POLITÉCNICA

Escuela Técnica Superior de Ingenieros Industriales
Universidad Politécnica de Madrid

José Gutiérrez Abascal, 2. 28006 Madrid
Tel.: 91 336 3060
info.industriales@upm.es

www.industriales.upm.es

Ane Corral Margeli

05 TRABAJO FIN DE MASTER

INDUSTRIALES

TRABAJO FIN DE MASTER

Hardware-Accelerated RISC-V Vector Extension for High-Performance Embedded Computing

TRABAJO FIN DE MASTER PARA LA OBTENCIÓN DEL TÍTULO DE MASTER EN ELECTRÓNICA INDUSTRIAL

FEBRERO 2026

Ane Corral Margeli

DIRECTORES DEL TRABAJO FIN DE MASTER:

Alfonso Rodríguez Medina

Iñigo Díez de Ulzurrun Aquerreta

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES

DEPARTAMENTO DE ELECTRÓNICA INDUSTRIAL

MADRID

TRABAJO DE FIN DE MASTER

Hardware-Accelerated RISC-V Vector Extension for High-Performance Embedded Computing

Autor: Ane Corral Margeli

Directores: Alfonso Rodríguez Medina e Iñigo Díez de Ulzurrun Aquerreta

Titulación: Máster en Electrónica Industrial

FEBRERO 2026

ACKNOWLEDGEMENTS

Quiero dar las gracias a mis tutores, Alfonso e Iñigo, por darme la oportunidad de trabajar en este proyecto. Gracias por vuestro apoyo y por permitirme aprender a vuestro lado cada día.

A mis compañeros del departamento, gracias por compartir este camino. Formar parte de este equipo hace que este proceso haya sido más enriquecedor y agradable. Gracias por compartir con vosotros el día a día.

ABSTRACT

The contemporary computational landscape is increasingly defined by the intensive data-processing requirements of Artificial Intelligence (AI) and Machine Learning (ML). To address these demands, Single Instruction, Multiple Data (SIMD) strategies have emerged as a critical approach for accelerating data-intensive workloads through the parallelisation of operations. Within this context, the open-source RISC-V Instruction Set Architecture (ISA) facilitates efficient SIMD computation through its RISC-V Vector Extension (RVV).

This thesis presents the design and validation of a vector accelerator tailored for high-performance tasks within embedded systems. The architecture is developed with a modular foundation to support future scalability and implements the Zve32x vector sub-extension, providing support for 32-bit integer operations. Integrated as a coprocessor to the CV32E20 core within the eXtendable Heterogeneous Energy-efficient Platform (X-HEEP) ecosystem, the accelerator extends the scalar core's capabilities via the Core-V eXtension Interface (CV-X-IF) 1.0. Data memory accesses to perform load/store operations are handled through the Open Bus Interface (OBI) v1.0 protocol to ensure efficient data throughput.

A first implementation of the accelerator, constrained to a Vector Register Length (VLEN) of 128 bits, was validated via simulation and on Xilinx Pynq-z2 Field-Programmable Gate Array (FPGA). Performance was evaluated using standard data-parallel kernels: SAXPY and Indexed Arithmetic, which perform scalar-vector multiplication; and Matmul, executing matrix multiplication. Furthermore, this research evaluates the RISC-V GNU Compiler Toolchain, specifically investigating its auto-vectorisation capabilities in C-based applications. Comparative analysis was performed between standard C implementations and those utilising "RISC-V Vector C Intrinsics". Results from simulation and FPGA execution demonstrate that the proposed accelerator achieves a maximum speed-up of 3.83× for the Indexed Arithmetic algorithm when employing the C Intrinsics library, highlighting the performance advantages of manual vectorisation in specialised embedded hardware.

Key words: Parallel computing, High-Performance Computing, Vector Accelerator, RISC-V, CV-X-IF 1.0, Vector Extension.

UNESCO Codes

- 1203 COMPUTER SCIENCES
  - 120311 Computer Software
- 2203 ELECTRONICS
- 3304 COMPUTER TECHNOLOGY
  - 330403 Arithmetic and Machine Instructions
  - 330406 Computer Architecture
- 3307 ELECTRONIC TECHNOLOGY
  - 330790 Microelectronics

Contents

1 Introduction **11**

1.1 Context and Motivation 11

1.2 Goals of the Thesis 12

1.3 Technology Framework 12

1.3.1 RISC-V Instruction Set Architecture 13

1.3.2 Core-V eXtension Interface (CV-X-IF) 19

1.3.3 Open Bus Interface (OBI) 23

1.3.4 X-HEEP 26

2 State of the Art **29**

2.1 RISC-V Based Vector Processors 29

2.1.1 Ara 30

2.1.2 Ara XL 31

2.1.3 Implementation by Johns et al. 32

2.1.4 Vicuna 33

2.1.5 Spatz 33

2.2 Comparison of Vector Processing Units (VPUs) 35

3 Hardware Development **37**

3.1 Methodology 37
3.2 Vector Accelerator Interfaces 37
3.2.1 The X-Fusion Integration Platform 37
3.2.2 Core-V eXtension Interface 38
3.2.3 OBI 39
3.3 Accelerator Core 39
3.3.1 Main Characteristics 39
3.3.2 Proposed VPU Architecture 40

4 Software Development Environment and Methodologies 49

4.1 RISC-V Compilation Toolchain and Configuration 49
4.2 Programming Methodologies for Vector Acceleration 50
4.2.1 RISC-V Vector C Intrinsic 50
4.2.2 Standard C and Auto-vectorization 50
4.2.3 Summary of Programming Approaches 52

5 Results 55

5.1 Tests Applications 55
5.1.1 SAXPY Kernel 55
5.1.2 Indexed Arithmetic Kernel 56
5.1.3 MatMul Kernel 58
5.2 Performance Evaluation 59
5.2.1 SAXPY Performance 59
5.2.2 Indexed Arithmetic Performance 60
5.2.3 Matrix Multiplication (MatMul) Performance 61
5.2.4 Execution in FPGA 62

5.2.5 General Conclusions 63

5.3 Hardware Resource Utilisation 64

5.4 Implementation Limitations and Compiler Constraints 65

6 Conclusions and Future Lines 67

Appendix A RVV Instructions 69

A.1 Instruction Formats 69

A.2 Supported Instructions for Zve32x subextension 71

Bibliography 73

TIME PLANNING AND BUDGETING 75

SOCIAL AND PROFESSIONAL RESPONSIBILITY 77

LIST OF FIGURES 82

LIST OF TABLES 84

ABBREVIATIONS 85

Chapter 1

Introduction

As edge-oriented high-performance embedded systems increasingly integrate AI/ML inference capabilities, vector processing has become a technique to exploit data-level parallelism under strict energy constraints. In parallel, open-source architectures based on RISC-V have gained significant traction as a flexible and innovative solution for such acceleration. This chapter introduces the background and motivation for the project and outlines the primary objectives of the work. Finally, a detailed analysis of the technological landscape to enable hardware acceleration is presented.

1.1 Context and Motivation

The rapid adoption of AI with ML applications has transformed the computing landscape, giving rise to predominantly data-intensive workloads and significantly increasing the demand for computational resources. These workloads typically require massive data processing capabilities that exceed the efficiency of traditional scalar architectures. Consequently, to meet these throughput demands, system designers are increasingly relying on SIMD architectures, such as vector processors, to exploit data-level parallelism.

Although vector architectures have traditionally been a cornerstone of High-Performance Computing (HPC), a critical challenge lies in adapting these capabilities for the embedded systems domain. Modern embedded systems face a dual constraint: they must handle complex workloads, such as real-time image processing and edge-AI, while strictly adhering to the low-power consumption and limited resource availability typical of Internet of Things (IoT) applications.

Addressing these multifaceted challenges has increasingly involved the adoption of RISC-V, which has emerged as the leading open-standard ISA for both industry and academia. A key development within this ecosystem is the RVV extension, which introduces scalable SIMD capabilities that allow embedded devices to accelerate complex mathematical operations in a standardised manner. However, implementing full-scale vector extensions in resource-constrained embedded devices requires a careful balance between architectural complexity and power efficiency.

Within this high-performance computing context, the long-term objective is the implementation of a multi-core vector accelerator. This will be integrated as a tile which, alongside a Network-in-Package (NiP) router, will form a chiplet within a System

in Package (SiP). This architectural integration constitutes the scope of a doctoral research project. The work presented in this Master's thesis focusses on the design and implementation of a baseline vector accelerator, intended to serve as the starting point for the subsequent development of the multicore vector accelerator within SiP. This research is supported by the Indra-UPM Chair in Microelectronics,[1] aligning with the broader industry push toward independent and open-source silicon design.

1.2 Goals of the Thesis

The research in this project is situated within a technological landscape where open-source hardware and modular architectures are increasingly used to meet the demands of data-intensive computing. Based on this framework, the following key objectives have been proposed for this research line to enable parallel processing capabilities within the constraints of the embedded domain:

- Analysis of the RISC-V RVV extension and definition of the target architecture, taking into account state-of-the-art implementations.
- The design and implementation of a vector accelerator in SystemVerilog based on Zve, a subset of the RVV specifically tailored for embedded systems. Compatibility with the Core-V eXtension Interface (X-IF) interface to enable reuse across different system environments.
- Development of the vector accelerator as an Intellectual Property (IP) of the X-HEEP fast-prototyping platform, leveraging its native support for RISC-V and the X-IF interface.
- Validation of the vector accelerator through data-parallel standard benchmarks oriented to embedded systems.

1.3 Technology Framework

This section outlines the primary technological components utilised in this work. It examines the RISC-V ISA specification and its relevant vector extensions, followed by an analysis of the CV-X-IF and OBI interface protocols; these serve as the primary mechanisms for scalar–vector coupling within the proposed architecture. Furthermore, the X-HEEP platform is detailed to provide the necessary context for hardware integration and subsequent software development.

[1] `https://blogs.upm.es/catedrachip/`

1.3.1 RISC-V Instruction Set Architecture

The RISC-V ISA represents the fundamental interface between hardware and software,[2] dictating how software instructions are executed by the underlying microarchitecture. The first RISC-V specification was released in 2011 by researchers at UC Berkeley [1], and since then, RISC-V has facilitated a paradigm shift in the semiconductor industry by providing a modular, open-source, and royalty-free alternative to proprietary ISAs. Unlike traditional architectures, the open-standard nature of RISC-V allows any entity to leverage it as a fundamental building block for both open-source and commercial solutions without the constraints of restrictive licencing fees.

A defining characteristic of RISC-V is its modularity. It is structured around a minimal fixed base ISA, which can be augmented with various standard extensions ratified by the RISC-V Foundation or custom application-specific extensions developed by individual designers to tailor the processor for specialised workloads [2]. Common standard extensions include:

- **M (Integer Multiplication and Division):** Essential for general-purpose computational tasks.
- **F/D (Single and Double-Precision Floating Point):** Required for high-precision scientific and engineering calculations.
- **C (Compressed Instructions):** Designed to reduce the code footprint, which is vital in resource-constrained embedded systems.

Beyond these, the ISA continues to evolve to meet the requirements of modern data-intensive workloads. A significant milestone in this evolution was the formal ratification of the Vector (V) extension, which provides a standardised and scalable framework for high-performance data parallelism. Furthermore, other specialised extensions, such as the Matrix extension, are currently under development to further accelerate emerging applications in AI and ML.

The governance and evolution of the architecture are managed by RISC-V International (formerly the RISC-V Foundation, established in 2015). This global organisation oversees a robust community of members who contribute to the continuous development of the ISA specifications and the broader software ecosystem. The widespread interest in RISC-V stems from its status as a truly global open standard. This standard ensures that software remains portable across diverse hardware implementations, empowering designers to develop bespoke hardware that remains compatible with a unified software stack.

[2] `https://riscv.org/about/`

RISC-V Vector Extension (RVV)

In RVV, instructions are integrated into the standard 32-bit instruction space [1], ensuring compatibility with the base RV32I ISA encoding constraints. This extension significantly expands the ISA by introducing over 200 instructions. To support these operations, the architectural state is augmented with 32 dedicated vector registers. Furthermore, the extension introduces seven specific Control and Status Registers (CSRs): `vl`, `vlenb`, `vtype`, `vxsat`, `vxrm`, `vstart`, `vcsr`.

Table 1.1 outlines the fundamental parameters of the RVV categorising them by their source of definition [1].

Table 1.1: Main Characteristics and Parameters of the RVV.

Parameter	Type	Description
EEW	Specification	**Effective Element Width**. The required width for specific instruction operands.
SEW	Runtime	**Selected Element Width**. Defines the size of the elements currently being processed.
VLEN	Design-time	**Vector Register Length.** The total bit-length of a single vector register.
ELEN	Specification	**Element Lenght.** The maximum Effective Element Width (EEW) supported.
LMUL	Runtime	**Vector Register Grouping**. Allows multiple registers to be treated as a single unit (when LMUL $>$ 1) or fractions (when LMUL $<$ 1) for optimal resource usage.
VL	Runtime	**Vector Length**. Specifies the number of elements to be processed in the current loop or instruction. Adopting values of 0 to Maximum Number of Elements (VLMAX).

Considering the main parameters described in Table 1.1, the relationship between the Vector Register Grouping (LMUL), the Vector Register Length (VLEN), and the Selected Element Width (SEW) determines the Maximum Number of Elements (VLMAX) defined in the specification, calculated as:

$$VLMAX = \frac{LMUL \times VLEN}{SEW} \tag{1.1}$$

This mathematical relationship allows the Vector Register File (VRF) to be dynamically reconfigured, providing a flexible interpretation of the stored data based on the application's requirements, as illustrated in Figure 1.1.

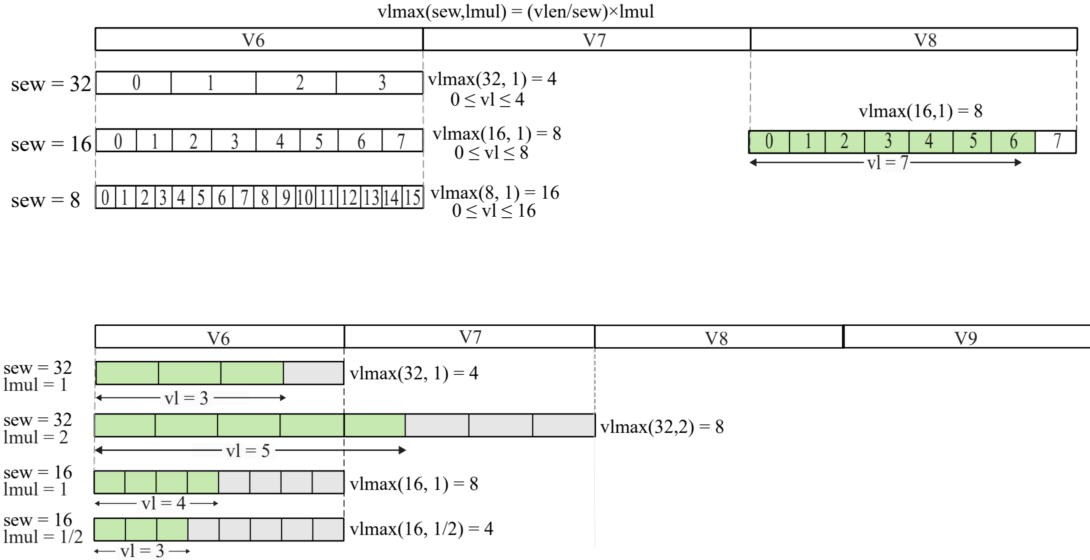

Figure 1.1: Element mapping within vector registers as a function of LMUL, SEW, and Vector Length (VL) being VLEN = 128.

CSRs Configuration and Management. As mentioned, the RVV extension defines seven dedicated CSRs essential for the configuration and operation of the vector processor. These registers define the state and data-handling characteristics:

- **vl**: Defines the number of elements to be updated with results during a vector operation.
- **vlenb:** Indicates the hardware Vector Register Length (VLEN) in bytes.
- **vtype:** Encodes the current settings for the vector unit, including element width (SEW) and register grouping (LMUL). Furthermore, it manages the Vector Tail Agnostic (vta) and Vector Mask Agnostic (vma) policies, which dictate how the hardware handles inactive elements.
- **vxrm and vxsat:** Manage fixed-point rounding modes and saturation flags, respectively.
- **vcsr:** A combined register providing access to both vxsat and vxrm.
- **vstart:** Specifies the index of the first element to be executed, facilitating the continuation of interrupted operations.

The vset{i}vl{i} Instruction Family. The `vset{i}vl{i}` instruction set serves as the primary mechanism for modifying the vector CSRs. It dynamically configures key execution parameters, including the SEW and the LMUL, as well as the vta and vma policies.

Vector Length Negotiation and Stripmining. A critical function of the `vset{i}vl{i}` instruction is the negotiation between the Application Vector Length

(AVL) and the hardware-supported vector length. The instruction returns a granted VL value to the Central Processing Unit (CPU), which may be less than the AVL requested by the application due to hardware constraints or the LMUL configuration.

This feedback loop enables stripmining, a fundamental concept in RVV architecture. In this paradigm, the CPU calculates the remaining workload by comparing the AVL with the returned VL. The loop continues until all elements have been processed. By increasing the LMUL value, the accelerator can process more data per iteration, thereby reducing the overhead of loop control. If an unsupported configuration is requested, the hardware asserts the Vector Instruction Length Illegal (`vill`) bit in `vtype` CSR to notify the system of an invalid state.

Source Operators. According to the specification [1], encoded parameters on instructions are adjusted based on the instruction type. Instructions are classified into three primary categories based on their major opcode ([6:0] bits of the instruction):

- **Load instructions:** For memory-to-VRF data movement.
- **Store instructions:** For VRF-to-memory data movement.
- **Arithmetic instructions:** Arithmetic operations and configuration of certain CSRs, which are discussed in subsequent sections.

Source operands depend on the instruction type (e.g., arithmetic versus load-store instructions) and the nature of the data. The following sources are utilised as operands:

- **Vector Source (vs):** The instruction provides the vector address required to read data from the VRF.
- **Scalar Source (rs):** Data contained on a 32-bit scalar register within the CPU.
- **Immediate (imm):** A 5-bit immediate value encoded directly within the instruction.

Arithmetic Instruction Classifications. Bits [14:12] of the instruction define the combination of source operands in arithmetic operation. Based on these bits, instructions are divided into three distinct types. The only exception is configuration instructions ([14:12] = 3'b111), which do not require a vector. These categories establish the standard naming conventions for the instructions.

- **Vector-Vector (vv):** Utilises two vector sources.
- **Vector-Scalar (vx):** Utilises one vector operand and one scalar operand.
- **Vector-Immediate (vi):** Utilises one vector operand and one immediate operand.

For these instruction types, bits [31:26] of the instruction encode the specific arithmetic operation to be performed.

For most operations, the final result is stored in the VRF. In the case of `vx` or `vi` instructions, before the operation, the scalar or immediate operand must be replicated across the entire VL, as illustrated in Figure 1.2. This replication is performed according to the SEW and the current VL, ensuring the single scalar value operates on every element within the vector.

