DE-42 (DC & SE)

NUST COLLEGE OF
ELECTRICAL AND MECHANICAL ENGINEERING

VECTOR PROCESSING UNIT INTEGRATED RISC-V PROCESSOR

A PROJECT REPORT

DE-42 (DC & SE)

Submitted by

NS MUHAMMAD NADEEM

NS AHSAN ALI

NS SHAHZAD AKHTER

NS MUHAMMAD HARIS

BACHELORS

IN

COMPUTER ENGINEERING

YEAR

2024

PROJECT SUPERVISOR

DR. SAJID GUL KHAWAJA

A/P DR. MUHAMMAD YASIN

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING
COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY,
ISLAMABAD, PAKISTAN

YEAR 2024

Certification

This is to certify that Muhammad Nadeem, 345096 and Ahsan Ali, 340661 and Shahzad Akhter, 356285, and Muhammad Haris, 342045 have successfully completed the final project Vector Processing Unit Integrated RISC-V Processor, at the College of Electrical and Mechanical Engineering,(CEME) to fulfill the partial requirement of the degree 42.

Sajid
Signature of Project Supervisor
Dr. Sajid Gul Khawaja
Associate Professor

Sustainable Development Goals (SDGs)

SDG No	Description of SDG	SDG No	Description of SDG
SDG 1	No Poverty	**SDG 9**	**Industry, Innovation, and Infrastructure**
SDG 2	Zero Hunger	SDG 10	Reduced Inequalities
SDG 3	Good Health and Well Being	SDG 11	Sustainable Cities and Communities
SDG 4	Quality Education	SDG 12	Responsible Consumption and Production
SDG 5	Gender Equality	SDG 13	Climate Change
SDG 6	Clean Water and Sanitation	SDG 14	Life Below Water
SDG 7	Affordable and Clean Energy	SDG 15	Life on Land
SDG 8	Decent Work and Economic Growth	SDG 16	Peace, Justice and Strong Institutions
		SDG 17	Partnerships for the Goals

Sustainable Development Goals

Complex Engineering Problem

Range of Complex Problem Solving

	Attribute	**Complex Problem**	
1	Range of conflicting requirements	Involve wide-ranging or conflicting technical, engineering and other issues.	
2	Depth of analysis required	Have no obvious solution and require abstract thinking, originality in analysis to formulate suitable models.	
3	Depth of knowledge required	Requires research-based knowledge much of which is at, or informed by, the forefront of the professional discipline and which allows a fundamentals-based, first principles analytical approach.	×
4	Familiarity of issues	Involve infrequently encountered issues	×
5	Extent of applicable codes	Are outside problems encompassed by standards and codes of practice for professional engineering.	
6	Extent of stakeholder involvement and level of conflicting requirements	Involve diverse groups of stakeholders with widely varying needs.	
7	Consequences	Have significant consequences in a range of contexts.	×
8	Interdependence	Are high level problems including many component parts or sub-problems	

Range of Complex Problem Activities

	Attribute	**Complex Activities**	
1	Range of resources	Involve the use of diverse resources (and for this purpose, resources include people, money, equipment, materials, information and technologies).	×
2	Level of interaction	Require resolution of significant problems arising from interactions between wide ranging and conflicting technical, engineering or other issues.	×
3	Innovation	Involve creative use of engineering principles and research-based knowledge in novel ways.	
4	Consequences to society and the environment	Have significant consequences in a range of contexts, characterized by difficulty of prediction and mitigation.	
5	Familiarity	Can extend beyond previous experiences by applying principles-based approaches.	

Dedicated to our parents, teachers, and family for their support, inspiration, and guidance.

Acknowledgment

We begin by expressing our gratitude to Allah, whose boundless love and generosity have showered blessings upon us, providing the strength and determination to successfully complete this assignment.

We extend our sincere thanks to NUST (CEME) for providing a conducive academic environment for our growth.

We deeply appreciate Dr. Sajid Gul Khawaja, our supervisor, whose insightful guidance and encouragement have played a pivotal role in our academic journey. His motivating lectures and unwavering support have been invaluable throughout this project. Dr. Khawaja's constructive criticism has challenged us to enhance our critical thinking skills and fulfill our research goals.

We would also like to acknowledge Dr. Muhammad Yasin, our co-supervisor, for his helpful advice and recommendations, which have contributed significantly to the development of this work.

