RISC-V Vector Extension

Charalampos Zachariadis - 03734
Electrical and Computer Engineering
University of Thessaly
Volos, Greece
zachariadis.charis@gmail.com

Stefanos Ziakas - 03568
Electrical and Computer Engineering
University of Thessaly
Volos, Greece
sziakas@uth.gr

Filippos Markovitsis - 03488
Electrical and Computer Engineering
University of Thessaly
Volos, Greece
fmarkovitsis@uth.gr

Eleni Athanailidi - 03453
Electrical and Computer Engineering
University of Thessaly
Volos, Greece
eathanailidi@uth.gr

***Abstract*—Vector instructions allow modern CPUs to perform the same operation on multiple data points at once, using a model known as Single Instruction, Multiple Data (SIMD). This parallelism helps to speed up common operations in areas such as scientific computing, graphics, and machine learning.**

Because of these advantages, modern processors and compilers are increasingly built to support and take advantage of vectorization. In this project, we explore the role of vector instructions and implement vectorized code to better understand how they work and why they matter in modern computing.

I. Introduction

The RISCY SoC is a compact RISC-V-based system-on-chip built around a six-stage pipelined core. Although originally developed as a simple RV32I implementation, its straightforward architecture makes it a practical platform for exploring custom instruction set extensions [1]. In this project, we enhance RISCY with hardware support for vector processing to enable Single Instruction, Multiple Data (SIMD) execution, addressing the limitations of its scalar-only design for data-parallel workloads.

We introduce a custom Vector Processing Unit (VPU), a dedicated vector register file, a vector-aware memory access logic and new control and status registers to manage vector operations. The instruction set includes vector arithmetic, data movement and control instructions for configuring vector length and grouping. The scalar pipeline is extended to decode and dispatch these instructions while preserving compatibility with the baseline RV32I ISA. All enhancements are implemented in Verilog and fully integrated into the existing SoC without disrupting scalar functionality.

By adding vector capabilities, this work increases the performance of parallelizable workloads and demonstrates how SIMD execution can be realized on lightweight RISC-V platforms.

II. Background and Terminology

To enable vector operations in hardware, it is important to understand the foundational elements of the RISC-V Vector Extension (RVV). These components define how vector instructions behave and how data is handled in parallel. In this section, we explain the key terms and concepts that are central to our implementation.

A. Vector Register Length (VLEN)

VLEN is the fixed size (in bits) of each physical vector register in the hardware. In our project, we set VLEN to 64 bits. This means each register holds a total of 64 bits, which are then divided into elements based on the selected SEW. For example, a SEW of 8 results in 8 elements per register.

B. Vector Length (VL)

The Vector Length (VL) determines how many elements a vector instruction will operate on during a single execution. VL is set at runtime and depends on both the application's request and the available hardware resources. If the hardware can't process all elements at once, the remaining elements are handled in subsequent iterations.

C. Application Vector Length (AVL)

The Application Vector Length is the number of elements that the program wants to process. However, this number may be larger than what the hardware can support in one go. The actual number of elements processed at a time is capped by the hardware-defined VLEN, and the remaining work is done over multiple cycles.

D. Standard Element Width (SEW)

SEW refers to the size of each individual data element in a vector register. For example, if SEW is 8, the processor treats the data as a collection of 8-bit values. Changing SEW allows the same hardware to process different types of data - like bytes, half-words, or full words - depending on the operation.

E. Register Grouping (LMUL)

LMUL (Length Multiplier) allows vector registers to be grouped together to handle wider operations or higher element counts. For example, an LMUL of 2 means the instruction operates over two vector registers as one logical vector. This is necessary when SEW is large or when the operation requires more data than a single register can hold. The decoded value of LMUL also determines how many iterations the vector unit must go through to complete a full instruction.

F. VTYPE Register

The vtype register holds configuration values that define the behavior of vector instructions. Specifically, it encodes the SEW (Standard Element Width) in bits [5:3] and the LMUL (Register Grouping Multiplier) in bits [2:0]. These values are extracted in our hardware logic to determine how data is partitioned and processed. The vtype register is updated by vector configuration instructions, which we will discuss in the following sections.