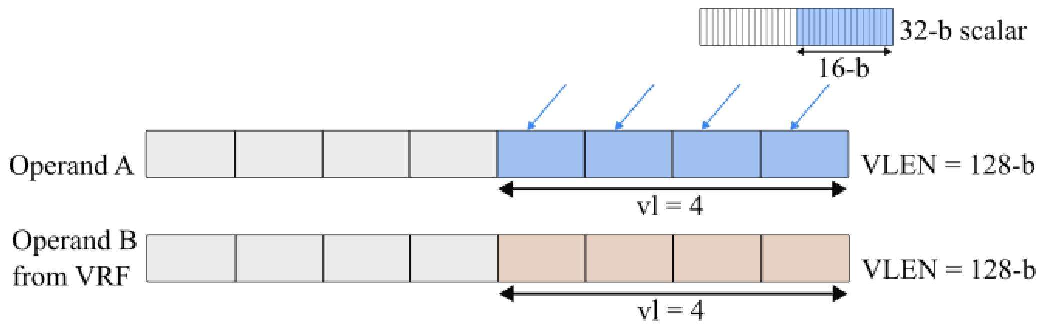

Figure 1.2: Replication of scalar/immediate operator over the input vector. Example of `vx` instruction: SEW = 16, VL = 4.

Further details regarding vector instruction formats and their respective encoded parameters are presented in section A.1.

Vector Element Partitioning and Policies. In accordance with the RVV specification, a vector register is partitioned into three distinct segments during instruction execution: the *prestart*, the *body*, and the *tail*. This partitioning, illustrated in Figure 1.3, is determined by the values held in the `vstart` and `vl` CSRs, as well as VLMAX.

The segments are defined by the element index (i) as follows:

- **Prestart (i < `vstart`):** Elements with an index lower than the value in the `vstart` CSR. These elements are never updated by the instruction; they remain undisturbed.

- **Body (`vstart` $\leq i <$ `vl`):** This segment contains the elements to be processed. Within the body, updates depend on the masking bit `vm` of arithmetic instructions. Active elements (those where the mask is enabled) are updated with the results of the operation. Inactive elements (masked-off) are handled according to the **mask policy**.

- **Tail (`vl` $\leq i <$ VLMAX):** Elements with an index between the current vector length and the maximum possible vector length. The handling of these elements is governed by the **tail policy**.

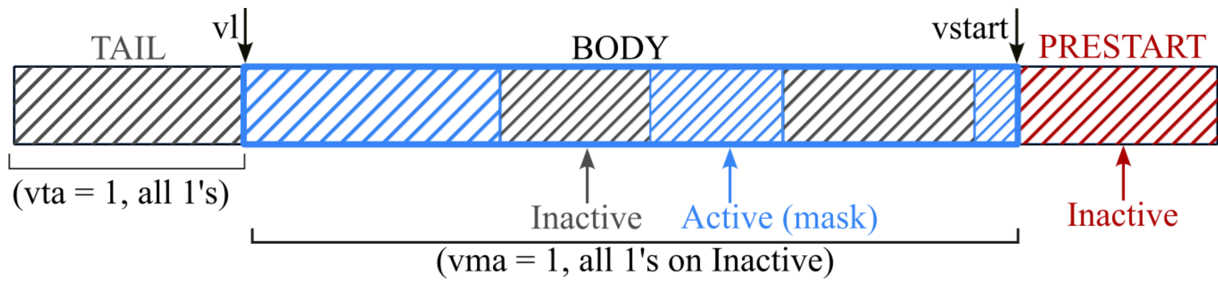

Figure 1.3: Application of tail and mask policies to vector elements.

The RVV specification defines two states in tail and mask policies that dictate the behaviour of inactive and out-of-range elements: undisturbed and agnostic. These are configured via the `vtype` register using the `vta` (tail agnostic) and `vma` (mask agnostic) bits. By default, the state is typically set to agnostic for both. Table 1.2 summarises how these policies handle elements depending on the register state.

Table 1.2: Vector tail and mask policy configurations.

Policy	Behaviour and Implementation
Undisturbed	Inactive elements retain their prior values, ensuring data persistence. Used for *tail undisturbed* (`tu`) and *mask undisturbed* (`mu`).
Agnostic	Inactive elements are treated as “don’t care” and are typically overwritten with 1s. Used for *tail agnostic* (`ta`) when $i \geq vl$, and *mask agnostic* (`ma`) for inactive elements.

Vector Extension Subsets

Within the RISC-V ecosystem, vector extensions are categorised based on the target application domain [1]. While the standard V extension is designed for high-performance application processors, a specific subset of extensions, identified by the “Zve” prefix, is intended for resource-constrained embedded use. These include Zve32x, Zve32f, Zve64x, Zve64f, and Zve64d, where the suffixes denote the supported data types; however, all of them support fixed-point operations: “x” for integer and logic operations, “f” for single-precision floating-point, and “d” for double-precision floating-point.

Complementing these are the “Zvl” extensions, which define the minimum supported vector length (VLEN) for a given hardware implementation. Supported values consist of Zvl32b, Zvl64b, Zvl128b, Zvl256b, Zvl512b and Zvl1024b. In the RVV, the Zvl extension acts as a guarantee to the software toolchain regarding the hardware’s capabilities. This is a critical parameter for the compiler, as it allows for the generation of optimised code that can safely assume a specific data width and distribution for vector registers in each design application. If this extension is omitted, the system defaults to Zvl32b; consequently,

the compiler assumes a 32-bit vector register width, which may result in the hardware's capacity being underutilised if the physical registers are larger.

1.3.2 Core-V eXtension Interface (CV-X-IF)

The CV-X-IF [3] is a standardised protocol designed to facilitate the expansion of a CPU through the integration of external coprocessors. It enables the execution of both standard RISC-V extensions, such as M (Integer Multiplication and Division), F/D (Single and Double-Precision Floating Point), and V (Vector), as well as user-defined custom instructions.

The primary architectural objective of the CV-X-IF is to enable the design, implementation, and verification of unprivileged instruction extensions in a modular fashion. By utilising this standardised interface, hardware designers can extend CPU functionality without necessitating intrusive modifications to the base CPU Register Transfer Level (RTL) code. This decoupling ensures that coprocessor IP can be seamlessly re-used across diverse RISC-V environments, significantly reducing verification overhead.

The development and industrialisation of the CV-X-IF are currently supported by the TRISTAN project.[3] This consortium focuses on expanding the European RISC-V ecosystem with the explicit goal of establishing a mature, competitive, and open-source alternative to existing proprietary commercial architectures. By fostering a robust ecosystem of interoperable IPs, the TRISTAN project aims to strengthen European technological sovereignty in the semiconductor domain.

Originating from the PULP Project via ETH Zurich and the University of Bologna,[4] the CV-X-IF was initially conceived to decouple the Floating-Point Unit (FPU) from the primary CPU pipeline, first manifesting as the "Auxiliary Processing Unit (APU) Interface" within the CV32E40P core. Although this early iteration established the groundwork for functional offloading, it was hampered by rigid, tight coupling that necessitated intrusive modifications to the CPU decoder and pipeline for any new extensions; however, subsequent architectural refinements through the CVA6 [4], ARA [5], and Snitch projects [6] transitioned the interface toward a modular, pipeline-agnostic framework with minimal integration overhead. Now formalised by the OpenHW Foundation, the CV-X-IF serves as a standardised, vendor-neutral protocol that eliminates structural dependencies between the host core and coprocessors, natively featuring in the CV32E40X and offered as a configurable option for the CVA6 to facilitate seamless RISC-V IP reuse across the industry.

[3] `https://tristan-project.eu/`
[4] `https://pulp-platform.org/`

Architecture

CV-X-IF is architecturally structured into several modular sub-interfaces, each managing a specific stage of the instruction offloading process. These channels facilitate a decoupled handshake protocol between the host CPU and the coprocessor:

- **Compressed Interface:** Handles the offloading of compressed (16-bit) instructions.
- **Issue Interface:** Manages the initial request and acceptance of uncompressed instructions.
- **Register Interface:** Facilitates the transfer of source operands from the CPU's General-Purpose Registers (GPRs) or CSRs.
- **Commit Interface:** Provides control signals to validate speculative instructions or if they should be killed.
- **Result Interface:** Transmits the final execution results and write-back data to the host.
- **Memory Interface:** Historically used for direct memory access (now deprecated in the core standard).

One of the most significant architectural shifts between the v0.2.0 draft and the ratified v1.0.0 is the fundamental change in how memory-accessing instructions are handled.

In the early v0.2 specification, the protocol was designed to be highly permissive, explicitly supporting custom load/store instructions. To facilitate this, it included a dedicated Memory (Request/Response) Interface. This allowed the external coprocessor to initiate memory transactions that were then routed through the host CPU's Load-Store Unit (LSU). While this offered flexibility for complex accelerators, it introduced significant dependencies on the CPU's internal memory management and consistency models.

With the release of v1.0, the OpenHW Foundation removed support for custom load/store instructions. The interface has subsequently pivoted to a strictly register-based offloading model. This demanded a high throughput protocol that enables data transfer between the coprocessor and the system's memory.

Table 1.3 details the signals included in the issue interface.

Table 1.3: Issue Interface signals [3].

Signal	Direction (CPU)	Description
`issue_valid`	output	Indicates the CPU is ready to offload an instruction.
`issue_ready`	input	A transaction is accepted only when both `issue_ready` and `issue_valid` are high.
`issue_request`	output	Contains the RISC-V instruction to be decoded and processed.
`issue_response`	input	Includes signals that communicates instruction acceptance (`accept`), data return requirements (`writeback`), or requests to read scalar registers (`register_read`).

Table 1.4 specifies the signals that compose the register interface.

Table 1.4: Register Interface signals [3].

Signal	Direction (CPU)	Description
`register_valid`	output	Indicates that CPU provides register contents related to an instruction.
`register_ready`	input	Handshake signal indicating the VPU has accepted the register data.
`register`	output	Groups `registers[1:0]`, which transfers up to two 32-bit data elements from the GPRs, and `rs_valid`, which indicates the validity of the register file source operand(s).

Table 1.5 shows the signals involved in the commit interface.

Table 1.5: Commit Interface signals [3].

Signal	Direction (CPU)	Description
`commit_valid`	output	Indicates that CPU has valid commit or kill information for an offloaded instruction.
`commit`	output	Apart from the instruction `hartid` and the `id`, includes `commit_kill`, that when `commit_valid` and `commit_ready` are 1, it signals the VPU to terminate the instruction.

Table 1.6 presents the signals associated with the result interface.

Table 1.6: Result Interface signals [3].

Signal	Direction (CPU)	Description
`result_valid`	input	Asserted when the coprocessor has completed processing and a valid result is available.
`result_ready`	output	Indicates the CPU is ready to accept the result provided when `result_valid` and `result_ready` are 1.
`result`	input	Contains the 32-bit writeback data, the destination address (`rd`), and the write-enable (`we`) flag for the GPR.

Protocol

Figure 1.4 provides a detailed timing diagram of the CV-X-IF protocol, visualising the synchronisation and handshakes between the issue, register, and result interfaces.

When the CPU encounters an instruction it does not recognise, it offloads the request through the CV-X-IF (`issue_valid`) to determine if a connected coprocessor can handle the task. The coprocessor must then signal its acceptance using the `accept` and `issue_ready` handshakes; if no coprocessor acknowledges the request, the CPU instead raises an illegal instruction exception.

Once the instruction is accepted, the CPU transfers the necessary operands from its register file through the dedicated register interface or via encoded instruction bits. During this phase, the coprocessor also identifies which specific registers in the core's register file it intends to target in case there is writeback. Based on CV-X-IF protocol, to manage speculative execution, the CPU should use a commit interface to inform the coprocessor whether the offloaded instruction should be killed or if it has become non-speculative and is permitted to commit.

The final stage of the process depends on the writeback requirements established during the initial issue phase. If the instruction does not require data to be returned to the CPU, the process concludes as soon as the `result_valid` signal is received, allowing the core to proceed with the program workflow. In contrast, if the CPU is expecting return data, the instruction only finishes when the `result_valid` signal is asserted alongside the appropriate write-enable signals.

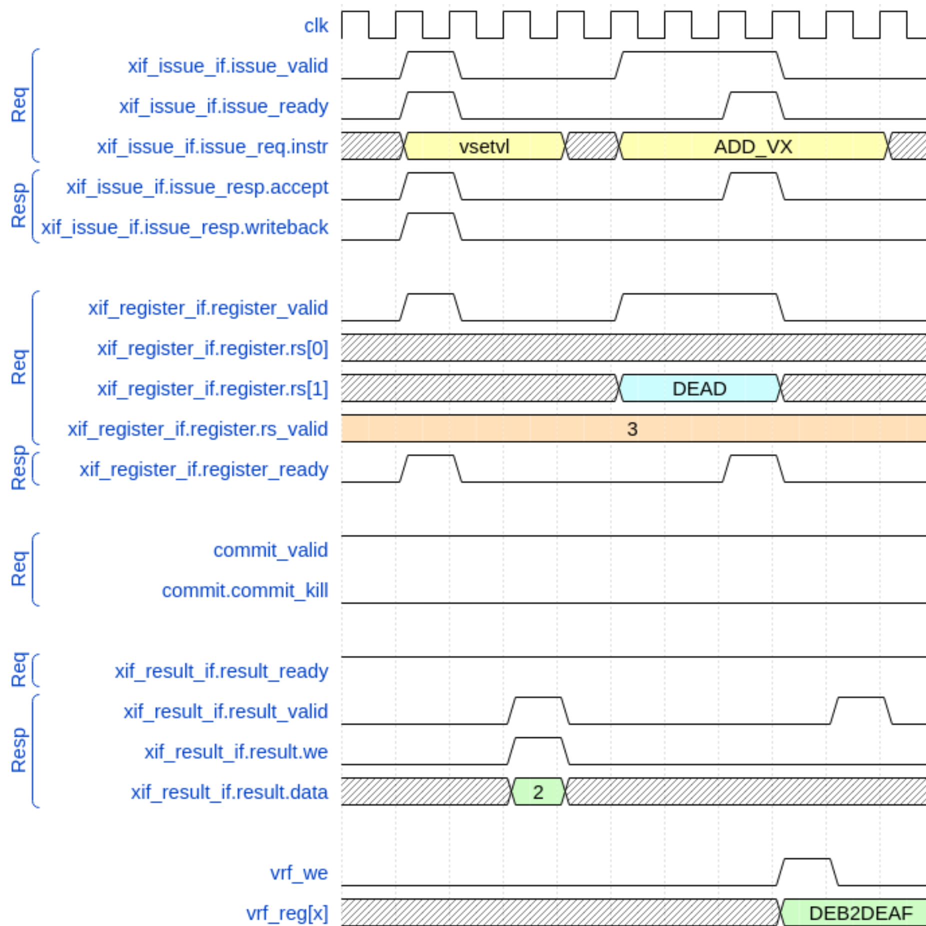

Figure 1.4: CV-X-IF handshake during vector instruction requests indicated by `xif_issue_if.issue_valid`. Request 1: Vector configuration instruction for VLEN = 32 and SEW = 16. Returning VL = 2 to the scalar core. Request 2: `vadd.vx` instruction. The two 16-bit elements contained on the vector register (5 and 2) and the scalar register data (0xDEAD) are summed.

1.3.3 Open Bus Interface (OBI)

OBI [7] is a high-performance, point-to-point protocol designed for connecting CV32E40-series CPUs (originally based on the PULP RI5CY [8] architecture) to bus infrastructure components.

Derived from the custom interfaces used in the RI5CY and Ibex RISC-V cores, OBI has been refined to be more robust and interoperable. It clarifies and extends those earlier designs to simplify the creation of bus interconnects and ease integration with industry-standard Advanced Microcontroller Bus Architecture (AMBA) protocols (Advanced High-performance Bus (AHB) [9] and Advanced eXtensible Interface (AXI) [10]). This standardisation is achieved through a direct mapping of handshaking mechanisms; for instance, the OBI `req` and `gnt` signals correspond directly to the AXI `VALID` and `READY` signals during the address phase. Similarly, the response phase is synchronised using the OBI `rvalid` and `rready` signals, which function as the AXI `VALID`

and `READY` equivalents. By standardising these connections, OBI improves system-level timing and ensures that masters and slaves can communicate reliably.

Furthermore, OBI facilitates multiple channels and parallelism, which significantly boosts bus throughput across the system.

This interface is a synchronous, request-grant based protocol designed for high-efficiency communication between masters and slaves. Its architecture is built upon the following core properties:

- Link Structure: Each physical connection within an OBI crossbar is defined as an OBI link. Each link is decoupled into two distinct, independent channels:
  - **Address Channel (A):** Handles the initiation of requests, including address, control, and write data signals.
  - **Response Channel (R):** Handles the completion of transactions, conveying read data and status information.
- Two-Phase Transaction Model: Every OBI transaction consists of two primary transfers:
  - **The Address Phase:** The master asserts the validity of the address and control signals. A transfer is considered successful only when both the master's valid signal and the slave's grant/ready signal are high.
  - **The Response Phase:** Once a request has been granted, the slave provides the response (and read data, if applicable). The transaction concludes when the master acknowledges the acceptance of this response.
- Decoupled Operation: Because the Address and Response phases are decoupled, the protocol inherently supports multiple outstanding transactions, allowing the system to achieve high parallelism and bus throughput.

The signal definitions and channel descriptions for the OBI transfer protocol are summarised in Table 1.7.

Table 1.7: OBI Port List [7]

Name	Source	Destination	Description
Global signals			
`clk`	Clock source	All	The bus clock times all bus transfers. All signal timings are related to the rising edge of `clk`.
`reset_n`	Reset controller	All	The bus reset signal is active LOW and resets the system and the bus. This is the only active LOW signal.
A Channel signals			
`req`	Master	Slave	Address transfer request. `req=1` signals the availability of valid address phase signals.
`gnt`	Slave	Master	Grant. Ready to accept address transfer. Address transfer is accepted on rising `clk` with `req=1` and `gnt=1`.
`addr`	Master	Slave	Address to access.
`we`	Master	Slave	Write enable, high for writes, low for reads.
`be`	Master	Slave	Byte enable. Set for the bytes to write/read.
`wdata`	Master	Slave	Write data. Only valid for write transactions. Undefined for read transactions.
R Channel signals			
`rvalid`	Slave	Master	Response transfer request. `rvalid=1` signals the availability of valid response phase signals. Used for both reads and writes.
`rdata`	Slave	Master	Read data. Only valid for read transactions. Undefined for write transactions.

The signal timing for OBI transfers is illustrated in Figure 1.5, detailing the decoupling of the *address* and *response phases*. In clock cycle 2, the address phase for a write operation begins; the master asserts both `req` and `we`, providing the target address and the corresponding write data. As the slave asserts `gnt` (signifying it is ready to receive), the address phase transfer is accepted. Subsequently, in cycle 3, the response phase occurs, where the valid data transfer to the memory is acknowledged through `rvalid`.