Furthermore, our heartfelt gratitude goes to our parents, whose unwavering support, love, and unshakable faith in our abilities have been the pillars of our perseverance and strength. Their encouragement has been a beacon of hope, instilling the courage and tenacity to pursue our goals. We are forever thankful for their selfless commitment to our well-being and careers, serving as a continuous source of inspiration.

Abstract

The open-source RISC-V instruction set architecture (ISA) is thoroughly examined in this thesis, emphasizing the ISA's special position as a result of its transparent development process. We investigate the unique security issues that RISC-V has because it is open-source and the several solutions that have been put in place to keep RISC-V secure. ISA extensions and Physical Memory Protection (PMP) are examples of hardware security measures. Cryptographic modifications are also made to ISA to protect RISC-V processors from potential vulnerabilities.

The paper highlights the security risks associated with the growing trend of bespoke chip designs that leverage the RISC-V ISA. It also looks at this trend in more detail. It should be noted that to improve security, commercial versions of RISC-V frequently include PMP and hardware security extensions.

The dissertation assesses the incorporation of a cryptographic core into the Picorv32 CPU, a design decision that increases security but consumes a large amount of space. A novel method is also investigated, namely the Custom Co-Processor (CCoP) equipped with a customized interface for the Picorv32. By shifting workloads to the co-processor, this technique uses ISA changes to facilitate CCoP operations, thereby lowering code complexity and improving performance significantly—particularly for activities like encryption and decryption. The incorporation of these customized instructions not only improves the design's latency but also boosts core performance, providing a potential remedy for the efficiency and security issues that come with RISC-V based systems.

This thesis serves as a vital resource for professionals and researchers in the field, providing insights into the security of the RISC-V CPU landscape. It establishes the framework for next research and advancements in protecting RISC-V based systems, guaranteeing their adaptability to changing security risks in the digital era.

Key words: custom ISA, loosely coupled, RISC-V, FPGA, security.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Looking back more than ten years, the University of California, Berkeley launched the open-core processor initiative to provide academics and developers access to design modifications [2]. The goal of the RISC-V (Reduced Instruction Set Computing Five) open-core processor project is to create an extendable and open-source instruction set for business and academic uses [3].

In recent times, the industry has embraced the RISC-V processor[4] and its instruction set architecture (ISA) specifications[5], which have roots in both academia and industry [6]. Numerous open-source and proprietary RISC-V implementations have surfaced, challenging the dominance of industry giants such as Advanced RISC Machine (ARM), Intel, and AMD.

The RISC-V Architecture offers an open-source implementation to rival these industry titans. Conventional processors often lack design flexibility and are proprietary, necessitating licensing or authorization costs. These limitations increase the price of developing new processors and add to the challenge of gaining a substantial market share [7].

This is where open-source RISC-V processors come into play. These implementations enable the reusability of projects, where project-specific requirements are typically satis-

fied by unique architectural modifications and extensions. Due to this feature, RISC-V is an ideal choice for various niche applications specific to different sectors. For example, RISC-V is used in low-power Internet-of-Things (IoT) applications [8, 9] and computationally expensive healthcare applications [10], as well as in geographical areas with strict dependability requirements [11, 12]. RISC-V finds applications in image processing, machine learning (ML), and artificial intelligence (AI) [13, 14, 15], high-performance computing (HPC) [16], and many more.

Table 1.1: RISC-V boards available in market and their features, applications and support of operating system

Platforms	**SoC & Processor**	**ARM Peer**	**Target App**	**Operating System Support**
ICE EVB	XuanTie C910; 64 - bit Dual cores 1.2GHz	Cortex-A55	5G , AI, Mobile	Linux , Android
HiFive Unmatched	SiFive U740; 64 - bit Quad cores 1.4GHz	Cortex-A55	Generic PC	Linux
HiFive Unleashed	SiFive U54; 64 - bit Quad cores 667MHz	Cortex - A53	AI, IoT	Linux, VxWorks
BeagleV	SiFive U74; 64 - bit Dual cores 1.0GHz	Cortex - A55	AI	Linux , Zephyr
PolarFire Icicle	SiFive U54; 64 - bit Quad cores 667MHz	Cortex - A53	AI, IoT	Linux, seL4
Kendryte KD233	Kendryte K210; 64 - bit Dual cores 400MHz	Cortex - M7	AI, IoT	FreeRTOS
HiFive 1 RevB	SiFive E310; 32 - bit core 320MHz	Cortex-M4	IoT	Bare-metal, embOS, FreeRTOS, Mynewt, RT - Thread, Zephyr
Gigadevice RV-STAR	GD32VF103; 32 - bit core 108MHz	Cortex-M3	Low Power	Bare-metal