III. Design Acceptances

To simplify implementation, a number of design-level assumptions and simplifications have been made. These acceptances are intentionally chosen to preserve functional correctness while reducing system complexity.

A. CSR Handling Simplification

In the standard RISC-V vector extension, control and status registers (CSRs) such as VL, vtype, and others are accessed via dedicated CSR instructions and identified through specific CSR addresses.

However, in this implementation, only a minimal subset of CSRs is used - specifically those required for vector arithmetic instruction execution (VL, vtype). These CSRs are not accessed through standard CSR addressing or individual CSR read/write instructions. Instead, all relevant vector CSRs are configured simultaneously using a single, dedicated instruction. This acceptance simplifies CSR logic by avoiding the need for a full CSR decoding mechanism and register bank.

In addition to this, AVL holds its own dedicated CSR register. This decision was taken in the early stages of the implementation of the RVV extension, as it was suggested by numerous manuals. Later on, manuals that do not include this extra CSR register were discovered. In any case, the inclusion of the AVL CSR register was considered a good way for the VPU to have exclusive control over its configuration parameters.

B. Limited AVL Range

In this implementation, the AVL (Application Vector Length) is restricted to a 5-bit immediate value, used in the vsetivli instruction. As such, the maximum representable AVL is 31, regardless of the hardware's VLMAX.

This differs from the official RISC-V Vector Extension (RVV), which allows AVL to be set via a register and supports a range of:

$vl \leq AVL \leq 2 \times vl_{\text{max}}$

Our implementation does not currently support this full range of register-based AVL values. As a result, while multiple iterations are functionally supported in the logic, the total workload must still be encoded within the limited immediate AVL field.

C. Unimplemented Configuration Instructions Support

Although the hardware decoding logic includes recognition for the vsetivli, vsetvli, and vsetvl instructions, only vsetivli is currently supported and fully implemented.

While vsetvli and vsetvl are decoded and partially handled, their full functionality-particularly the ability to read the AVL value from a scalar register- is not yet supported in the pipeline. As a result, vector configuration in the current implementation relies solely on immediate values provided by vsetivli.

D. Partial vtype Field Support

The vtype field in the RISC-V Vector Extension contains several configuration bits, including those related to masking.

In this implementation, only a subset of the vtype bits is used, specifically, those that encode the SEW (Standard Element Width) and LMUL (Register Grouping). Features such as multiple masking options are not currently supported, and the corresponding vtype bits are ignored.

Additionally, the most significant bit of vtype (bit 6) is repurposed as a validity flag. This bit is used internally to indicate whether the current vtype is valid and whether VL, AVL and vtype should be updated. This simplifies control logic, but diverges from the official RVV specification where vtype[6] is reserved.

E. Single Masking Option

Originally, RVV allows several masking options, concerning the elements that fall out of the most recently updated VL, such as simply ignoring these elements and the operations that are done with them, replacing them with zeros etc. To avoid implementing redundant logic, it was agreed to simply replace excess elements with zeros, which is one of the masking options.

IV. Supported Instructions

Our vector extension introduces a set of vector instructions, designed to work on both vector-vector and vector-scalar data, and to support configurable vector lengths and element widths. The design focuses on simplicity, flexibility, and seamless integration with an existing scalar-only RISC-V core.

A. Configuration Instruction

To configure the vector unit, we implemented the following instruction:

- **vsetivli (rd, uimm, vtype)** Sets the vector length (VL) and vector type (vtype) based on an immediate AVL (Application Vector Length) and vtype. This instruction allows the program to specify how many elements to operate on and configure properties such as SEW (Standard Element Width) and LMUL (Length Multiplier) without depending on general-purpose registers.

This instruction enables dynamic adaptation to varying data widths and register grouping configurations while keeping control logic simple and efficient.