A subsequent read request begins in cycle 3. The master asserts req together with the target address; since the slave maintains `gnt` at high, the request is registered. The validity of this read operation is confirmed in cycle 4 via the assertion of `rvalid` by the

slave, at which point the requested data is available on the `rdata` bus.

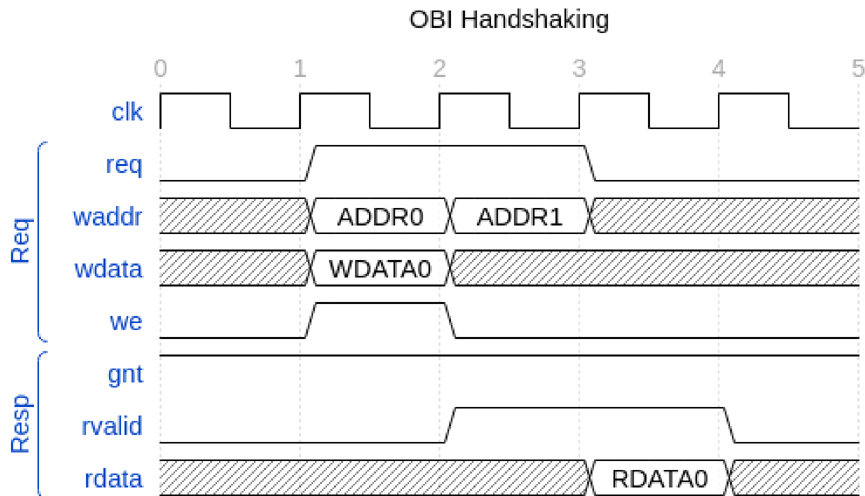

Figure 1.5: OBI handshake protocol timing.

1.3.4 X-HEEP

X-HEEP [11] is an open-source RISC-V (subsection 1.3.1) Microcontroller (MCU) framework described in SystemVerilog. It is specifically engineered to target ultra-low-power embedded systems, offering a highly modular architecture that supports both minimal configurations for small-scale platforms and extensive customisation through the integration of external accelerators.

The platform provides a flexible design space, allowing designers to configure the MCU by selecting specific CPU cores, integrating custom IP blocks, and tailoring the peripheral and memory subsystems to meet specific application requirements. The architecture integrates validated components from leading open-source initiatives, including the PULP Platform,[5] OpenHW Foundation,[6] and lowRISC.[7] X-HEEP supports a robust verification and simulation flow across multiple industry-standard tools, including Verilator and Siemens Questa. Furthermore, the architecture is silicon-proven and suitable for implementation on both FPGAs and Application-Specific Integrated Circuits (ASICs).

The internal architecture, illustrated in Figure 1.6, is composed of three primary pillars:

[5] `https://pulp-platform.org/`
[6] `https://openhwfoundation.org/projects/`
[7] `https://lowrisc.org/`

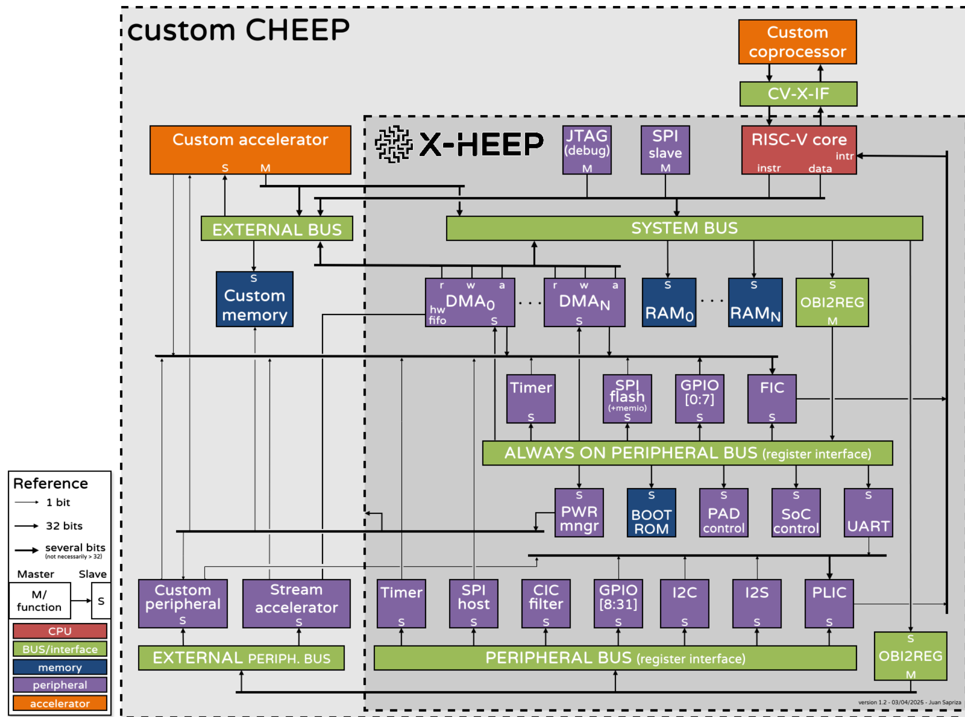

Figure 1.6: Internal architectural block diagram of the X-HEEP platform [11].

- **CPU Subsystem:** Based on the OpenHW Foundation RISC-V ecosystem, the platform allows for the selection of various 32-bit low-power cores, such as the CV32E20 or the CV32E40P. For designs requiring hardware acceleration, the CV32E40PX and CV32E40X variants are supported, as they incorporate the X-IF. These cores utilise a Harvard architecture with independent instruction and data buses adhering to the OBI protocol (subsection 1.3.3). The use of on-chip Static Random Access Memorys (SRAMs) ensures high-speed data availability, removing the necessity for a cache layer to speed up external memory access.

- **Memory Subsystem:** The architecture features a multi-banked configuration. These banks are parameterisable in terms of size and quantity, serving as unified storage for both instructions and data. The multi-banked approach facilitates concurrent access to different memory regions, significantly reducing bus contention and enhancing overall system throughput.

- **Peripheral Subsystem:** Peripheral management is divided into two distinct domains to optimise power consumption. The "Standard Peripheral domain" contains non-critical utilities such as timers, I2C and SPI interfaces, which may be clock-gated when idle. In contrast, the "Always-On (AON) domain" incorporates mission-critical components, including the SoC Controller, boot ROM, DMA controller, and UART, ensuring the system remains responsive to wake-up events and maintains essential orchestration logic. Furthermore, the platform leverages the X-IF protocol to facilitate modular CPU expansion, while employing the OBI to manage robust memory transactions.

Chapter 2

State of the Art

This chapter provides an analysis of existing state-of-the-art vector architectures to contextualise the design and identify the prevailing trends within the RISC-V ecosystem.

2.1 RISC-V Based Vector Processors

To provide a conceptual framework for the following state-of-the-art analysis, this section establishes the necessary architectural context. The discussion begins by the categorisation of accelerator types to distinguish between various execution models. Following this classification, specific implementations from recent literature are reviewed, concluding with a comparative synthesis of their primary characteristics.

Array Processors

As an alternative to parallelising tasks across multiple cores or threads, SIMD arrays have been integrated into various ISAs. Execution pattern of this type of processor is illustrated in Figure 2.1. These typically consist of fixed-size arrays that utilise dedicated functional units for each array element, allowing a single operation to be applied across the entire dataset simultaneously. A defining characteristic of array processors is the requirement for computational resources to be replicated for every element of the maximum supported array length [5].

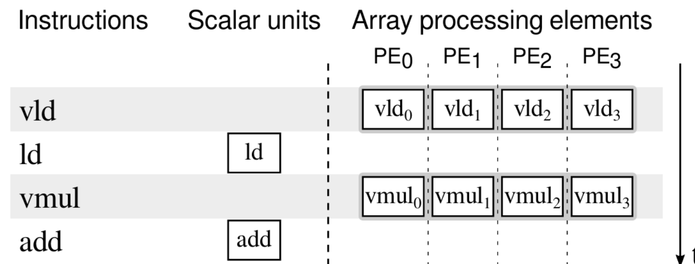

Figure 2.1: Execution pattern of an array processor [12].

Vector Processors

Vector processors represent a time-multiplexed evolution of array processors. They utilise functional units that process elements of the same vector over multiple clock cycles. This architectural approach enables greater flexibility, allowing for the dynamic configuration of vector lengths and more efficient hardware utilisation [5]. The architecture of a vector processor can also distribute computational tasks across multiple functional units, enabling them to operate in parallel with one another and concurrently with the host scalar processor [12]. By utilising a combination of spatial hardware replication and time-multiplexing to optimise performance, computational throughput is enhanced.

Three fundamental characteristics of this execution model are observable in Figure 2.2. Firstly, the design can incorporate spatial replication to increase the number of elements processed per cycle, thereby integrating array-style parallelism within the vector paradigm to further scale performance. Secondly, vector elements are processed sequentially when hardware resources are finite, allowing a predefined number of functional units to handle large vectors over time. Finally, the scalar unit may proceed with independent instructions while the vector unit completes long-latency operations.

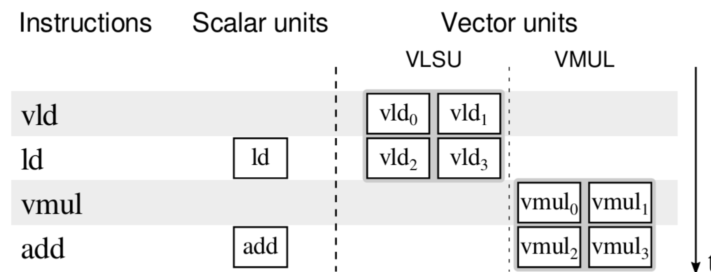

Figure 2.2: Execution pattern of an vector processor [12].

These are some RISC-V vector processors found in the state of the art:

2.1.1 Ara

Ara [5] is a 64-bit vector processor designed to handle operands up to 64 bits in width, compliant with version 0.7 of the RISC-V Vector specification. It operates as a tightly coupled coprocessor to the open-source Ariane (RV64GC) scalar core.

The architecture is built around scalable and replicable lanes as depicted in Figure 2.3. Each lane features a 64-bit datapath containing an Arithmetic Logic Unit (ALU), a FPU, and a Multiplier (MUL). To optimise throughput, each datapath can be subdivided to support various element widths, specifically: 1 × 64-bit, 2 × 32-bit, 4 × 16-bit, or 8 × 8-bit operations. Internally, each lane's sequencer can manage up to eight parallel instructions.

Communication between the Ariane core and Ara is managed via an interface similar

to the Rocket Custom Coprocessor Interface (RoCC); however, a key distinction exists in the division of labour between the units. While a standard RoCC implementation delegates the entirety of the decoding task to the coprocessor, the Ariane-Ara interface utilises a dispatcher to push pre-decoded instructions directly to the accelerator. Data transfers, meanwhile, are handled through AXI protocol.

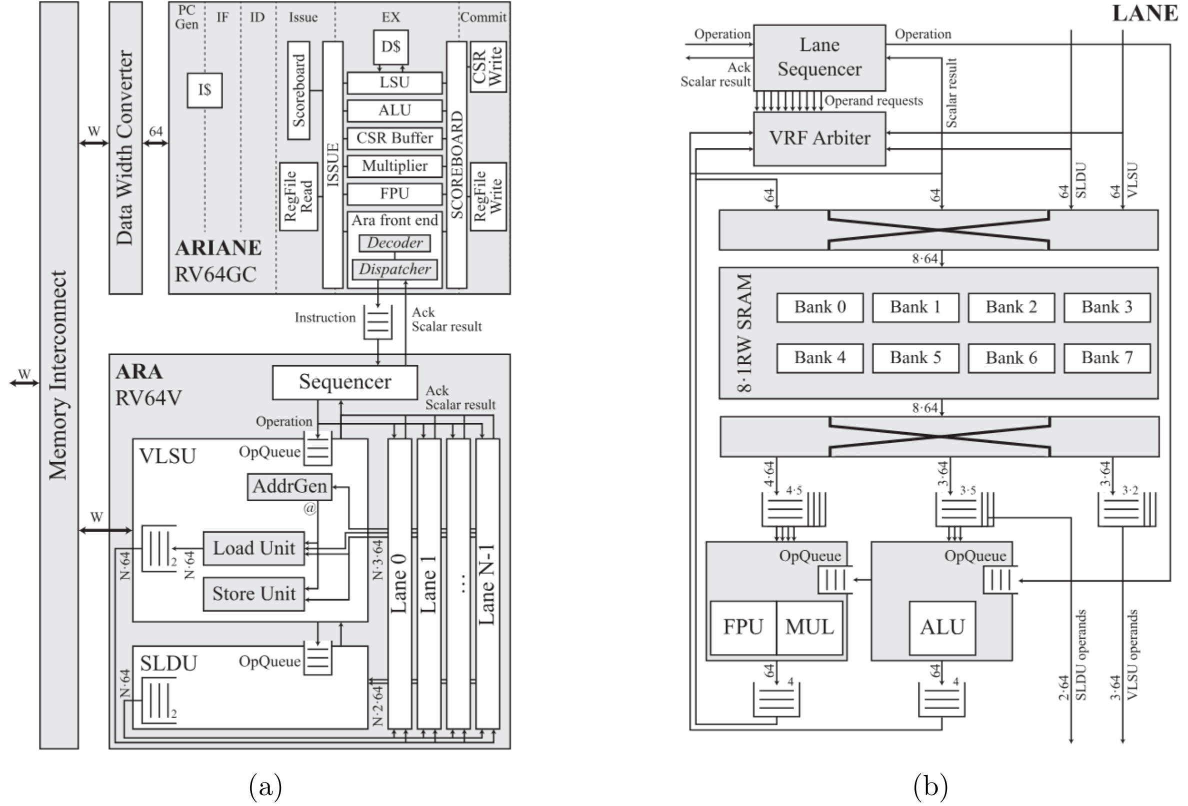

Figure 2.3: Top-level block diagram of Ara [5]. (a) Block diagram of an Ara instance with N parallel lanes. (b) Block diagram of one lane of Ara.

2.1.2 Ara XL

Ara XL [13] is a 64-bit vector processor that advances the original design by adhering to version 1.0 of the RISC-V Vector specification. Unlike the standard Ara, the XL variant is a cluster-based coprocessor (see Figure 2.4), where each cluster is based on the Ara2 architecture, all of which are interfaced with a central CVA6 open-source scalar core.

The system is highly scalable for massive data processing, supporting up to 64 parallel vector lanes and providing full support for double-precision floating-point elements. Communication between the clusters and the CVA6 core is facilitated by three primary interfaces:

- Request Interface (REQI): Manages synchronisation across the vector processors.
- Global Load Store Unit (GLSU): Facilitates memory-to-Vector VRF operations via an AXI interface.

- Ring Interface (RINGI): Enables efficient data movement between different clusters.

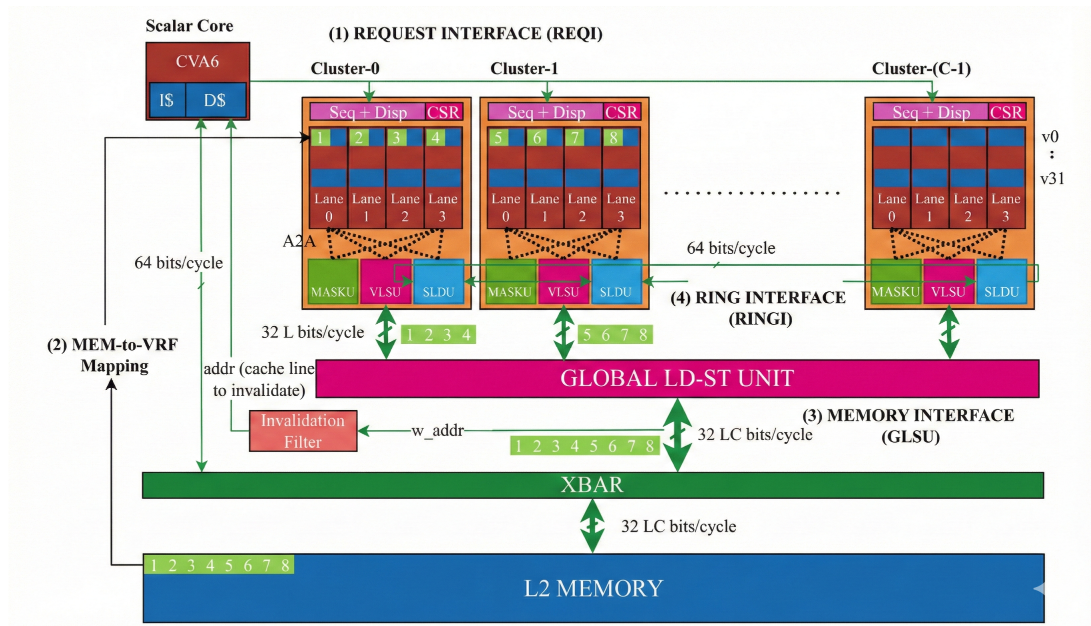

Figure 2.4: Top-level block of Ara XL and its interfaces [13].

2.1.3 Implementation by Johns et al.

The architecture developed by University of Southampton is built upon the RISC-V Specification v0.9, featuring vector functionality that is natively integrated into the processor, as illustrated in Figure 2.5 [14]. By embedding these capabilities directly into the core, the design avoids the inherent latency and communication overhead typically found in traditional interfaces between a main scalar core and a separate coprocessor. This tightly coupled approach ensures more efficient data handling and streamlined execution of vector instructions.

To further optimise hardware efficiency, the processor utilises the RV32E specification. This is a variant of the RV32I base instruction set that reduces the scalar register count from 32 to 16. This specific configuration is employed to minimise silicon area and avoid the inclusion of unused registers within the vector register file, ensuring that the hardware remains lean. Furthermore, the design focuses on enhanced scalar processing to boost performance while keeping overall system complexity to a minimum.

Data processing versatility is achieved through a flexible execution stage capable of handling 8-bit, 16-bit, and 32-bit elements. The hardware layout consists of a specialised array of Processing Unit (PU), comprising one 32-bit PU, one 16-bit PU, and two 8-bit PUs. This arrangement allows the processor to adapt its throughput based on the precision required by the specific computational task, maximising the utility of the available arithmetic logic.