The RISC-V ecosystem, by creating software compilers, system-on-a-chip (SoC) peripherals, and other related components, aids in simplifying FPGA or ASIC-based CPUs. Consequently, numerous low-scale Internet of Things chips, such as the HiFive and HiFive1 Rev B SoCs from SiFive [17], have been taped out for desktop and IoT applications [18]. Examples of other tapouts include the Alibaba-produced Xuantie-910 cloud and edge computing tapout [19], the Microchip PolarFire SoC FPGA Icicle Kit, and the Black-

parrot multi-core accelerator [20]. These implementations effectively portray RISC-V as a product rather than just an idea. Table 1.1 shows an overview of the platforms.

1.2 Emergence of RISC-V and Open-Source Software Environment

With the emergence of RISC-V and the creation of an open-source software environment that anybody may join, it is predicted that the true paradigm will change. The main goal is to enable various implementations, giving companies, educational institutions, or research centers the flexibility to develop their hardware solutions with the knowledge that this software layer is accessible and may be utilized without restrictions. If RISC-V proves to be effective, it could eventually be found in processors across several platforms, including computers, smartphones, and other kinds of microcontrollers [7].

1.3 Motivation

Teachers have been debating what and how to teach computer architecture and organization[21] for decades. One of the first things these courses ask, for instance, is what architecture to use as the model system. Various textbooks make use of ARM[22], PIC[23], and x86[24], among others, and these are only a small sample of different architectures (e.g., LC-2, MIPS, and Hack).

After selecting an architecture, the next step is to determine how to analyze the system. Some options include using simulators[25]-[26] for assembly programming, FPGA processor implementations, or processor function experiments. Following these choices, students can complete coursework in various ways, including lectures, assignments, tests, and other undertakings. As universities look to student-centered approaches to learning, project-based learning and experiential learning have recently seen a resurgence, and computer architecture is no exception to these trends.

The ISA implementation establishes the power and weakness of a CPU. ISA-based processor selection considers application-driven constraints such as power consumption, performance, ease of design, etc. In this sense, there are many implementations of RISC-V with distinct trade-offs. Research projects often present comparative outcomes based on certain ASIC technology tapout or FPGA, and these outcomes are significantly influenced by design parameters that differ among research groups (e.g., cache size).

Students and academics can investigate several design concepts for enhancing CPU efficiency with RISC-V. This is due to several important factors, such as the design flexibility of RISC-V, which can be tailored to a wide range of devices and applications thanks to its highly modular and customizable architecture; the openness of its source codes, which allow anyone to use it without having to pay licensing fees or obtain permission from a proprietary vendor; and the architecture's base performance, attributed to RISC-V's highly efficient instruction set and streamlined design. RISC-V provides designers with an open-source, shared ISA, freeing them up to concentrate on developing cutting-edge new hardware and software, as opposed to investing time and energy in circumventing proprietary limitations.

1.4 Problem Statement

The growing volume and complexity of data in today's computer tasks—particularly in domains like artificial intelligence, data analytics, and scientific computing—makes vector processors essential. Large data sets can be processed concurrently by vector processors, which take advantage of parallelism to greatly speed up calculations. In scientific research, data analysis, and machine learning, tasks like matrix operations, signal processing, and simulations are common and require this capability to be handled well. Furthermore, by maximizing the use of computing resources, vector processors enhance energy efficiency and are therefore perfect for tasks running in low-power contexts like embedded systems and mobile devices.

Furthermore, vector processors' performance and scalability make them essential for fulfilling the demands of high-performance computing activities in a variety of disciplines, spurring innovation and improvements in computational capabilities. In general, the demand for vector processors arises from their capacity to effectively handle vast amounts of data and carry out intricate calculations, facilitating advancements in science, technology, and computational capacities in various domains.

1.5 Objectives

The objective of our final year project are as follows:

- To understanding and simulation of VPU
- To design and Implementation of FPGA supported VPU
- Integration of VPU with RISC-V
- Execution on FPGA and analysis of the final core

1.6 Organization of Report

This thesis report is divided into the following chapters. The remaining of this document is organized as follows:

- Chapter 2 covers the RISC-V ISA, existing extensions, and analysis/comparison of RISC-V cores.
- Chapter 3 discusses design of our proposed vector processing unit and its implementation aspects.
- Chapter 4 outlines the integration process which eenabled RISC-V PICORV32 core to utilize the proposed VPU.
- Chapter 5 discusses the cross-compilation toolchain which is required for analysis

in terms of latency and line of code.