B. Arithmetic Instructions

We implemented a wide range of vector arithmetic and logical operations to support parallel data processing:

- **vadd.vv (vd, vs1, vs2)** Performs element-wise addition between two vector registers (vs1 and vs2) and stores the result in vd.
- **vadd.vx (vd, vs2, rs1)** Adds each element of the vector register vs2 with a scalar value from register rs1 and stores the result in vd.
- **vadd.vi (vd, vs2, imm)** Adds each element of the vector register vs2 with an immediate scalar value and stores the result in vd.
- **vsub.vv (vd, vs1, vs2)** Performs element-wise subtraction between two vector registers (vs2 - vs1) and stores the result in vd.
- **vsub.vx (vd, vs2, rs1)** Subtracts a scalar value from register rs1 from each element of the vector register vs2 and stores the result in vd.
- **vsub.vi (vd, vs2, imm)** Subtracts an immediate scalar value from each element of the vector register vs2 and stores the result in vd.
- **vmul.vv (vd, vs1, vs2)** Performs element-wise multiplication of two vector registers and stores the result in vd.
- **vmul.vx (vd, vs2, rs1)** Multiplies each element of vector register vs2 with a scalar value from register rs1 and stores the result in vd.
- **vmul.vi (vd, vs2, imm)** Multiplies each element of vector register vs2 with an immediate scalar value and stores the result in vd.
- **vand.vv (vd, vs1, vs2)** Performs element-wise bitwise AND between two vector registers and stores the result in vd.
- **vand.vx (vd, vs2, rs1)** Performs element-wise bitwise AND between vector register vs2 and scalar value from register rs1 and stores the result in vd.
- **vand.vi (vd, vs2, imm)** Performs element-wise bitwise AND between vector register vs2 and an immediate scalar value and stores the result in vd.
- **vor.vv (vd, vs1, vs2)** Performs element-wise bitwise OR between two vector registers and stores the result in vd.
- **vor.vx (vd, vs2, rs1)** Performs element-wise bitwise OR between vector register vs2 and the scalar value from register rs1 and stores the result in vd.
- **vor.vi (vd, vs2, imm)** Performs element-wise bitwise OR between vector register vs2 and an immediate scalar value and stores the result in vd.
- **vxor.vv (vd, vs1, vs2)** Performs element-wise bitwise XOR between two vector registers and stores the result in vd.
- **vxor.vx (vd, vs2, rs1)** Performs element-wise bitwise XOR between vector register vs2 and the scalar value from rs1 and stores the result in vd.
- **vxor.vi (vd, vs2, imm)** Performs element-wise bitwise XOR between vector register vs2 and an immediate scalar value and stores the result in vd.

These arithmetic vector instructions operate on entire vectors in parallel, applying the same operation element-wise across all active elements, thereby enabling efficient data-level parallelism as determined by the configured vector length.

C. Memory Instructions

For transferring data between vector registers and memory:

- **vsb.v (vs3, (rs1 + imm))** A premature version of vsb (storing a vector register byte-by-byte in memory) was created to project our results in the memory and ultimately on a screen. It is currently not capable of storing the whole vector in the memory with just one instruction, as the understanding of the already implemented API related to memory handling caused an overall slowdown in the integration and implementation process.

V. Detailed Module Descriptions

A. vsetvl

The vsetvl module is responsible for extracting vector configuration parameters from vector setup instructions. It does not perform full instruction decoding, but instead isolates the key fields required for configuring the vector execution environment, SEW (Standard Element Width), LMUL (Length Multiplier), and AVL (Application Vector Length).

This module operates only when the incoming instruction matches both the vector instruction format and the vector configuration function code:

- The opcode must match VR_FORMAT (7'b1010111), which identifies all vector instructions.
- The funct3 field must match VC_FORMAT (3'b111), indicating that the instruction is intended for vector configuration.

When these conditions are met, the module extracts the relevant fields as follows:

- MUL is extracted from bits [2:0] of the vtypei field.
- SEW is extracted from bits [5:3] of the vtypei field.
- AVL is taken from the 5-bit immediate input AVL_imm and zero-extended to 7 bits for compatibility with the internal vector length representation

To enable the update of vector control registers (vl, vtype) in downstream logic, the module asserts the vcsr_wen signal when a valid configuration instruction is detected. If the instruction does not match the expected pattern, the module outputs default values (ones) and deasserts vcsr_wen, effectively preventing any change to the current vector configuration.