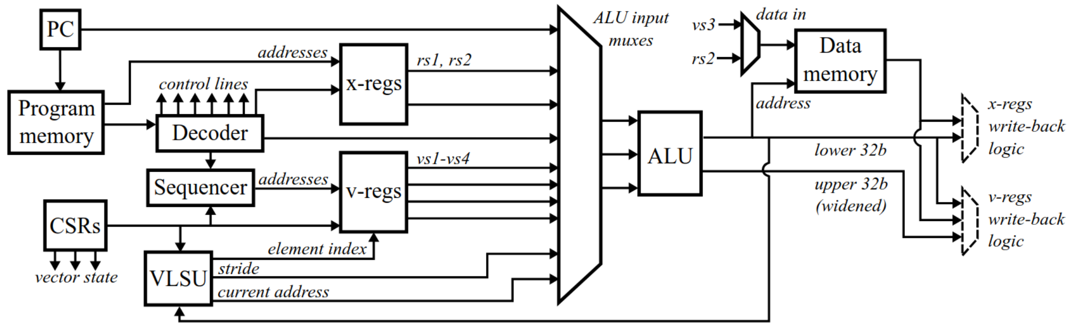

Figure 2.5: Schematic of the integrated vector processor [14].

2.1.4 Vicuna

Vicuna [12] is a 32-bit vector coprocessor based on version 1.0 of the RISC-V Vector specification. Its primary design objectives are timing predictability and hardware simplicity, making it highly suitable for real-time applications. As illustrated in Figure 2.6, the system is designed to interface with the Ibex open-source scalar core (formerly known as Zero-riscy),[1] providing a scalable solution for high-performance embedded tasks.

Connectivity between the scalar core and the vector unit is established through the X-IF v0.2 [15]. By utilising this standardised interface rather than a custom-built solution, Vicuna achieves a more modular and interoperable design. This allows for streamlined instruction offloading and efficient communication between the Ibex core and the coprocessor without the overhead of bespoke logic.

In terms of instruction support, Vicuna focuses specifically on integer and fixed-point operations to maintain its goal of hardware simplicity. Despite this targeted focus, it remains a versatile tool for data processing by supporting essential vector capabilities, including masking, permutations, and reductions. This ensures that the processor can handle complex data manipulation while remaining optimised for silicon area and energy efficiency.

2.1.5 Spatz

The Snitch scalar core and the Spatz vector processor [16] constitute the two distinct computational units within the open-source and scalable Core Cluster (CC) illustrated in Figure 2.7a. Both, Snitch and Spatz, are compliant with the RISC-V 1.0 specification. The Snitch unit functions as a small scalar cluster, configurable to support either the RV32I or RV32E integer base instruction sets, while handling control flow and coordination.

[1] `https://pulp-platform.org/`

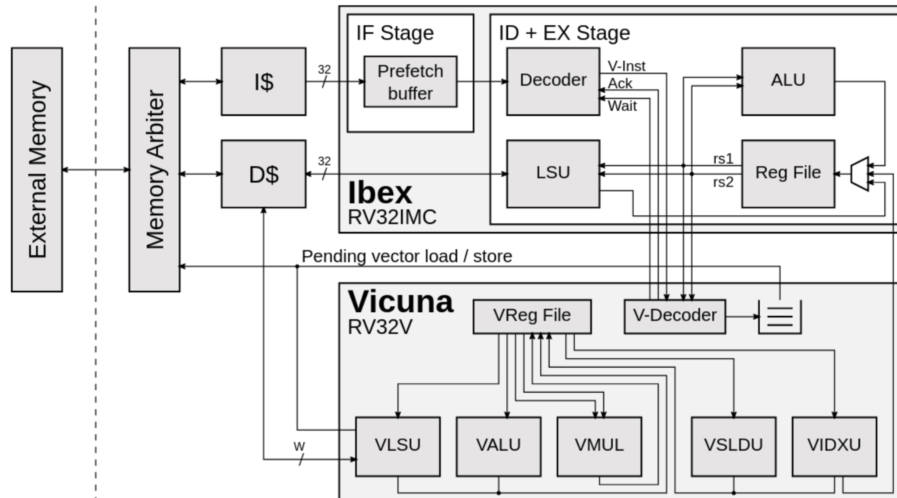

Figure 2.6: Overview of Vicuna architecture integrated with Ibex core [12].

Communication between the scalar Snitch core and the Spatz vector accelerator is facilitated by the X-IF v1.0. System memory access is managed via the AXI interconnect protocol.

The Spatz processor has been developed in iterative versions to address varying computational requirements:

- **Initial Configuration (Zve32x):** The first iteration of Spatz was designed for integer-only arithmetic with no native floating-point support. This configuration implements the Zve32x extension, enabling vector operations for 8, 16, and 32-bit integer elements.

- **Advanced Configuration (Zve64d):** A subsequent version was developed to widen processing capabilities by implementing the Zve64d extension. This enhancement enables the PUs to drive larger functional units, supporting double-precision floating-point computation. The Zve64d extension standardises support for 8, 16, 32, and 64-bit integer and floating-point elements.

(a)

(b)

Figure 2.7: Schematic of the Spatz CC [16]. (a) Shared L1-Cluster with two PUs. (b) Spatz microarchitecture.

2.2 Comparison of VPUs

Table 2.1 presents a comparative overview of VPUs analysed in the state of the art, alongside the intended architectural characteristics of the proposed vector accelerator developed in this master's thesis. The comparison highlights key design aspects, including supported RVV, core integration mechanisms, vector length configurations, and floating-point support, thereby positioning the proposed design within the existing research landscape.

The selection of the accelerator's communication protocols in this work was primarily guided by the project's open-source design philosophy. In line with the Vicuna and Spatz processors, CPU extension is implemented via the X-IF interface (subsection 1.3.2). However, in contrast to the majority of vector processors found in the literature, the VPU developed in this thesis utilises the OBI for data management (subsection 1.3.3), rather than the more prevalent AXI protocol. OBI provides a more simplified, open-source alternative to the complexity of the AXI standard, while additionally supporting burst transactions required for efficient Direct Memory Access (DMA) operations. Unlike other high-performance accelerators that rely on complex cores, the CV32E20 was selected for this work; it does not implement floating-point units and employs a streamlined two-stage pipeline, in contrast to the six-stage architecture found in designs such as the CVA6/Ariane. This selection facilitates a more area-efficient and lightweight implementation, aligning with the project's focus on resource-constrained environments.

Table 2.1: Comparison of state-of-the-art vector processors and the proposed Vector Processing Unit design.

VPUs from the state of the art						
Characteristics	**Ara**	**Ara XL**	**Min. RVE32**	**Vicuna**	**Spatz**	**This work**
RVV Specification Version	0.7	1.0	0.9	1.0	0.7	1.0
Scalar Core	Ariane	CVA6	Own RV32E	CVE2	Snitch	CV32E20
Vector Extension	Zve64d*	Zve64d	Zve32x*	Zve32x	Zve64d	Zve32x
Communication with Core	RoCC AXI	REQI RINGI AXI	–	X-IF 0.2	X-IF 1.0 AXI	X-IF 1.0 OBI
SEW	8, 16, 32, 64	8, 16, 32, 64	8, 16, 32	8, 16, 32	8, 16, 32, 64	8, 16, 32
VLENB	4–128	8–8192	4	16–256	8–64	16
FPU	✓	✓	×	×	✓	×

*Although not explicitly designated as such, the implementation conforms to the RISC-V v1.0 Zve64d/Zve32x specification.

Chapter 3

Hardware Development

3.1 Methodology

The hardware architecture was developed using the SystemVerilog Hardware Description Language (HDL).

3.2 Vector Accelerator Interfaces

This section describes the vector accelerator's general architecture and the interfacing mechanisms required to facilitate its operation alongside the scalar processor.

3.2.1 The X-Fusion Integration Platform

To facilitate the composition of the vector accelerator, the X-Fusion platform is employed.[1] X-Fusion serves as a scalable infrastructure layer designed to simplify the integration of custom hardware into the X-HEEP [11] ecosystem, as depicted in Figure 6.2.

The platform provides several critical advantages for system designers:

- **Decoupled Interconnection:** It simplifies the interfacing of peripherals and accelerators with X-HEEP, removing the requirement for direct modifications to the X-HEEP RTL or core logic.
- **Scalability and Modular Design:** The structure is scalable, allowing for the addition of multiple processing elements or accelerators within a unified framework.
- **Parameterisable Framework:** Key architectural parameters are highly configurable, including memory hierarchies, address space dimensions, and expansion bus capabilities, allowing the system to be tailored to specific application requirements.

[1] `https://github.com/x-heep/x-fusion`

Figure 3.1: Vector accelerator and X-HEEP integration in X-Fusion.

As illustrated in Figure 3.1, this architecture integrates the CV32E20 as the scalar core within X-HEEP.[2] It is a compact 32-bit, area-optimised, two-stage RISC-V scalar processor. The CV32E20 core was proposed for this work primarily for its simplicity and because, despite being under active development, it incorporates the X-IF 1.0 interface. Within this framework, the scalar processor manages the control flow, while the VPU represents the primary contribution of this work, engineered specifically to accelerate data-parallel workloads.

3.2.2 Core-V eXtension Interface

The X-IF serves as the interface to manage the control flow and instruction offloading from the scalar core. The CV-X-IF interface was implemented in accordance with the official specification [3]; an overview of this protocol is provided in subsection 1.3.2. This vector accelerator developed in this work is based on version v1.0.0-rc.2. To ensure compliance with the X-IF 1.0 specification, the X-HEEP platform was patched to remove the memory interface, which is no longer supported in this specific version of the protocol.

Regarding the X-IF implementation within this accelerator, it should be noted that the `commit interface` logic is not fully implemented on the CV32E20. Consequently, the `commit_valid` signal is hardwired to 1 and `commit_kill` to 0. While the instruction `hartid` and `id` remain undriven in the current iteration, subsequent updates to the commit stage have integrated these identifiers, though the validation and kill signals remain statically assigned.

[2] `https://pulp-platform.org/`

3.2.3 OBI

The OBI (v1.0) serves as a high-performance communication interconnect, facilitating efficient data exchange between master and slave components as detailed in subsection 1.3.3 [7]. Within the X-Fusion implementation, the OBI enables DMA capabilities, allowing the coprocessor to perform autonomous data fetches independently of the instruction offloading managed via the X-IF.

The protocol is configured as a point-to-point interface between the coprocessor, acting as the master, and the X-HEEP memory subsystem, acting as the slave. Furthermore, the OBI interface is the native interconnect used throughout the X-HEEP platform; for instance, the scalar core itself functions as an OBI master to access the system memory. Currently, the vector accelerator utilises a single OBI link for all load–store operations; these interconnects are illustrated in Figure 3.1.

3.3 Accelerator Core

This section details the architectural design of the vector accelerator developed in this work. It outlines the key characteristics of the system and provides a comprehensive description of the primary components engineered to facilitate vector processing.

3.3.1 Main Characteristics

The vector accelerator core or Vector Processing Unit (VPU) presented in this thesis is developed in compliance with the RISC-V Zve32x subextension [1]. Zve32x supports integer and fixed-point operations but excludes hardware-based floating-point vector support to maintain a minimal silicon footprint. Though, the vector accelerator developed in this work only targets integer operations.

Furthermore, the design incorporates the Zvl128b extension meaning that the hardware supports a minimum vector length of 128 bits in the vector accelerator's VRF, which is exactly the VLEN defined on this design. The implementation of this extension enables the compiler determine the available hardware resources, allowing it to optimise the number of elements processed and their distribution and limitations in the register file for this specific VRF. This is a critical parameter for the compiler, as it allows for the generation of optimised code that can safely assume a specific data width for vector registers.

Table 3.1 details the architectural parameters mandated by the Zve32x sub-extension, the design-time constants selected for this specific implementation, and the dynamic parameters configured by the user at runtime.

Table 3.1: Main Characteristics the vector accelerator.

Parameter	Type	Value(s)
VRF	Specification	v0-v31
EEW	Specification	8, 16, 32 bits
SEW	Runtime	8, 16, 32 bits
VLEN	Design-time	128 bits
ELEN	Specification	32 bits
LMUL	Runtime	1/4, 1/2, 1
VL	Runtime	0 to VLMAX

3.3.2 Proposed VPU Architecture

This section provides a detailed technical overview of the primary modules constituting the vector accelerator, focusing on their architecture, functional responsibilities, and integration within the wider system. Figure 3.1 serves as a visual guide to the system's organisation.

Control Unit and Instruction Decoder

The Control Unit serves as the central orchestrator for the vector processing unit, managing the execution flow and interfacing with the host CPU.

Finite State Machine (FSM) and Orchestration. The core logic of the accelerator is governed by a FSM (see Figure 3.2) comprising two primary states: *Idle* and *Busy*. The orchestration process follows a specific handshake protocol via the X-IF.

1. **Idle State:** The unit awaits a request from the CPU. Upon receiving a valid instruction, the unit asserts `issue_ready`.

2. **Busy State:** Once the instruction is accepted, the FSM transitions to the Busy state. During this phase, the instruction is decoded and executed.

3. **Completion:** The result is processed and the `result_valid` signal is returned to the core in the final cycle, after which the FSM returns to Idle.

For instance, a standard arithmetic operation typically completes this cycle over three clock cycles.

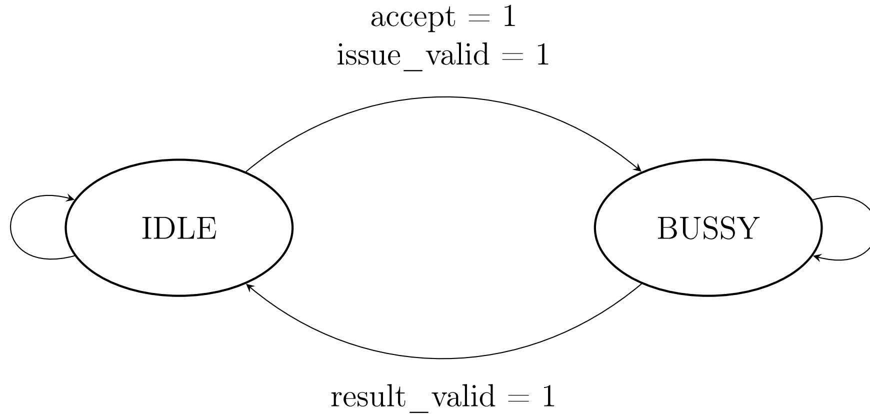

Figure 3.2: VPU Finite State Machine.

Instruction Dictionary. The Instruction Dictionary is a structural component containing the definitions for all supported vector instructions (shown in section A.2). When an instruction request is received, it is compared against the instruction list; a successful match triggers an acceptance signal via the X-IF interface. Once identified, the operation's requirements are determined by the dictionary; this includes whether a result writeback is necessary via the `register interface` and, based on the `register_read` status, if operands must be fetched through the `register interface`. This example demonstrates the encoding for an 8-bit unit-stride load (`vle8.v`) followed by a vector-vector addition (`vadd.vv`).

```
...
'{
    // vle8.v vd, (rs1)
    instr: 32'b000_0_00_0_00000_00000_000_00000_0000111,
    mask:  32'b111_1_11_0_11111_00000_111_00000_1111111,
    resp : '{accept : 1'b1, writeback : 1'b0, register_read : {1'b1,1'b1}},
    opcode : VLE8
},
'{
    //vadd.vv vd, vs2, vs1, vm
    instr: 32'b0000000_00000_00000_000_00000_1010111,
    mask:  32'b1111110_00000_00000_111_00000_1111111,
    resp : '{accept : 1'b1, writeback : 1'b0, register_read : {1'b1,1'b1}},
    opcode : VADD_VV
},
...
```

Source Operators and Decoding Logic. The control unit decodes instructions according to the RVV specification, following the key features detailed in section 1.3.1. The logic first extracts general parameters, such as the major opcode, to categorise the operation as a load-fp, store-fp, or arithmetic instruction. Details of the instruction parameters can be found in section A.1.

For load–store operations, the specific transfer type is decoded from these fields, and the base memory address is retrieved from the scalar register interface. For arithmetic instructions, the unit identifies the specific functional operation and the required addressing mode, managing the operand sources as follows:

- Vector-Vector (*vv*): Decodes register addresses for subsequent VRF fetches.
- Vector-Scalar (*vx*): Retrieves the scalar value from the register interface and replicates it across the vector elements. In this implementation, the address encoded within the instruction is not used to read from scalar registers directly; instead, the data is transferred via the X-IF `register interface`. However, these scalar register addresses remain necessary for the configuration of CSRs.
- Vector-Immediate (*vi*): Extracts the immediate value directly from the instruction encoding.

Control and Status Registers (CSRs). The CSRs define the architectural state of the vector processor, providing the hardware with critical configuration data regarding element processing and vector register management. In this implementation, the `vlenb` register is set to 16, reflecting the 128-bit VLEN chosen for the accelerator.

For the purposes of this thesis, certain aspects of the RVV specification were omitted or only partially integrated to focus on core arithmetic performance. Specifically:

- `vxsat`, `vxrm` and `vcsr`: These fixed-point specific features are not implemented in the current design since no floating-point is supported.
- `vstart`: Support for this register is currently limited, as the current implementation assumes vector operations are non-restartable following an exception.
- LMUL: Supported values are restricted to 1, 1/2, and 1/4. Vector register grouping (LMUL > 1) is not implemented in this version to simplify register file mapping.

Consequently, based on the relationship defined in Equation 3.1, the supported vector length (`vl`) starts from 1, while the maximum vector length (VLMAX) ranges from 4 (for SEW = 32, LMUL = 1) to 16 (for SEW = 8, LMUL = 1).

$$VLMAX = \frac{LMUL \times VLEN}{SEW} \tag{3.1}$$

Load-Store Unit (LSU)

The primary function of the LSU is to manage data movement between the system memory and the VRF. Upon receiving a single instruction request, the unit orchestrates either the loading of memory data into the VRF or the storage of active vector elements from the VRF into the X-HEEP memory space.

The unit employs a single OBI port, consisting of dedicated master request and slave response channels. The VPU operates as the bus master throughout these transactions, initiating all memory access requests.

While standard scalar load-store operations typically manage data widths of 8, 16, 32, or 64 bits, vectorised operations require the processing of VL elements of SEW size. Consequently, the design of this module was engineered to meet the following technical specifications:

- VLEN Throughput: Logic module to facilitate the loading or storing of VLMAX elements within a single instruction cycle.

- Subextension Compatibility: Support for 8, 16, and 32-bit element widths to ensure compliance with the Zve32x RVV subextension SEW options.

- Addressing Flexibility: Support for indexed memory access and misalignment handling. Because the placement of data within a memory word is contingent upon both VL and SEW, elements are not guaranteed to be aligned to the byte-zero position of the memory word.

Architectural contributions. The coprocessor incorporates a dedicated LSU derived from the CVE2 architectural design.[3] While the core RTL is based on the CVE2 LSU, it functions as a standalone instance within the coprocessor's logic. This module is encapsulated within a custom control wrapper designed to translate scalar memory operations into the iterative access patterns required for vector processing. To maintain a clear distinction from the primary CPU's memory interface, this internal component is hereafter referred to as the `scalar LSU`.

The additional architectural logic primarily involves a 128-bit intermediate buffer, designed to facilitate the transition between the wide vector registers and the narrower memory bus. This buffer serves as a staging area that is dynamically managed based on the SEW and the current VL (see Figure 3.3).