- Finally, Chapter 6 concludes the project report while higlighting future directions from this work.

Chapter 2

Background and Related Work

The RISC-V ISA specification is analyzed at the beginning of this chapter, with a focus on the base integer instruction set, current extensions, and their corresponding operational mechanisms. Afterward, a number of previously developed RISC-V cores will be examined and contrasted. To identify what is missing from the solutions that are now in use and what needs to be added to a new one, it is crucial to understand the capabilities and limitations of each core. The main areas of emphasis for this evaluation will be FPGA support, the caliber of the documentation, and how simple it is to adapt and modify the cores [27]. Since proprietary ISAs are the intellectual property of the companies that possess them, users' ability to modify hardware is limited [28]. First of all, utilizing proprietary ISAs may require purchasing a license, which might cost several million dollars [28]. Secondly, it is often prohibited under these agreements for licensees to alter the circuitry and hardware [28, 29]. Hardware manufacturers and builders may find it difficult to employ proprietary ISAs due to these features [27]. The Intel x86 architecture, which rules the desktop PC market, and the ARM architecture, which is used in embedded mobile devices, are contrasted with the RISC-V architecture, which offers the strong advantages of simplicity, flexibility, customizability, high energy efficiency, and is free and open-source.

2.1 RISC-V

As the name implies, RISC-V is a base-integer ISA that is based on a reduced instruction set computer (RISC) [30]. The load-store design of RISC-V [31] allows for a clear division between instructions that perform calculations only in memory and those that additionally reduce the number of instructions per program by lowering the number of cycles per instruction. This contrasts with CISC (complex instruction set computer) designs, in which the hardware may further subdivide one instruction into several sub-instructions. Figure 2.1 provides a summary of the RISC-V ecosystem and its associated utilities.

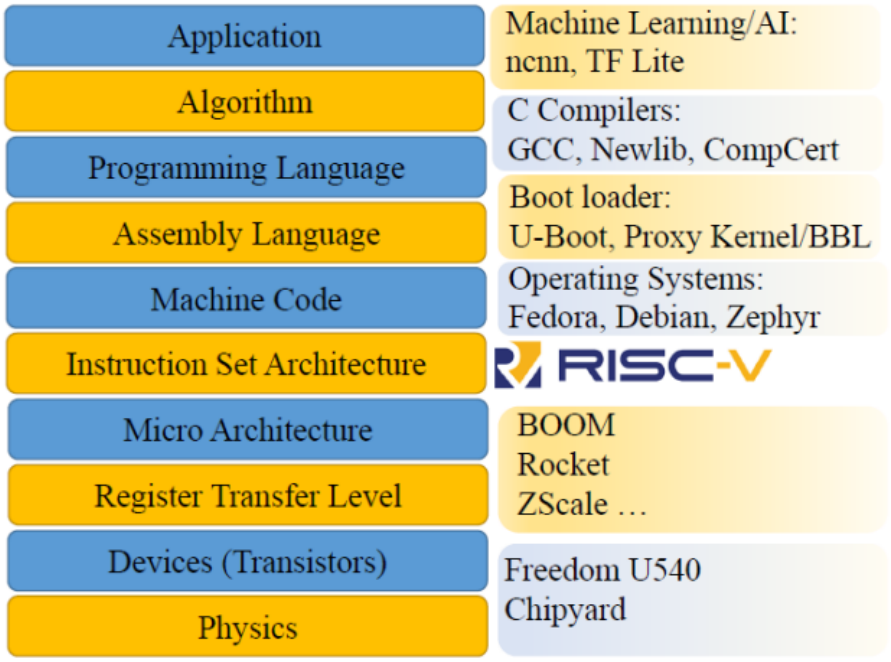

Figure 2.1: Summary of the RISC-V ecosystem and its associated utilities.

2.1.1 RISC-V Instruction Set Architecture (ISA)

The RISC-V architecture requires a fundamental integer instruction set (ISA) for all implementations, with optional additions that make up the ISA. Like early RISC processors, the basic integer ISA has variable-length instruction encoding[32] options but no branch delay slots. A basic set of instructions is carefully selected to create an ISA and soft-

Table 2.1: Summary of naming convention used in the RISC-V ecosystem

Nomenclature	Details
*RV	(RISC-V)
*32I	(32-bit integer)
*32E	(32-bit integer -embedded,16registers)
*64I	(64-bit integer)
*128I	(128-bit integer)

ware toolchain "skeleton" that provides a strong foundation for software development. Compilers, assemblers, linkers, and operating systems can all aim to support this skeleton, which also includes additional supervisor-level functions. RISC-V supports four ISA variations, identified by the number and breadth of integer registers they support, as well as optional extensions that provide additional features such as division, multiplication, and floating-point operations. Because of its modular architecture, implementations may devote resources only to the functionality that is really needed, allowing for maximum energy economy and customization for particular applications. The version and extensions of a RISC-V ISA implementation are denoted by naming conventions. For example, "RV32I" is the designation given to a 32-bit basic integer ISA implementation with 32 registers; extra letters are attached to denote any extensions utilized. Some of the well-known nomenclatures and naming conventions are mentioned in Table 2.1.