By isolating configuration logic into this module, the design cleanly separates vector setup from the scalar pipeline and ensures that only properly formed configuration instructions affect the processor's vector state.

B. vl_setup

The vl_setup module is responsible for determining the effective vector length (vl) to be used by the processor during vector operations. It computes this value based on the currently configured element width (SEW), register grouping (LMUL), and the application's requested vector length (AVL), while also ensuring that vector execution is only enabled under valid conditions.

Before any calculation, the module checks whether vector configuration is allowed by verifying that the vcsr_wen signal is high (vcsr_wen == 1'b1). This signal is asserted by the vsetvl module when a valid vector configuration instruction (vsetivli) is encountered. It ensures that vector control registers are only updated when explicitly instructed by the program.

If vector setup is permitted, the module then validates the SEW and LMUL values. Specifically, it checks that the most significant bit of each is unset. This ensures that only supported values are used: SEW values corresponding to 8, 16, 32, or 64-bit elements, and LMUL values of 1, 2, 4, or 8.

The core of the module's logic lies in calculating vlmax, the maximum number of elements that can fit into the vector register group under the current configuration. The calculation is defined as:

$$\texttt{vlmax} = \left(\frac{\texttt{VLEN}}{2^{\texttt{SEW}+3}}\right) \times 2^{\texttt{LMUL}}$$

Here, VLEN is the total number of bits available in the hardware vector register file, fixed at 64 in this design. The term $2^{\text{SEW}+3}$ converts the encoded SEW value into the actual bit-width of each vector element (e.g., SEW = 0 means 8 bits, SEW = 1 means 16 bits, etc.), and 2^{LMUL} scales the number of registers used for wide vectors.

In the Verilog code, this is implemented as:

$\texttt{curr\_vlmax} \leftarrow (\texttt{VLEN} \gg (\texttt{SEW} + 3)) \times (1 \ll \texttt{lmul})$

After computing vlmax, the actual vector length (vl) is determined as the smaller of AVL and vlmax. If the application's requested AVL exceeds what the hardware can support in a single operation, the module updates a residual AVL value (new_AVL) to reflect the number of elements that remain to be processed in future iterations.

Moreover, we should also consider that AVL needs to match with LMUL and SEW. For example we should not group more registers than the ones we need based on VL and SEW. For example, if we have SEW = 8 and VL = 3, we need a total of $3*8 = 24$ bits, which are available from a single register. That means that LMUL values bigger than 1 (2, 4 or 8) are invalid since multiple vector registers are unnecessary to provide the necessary outputs.

If vector configuration is not permitted (vcsr_wen is low or an invalid SEW/LMUL setting is detected), the module safely disables vector execution by outputting zeros for vl, AVL, and vtype.

C. *vRegFile*

The vRegFile module implements the vector register file - the core storage component for all vector data in the processor. It allows flexible access to vector registers, supporting both standard and grouped access modes depending on the current LMUL register.

In addition to storing the actual vector values, this module also supports dynamic updates of vl and vtype, which control vector processing modes and element grouping, adapting to the current instruction context.

It works in tandem with the masking logic to zero out elements that fall outside the active vector length, preserving correctness during segmented operations.

Read operations include bypassing logic to forward recently written data during the same clock cycle, ensuring data consistency and reducing pipeline stalls.

D. *vcontrol_bypass*

The vcontrol_bypass module implements data forwarding (bypassing) logic to resolve data hazards within the vector pipeline. It continuously monitors register dependencies by comparing source register indices of the instruction currently in the decode/execute stage (IDEX) against destination registers of instructions in subsequent pipeline stages (EXMEM and MEMWB).

When the source registers (vrs1 or vrs2) of the current instruction match the destination register (vrd) of an instruction further along the pipeline, the module forwards the most recent vector ALU output directly from the EXMEM or MEMWB stage. This ensures that the current instruction uses the latest available data without waiting for the register file to be updated.