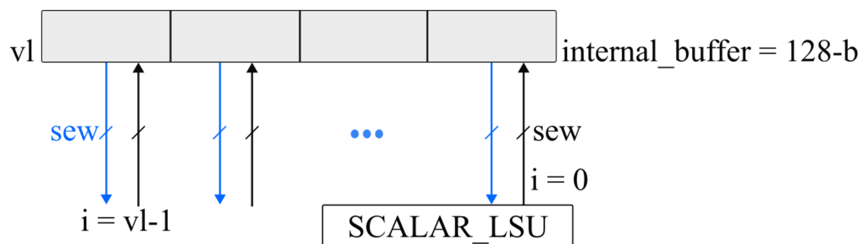

Figure 3.3: Indexed access and data transfer to internal buffer and the scalar load-store unit.

The control layer orchestrates the data movement through an iterative state machine, ensuring all vector elements are processed before the instruction is retired. In load

[3] `https://github.com/openhwgroup/cve2/blob/main/rtl/cve2_load_store_unit.sv`

operations, upon receiving a load instruction and a base address, the unit initiates a sequence of memory read requests. Data retrieved from memory through the `scalar LSU` is placed into the 128-bit buffer in an indexed manner. The unit continues these iterations until the number of loaded elements matches VL. Once the buffer is fully populated with the required active elements, the entire vector is written to the VRF in a single transfer, and the instruction is terminated. In store operations, the process begins by accessing the relevant vector from the VRF and staging it within the intermediate buffer. The control logic then enters a loop, iteratively providing the memory addresses and the corresponding data segments to the underlying `scalar LSU` module. The instruction is only marked as complete once the OBI interface acknowledges the successful storage of all active elements.

A critical function of the wrapper logic is the modification of the memory addresses sent to the `scalar LSU` within the VPU. The system must calculate the target address for each element by incrementing the base address according to the instruction type and SEW. Currently, the accelerator supports unit-stride (contiguous) and strided memory instructions. In unit-stride operations, the address increment is directly tied to the SEW (e.g., an increment of 1, 2, or 4 bytes). In strided operations, the increment is determined by the specific stride value provided by the instruction.

This iterative hardware loop ensures that the VPU can handle misaligned data and non-contiguous memory access patterns transparently, providing the programmer with a consistent vector memory model.

The necessity of this support stems from the fact that vector instructions often request a variable number of elements that do not align with the 32-bit word boundaries of the memory system. Whilst the memory subsystem transactions occur in contiguous words, the VPU must maintain the flexibility to access specific bytes within those words based on the current Selected Element Width.

As illustrated in Figure 3.4, the hardware manages these byte-offsets by tracking the starting position within each memory word. For instance, if a unit-stride load is executed with a vector length (VL) of 3 using 8-bit elements, the hardware retrieves bytes at indices 0, 1, and 2 of the first memory word. A subsequent 4-element load must then initiate at byte index 3 of that same word before transitioning to the next memory address to retrieve the remaining three elements.

Without this granular byte-addressing capability, data located at non-word-aligned offsets would be bypassed, leading to incorrect operand fetches. This mechanism ensures that the VPU handles misaligned data and non-contiguous access patterns seamlessly, providing a consistent vector memory model to the programmer.

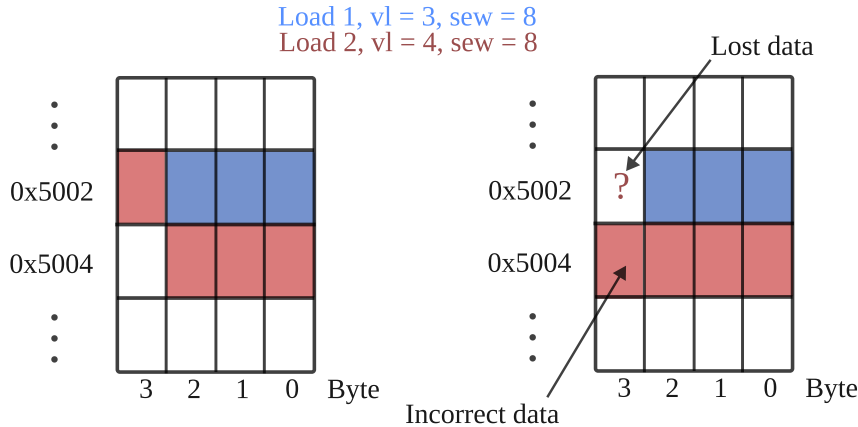

Figure 3.4: Misaligned access to memory.

Vector Register File (VRF)

In compliance with the RVV extension [1], the design implements a VRF consisting of 32 architectural vector registers. Each register possesses a fixed bit-length, denoted as VLEN, which is defined as 128 bits in this thesis.

The VRF is implemented using a packed array structure (`logic [NUM_REGS-1:0][VLEN-1:0] vreg`). Unlike unpacked arrays, the fundamental advantage of a packed representation is that it guarantees contiguous memory allocation across the entire array [17, §7.4, p. 144].

This contiguous storage is particularly advantageous when implementing the LMUL feature of the RVV standard. Although register grouping (LMUL > 1) is outside the current implementation scope, the adopted packed format ensures that the architecture remains fundamentally compatible with such features. By storing registers in adjacent bit positions, the hardware could eventually access grouped registers as a unified block without the overhead of complex indexing or non-contiguous addressing. For the current implementation, this approach ensures a straightforward register file mapping for the supported LMUL ≤ 1 configurations.

The VRF features three read ports and one write port. This configuration is specifically adjusted to support the throughput requirements of the RVV ISA:

- Two Read Ports: Dedicated to vector-vector (`vv`) instructions, which require two source operands to be fetched simultaneously from the VRF.
- Third Read Port: Necessary for instructions that require the current data in the destination vector to be read before performing an operation (e.g., masked operations or certain accumulative arithmetic instructions).
- Single Write Port: Used to commit the results of arithmetic or load instructions back to the VRF.

As detailed in section 3.3.2, almost all arithmetic and store instructions necessitate at least one vector read. Specifically, vector-vector (`vv`) instructions require a minimum of two vector operands, while vector-scalar (`vx`) and vector-immediate (`vi`) instructions require at least one. Depending on the specific opcode and masking requirements, the third read port is utilised to fetch the destination vector's contents to complete the operation.

To support the hardware-level management of vector segments (prestart, body and tail), the VRF includes dedicated logic to enforce tail and mask policies. The implementation utilises a combination of mask-generation logic to ensure that prestart, elements in the tail and inactive body regions are handled as either undisturbed or agnostic, as configured by the user at runtime. This integration facilitates a structured hardware implementation while ensuring compliance with the architectural state definitions summarised in Table 1.2.

SIMD Architecture and VAU Organisation

This section details the Single Instruction, Multiple Data (SIMD) architecture, specifically the management of vector elements and the internal organisation of the Vector Arithmetic Units (VAUs). Adhering to the RISC-V Zve32x vector subextension [1], the processor supports SEWs of 8, 16, and 32 bits. The vector coprocessor must dynamically adapt its functional units to process these varying widths based on the instruction request.

The architecture is defined by a minimum vector length (VLEN_{min}) of 32 bits by the standard. The primary SIMD structure consists of N lanes, where $N = \text{VLEN}/32$. Each base SIMD module (or lane) is designed to process a maximum of 32 bits per clock cycle. To maintain high throughput across different SEWs, each lane incorporates a heterogeneous selection of functional units: one 32-bit VAU, one 16-bit VAU, and two 8-bit VAUs, Figure 3.5 outlines this internal organisation.

Data is dispatched to these units based on the SEW specified by the instruction:

- **SEW = 32:** A single 32-bit element is processed using the 32-bit VAU path.
- **SEW = 16:** Two elements are processed per lane by utilising the dedicated 16-bit VAU and the lower 16 bits of the 32-bit VAU.
- **SEW = 8:** Four elements are processed per lane by activating the two 8-bit VAUs, the lower 8 bits of the 16-bit VAU, and the lower 8 bits of the 32-bit VAU.

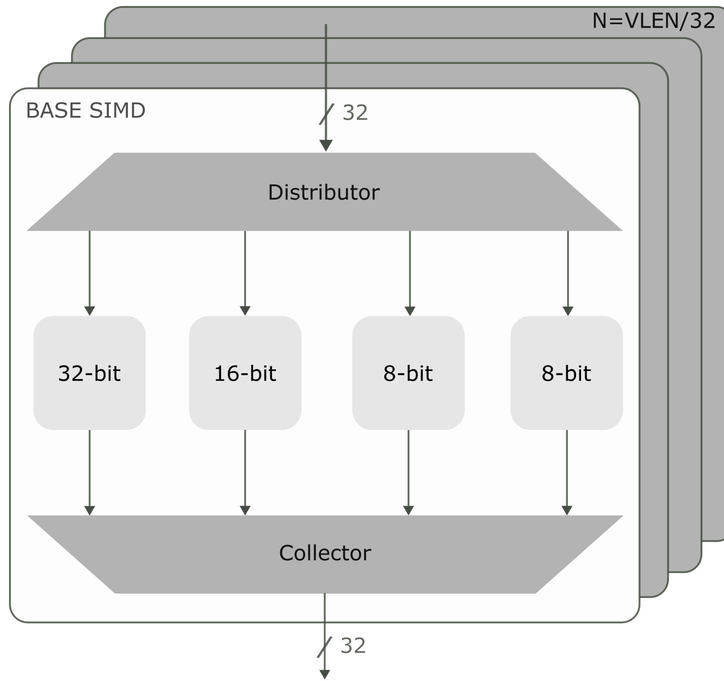

Figure 3.5: VAU and SIMD lanes distribution.

This multi-width functional unit mapping is inspired by the SPATZ [16] vector processor reference design, ensuring efficient hardware utilisation regardless of the operand size.

Scalability and VLEN Configuration. In accordance with the RVV specification [1], the SIMD structure is designed to be highly configurable at design-time. The base SIMD module acts as a scalable building block; it is replicated as many times as necessary to meet the target VLEN. In this specific implementation, where VLEN = 128, the VPU instantiates four base SIMD blocks. This modular approach allows the design to scale seamlessly for different performance requirements while maintaining a consistent internal lane logic.

Vector Movement and Permutation Logic

This module manages element and vector data movement within the VRF. While the RVV defines a wide range of permutation instructions [1], the current implementation focuses on a specific subset (see section A.2). The module is also responsible for address assignment and incremental logic during multi-vector register operations. The supported instructions are categorised into four primary types:

- **Vector Integer Move Instructions:** These instructions copy a source operand (which may be a vector, scalar, or immediate) into the active elements of a destination vector register group. Although they involve data movement, these are functionally categorised as arithmetic instructions.

- **Permutation Instructions:** These operations reorder or move elements within and between vector registers.

  - **Integer Scalar Move Instructions:** These handle the transfer of a single value between a scalar register and element 0 of a target vector register (see Figure 3.6).

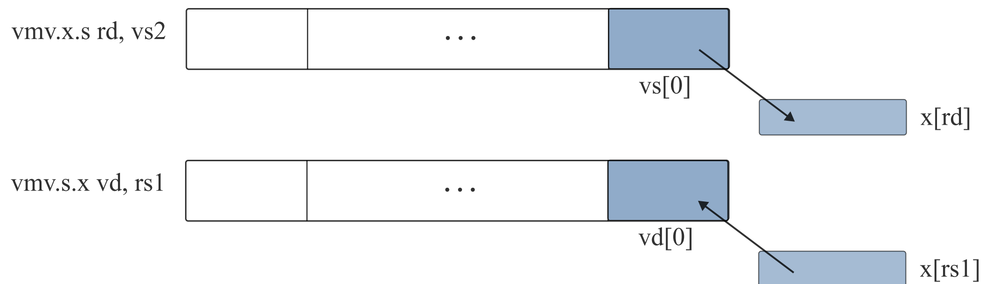

Figure 3.6: `vmv.x.s rd, vs2` instruction moving element 0 from VRF to scalar register. `vmv.s.x vd, rs1` instruction moving scalar register data to element 0 of the VRF.

  - **Whole Vector Register Move Instructions:** These facilitate the multi-register transfers of entire vector registers into a destination group. The architecture supports the concurrent copying of up to eight vector registers (see Figure 3.7).

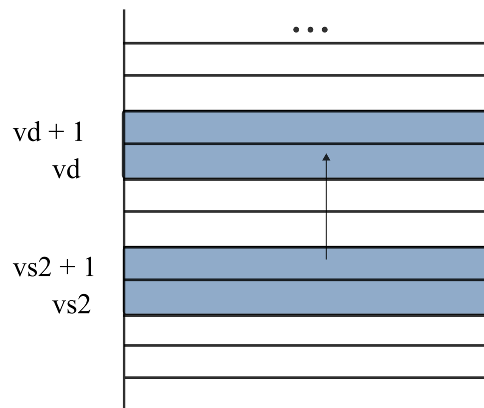

Figure 3.7: `vmv2r.v vd, vs2` instruction moving two whole vector registers within the VRF.

Chapter 4

Software Development Environment and Methodologies

4.1 RISC-V Compilation Toolchain and Configuration

The RISC-V GNU Compiler Toolchain was utilised to compile applications written in C, allowing them to interact transparently with the vector accelerator.[1] The configuration and installation of this toolchain represented a critical phase of the development process, as the compiler's settings directly govern its behaviour, its interpretation of vector instructions, and the subsequent machine code dispatched to the vector accelerator.

To ensure compliance with the RV32I base instruction set, alongside the Zve32x vector subextension and the Zvl128b length requirement, the target architecture was specifically defined to match the hardware capabilities. By explicitly targeting the Zve32x subextension, the configuration ensures the cross-compiler interprets vector-scalar instructions correctly without attempting to implement unsupported floating-point operations or any other subextension instructions. This targeted approach avoids the overhead of the full "V" extension, ensuring that hardware resources are not allocated to unsupported architectural features.

By targeting the required sub-extension, the "installation" of the whole "V" extension is avoided, and the use of unnecessary resources.

The toolchain was built using GCC v15.1.0 as the cross-compiler, configured with the following parameters:

```
./configure \
--target=riscv32-unknown-elf \
--prefix=/tools/rv32imc_zve32x_zvl128b \
--with-arch=rv32imc_zve32x_zvl128b \
--with-abi=ilp32 \
--enable-multilib \
--with-cmodel=medlow
```

During the execution of C code, the cross-compiler correctly identifies 32-bit vector-scalar instructions defined by the Zve32x extension. Any instructions not

[1] https://github.com/riscv-collab/riscv-gnu-toolchain

supported by the extension are handled by the standard hardware core. Within the X-HEEP environment, the architecture is explicitly defined as follows:

```
ARCH = rv32im_zve32x_zvl128b
```

4.2 Programming Methodologies for Vector Acceleration

To evaluate the performance and flexibility of the vector accelerator in the X-Fusion environment, two primary coding methodologies were considered: the use of "RVV C Intrinsics" and Standard (Pure) C coding with auto-vectorisation. While "RVV C Intrinsics" provide low-level precision in hardware execution, standard C coding allows the compiler to manage vectorisation automatically. This ensures code transparency and enables the use of standard benchmarks, promoting standardisation and compatibility with general-purpose applications that are not tailored for the RISC-V Vector ISA.

4.2.1 RISC-V Vector C Intrinsic

The first approach utilises the "RISC-V Vector C Intrinsics" library, which serves as a functional interface between the C language and the hardware implementation [18]. This method is often preferred in high-performance applications as it simplifies RVV implementation for the user by the compiler handling instruction scheduling and register allocation and intrinsics ensuring the correct hardware configuration for execution. An example of this instruction mapping is shown below:

C Intrinsic: `__riscv_vmacc_vx_i32m1(vy, a, vx, vl);`

Generated Assembly with SEW = 32, LMUL = 1: `vmacc.vx vd, rs1, vs2`

A key advantage of this style, similar to Inline Assembly, is that it allows the programmer to determine the precise instructions the vector processor receives. This avoids reliance on the compiler's optimisation decisions, which may not always choose the most efficient strategy. Furthermore, configuration parameters such as LMUL and SEW are explicitly defined by the user. The implementation of the kernels developed using this methodology can be found in subsection 5.1.1.

4.2.2 Standard C and Auto-vectorization

The second approach involves developing algorithms in Standard C and delegating the vectorisation process entirely to the compiler, which is already configured with the target vector subextension (`ARCH = rv32imc_zve32x_zvl128b`). Although the compiler

is capable of performing auto-vectorisation, several architectural and logical constraints often impede this optimisation.

A primary factor is semantic preservation, in which the compiler must ensure that the output of the vectorised code precisely matches the behaviour of the scalar implementation. This becomes problematic when using `int8_t` or `int16_t` for both operands and the accumulator, as the C standard mandates that results wrap (overflow) at their respective bit-widths. High-performance hardware instructions, such as Widening Multiply-Accumulates, prevent this wrapping by utilising wider internal registers, such as 32-bit accumulators. Vector hardware, however, is designed to accumulate multiple operations in wider registers and apply truncation only once after several iterations. When the accumulator in the source program is declared with a narrow type, such as `int8_t` or `int16_t`, using widening instructions would change the point at which wraparound occurs and potentially alter the result. Since no vector execution order can naturally preserve per-iteration truncation, the compiler cannot legally apply vectorization and retains the scalar form.

In addition to these semantic constraints, the Cost Model Profitability plays a decisive role in the compiler's decision-making process. Even when a mathematically correct vectorization strategy exists, the compiler estimates whether the transformation is profitable relative to scalar execution. Smaller data types such as `int8` increase vector density, allowing many elements to be processed in parallel within a single vector register. Despite this increased density, for smaller loops or unfavorable memory patterns, the compiler may find that the cost of preparing vector registers and managing loop boundaries exceeds the speedup provided by parallel execution. In such cases, the transformation is considered unprofitable compared to the scalar baseline, leading the compiler to favour a non-vectorised output.

To ensure a consistent performance baseline and to mandate that the compiler aggressively attempts these vector transformations, the `-O3` flag was applied as a required attribute for all Standard C implementations.

Compiler Control and Vectorisation Attributes

Under this methodology, the compiler analyses the C code and attempts to map operations to the vector instructions supported by the target architecture (in this case, the Zve32x subextension). If the compiler cannot find a suitable vector instruction mapping, it defaults to standard scalar execution. For code segments where vectorisation is specifically not desired, such as when establishing a scalar "Golden" reference, the following attribute was implemented to maintain scalar execution: `__attribute__((optimize("no-tree-vectorize")))`.