Specific performance needs may be met with flexibility when building implementations because of RISC-V's ability to keep standard extensions, additional extensions, and the underlying integer ISA separate [33]. Because new extensions may be developed to accommodate evolving application demands without compromising backward compatibility, this technique also facilitates easy scaling. Additionally, by combining several functions into a single instruction set architecture, this modularity lowers the possibility of mistakes and streamlines the design process.

2.1.1.1 Base Integer Instruction Set

With no branch delay slot, RISC-V basic integer ISA can be used as a general-purpose plus/or optional extension to the base ISA. It also supports variable length instructions that encode 32-, 64-, and 128-bit variations of the base ISA [33]. RISC-V's basic integer instruction set is designated as RV32I for 32-bit address spaces and RV64I for 64-bit address spaces, respectively. The majority of typical operations performed in general-purpose computing are covered by its 47 instructions. An instruction set that is suitable for implementing operating systems, higher-level programming languages, and other software applications is offered by the RV32I and RV64I. Optional instruction extensions can be added to the underlying ISA to enable significant customisation and specialisation [33]. To further improve the capabilities of the CPU, further instruction set extensions, such as the vector (V), floating-point (F), and multiplication and division (M) extensions, can be introduced as needed.

The six formats for instructions that are 32 bits long are R, I, S, B, U, and J. The order of the immediate word inside the instruction is the sole difference between the S and B forms. This also applies to the U and J forms. R-type, I-type, S-type, and U-type instructions are the four general types of instructions that RISC-V ISA provides. These are given below: Every kind of instruction has a unique operand set and opcode that designate the operation to be carried out.

- R-type instructions use two registers to carry out logical and arithmetic operations. For instance, `ADD rd, rs1, rs2`.
- Arithmetic and logical operations are carried out instantly via I-type instructions. For instance, `ADDI rd, rs1, imm`.
- S-type instructions use memory to save data from a register. For instance, `SW rs2, imm(rs1)`.
- Immediate values are loaded into a register via U-type instructions. For instance,

`LUI rd, 10 imm.`

The RISC-V architecture's base instruction format is seen in Figure 2.2. The registers are in the same place for all formats. There are several methods to encode it, depending on the format [16].

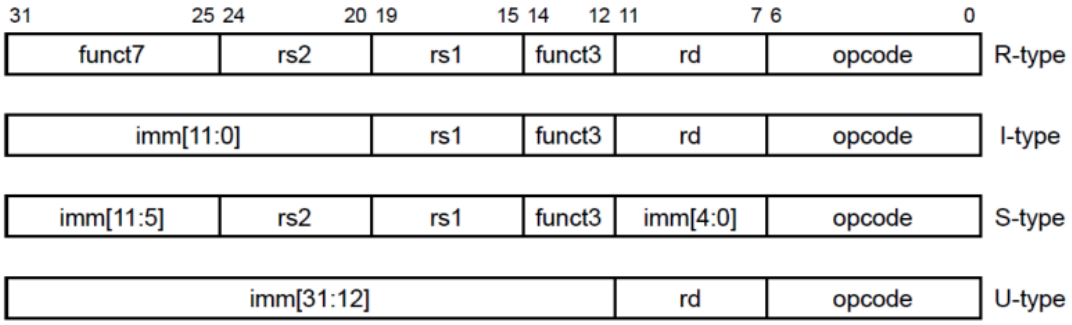

Figure 2.2: RISC-V base instruction formats [2]

Instructions for the following load and store operations are part of the RISC-V basic integer instruction set:

- LW, SW, LH, SH, LB, SB, LBU, LHU

Arithmetic and logic operations include:

- ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU

Immediate operations include:

- ADDI, ANDI, ORI, XORI, SLLI, SRLI, SRAI, SLTI, SLTIU

Control flow procedures include:

