, and are expected to contribute to the development of one or more research articles for submission to leading conferences in Computer Architectures.

9.1 Future work

9.1.1 System-level evaluation and FPGA emulation

The current evaluation of the proposed optimization has been limited to the VPU, which operates as a co-processor to a scalar core. The VPU UVM environment integrates a Spike-based model of the scalar core; however, this model is highly idealized and does not accurately reflect real-world system behavior. As a result, the performance improvements measured in this setup may not fully capture the impact of the optimization at the system level. For this reason, future work should include:

1. **Full-system evaluation with RTL simulation**
   A complete system-level RTL simulation should be conducted by integrating the VPU with the scalar core. Running microbenchmarks in this environment would allow for a more accurate analysis of performance improvements and any possible interactions between subsystems.

2. **Full-System FPGA emulation**
   FPGA-based emulation of the entire system should be explored. Due to its significantly faster execution compared to RTL simulation, this approach would

enable the evaluation of full application benchmarks. This would provide deeper insights into the real-world performance and energy efficiency impact of the optimization under realistic workloads.

9.1.2 Optimization of Vector-Scalar operation

The current design does not fully exploit the potential benefits offered by the availability of the Squeezed Register File and the optimized register renaming mechanism. The RISC-V Vector ISA supports vector-scalar arithmetic operations, which involve instructions that operate between a vector and a scalar operand. When such an instruction reaches the renaming stage, a squeezed physical register can be used to delay the actual execution of the operation.

Specifically, the execution can be transformed into a fast move: the destination register's entry in the Register Alias Table (RAT) is updated to point to the same physical register as the vector source operand, effectively allowing the two vector registers to share the same physical register. Meanwhile, a new squeezed physical register is allocated from the Free Squeezed Register List (FSRL), and the scalar data is stored into this squeezed register.

The Register Map Table is extended to distinguish not only between vector and squeezed mappings but also to recognize a new mapping type related to vector-scalar operations. Additionally, it stores the type of arithmetic instruction to be executed (e.g., add, sub, mul). The execution of the arithmetic operation is postponed and is only performed if a store targeting the destination register occurs.

Otherwise, if the destination register is used as a source by a subsequent instruction, that instruction can be transformed into a three-operand instruction, using the new vector source register, the squeezed register (original scalar operand), and the original vector source register. The hardware already supports three-operand operations, as it is used in fused multiply-add instructions. This approach can save many clock cycles by delaying the execution of the original instruction and avoiding unnecessary immediate computation.

Bibliography

[1] Richard M. Russell. «The CRAY-1 computer system». In: *Commun. ACM* 21.1 (Jan. 1978), pp. 63–72. ISSN: 0001-0782. DOI: 10.1145/359327.359336. URL: https://doi.org/10.1145/359327.359336 (cit. on p. 1).

[2] Mateo Valero Roger Espasa and James E. Smith. «Vector Architectures: Past, Present and Future». In: *ICS '98: Proceedings of the 12th international conference on Supercomputing* (1998) (cit. on p. 1).

[3] Lieven Eeckhout. «Is moore's law slowing down? what's next?» In: *IEEE Micro* 37.04 (2017), pp. 4–5 (cit. on p. 1).

[4] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. «Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators». In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 129–140 (cit. on p. 1).

[5] Krste Asanović and David A Patterson. «Instruction sets should be free: The case for risc-v. EECS Department». In: *University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014) (cit. on p. 3).

[6] RISC-V Foundation. *Ratified Extensions*. https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154732/Ratified+Extensions. Accessed: 2025-03-11. 2023 (cit. on p. 3).

[7] RISC-V International. *RISC-V Vector Extension Specification*. Accessed: 2025-07-01. 2025. URL: https://github.com/riscvarchive/riscv-v-spec (cit. on p. 3).

[8] DRAC Project, Barcelona Supercomputing Center. *Lagarto KA: The High Performance Core for DRAC*. https://drac.bsc.es/en/bsc2. Accessed: 2025-07-11. Nov. 2021 (cit. on p. 9).

[9] David I Rich. «The evolution of SystemVerilog». In: *IEEE Design & Test of Computers* 20.04 (2003), pp. 82–84 (cit. on p. 14).

[10] Francesco Minervini et al. «Vitruvius+: an area-efficient RISC-V decoupled vector coprocessor for high performance computing applications». In: *ACM Transactions on Architecture and Code Optimization* 20.2 (2023), pp. 13–15 (cit. on p. 19).

[11] riscv-collab. *riscv-gnu-toolchain*. `https://github.com/riscv-collab/riscv-gnu-toolchain`. Accessed: 2025-07-04. 2025 (cit. on p. 34).

[12] riscv-software-src. *riscv-isa-sim*. `https://github.com/riscv-software-src/riscv-isa-sim`. Accessed: 2025-07-03 (cit. on p. 34).

[13] Saeed Maleki, Yaoqing Gao, María J. Garzarán, Tommy Wong, and David A. Padua. «An Evaluation of Vectorizing Compilers». In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. 2011, pp. 372–382. DOI: `10.1109/PACT.2011.68` (cit. on p. 36).

[14] Hsiangkai Wang, Zakk Chen, Kito Cheng, Yi-Hsiu Hsu, Roger Ferrer Ibanez, Nick Knight, and Mingjie Xing. *RVV Intrinsic API Documentation (Version 0.11.x)*. `https://github.com/riscv-non-isa/rvv-intrinsic-doc/blob/v0.11.x/rvv-intrinsic-api.md`. Accessed: 2025-07-11. 2021 (cit. on p. 36).

[15] Ramirez Cristobal, Hernandez Cesar Alejandro, Palomar Oscar, Unsal Osman, Ramirez Marco Antonio, and Cristal Adrian. «A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures». In: *ACM Trans. Archit. Code Optim.* 17.4 (Nov. 2020). ISSN: 1544-3566. DOI: `10.1145/3422667`. URL: `https://doi.org/10.1145/3422667` (cit. on p. 36).

[16] Matteo Perotti, Matheus Cavalcante, Renzo Andri, Lukas Cavigelli, and Luca Benini. «Ara2: Exploring Single- and Multi-Core Vector Processing With an Efficient RVV 1.0 Compliant Open-Source Processor». In: *IEEE Trans. Comput.* 73.7 (July 2024), pp. 1822–1836. ISSN: 0018-9340. DOI: `10.1109/TC.2024.3388896`. URL: `https://doi.org/10.1109/TC.2024.3388896` (cit. on p. 37).

[17] P.M. Djuric, J.H. Kotecha, Jianqui Zhang, Yufei Huang, T. Ghirmai, M.F. Bugallo, and J. Miguez. «Particle filtering». In: *IEEE Signal Processing Magazine* 20.5 (2003), pp. 19–38. DOI: `10.1109/MSP.2003.1236770` (cit. on p. 38).

[18] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. «Layer normalization». In: *arXiv preprint arXiv:1607.06450* (2016) (cit. on p. 38).

[19] W.M. Elgharbawy and M.A. Bayoumi. «Leakage sources and possible solutions in nanometer CMOS technologies». In: *IEEE Circuits and Systems Magazine* 5.4 (2005), pp. 6–17. DOI: `10.1109/MCAS.2005.1550165` (cit. on p. 51).