The code snippet below provides an example of this application, detailing how the "Golden" functions are forced to execute in the scalar unit. In contrast, for the kernels that were intended to be tested by the vector accelerator, `attribute((optimize("O3")))` was applied. While the compiler is generally capable of understanding and optimising loops without additional attributes, it may still defaults to scalar code due to the semantic

and cost-based limitations discussed previously. Furthermore, in this example was observed that the comparison logic used to verify the scalar and vector results was itself automatically translated into vector instructions, demonstrating the compiler's tendency to utilise the vector unit for data-parallel tasks.

```
...
__attribute__((optimize("O3", "noinline", "no-tree-vectorize")))
void saxpy_golden_int32(size_t n, int32_t a,
                        const int32_t *x, int32_t *y)
{
    for (size_t i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}

__attribute__((optimize("O3")))
void saxpy_vec_int32(size_t n, int32_t a,
                        const int32_t *__restrict x, int32_t *__restrict y)
{
    for (size_t i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}

int verify_results_int32(int n, int32_t *golden, int32_t *vector) {
    int errors = 0;
    for (int i = 0; i < n; i++) {
        if (golden[i] != vector[i]) {
            errors++;
        }
    }
    return errors;
}

int main() {

    int32_t a = 4;
    int32_t input_int32[N] = {
        0xDEADBEEF, 0xCAFEBABE,
        ...
    };

    for (int i = 0; i < N; i++) {
        output_golden_int32[i] = input_int32[i];
        output_vector_int32[i] = input_int32[i];
    }

    saxpy_golden_int32(N, a, input_int32, output_golden_int32);
    saxpy_vec_int32(N, a, input_int32, output_vector_int32);

 int fails_t1 = verify_results_int32(N, output_golden_int32, output_vector_int32);

...
```

4.2.3 Summary of Programming Approaches

These two programming possibilities represent a trade-off between development effort and hardware control. While intrinsic code requires a deeper understanding of the vector processor's capabilities and microarchitecture, it ensures a predictable instruction sequence and often results in more optimal execution. In contrast, standard C facilitates rapid development and greater code portability but leaves performance entirely dependent

on the compiler's auto-vectorisation capabilities. An empirical comparison of these methods is further discussed in section 5.2.

Chapter 5

Results

This chapter presents the performance evaluation and experimental findings for the vector accelerator developed in this thesis. The results comprise data obtained from functional simulations conducted in QuestaSim and hardware synthesis reports. While the initial goal involved standard benchmarks, considerable portion of the research was dedicated to investigating the compiler's auto-vectorisation behaviour and limitations for C code. Consequently, validation was performed by implementing data-parallel algorithms using the "RISC-V Vector C Intrinsics" library alongside standard C code.

5.1 Tests Applications

The performance of the accelerator was evaluated using three primary kernels: the **SAXPY** algorithm (`y[i]= a·x[i]+ y[i]`), **Indexed Arithmetic** algorithm (`a[i]= i·b[i]+ c[i]`), and a standard Matrix Multiplication (**MatMul**) algorithm. To assess the efficiency of the vector unit and the software toolchain, two distinct implementation approaches were employed:

1. **"RISC-V Vector C Intrinsics":** Manual implementation using RVV C intrinsic instructions to provide fine-grained control over the vector unit. Examples in "RISC-V Vector C Intrinsiscs" library were taken as a reference from the open-source repository.[1]

2. **Auto-vectorisation:** Standard C code compiled with GCC auto-vectorisation features to evaluate the compiler's ability to target the custom hardware.

5.1.1 SAXPY Kernel

The implementation using RISC-V Vector C Intrinsics is presented below:

```
...
__attribute__((optimize("O3", "noinline", "no-tree-vectorize")))
void saxpy_golden_int32(size_t m, const int32_t a, const int32_t *x, int32_t *y)
{
    for (size_t i = 0; i < m; i++) {
```

[1] https://github.com/riscv-non-isa/rvv-intrinsic-doc/tree/main/examples

```
        y[i] = a * x[i] + y[i];
    }
}

void saxpy_vec_int32(size_t n, const int32_t a, const int32_t *x, int32_t *y)
{
    for (size_t vl; n > 0; n -= vl, x += vl, y += vl)
    {
        vl = __riscv_vsetvl_e32m1(n);

        // *** LOAD 32-bit words even though elements are int32 ***
        vint32m1_t vx = __riscv_vle32_v_i32m1(x, vl);
        vint32m1_t vy = __riscv_vle32_v_i32m1(y, vl);

        // y = a * x + y
        vint32m1_t vres = __riscv_vmacc_vx_i32m1(vy, a, vx, vl);

        // *** STORE 32-bit ***
        __riscv_vse32_v_i32m1(y, vres, vl);
    }
}

...
```

The following code segment demonstrates the Standard C implementation used for auto-vectorisation:

```
...

__attribute__((optimize("O3", "noinline", "no-tree-vectorize")))
void saxpy_golden_int32(size_t n, int32_t a,
                        const int32_t *x, int32_t *y)
{
    for (size_t i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}

__attribute__((optimize("O3")))
void saxpy_vec_int32(size_t n, int32_t a,
                     const int32_t *__restrict x, int32_t *__restrict y)
{
    for (size_t i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}

...
```

5.1.2 Indexed Arithmetic Kernel

While architecturally similar to the SAXPY kernel, the Indexed Arithmetic algorithm was implemented to evaluate how minor algorithmic variations influence execution efficiency and vector unit utilisation. Unlike SAXPY, which employs the destination vector as one of the source operands (e.g., $y = ax + y$), this kernel treats the destination vector as a separate entity. The implementation modifies all operands by incorporating

the loop index (i) directly into the calculation, as defined by the following expression: $a[i] = i \cdot b[i] + c[i]$.

The implementation using RISC-V Vector C Intrinsics is presented below:

```
...

__attribute__((optimize("O3", "noinline", "no-tree-vectorize")))
void index_golden32(uint32_t *a, uint32_t *b, uint32_t *c, int n) {
  for (int i = 0; i < n; ++i) {
    a[i] = b[i] + i * c[i];
  }
}

void index_vec32(uint32_t *a, uint32_t *b, uint32_t *c, int n) {
  size_t vlmax = __riscv_vsetvlmax_e32m1();

  vuint32m1_t vec_i = __riscv_vid_v_u32m1(vlmax);

  for (size_t vl; n > 0; n -= vl, a += vl, b += vl, c += vl) {
    vl = __riscv_vsetvl_e32m1(n);

    vuint32m1_t vec_b = __riscv_vle32_v_u32m1(b, vl);
    vuint32m1_t vec_c = __riscv_vle32_v_u32m1(c, vl);

    vuint32m1_t vec_a = __riscv_vmacc_vv_u32m1(vec_b, vec_c, vec_i, vl);

    __riscv_vse32_v_u32m1(a, vec_a, vl);

    vec_i = __riscv_vadd_vx_u32m1(vec_i, vl, vl);
  }
}

...
```

The following code segment demonstrates the Standard C implementation used for auto-vectorisation:

```
...

__attribute__((optimize("O3", "noinline", "no-tree-vectorize")))
void index_golden32(uint32_t *a, uint32_t *b, uint32_t *c, int n) {
  for (int i = 0; i < n; ++i) {
    a[i] = b[i] + i * c[i];
  }

__attribute__((optimize("O3")))
void index_vec32(uint32_t *__restrict a, uint32_t *__restrict b, uint32_t *c, int n) {
  for (int i = 0; i < n; ++i) {
    a[i] = b[i] + i * c[i];
  }
}

...
```

5.1.3 MatMul Kernel

The matrix multiplication kernel implemented for "RVV C Intrinsics" employs a dot-product strategy. This implementation computes $C = A \times B$ by accepting matrix B in a pre-transposed format (B^T). This layout facilitates unit-stride vector loads, accessing contiguous memory elements, which is theoretically optimal for memory throughput. The implementation using RISC-V Vector C Intrinsics is presented below:

```
...

void matmul_golden32(int32_t **a, int32_t **b_t, int32_t **c, int n, int m, int o) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j) {
      c[i][j] = 0;
      for (int k = 0; k < o; ++k)
        c[i][j] += a[i][k] * b_t[j][k];
    }
}

void matmul32(int32_t **a, int32_t **b_t, int32_t **c, int n, int m, int o) {
  size_t vlmax = __riscv_vsetvlmax_e32m1();
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      int32_t *ptr_a = &a[i][0];
      int32_t *ptr_b = &b_t[j][0];
      int k = o;
      vint32m1_t vec_s    = __riscv_vmv_v_x_i32m1(0, vlmax);
      vint32m1_t vec_zero = __riscv_vmv_v_x_i32m1(0, vlmax);

      for (size_t vl; k > 0; k -= vl, ptr_a += vl, ptr_b += vl) {
        vl = __riscv_vsetvl_e32m1(k);

        vint32m1_t vec_a = __riscv_vle32_v_i32m1(ptr_a, vl);
        vint32m1_t vec_b = __riscv_vle32_v_i32m1(ptr_b, vl);

        vec_s = __riscv_vmacc_vv_i32m1(vec_s, vec_a, vec_b, vl);
      }

      vint32m1_t vec_sum = __riscv_vredsum_vs_i32m1_i32m1(vec_s, vec_zero, vlmax);

      int32_t sum = __riscv_vmv_x_s_i32m1_i32(vec_sum);
      c[i][j] = sum;
    }
  }
}

...
```

In contrast, the original algorithm could not be auto-vectorised because the input was structured as a matrix rather than a linear vector. This increased complexity prevented the compiler from identifying as vector operands, thereby forcing a fallback to scalar execution. Consequently, an alternative strategy was implemented to evaluate auto-vectorisation capabilities (algorithm detailed below). This configuration processes the input matrices as flattened 1D arrays using a standard $i - j - k$ loop order.

The following code segment demonstrates the Standard C implementation used for auto-vectorisation:

```
...
__attribute__((optimize("O3", "noinline", "no-tree-vectorize")))
void matmul_golden_int32( int32_t *a,
                          int32_t *b,
                          int32_t *c, size_t size_m, size_t size_n, size_t size_k)
{
    for (size_t m = 0; m < size_m; m++)
    {
        for (size_t n = 0; n < size_n; n++)
        {
            int32_t acc = 0;
            for (size_t k = 0; k < size_k; k++)
            {
                acc += a[m*size_k + k] * b[k*size_n + n];
            }
            c[m*size_n + n] = acc;
        }
    }
}

__attribute__((optimize("O3")))
void matmul_hw_int32( int32_t *a,
                      int32_t *b,
                      int32_t *c, size_t size_m, size_t size_n, size_t size_k)
{
    for (size_t m = 0; m < size_m; m++)
    {
        for (size_t n = 0; n < size_n; n++)
        {
            int32_t acc = 0;
            for (size_t k = 0; k < size_k; k++)
            {
                acc += a[m*size_k + k] * b[k*size_n + n];
            }
            c[m*size_n + n] = acc;
        }
    }
}
...
```

5.2 Performance Evaluation

5.2.1 SAXPY Performance

The experimental process began with functional simulation, which was subsequently corroborated through physical deployment on the FPGA. The following results detail the clock cycle counts obtained for the SAXPY kernel across varying data widths (32-bit, 16-bit, and 8-bit integers). Reference "Golden" values represent standard scalar execution on the base processor core.

The VPU passed all functional verification tests, confirming its architectural integrity. The performance results are detailed in Table 5.1.

Table 5.1: SAXPY Results: Execution Cycles.

Implementation	Data Type	Vector Size	Scalar (Cycles)	Vector (Cycles)	Speedup
"RVV C Intrinsics"	32-bit	32	490	255	1.92×
	16-bit	23	377	124	3.04×
	8-bit	41	626	180	3.48×
Auto-vectorisation	32-bit	32	490	267	1.84×
	16-bit	23	378	137	2.76×
	8-bit	41	624	185	3.37×

While the performance gap between manual intrinsics and auto-vectorisation remains narrow, the transition from scalar to vector execution achieves significant gains that scale with data density. In the 32-bit configuration, the speedup is limited to approximately 1.9×. However, as the operand width decreases, the speedup improves to 3.04× for 16-bit data and reaches a peak of 3.48× for 8-bit integers.

This scaling behaviour is directly attributable to the SIMD nature of the RVV extension. With a fixed vector length, reducing the bit-width of the data types increases the number of elements processed per clock cycle. Specifically, an 8-bit operation allows for four times the throughput of a 32-bit operation within the same register space.

5.2.2 Indexed Arithmetic Performance

The Indexed Arithmetic kernel was validated using the same methodology as the SAXPY benchmarks, with simulation results successfully corroborated through FPGA deployment. Table 5.2 presents the execution cycles for a fixed vector length of 31 elements. Consistent with previous observations, the speedup scales inversely with operand bit-width; the highest performance gain of 3.76× was achieved using 8-bit integers with "RISC-V Vector C Intrinsics", while the lowest acceleration occurred in the 32-bit auto-vectorised configuration.

A clear difference in performance appears when comparing these results to the SAXPY kernel. While SAXPY showed almost no difference between manual intrinsics and auto-vectorisation, the Indexed Arithmetic kernel (`a[i] = i · b[i] + c[i]`) shows a significant gap across all configurations. The auto-vectorised speedups are consistently lower than those achieved with manual intrinsics, reaching only 3.01× for 8-bit data compared to the 3.76× manual peak.

Table 5.2: Index Arithmetic Results: Execution Cycles.

Implementation	Data Type	Vector Size	Scalar (Cycles)	Vector (Cycles)	Speedup
"RVV C Intrinsics"	32-bit	31	536	298	1.8×
	16-bit	31	567	197	2.88×
	8-bit	31	537	143	3.76×
Auto-vectorisation	32-bit	31	536	443	1.21×
	16-bit	31	536	268	2.0×
	8-bit	31	536	178	3.01×

5.2.3 Matrix Multiplication (MatMul) Performance

Several key observations were noted regarding the Matrix Multiplication (MatMul) kernel. Unlike the implementation using "RISC-V Vector C Intrinsics", the Standard C version necessitates strided memory access due to its algorithmic structure. While strided memory access typically results in lower throughput compared to the unit-stride patterns in many architectures, benchmarks on this specific vector accelerator indicate that memory operation latency remains uniform regardless of stride type (as illustrated in Figure 5.1). For consistency, both implementations were executed using matrix dimensions of 8×7 (A) and 7×8 (B), producing an 8×8 result matrix (C).

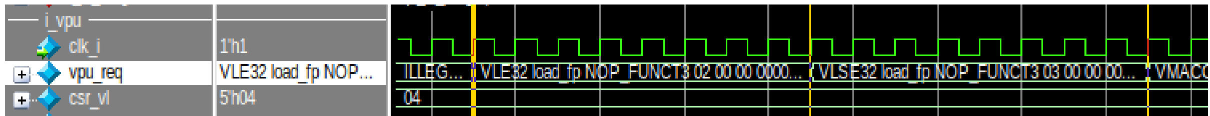

Figure 5.1: Execution time comparison between unit-stride (VLE32) and strided (VLSE32) loads for SEW=32 and VL=4. Both instructions execute in 8 clock cycles.

As with the SAXPY and Indexed Arithmetic kernel, the MatMul implementation was validated through functional simulation and subsequently corroborated on the FPGA, yielding identical results across both simulation and physical hardware. Consistent with the trends observed in the SAXPY and Indexed Arithmetic applications (Table 5.1), the highest performance gains were achieved using the 8-bit integer implementation with "RISC-V Vector C Intrinsics", which reached a speedup of 3.16× under these specific test conditions.

Notably, the compiler's auto-vectorisation did not achieve the same level of performance improvement as the manual intrinsics. This discrepancy suggests that the compiler struggled to map the nested loop structure of the Standard C MatMul to the most efficient vector instructions. To further investigate this, a more rigorous Standard C implementation, structured to more closely mirror the manual intrinsic approach, could be explored to determine if the performance gap can be narrowed.

The following analysis (Table 5.3) compares these execution times to quantify the performance differential between scalar and vector processing, as well as between manual intrinsics and compiler auto-vectorisation.

Table 5.3: MatMul Results: Execution Cycle Comparison

Implementation	Data Type	Scalar (Cycles)	Vector (Cycles)	Speedup
"RVV C Intrinsics"	32-bit	7,535	4,525	1.67×
	16-bit	7,464	3,507	2.13×
	8-bit	11,900	3,770	3.16×
Auto-vectorisation	32-bit	7,400	5,022	1.46×
	16-bit	7,399	5,022	1.47×
	8-bit	7,287	5,007	1.46×

5.2.4 Execution in FPGA

To evaluate the scalability of the previously discussed applications, a series of higher-volume tests were conducted. Due to the significant computational intensity of these larger datasets, which would result in impractical execution times in a simulation environment, the testing was performed directly on the FPGA hardware. The Indexed Algorithm was selected for this scalability study, as it demonstrated the most efficient performance during the preliminary small-scale benchmarks. The results of these hardware-based tests are presented in Table 5.4, providing a quantitative assessment of how the vector unit handles increased data loads.

Table 5.4: Indexed Algorithm Execution Cycle Comparison with 'RVV C Instrinsics".

Vector Length (N)	Data Type	Scalar (Cycles)	Vector (Cycles)	Speedup
N = 64	32-bit	1099	587	1.87×
	16-bit	1099	391	2.81×
	8-bit	1099	287	3.83×
N = 128	32-bit	2187	1163	1.88×
	16-bit	2187	775	2.82×
	8-bit	2187	575	3.80×
N = 256	32-bit	4363	2315	1.88×
	16-bit	4363	1543	2.83×
	8-bit	4363	1151	3.79×
N = 512	32-bit	8715	4619	1.89×
	16-bit	8715	3079	2.83×
	8-bit	8715	2303	3.78×
N = 1024	32-bit	17419	9227	1.89×
	16-bit	17419	6151	2.83×
	8-bit	17419	4607	3.78×

The results presented in Table 5.4 corroborate the trends previously discussed. Performance improves as the element width decreases, as a greater number of elements

can be packed into a single vector register. While the speedups for 16-bit and 32-bit elements remain relatively constant, 8-bit elements exhibit a minimal decrease in speedup as the vector length increases. Despite this slight variation, the best overall performance was achieved using 8-bit elements with a vector length of 64, a speedup of 3.83 is obtained.

It was not possible to present results for standard C coding executions due to architectural constraints. To support the full range of vector lengths (64 to 1024), the loop index must be defined as a standard integer. However, this triggers the `vnsrl.wi` instruction, which is currently unsupported. Attempts to circumvent this by constraining the index to the int8 type for 8-bit element execution resulted in erroneous outputs. Specifically, when the vector length reaches or exceeds 256, these smaller types suffer from integer overflow, as previously noted.

5.2.5 General Conclusions

A definitive correlation exists between element size and performance gains: as the element width decreases, the differential in execution time becomes more pronounced. This is primarily attributed to the increased packing density within the vector registers, which enables a greater number of elements to be processed per single instruction.

This effect is particularly significant when utilising ‘RVV C Intrinsics”. In such implementations, the LMUL is typically called with LMUL = 1. This ensures that the full capacity of the vector register is available for processing, meaning the actual number of processing elements is dependent only on the AVL and the stripmining process.

In contrast, when executing kernels in standard C, the compiler optimiser frequently configures LMUL < 1. This configuration effectively deactivates portions of the vector register, decreasing its capacity to accommodate a larger number of elements. Consequently, to process the same AVL, a greater number of stripmining iterations are required, increasing loop overhead.