- BEQ, BNE, BLT, BGE, BLTU, BGEU, JAL, JALR, BEQ

Other operations include:

- NOP, AUIPC, LUI

Designed to be easy to use, effective, and expandable, the RISC-V base integer ISA offers a standard framework for program development on RISC-V processors. Compilers,

assemblers, and other software tools for the RISC-V architecture are simple to create and optimize because of its clear set of instructions, which can be effectively implemented in both hardware and software. The basic integer ISA makes guarantee that software created for one RISC-V processor may be readily transferred to another by offering a uniform foundation for RISC-V processor software implementation. This speeds up innovation and makes software development simpler. Furthermore, the base integer ISA is purposefully intended to be expandable, allowing for the inclusion of new features and instructions as needed. As a result, the design may be modified to accommodate new computer environments and developing applications. To summarize, the base integer of RISC-V Because of its ease of use, effectiveness, and expandability, ISA is a great place to start when developing RISC-V processors and software applications. Its versatility allows for adaptation to evolving computing demands, and its succinct set of instructions offers a strong basis for creating operating systems, other software applications, and higher-level programming languages.

2.1.1.2 Extensions

The open-source RISC-V Foundation is always creating new extensions to increase the functionality of its instruction set design. A complete list of all the extensions that are presently being developed, including those that have been authorized and those that are still in the draft stage, may be seen in Table 2.2. Integer multiplication and division, Control and Status Register instructions, and single, double, and quad floating-point are among the authorized additions. It is anticipated that updates and revisions will be made soon because a number of these extensions are still in the draft stage and under development. These additions are being developed with an eye on boosting security, cutting power consumption, and meeting the changing demands of the computer industry. A few of the additions that are presently in the draft stage are hardware virtualization and vector processing. These enhancements could completely change the way computers handle data, increasing processing speed and efficiency while protecting the security and integrity

of sensitive data. Summary of the extensions is given in Table 2.2.

Table 2.2: Extensions available for RISC-V ISA [1]

Extension	Description	Version	Status
Zifencei	Instruction - Fetch Fence	2.0	Ratified
Zicsr	Control and Status Register (CSR) Instructions	2.0	Ratified
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified
A	Standard Extension for Atomic Instructions	2.0	Frozen
F	Standard Extension for Single - Precision Floating - Point	2.2	Ratified
D	Standard Extension for Double - Precision Floating - Point	2.2	Ratified
Q	Standard Extension for Quad - Precision Floating - Point	2.2	Ratified
C	Standard Extension for Compressed Instructions	2.0	Ratified
Ztso	Standard Extension for Total Store Ordering	0.1	Frozen
Counters	Performance Counters and Timers	2.0	Draft
L	Standard Extension for Decimal Floating - Point	0.0	Draft
B	Standard Extension for Bit Manipulation	0.0	Draft
J	Standard Extension for Dynamically Translated Languages	0.0	Draft
T	Standard Extension for Transactional Memory	0.0	Draft
P	Standard Extension for Packed - SIMD Instructions	0.2	Draft
V	Standard Extension for Vector Operations	0.7	Draft
N	Standard Extension for User - Level Interrupts	1.1	Draft
Zam	Standard Extension for Misaligned Atomics	0.1	Draft

A few of the enhancements, like as the scalar cryptography instructions, are aimed at improving RISC-V's security [34]. The RISC-V Foundation has set very high requirements, and all additions have been rigorously tested and confirmed to fulfil those criteria.

To sum up, the RISC-V Foundation is dedicated to developing and growing its instruction set architecture throughout time, with an emphasis on cooperation and creativity. The extensions that are being created present fascinating prospects for computing's future, and in the upcoming years, even more developments should be anticipated. A summary of some of the main benefits that the community receives from RISC-V is given in Table 2.3.

Table 2.3: Summary of some of the key features of the RISC-V instruction set architecture

Feature	Description
Open standard	RISC-V is an open, royalty-free ISA developed by the RISC-V Foundation.
Modular design	RISC-V is designed with a modular structure, with optional extension modules.
Fixed instruction length	RISC-V instructions are of fixed length, making decoding simple.
Reduced instruction set	RISC-V has a reduced instruction set, with a focus on simple, orthogonal instructions.
User-level ISA	RISC-V has a user-level ISA for general-purpose computing.
Privileged ISA	RISC-V has a privileged ISA for system-level operations.
32/64-bit support	RISC-V supports both 32-bit and 64-bit architectures.
Scalable vector extension	RISC-V has a scalable vector extension for efficient vector processing.
Multiple implementations	RISC-V has multiple implementations from various vendors, providing flexibility and choice.