By dynamically forwarding operands through the bypassOutvA and bypassOutvB outputs, the module effectively eliminates pipeline stalls caused by data hazards. This mechanism is essential for maintaining high throughput and low latency in vector processing.

It is also important to mention that the already implemented bypass unit of the vanilla RISC-V processor was also used in order to eliminate data dependencies during the execution of vector instructions, whenever they include the reading of data of the original RegFile.

E. *vgrouping_selector*

The vgrouping_selector module controls how vector instructions operate over groups of registers when LMUL (Length Multiplier) is greater than 1. In such cases, a single logical vector register spans multiple physical registers, and the instruction must be applied in stages across these grouped registers.

This module tracks which part of the group is currently being processed using an internal counter (cnt_in). It updates the register indices (raA, raB, and rdest) accordingly, so that the correct slice of the vector is accessed during each iteration. For example, if LMUL = 4, the same instruction will run four times, incrementing the register indices each time to cover the full group.

The stall signal is used to pause instruction fetch while the grouped operation is still in progress. Specifically, if grouping is enabled (meaning the current active instruction is a vector instruction that requires grouping) and the current counter value (cnt_in) indicates that there are still more sub-operations to perform (cnt_in $<$ LMUL - 1), the module asserts a stall. This ensures that the current vector instruction remains active in the pipeline until all required sub-iterations are completed. During each cycle, the counter (cnt_out) is incremented, advancing to the next slice of the grouped registers. Once all slices have been processed, the stall is deasserted and the counter resets, allowing the pipeline to proceed to the next instruction.

Internally, the module decodes the encoded LMUL value into a usable number (1, 2, 4, or 8), representing how many registers are grouped together.

F. control_vunit

The vcontrol_unit module is responsible for decoding vector instructions and generating the control signals necessary to direct the vector arithmetic logic unit (vALU). It takes as inputs the instruction fields opcode, funct3, and funct6, and produces outputs to manage operand selection, operation type, write enable, and element grouping.

When the opcode matches the vector instruction format (VR_FORMAT), the unit uses the funct3 field to determine the type of vector operation type:

- VV_FORMAT (vector-to-vector)
- VX_FORMAT (vector-to-scalar)
- VI_FORMAT (vector-to-immediate)
- VC_FORMAT (vector control/configuration)

Depending on the funct3 value, the module sets several control signals:

- valu_src: Selects the source of the scalar operand for the vALU. It is set to 0 by default and whenever the scalar operant is the data read from a scalar register, and to 1 when the scalar operand is an immediate.
- vRegWrite: Enables writing the result back to the vector register file. This is asserted (1) for all arithmetic and logical operations, and deasserted (0) for control/configuration instructions.
- grouping_enable: Enables grouped (parallel) processing of vector elements, set to 1 during arithmetic/logical operations and 0 otherwise.

Within each funct3 category, the control unit further identifies the specific arithmetic or logical operation by decoding the funct6 field. This determines the command sent to the vALU, specifying whether to perform addition, subtraction, multiplication, or bitwise logical operations such as AND, OR, and XOR.

For VC_FORMAT instructions, which handle vector control and configuration, the control signals disable writeback (vRegWrite = 0), set valu_op to 1111 (no operation), and disable grouping (grouping_enable = 0), as these instructions do not perform arithmetic or logical operations.

If the opcode does not correspond to the vector format, or the instruction is unrecognized, the control unit defaults to disabling all vector operations by setting valu_src to 1'b0, vRegWrite to 0, valu_op to 1111, and grouping_enable to 0.

G. sign_ext_64

The sign_ext_64 module performs sign extension on two types of inputs: a 5-bit immediate value (simm5) and a 32-bit scalar input (scalar_in). Both inputs are extended to 64 bits to match the processor's vector width.

- The 5-bit immediate is extended by replicating its most significant bit (sign bit) 59 times, forming a 64-bit signed immediate (simm64).
- The 32-bit scalar input is similarly extended by replicating its sign bit 32 times, producing a 64-bit signed scalar (scalar_in_64).

This ensures that subsequent vector operations correctly interpret the signed values, regardless of their original bit widths.