Comparative Example: Register Occupancy. To illustrate the impact of LMUL on iteration count, consider a scenario where VLEN = 128 bits, SEW = 16 bits, and AVL = 14. The maximum vector length (VL_{max}) is determined by the following relation:

$$\mathrm{VL}_{max} = \frac{\mathrm{VLEN}}{\mathrm{SEW}} \times \mathrm{LMUL} \tag{5.1}$$

- **Scenario A (LMUL = 1):** Here, $\mathrm{VL}_{max} = (128/16) \times 1 = 8$. The execution is completed in two iterations (e.g., an initial VL = 8 followed by VL = 6).
- **Scenario B (LMUL = 1/2):** Here, $\mathrm{VL}_{max} = (128/16) \times 0.5 = 4$. This configuration requires four iterations through the stripmining loop (e.g., two cycles of VL = 4 and two cycles of VL = 3) to process the same AVL.

Consequently, lower LMUL configurations lead to under-utilised vector registers, necessitating a greater number of iterations through the strip-mining loop to process the complete dataset.

5.3 Hardware Resource Utilisation

This section shows the results obtained when synthesising the X-Fusion with the implemented design on the Xilinx Pynq-z2 FPGA. Table 5.5 shows resource utilisation of the vector processor. In Figure 5.2, the implementation of main components is seen: X-HEEP and VPU. Table 5.6 shows full occupation of X-Fusion and its main components on FPGA.

Table 5.5: Resource utilisation of the vector accelerator sub-modules in PYNQ-Z2 FPGA.

VPU Resource Utilisation Distribution				
Module	**LUTs**	**FFs**	**DSPs**	**BRAMs**
Instruction Decoder	13093	311	-	-
Vector Register File	5200	4097	1	-
SIMD Controller	2599	-	20	-
OBI LSU	698	175	2	-
Total VPU	**21571**	**4622**	**23**	**0**

Table 5.6: Logic utilisation of primary sub-modules and total FPGA resource consumption for X-Fusion architecture.

X-Fusion Resource Utilisation Distribution				
Module	**LUTs**	**FFs**	**DSPs**	**BRAMs**
CPU	7242	1936	1	-
Vector Accelerator	21571	4622	23	-
External Bus	39	2	-	-
Memory Subsystem	273	5	-	32
Total X-Fusion	**48920**	**31257**	**24**	**32**

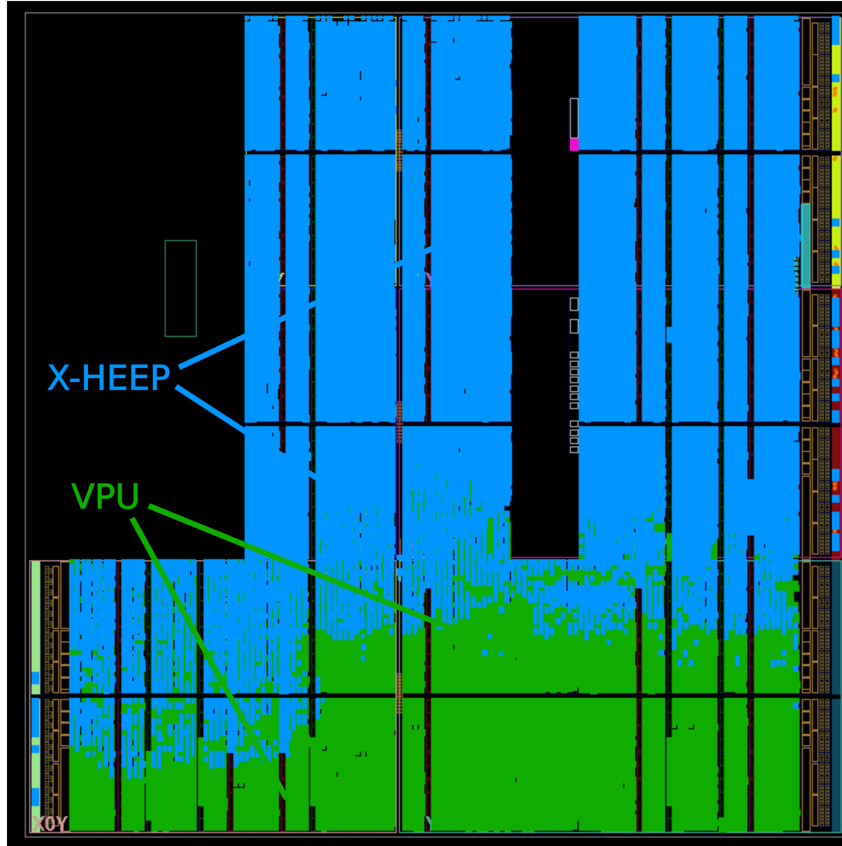

Figure 5.2: PYNQ-Z2 FPGA floorplan with X-HEEP(blue) and VPU (green) resource blocks.

5.4 Implementation Limitations and Compiler Constraints

The following section outlines current hardware limitations and compiler behaviours observed during the integration of the CV32E20 core within the X-HEEP system.

- **Incomplete Commit Interface:** In the current integration, the control commit interface signals for the CV32E20 core, `commit_valid` and `commit_kill`, are tied to static values to maintain basic functional flow. Furthermore, the instruction identification (`id`) signals remain unconnected.
  - Cause: This limitation arises from the current version of the CV32E20, which does not yet support the full handshaking protocol required for dynamic instruction tracking between the core and coprocessors.
  - Mitigation: To ensure architectural stability despite the lack of a functional "kill" mechanism, the system is constrained to a strictly in-order or non-speculative execution model. By hardwiring the commit signals, the design prevents the dispatch of vector instructions until their execution path is fully

resolved, thereby eliminating the risk of unrecoverable state inconsistencies caused by branch mispredictions.

- **Compressed Instruction Alignment:** The CV32E20 core triggers assertion failures when loading compressed instructions from memory.
  - Cause: This error originates from uninitialised data in the upper half-word of the instruction fetch (returning `XXXX`). The control logic interprets this undefined state as an illegal instruction or a bus error, flagging an assertion failure in the simulation environment.
  - Workaround: To ensure stable execution within the X-HEEP environment, support for the "C" extension was disabled. The system was reconfigured to use a standard 32-bit instruction alignment by omitting the compressed extension from the ISA string: `ARCH = rv32im_zve32x_zvl128b`.
- **Missing Vector CSRs:** The current CV32E20 integration does not implement Vector CSRs, specifically "`vlenb`".
  - Issue: The auto-vectorising compiler attempts to read '`vlenb`' to perform runtime memory dependency checks (ensuring vectors do not collide). Since the CSR is missing, this triggers an "Illegal Instruction" exception.
  - Workaround: To prevent the compiler from generating these CSR reads, the `__restrict` type qualifier was applied to input pointers. This explicitly informs the compiler that memory operands do not overlap, removing the need for memory overlapping checks.
- **Inconsistent Code Generation:** The compiler does not always automatically vectorise Standard C code, particularly when processing narrow data types.
  - Issue: If the accumulator operand is not sufficiently wider than the source operands, premature truncation can occur, compromising result accuracy. Additionally, for smaller datasets, the compiler's cost model may determine that the overhead of vector configuration outweighs the performance gains, resulting in a fallback to scalar execution.
  - Workaround: To force vector execution, the `attribute((optimize("O3")))` is applied to specific functions. However, a more robust solution involves adapting variable types to better utilise hardware widening instructions and ensuring the data alignment meets the compiler's vectorisation requirements.

Chapter 6

Conclusions and Future Lines

This thesis presents the design, basic functional verification, and hardware implementation of a vector accelerator based on the RVV Zve32x subextension, specifically targeting 32-bit integer operations and 128-bit VLEN. Developed as a coprocessor for the CV32E20 core within the X-HEEP platform, the implementation utilises the X-IF v1.0 for core-to-coprocessor communication without necessitating modifications of the CPU's RTL, and the OBI v1.0 as a lightweight mechanism for high-bandwidth memory load-store operations.

While the initial research objective was to perform verification using data-parallel standard benchmark suites, the hardware was validated by implementing data-parallel algorithms, such as SAXPY, Indexed Arithmetic and MatMul, using the "RISC-V Vector C Intrinsics" library in together with standard C implementations to provide a comparative baseline for performance and compiler efficiency.

Through the implementation of these HPC kernels, it was established that the adoption of data-parallel processing through the vector extension significantly enhances computational throughput. This was validated through simulation and FPGA implementation. For instance, the Indexed Arithmetic algorithm achieved a speedup of 3.83x using "RVV C Intrinsics". This performance gain is closely linked to element size, as narrower elements facilitate higher packing density within the vector registers, enhancing accelerator performance. This enables a greater number of elements to be processed per instruction. However, while hardware capacity provides the baseline, overall efficiency is also influenced by the compiler's management of register grouping and the specific number of elements processed during each stripmining iteration.

Observations regarding compiler behaviour highlight critical areas for further investigation. While the `-O3` flag was utilised to force auto-vectorisation. Without it, unless the output variables are explicitly cast or sized to guarantee safety, the compiler defaults to scalar execution to ensure numerical correctness. Further research is required to define the specific cost-model thresholds and code patterns that trigger these "safe" but computationally expensive scalar alternatives.

Regarding compiler behaviour and LMUL strategies, testing showed that the compiler frequently utilised fractional LMUL (where LMUL < 1). It remains unclear if the compiler effectively identifies scenarios where fractional LMUL is optimal or if it defaults to a conservative register usage of 32 bits, despite the configuration specifying a minimum vector length of 128 bits via `zvl128b`. Furthermore, the compiler did not implement LMUL > 1 (register grouping) in any test cases. The impact of register grouping specific effects on throughput and overhead requires further analysis to determine if such

configurations offer any advantage.

Extending the research presented in this work, several lines of future development have been established. From a hardware perspective, the primary goal is to complete the Zve32x extension by incorporating all remaining instructions and architectural requirements. This will be accompanied by microarchitectural optimisations aimed at reducing logic area and power consumption. To ensure industrial-grade reliability, a formal verification framework using SystemVerilog Assertions (SVA) will be integrated. Finally, the accelerator's performance will be evaluated against industry-standard machine learning data-parallel workloads for deeply embedded systems using the MLPerf Tiny benchmark suite [19].

At present, the vector accelerator utilises a single OBI port for memory operations. The interface is designed to facilitate the integration of additional OBI links for multi-port configurations without significant structural redesign. Increasing the number of memory access ports is a strategic objective to better leverage the available parallelism, as illustrated in Figure 6.1.

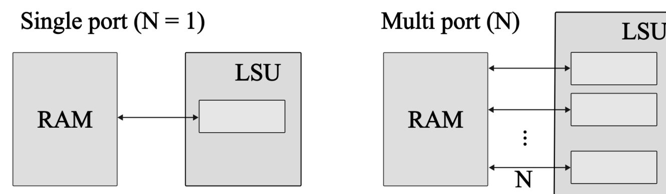

Figure 6.1: OBI multi-port configuration.

Looking towards long-term development, the research points towards a multi-core vector accelerator paradigm. The ultimate objective is to develop a modular cluster that can be replicated to provide tile-level scalability and enhanced processing power (Figure 6.2).

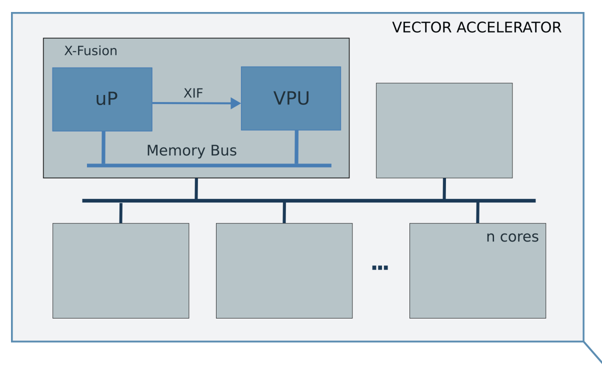

Figure 6.2: Modular cluster architecture for tile-level scalability.

Chapter A

RVV Instructions

A.1 Instruction Formats

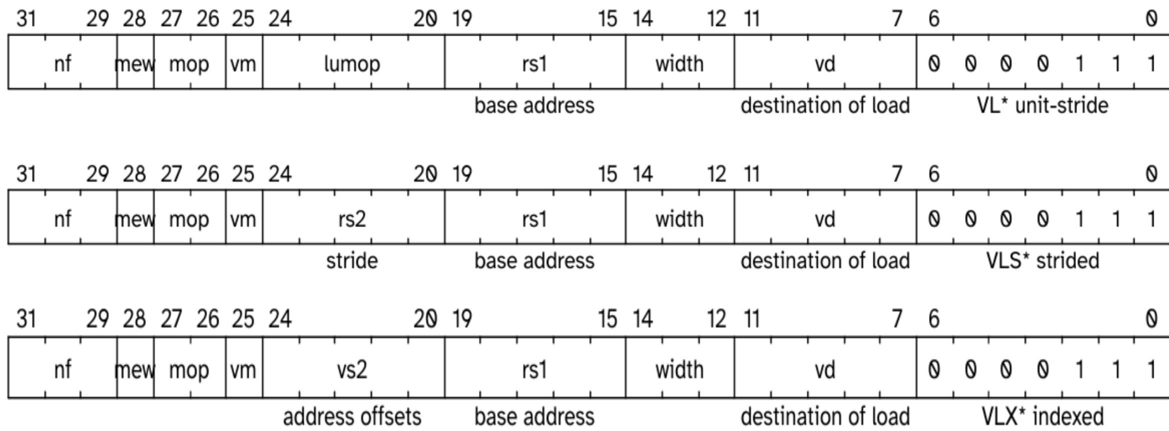

Figure A.1: Vector Load Instruction Format under LOAD-FP major opcode [1].

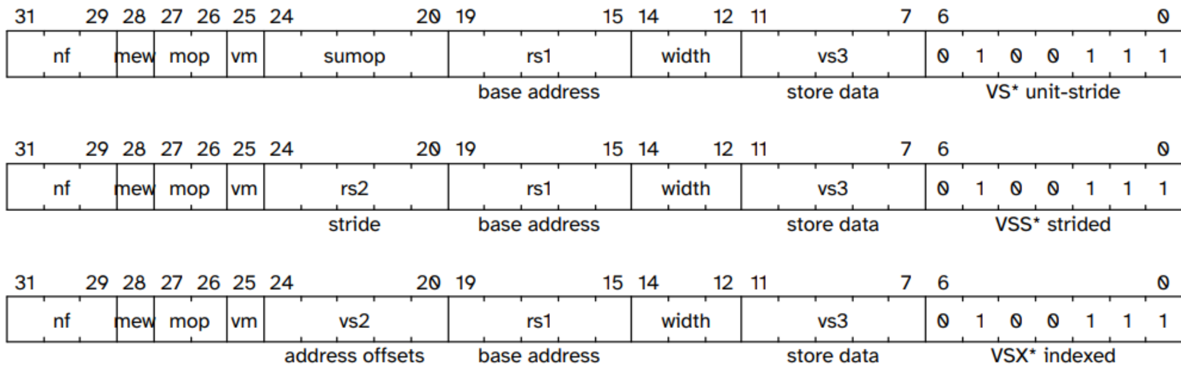

Figure A.2: Vector Store Instruction Format under STORE-FP major opcode [1].

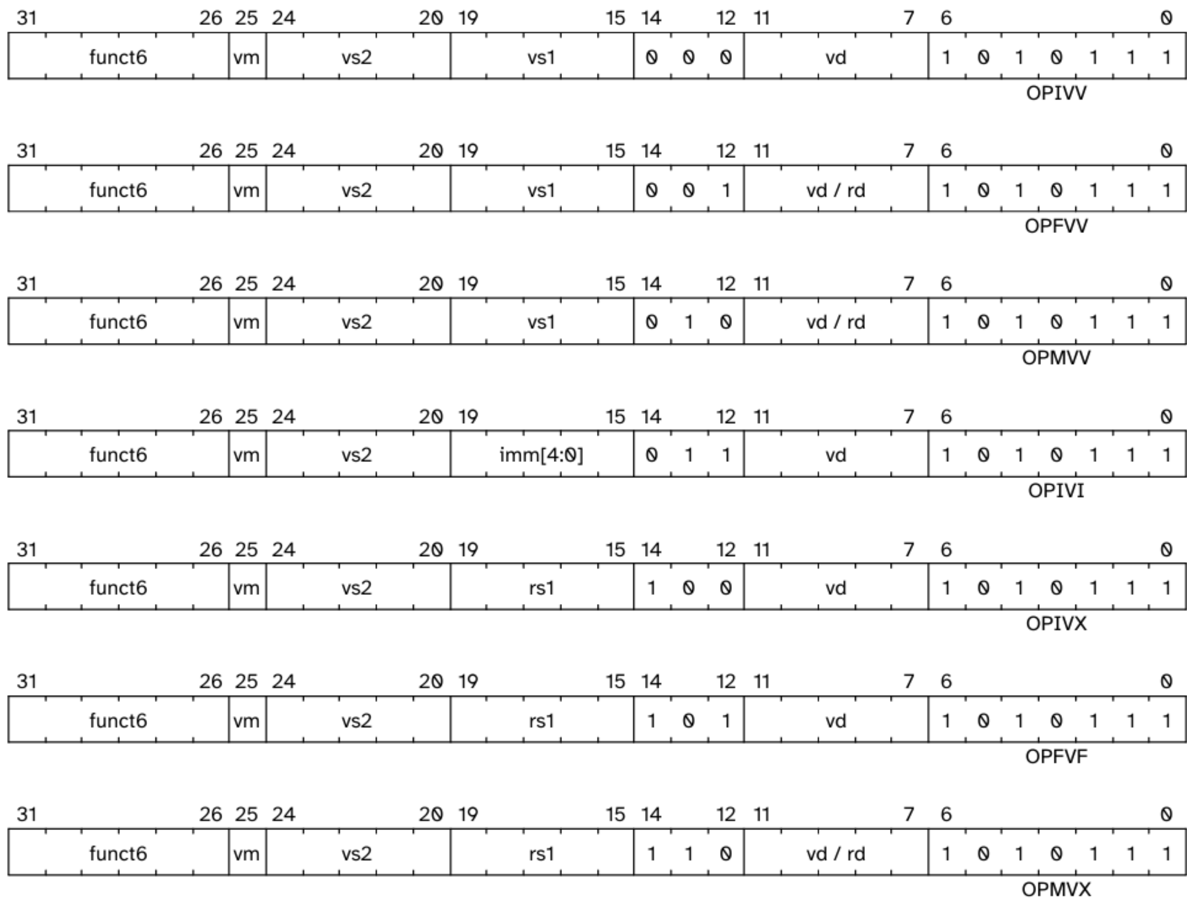

Figure A.3: Vector Arithmetic Instruction Format under OP-V major opcode [1].