2.2 RISC-V Cores

Many companies have created numerous RISC-V basic ISA cores that may be utilized as co-processor design, advanced SoCs, microcontrollers, and even more sophisticated implementations and integrations of customized accelerators. A number of them were chosen and contrasted for examination in this part on the basis of their characteristics, highest frequency of operation, support for FPGA, resource consumption, and documentation accessibility. The RISC-V cores listed below are examined and contrasted:

- PULPino
- VexRISC-V
- PicoRV32
- Rocket Chip
- ORCA
- SweRV

2.2.1 PicoRV32

The goal of the PicoRV32 is to act as an auxiliary CPU for FPGA and ASIC designs by reducing size and increasing achievable clock rates. It is an in-order, non-pipelined processor based on the RISC-V instruction set architecture [4].

An easy-to-use memory interface is provided by this 32-bit RISC-V core [36], which is written in Verilog and supports the following ISAs: integer reduced (RV32E), integer (RV32I), integer compressed (RV32IC), and integer multiplication and division (RV32IM).

The Pico Co-processor Interface (PCPI), an optional add-on for the PicoRV32 processor, functions as a closely connected 17-coupled accelerator to support the RISC-V ISA's multiplication and division operations [4].

Three versions of the PicoRV32 core are available: Advanced 10 extensible Interface (AXI) Lite, basic memory interface, and hybrid. PicoRV32 WB and the master interface for connecting peripherals. It employs about 760, 920, and 2020 Look-up Tables (LUTs) on Xilinx 7-Series FPGAs concurrently and offers a Wishbone master interface based on small, regular, and large configurations. According to the timing evaluation, the Xilinx Virtex UltraScale+ board with the xcvu3p-ffvc1517-3-e FPGA may reach a maximum frequency of 714 MHz. Its fundamental design is not described, despite the abundance of technical data on its features and assessment reports that are available [16].

An optional Pico Co-processor Interface (PCPI) on this processor acts as a tightly coupled accelerator for the multiplication and division instructions of the RISC-V ISA. This interface is required in order to enable the RISC-V "M" extension, which allows integer multiplication and division [4].

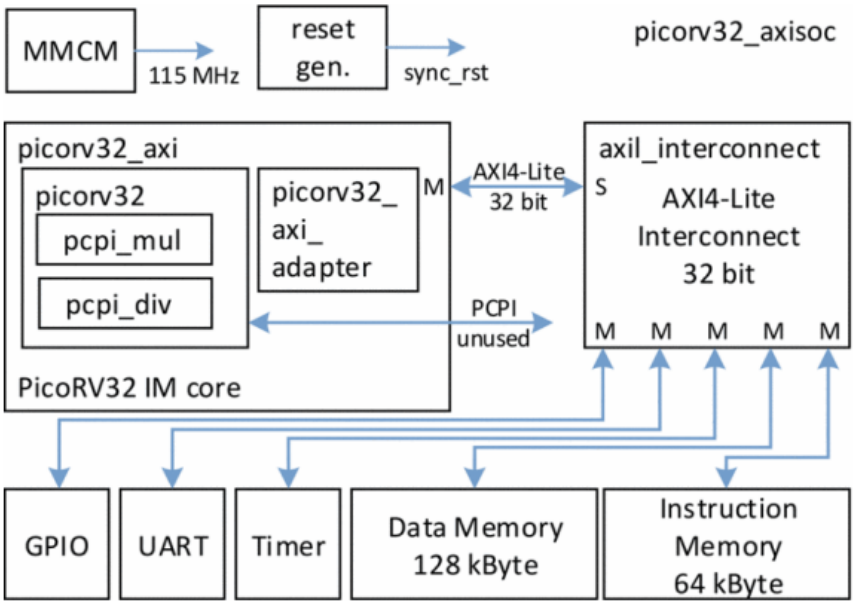

Figure 2.3: Block level representation of PICORV32, interfaces, and standard interconnects.