H. sign_ext_64_selector

The sign_ext_64_selector module chooses which 64-bit extended input to provide to the vector arithmetic logic unit (vALU) as the first operand, in case of a vector - scalar operation.

It receives a control signal, valu_src, which determines the source of the operand:

- When valu_src is 0, the scalar input (scalar_in_64) is selected.
- When valu_src is 1, the sign-extended immediate (simm64) is chosen.

This selection enables flexible handling of different instruction types, allowing the vALU to operate on either scalar or immediate values as needed.

I. vALU

The vALU (vector Arithmetic Logic Unit) is responsible for executing all vector arithmetic and logical operations in the processor. It is designed to efficiently support both fully vectorized instructions as well as operations that combine vector and scalar inputs.

The module uses the SEW (Standard Element Width) setting to dynamically divide each 64-bit vector input into smaller elements of 8, 16, 32, or 64 bits. For example, if SEW is set to 8 bits, the input is split into eight separate 8-bit elements, allowing the vALU to perform eight parallel operations such as additions or logical ANDs at once.

When an operation involves a scalar input, the scalar value is broadcast across all elements of the vector. This means the scalar is applied element-wise to each part of the vector, matching the element width defined by SEW.

All arithmetic operations, including addition, subtraction, and multiplication, are performed on a signed basis, ensuring correct handling of positive and negative values. Logical operations such as AND, OR, and XOR are also supported, with scalar-vector cases handled similarly through element-wise application.

The entire vALU is implemented as a purely combinational circuit, producing results within a single clock cycle. This design choice simplifies the control logic and improves performance by avoiding additional sequential stages.

J. *vl_masking*

The vl_masking module is a support unit responsible for tail agnostic zeroing masking, ensuring that vector operations only affect the active elements defined by the current vector length (VL). This is particularly important when a group of vector elements is only partially filled - for example in the final group of a segmented operation.

To achieve this, the module dynamically constructs a bit-mask based on both the current VL and SEW configuration. Elements beyond the VL boundary are automatically masked -effectively set to zero- so that they do not participate in the final result that will be stored in the vector register.

K. *Integration to Existing RISC-V Architecture*

Here is the dataflow that shows our own modules integated to an existing RISC-V 6 staged pipelined CPU that uses scalar regs:

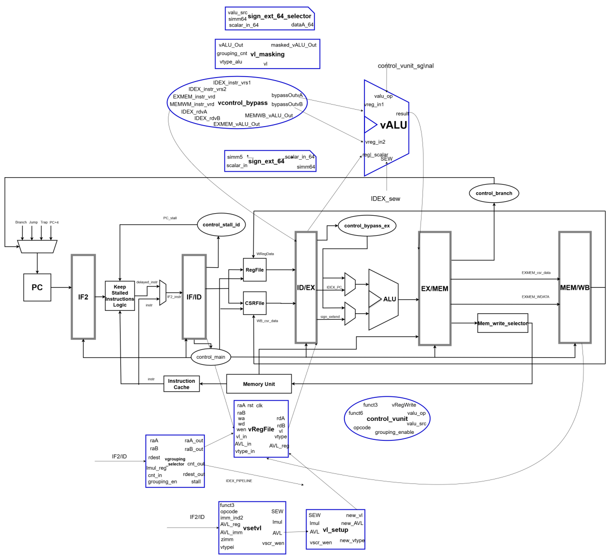

Fig. 1. Dataflow Design of the Integration

VI. Testing / Verification

The vector extension implementation was verified using the existing simulation infrastructure provided by the RISC-Y core. The process involved running a set of hand-crafted vector instructions, simulating and inspecting both scalar and vector register file states. The tests we have created include both small tests, targeted to stress "vulnerabilities" of our implementation and identify their chance to cause a bug, and larger tests with multiple instructions, aiming to simulate the processing unit's behavior while running an actual program. We also tested the the co-existence of the VPU with the core processing unit, making sure that swapping from one unit to the other is as seemless as possible.