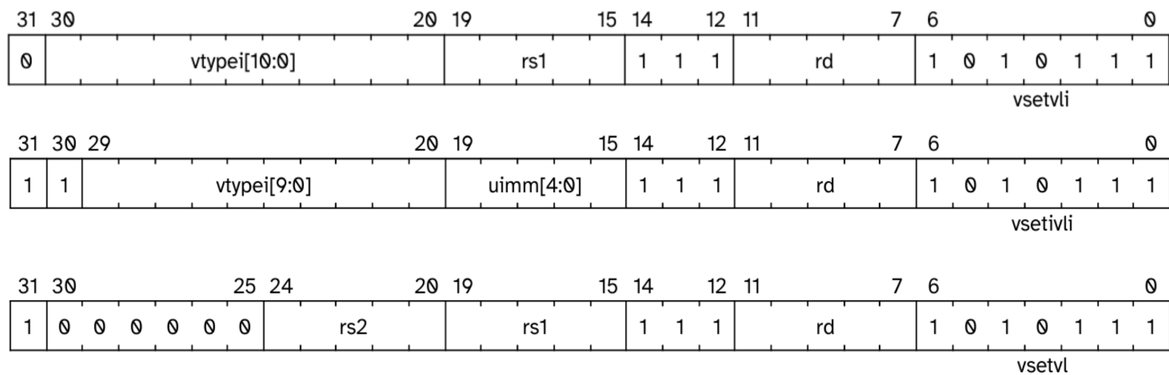

Figure A.4: Vector Configuration Instruction Formats under OP-V major opcode [1].

A.2 Supported Instructions for Zve32x subextension

Table A.1: Implemented Vector Accelerator Instructions.

Mnemonic	Type	Operands	Description
Configuration & CSRs			
`VSETVLI, VSETIVLI, VSETVL`	Config	rd, rs1(uimm), vtype	Set vector length(`vl`) and vector type (`vtype`).
Memory Operations (Load/Store)			
`VLE<8/16/32>.V`	Load	vd, (rs1), vm	Vector Unit-Stride Load (8, 16, or 32-bit elements).
`VLSE<8/16/32>.V`	Load	vd, (rs1), rs2, vm	Vector Strided Load (8, 16, or 32-bit elements).
`VL1RE32.V`	Load	vd, (rs1)	Whole Vector Register Load (1 register).
`VSE<8/16/32>.V`	Store	vs3, (rs1), vm	Vector Unit-Stride Store (8, 16, or 32-bit elements).
`VS1R.V`	Store	vs3, (rs1)	Whole Vector Register Store (1 register).
Integer Arithmetic			
`VADD`	ALU	vv, vx, vi	Integer Addition.
`VSUB`	ALU	vv, vx	Integer Subtraction.
`VRSUB`	ALU	vx, vi	Integer Reverse Subtraction.
`VAND, VOR, VXOR`	Logic	vv, vx, vi	Bitwise Logical AND, OR, XOR.
`VSLL, VSRL, VSRA`	Shift	vv, vx, vi	Logical Left/Right Shift, Arithmetic Right Shift.
`VMIN/MAX, VMINU/MAXU`	Compare	vv, vx	Signed/Unsigned Integer Minimum and Maximum.
`VMSEQ, VMSNE`	Compare	vv, vx, vi	Set mask if Equal, Not Equal.
`VMSLT(U)`	Compare	vv, vx	Less Than (Signed/Unsigned).
`VMUL, VMULH(U/SU)`	Mul	vv, vx	Integer Multiply (Low, High, High Unsigned, High Signed-Unsigned).
`VMACC, VNMSAC, VMADD`	MAC	vv, vx	Multiply-Accumulate variants (Fused/Non-fused).
Data Movement & Permutation			
`VMERGE`	Merge	vvm, vxm, vim	Vector Merge (conditional move based on mask).
`VMV`	Move	vv, vx, vi, xs, sx	Vector Move (Copy, splat scalar to vector, extract scalar).
`VMV<1/2/4/8>R.V`	Move	vd, vs2	Whole Register Move (copy 1, 2, 4, or 8 registers).
`VID.V`	Index		Write element ID to destination vector.
Reduction			
`VREDSUM.VS`	Reduce	vd, vs2, vs1	Vector Reduction Sum (sum all elements to scalar).

Bibliography

[1] RISC-V International, "The RISC-V instruction set manual, volume i: Unprivileged architecture," tech. rep., 2025. Technical Specification, Ratified State, Version 20250508.

[2] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume i: Base user-level ISA," Tech. Rep. EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[3] OpenHW Group, "Core-v extension interface (cv-x-if) documentation — preface." `https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/preface.html`, 2024. Accessed: 2026-01-10.

[4] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, pp. 2629–2640, Nov 2019.

[5] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2020.

[6] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Transactions on Computers*, 2020.

[7] OpenHW Group, "Open bus interface (obi) v1.0 documentation." `https://github.com/openhwgroup/obi/blob/main/OBI-v1.0.pdf`, 2020. Accessed: 2026-01-10.

[8] A. Traber, M. Gautschi, and P. D. Schiavone, *RI5CY: User Manual.* PULP Platform, rev. 4.0 ed., Apr. 2019. [Online]. Available: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf.

[9] ARM Limited, "ARM AMBA 5 AHB protocol specification AHB5, AHB-lite." `https://documentation-service.arm.com/static/5f91607cf86e16515cdc3b4b?token=`, 2015. Accessed: 2026-01-11.

[10] ARM Limited, "AMBA AXI protocol specification (IHI0022)." `https://developer.arm.com/documentation/ihi0022/latest/`, 2025. Accessed: 2026-01-11.

[11] S. Machetti, P. D. Schiavone, G. Ansaloni, M. Peón-Quirós, and D. Atienza, “X-heep: An open-source, configurable and extendible risc-v platform for tinyai applications,” in *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2025.

[12] M. Platzer and P. Puschner, “Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation,” in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)* (B. B. Brandenburg, ed.), vol. 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 1:1–1:18, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[13] N. Purayil, M. Perotti, T. Fischer, and L. Benini, “AraXL: A physically scalable, ultra-wide RISC-V vector processor design for fast and efficient computation on long vectors,” in *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pp. 1–7, 2025.

[14] M. Johns and T. J. Kazmierski, “A minimal risc-v vector processor for embedded systems,” in *2020 Forum for Specification and Design Languages (FDL)*, pp. 1–4, 2020.

[15] Vicuna Documentation Contributors, “Vicuna - a flexible and scalable RISC-V vector coprocessor.” `https://vicuna.readthedocs.io/en/latest/index.html`, 2025. Accessed: 2026-01-12.

[16] M. Cavalcante, M. Perotti, S. Riedel, and L. Benini, “Spatz: Clustering compact risc-v-based vector units to maximize computing efficiency,” Sept. 2023.

[17] “Ieee standard for systemverilog—unified hardware design, specification, and verification language,” 2018.

[18] RISC-V Vector C Intrinsic Task Group, “Risc-v vector c intrinsic specification document.” RISC-V International, Apr. 2025. Version 1.0, 25 April 2025.

[19] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, *et al.*, “Mlperf tiny benchmark,” *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[20] United Nations, “Communications materials - United Nations Sustainable Development,” 2026. Accessed: Jan. 27, 2026.

TIME PLANNING AND BUDGETING

Figure 6.5 provides an overview of the main tasks carried out throughout the development of this master's thesis. The overall project schedule and the time allocation for each task are illustrated.

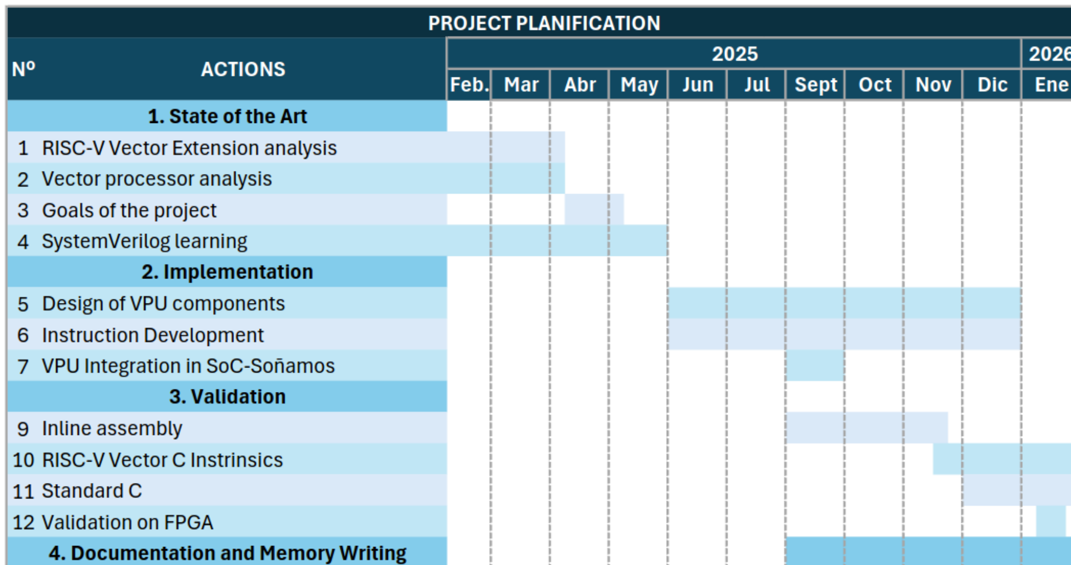

Figure 6.5: Gantt diagram of the project.

Table 6.2 summarises the estimated costs associated with the development of this master's thesis, considering only the most relevant resources.

Material	**Price/Unit(€)**	**Quantity**	**Total Price**
Xilinx Pynq-z2 FPGA	190 €	1	190 €
Vivado Licence	3.679,49 €	1	3.679,49 €
QuestaSim	0 €	1	0 €
Engineer hours	20 €	400	8.000
TOTAL	-	-	11.869,49 €

Table 6.2: Project budget.

SOCIAL AND PROFESSIONAL RESPONSIBILITY

The development of high-performance hardware architectures is linked to global challenges regarding energy consumption and the accessibility of technological infrastructure. As computational requirements for data-intensive applications, such as AI and radar signal processing, increase, the professional responsibility of the hardware engineer focuses on the creation of efficient and scalable systems. Utilising the RISC-V ISA and the Zve extension allows a research path towards high-throughput processing within the limited power envelopes of the embedded domain. This methodology employs an open-standard architecture to support technical reliability and international innovation.

This chapter evaluates the project's alignment with the United Nations Sustainable Development Goals (SDGs). The following sections analyse how the technical objectives of this thesis, specifically the reduction of instruction overhead and the use of vendor-independent hardware, contribute to global targets for industrial growth, energy efficiency, and the development of shared technical foundations.

Figure 6.6: The United Nations Sustainable Development Goals framework [20].

- **SDG 7: Affordable and Clean Energy** The primary technical objective of this vector accelerator is the advancement of computational efficiency, which remains

a critical professional responsibility in an era of rising computational energy costs. By implementing a vectorised architecture, the design targets a higher throughput of operations per clock cycle compared to scalar alternatives. Theoretically, vectorisation facilitates a higher "work per watt" ratio by reducing the instruction overhead.

Consequently, this research supports the global transition toward more sustainable computing by providing a technical reference for high-performance hardware that minimises the environmental impact of intensive data processing and AI workloads.

- **SDG 8: Decent Work and Economic Growth**

  This project contributes to economic productivity by developing specialised technical knowledge in the expanding field of open-standard hardware. By focusing on the RISC-V vector accelerator, the research supports an ecosystem that lowers the barriers to entry for smaller organisations and startups, who are often excluded by the prohibitive costs of proprietary silicon licences. The work encourages innovation focused on technical performance rather than capital-intensive development.

  Furthermore, the use of an FPGA for testing establishes a reproducible hardware-verification framework, allowing for rigorous testing of logic in a physical environment. This supports the development of advanced engineering capable of designing independent silicon solutions, which is essential for long-term economic stability and the creation of high-technology jobs in a diversifying global semiconductor market.

- **SDG 9: Industry, Innovation, and Infrastructure**

  This project supports the development of inclusive and scalable technological systems by adopting the open-standard RISC-V ISA. Traditional hardware ecosystems are frequently restricted by "vendor lock-in," which limits the flexibility and security of global computing frameworks. This research addresses these limitations by utilising open-source hardware description languages and public development tools, ensuring the design remains standards-compliant, vendor-neutral, and functionally consistent across diverse systems.

  Validating the accelerator on an FPGA confirms its portability, demonstrating that high-performance computation can be deployed across various hardware platforms without dependency on a single manufacturer. This methodology accelerates industrial progress by proving that standardised processor architectures can be successfully adapted for specialised tasks, thereby supporting the wider commercial adoption of open hardware frameworks in professional engineering.

- **SDG 17: Partnerships for the Goals**

  This project aligns with the objectives of global cooperation by engaging with the technical standards established by the international RISC-V community.

Traditionally, advanced hardware development was restricted to specific private entities, whereas this research utilises a collaborative framework where technical specifications are accessible through open documentation and shared repositories.

By evaluating and testing implementations based on RVV, the research validates a standardised toolset. The professional responsibility involves adhering to these universal specifications, which supports a wider network of academic and industrial interests. This methodology ensures that advancements in processor technology are based on shared technical foundations, encouraging a more balanced distribution of technological progress across the engineering sector.

List of Figures

1.1 Element mapping within vector registers as a function of LMUL, SEW, and VL being VLEN = 128. 15

1.2 Replication of scalar/immediate operator over the input vector. Example of vx instruction: SEW = 16, VL = 4. 17

1.3 Application of tail and mask policies to vector elements. 18

1.4 CV-X-IF handshake during vector instruction requests indicated by `xif_issue_if.issue_valid`. Request 1: Vector configuration instruction for VLEN = 32 and SEW = 16. Returning VL = 2 to the scalar core. Request 2: `vadd.vx` instruction. The two 16-bit elements contained on the vector register (5 and 2) and the scalar register data (0xDEAD) are summed. 23

1.5 OBI handshake protocol timing. 26

1.6 Internal architectural block diagram of the X-HEEP platform [11]. 27

2.1 Execution pattern of an array processor [12]. 29

2.2 Execution pattern of an vector processor [12]. 30

2.3 Top-level block diagram of Ara [5]. (a) Block diagram of an Ara instance with N parallel lanes. (b) Block diagram of one lane of Ara. 31

2.4 Top-level block of Ara XL and its interfaces [13]. 32

2.5 Schematic of the integrated vector processor [14]. 33

2.6 Overview of Vicuna architecture integrated with Ibex core [12]. 34

2.7 Schematic of the Spatz CC [16]. (a) Shared L1-Cluster with two PUs. (b) Spatz microarchitecture. 35

3.1 Vector accelerator and X-HEEP integration in X-Fusion. 38

3.2 VPU Finite State Machine. 41

3.3 Indexed access and data transfer to internal buffer and the scalar load-store unit. 43

3.4 Misaligned access to memory. 45

3.5 VAU and SIMD lanes distribution. 47

3.6 `vmv.x.s rd, vs2` instruction moving element 0 from VRF to scalar register. `vmv.s.x vd, rs1` instruction moving scalar register data to element 0 of the VRF. 48

3.7 `vmv2r.v vd, vs2` instruction moving two whole vector registers within the VRF. 48

5.1 Execution time comparison between unit-stride (VLE32) and strided (VLSE32) loads for SEW=32 and VL=4. Both instructions execute in 8 clock cycles. 61

5.2 PYNQ-Z2 FPGA floorplan with X-HEEP(blue) and VPU (green) resource blocks. 65

6.1 OBI multi-port configuration. 68

6.2 Modular cluster architecture for tile-level scalability. 68

A.1 Vector Load Instruction Format under LOAD-FP major opcode [1]. 69

A.2 Vector Store Instruction Format under STORE-FP major opcode [1]. . . . 69

A.3 Vector Arithmetic Instruction Format under OP-V major opcode [1]. . . . 70

A.4 Vector Configuration Instruction Formats under OP-V major opcode [1]. . 70

6.5 Gantt diagram of the project. 75

6.6 The United Nations Sustainable Development Goals framework [20]. 77

List of Tables

1.1 Main Characteristics and Parameters of the RVV. 14

1.2 Vector tail and mask policy configurations. 18

1.3 Issue Interface signals [3]. 21

1.4 Register Interface signals [3]. 21

1.5 Commit Interface signals [3]. 21

1.6 Result Interface signals [3]. 22

1.7 OBI Port List [7] 25

2.1 Comparison of state-of-the-art vector processors and the proposed Vector Processing Unit design. 36

3.1 Main Characteristics the vector accelerator. 40

5.1 SAXPY Results: Execution Cycles. 60

5.2 Index Arithmetic Results: Execution Cycles. 61

5.3 MatMul Results: Execution Cycle Comparison 62

5.4 Indexed Algorithm Execution Cycle Comparison with 'RVV C Instrinsics". 62

5.5 Resource utilisation of the vector accelerator sub-modules in PYNQ-Z2 FPGA. 64

5.6 Logic utilisation of primary sub-modules and total FPGA resource consumption for X-Fusion architecture. 64

A.1 Implemented Vector Accelerator Instructions. 71

6.2 Project budget. 75

ABBREVIATIONS

AI Artificial Intelligence
AHB Advanced High-performance Bus
ALU Arithmetic Logic Unit
AMBA Advanced Microcontroller Bus Architecture
APU Auxiliary Processing Unit
ASIC Application-Specific Integrated Circuit
AVL Application Vector Length
AXI Advanced eXtensible Interface
CC Core Cluster
CPU Central Processing Unit
CSR Control and Status Register
CV-X-IF Core-V eXtension Interface
DMA Direct Memory Access
EEW Effective Element Width
ELEN Element Length
FPGA Field-Programmable Gate Array
FPU Floating-Point Unit
FSM Finite State Machine
GLSU Global Load Store Unit
GCC GNU Compiler Collection
GPR General-Purpose Register
HDL Hardware Description Language
HPC High-Performance Computing
I2C Inter-Integrated Circuit
IoT Internet of Things
IP Intellectual Property
ISA Instruction Set Architecture
LMUL Vector Register Grouping

LSU Load-Store Unit
NiP Network-in-Package
MCU Microcontroller
ML Machine Learning
MUL Multiplier
OBI Open Bus Interface
PU Processing Unit
REQI Request Interface
RINGI Ring Interface
RoCC Rocket Custom Coprocessor Interface
ROM Read Only Memory
RTL Register Transfer Level
RVV RISC-V Vector Extension
SDG United Nations Sustainable Development Goal
SEW Selected Element Width
SIMD Single Instruction, Multiple Data
SiP System in Package
SoC System-on-Chip
SPI Serial Peripheral Interface
SRAM Static Random Access Memory
SVA SystemVerilog Assertions
UART Universal Asynchronous Receiver-Transmitter
VAU Vector Arithmetic Unit
vill Vector Instruction Length Illegal
VL Vector Length
VLMAX Maximum Number of Elements
VLEN Vector Register Length
VLENB Vector Register Length in Bytes
vma Vector Mask Agnostic
VPU Vector Processing Unit
VRF Vector Register File
vta Vector Tail Agnostic
X-HEEP eXtendable Heterogeneous Energy-efficient Platform
X-IF Core-V eXtension Interface