The tests provided for the confirmation of the correct functionality of the VPU are:

- Test vadd.vi and vadd.vv with multiple SEW and LMUL values, along with the correct functionality of AVL and forwarding
- Test vsub.vi and vsub.vv
- Test .vx, .vv and .vi instructions, along with multiple forwarding scenarios of both vRegs and classical Regs
- Use the vsb.v instruction to store the word "hell" in memory
- Program with infinite loop, recording the continuous increment of counters
- Test the functionality of the masking module

VII. Current Limitations and Future Work

While the current implementation provides full support for basic vector operations with LMUL values of 1 and above, it does not yet support fractional LMUL settings (like 1/2, 1/4, 1/8) as defined in the RISC-V vector specification. Fractional LMUL allows a single vector register to be subdivided into smaller logical registers, enabling multiple low-width operations to run in parallel and improving hardware utilization, especially for operations involving smaller data types.

Supporting fractional LMUL would require significant changes to the register access logic, including more advanced indexing and masking techniques to prevent resource conflicts when multiple logical registers share the same physical space. Although not included in the current design, supporting fractional LMUL remains a high-priority target for future development, as it would significantly improve the flexibility and coverage of our vector extension framework.

Additionally, only the vsetivli instruction is currently implemented for vector configuration. Full support for the standard vector setup instructions-such as vsetvli and vsetvl, which allow the use of general-purpose registers for dynamic AVL control is not yet included. Incorporating these instructions would enhance runtime configurability and bring the implementation closer to full compliance with the RISC-V vector specification.

Currently, the implementation does not include support for vector load instructions, which are essential for efficiently bringing data from memory into vector registers for subsequent computation. The absence of vector memory operations limits the practical usability of the vector unit, as real-world vectorized applications typically require frequent interaction with memory.

Implementing vector load support would involve extending the data path to include a vector memory interface, handling alignment and stride considerations, and supporting masking behavior for partial loads. Furthermore, integration with the scalar memory subsystem would be necessary to manage memory hazards and ensure correct ordering in mixed scalar-vector programs.

The current design includes only minimal support for control and status registers (CSRs) related to the vector extension, with limited decoding logic and restricted access patterns. To ensure full compatibility with the RISC-V specification and broader software toolchains, future work should include comprehensive implementation of standard CSR read/write instructions such as csrrw, csrrs, and csrrc for all relevant vector CSRs -including VL and vtype.

REFERENCES

[1] P. Nanousis and K. Varakliotis, “RISCY: A Custom RISC-V SoC,” University of Thessaly, Technical Report, Feb. 2025. [Online]. Available: https://github.com/Nanousis/RiscY/blob/main/Documentation/RiscY_project_report.pdf

[2] A. Amid *et al.*, “The RISC-V Vector Extension Version 1.0,” RISC-V International, Tech. Spec., Sept. 2021. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0

[3] Samsung Research, “RISC-V and Vectorization,” Samsung Research Blog, Aug. 2022. [Online]. Available: https://research.samsung.com/blog/RISC-V-and-Vectorization

[4] A. Kruppe and R. Espasa, “RISC-V Vectors and LLVM,” presented at the 2019 LLVM Developers’ Meeting, Apr. 2019. [Online]. Available: https://llvm.org/devmtg/2019-04/slides/TechTalk-Kruppe-Espasa-RISC-V_Vectors_and_LLVM.pdf

[5] A. Peralta, “Otter RISC-V Vector Extension Instruction Manual, Version 1.0,” Unpublished technical report, 2021. [Online]. Available (referenced GitHub links): https://github.com/riscv/riscv-v-spec/releases/tag/0.7.1

[6] “Unprivileged ISA for RISC-V,” University of Illinois at Urbana-Champaign, ECE391 Course Document, Apr. 2024. [Online]. Available: https://courses.grainger.illinois.edu/ece391/su2025/docs/unpriv-isa-20240411.pdf

[7] M. Ali, M. von Ameln, and D. Goehringer, “Vector Processing Unit: A RISC-V based SIMD Co-processor for Embedded Processing,” in *Proc. 2021 24th Euromicro Conf. on Digital System Design (DSD)*, Palermo, Italy, Sep. 2021, pp. 57–64. [Online]. Available: https://ieeexplore.ieee.org/document/9556410