2.3 RISC-V Toolset

In addition to the recommended design, a set of software tools are required to facilitate programme creation and communication between the host device (a personal computer) and the core (on the FPGA). This section explains which instruments must be installed on the computer in order to allow data and binary transfer, PCIe bus access, and software development. Additionally, depending on the user's preference, the processor's memory was expanded to include a Board Support Package, a makefile, libraries to interface with the processor's peripherals, scripts to transfer the binaries, read or clear the registers, and other utilities. We go over the resources and design decisions utilized to build FPGA-based RISC-V processors. This contains the design decisions taken to make the architecture tractable, the tools used to generate the processor on the FPGA, and the tools used to simulate the RISC-V programmes. A cross-compiler is needed in order to execute cross-compilation for RISC-V. A compiler that can produce code for a target platform other than the one it is now operating on is known as a cross-compiler. When it comes

to RISC-V, a cross-compiler creates executable code for RISC-V on a host computer that is usually built on a different architecture. In RISC-V, cross-compilation is accomplished by a series of stages. Using the cross-compiler, the source code is first compiled to produce RISC-V executable code. After that, the code is linked to the RISC-V libraries and any prerequisites, including device drivers or system calls. Eventually, a RISC-V platform can use the generated executable file for operation. In RISC-V, cross-compilation is especially crucial when creating software for embedded systems and other platforms with constrained hardware resources. Developers may still target the RISC-V platform while taking use of advantages like faster CPUs and more memory by using a more potent development machine to cross-compile the code. There are several cross-compilation toolsets available for RISC-V development, including:

- The most popular toolkit for RISC-V development is the RISC-V GNU Compiler Toolchain. It may be used to create software for RISC-V processors and comes with a GCC compiler, binutils, GDB debugger, and other libraries.
- LLVM: This well-known open-source compiler infrastructure supports the RISC-V instruction set. A debugger, linker, compiler, and other tools are part of the LLVM toolkit.
- Spike: Without actual hardware, RISC-V applications may be tested using Spike, a RISC-V ISA emulator. Both a standalone download and the RISC-V GNU Compiler Toolchain contain it. Here is a summary of some of Spikes's primary attributes:
  - – Multiple ISAs: RV32IMAFDQCV extensions
  - – Multiple memory models: Weak Memory Ordering (WMO) and Total Store Ordering (TSO)
  - – Privileged Spec: Machine, Supervisor, User modes (v1.11)
  - – Debug Spec
  - – Single-step debugging with support for viewing memory/register contents

  - – Multiple CPU support
  - – JTAG support
  - – Highly extensible (add and test new instructions)

- QEMU: To imitate RISC-V hardware, utilize QEMU, a virtual machine emulator. It may be used to execute RISC-V applications on many operating systems and has support for a variety of RISC-V processors.
- OpenOCD: For RISC-V processors, OpenOCD is an open-source on-chip debugging tool. It offers a means of interacting with the JTAG interface of the CPU and may be applied to code flashing and debugging.
- GDB: A well-known open-source debugger for RISC-V processors is called GNU Debugger (GDB). It may be used to debug RISC-V software that is executing in a simulator or on real hardware.
- Buildroot: For building embedded Linux systems, Buildroot is an easy-to-use and effective program. Software for cross-compiling RISC-V architectures is supported.

These are a few of the most popular toolkits for developing with RISC-V. Various tool sets and development environments can be accessed based on the particular requirements of the project.

2.3.1 Testing

To test a RISC-V program, you can follow these general steps:

- Either use assembly language or a higher-level language that translates to RISC-V instructions to write your RISC-V program. As an illustration, Code Segment 1 adds two numbers and records the outcome in a register.
- Your program should be assembled into machine code using a RISC-V assembler, such as GNU Assembler (GAS). For instance.

```
.data
num1: .word 5
num2: .word 7
sum: .word 0

.text
main:
    lw x1, num1
    lw x2, num2
    add x3, x1, x2
    sw x3, sum
    li a7, 10
    ecall
```

Figure 2.4: Listing 1: Example of Assembly Program adding 2 numbers in RISC-V

riscv64-unknown-elf-as -o add.o add.s

This would produce a machine code file named add.o after assembling add.s assembly code.

- Load the machine code into memory on a RISC-V processor, either on a real RISC-V hardware platform or in a RISC-V simulator. E.g. if you were using the Spike simulator, you could load the program using the following command:

  spike pk add.o

- Execute the program on the RISC-V processor, either by running it on real hardware or by running it in a simulator.

You can select the RISC-V development board or simulator that best suits your requirements from the wide variety that is available. Going into the details of testing a program, there are typically four ways:

2.3.2 Behavioral simulation:

Use the RISC-V ISA simulator (Spike) to run the program. Programs that do not access I/O devices or user mode programs that operate within RISC-V Linux can be executed on Spike.

2.3.3 RTL simulation:

simulate the program in Verilator. No I/O devices are available in RTL simulation.

2.3.4 FPGA simulation:

simulate the program using Xilinx ISim. Behavioral modules for I/O devices are provided by Xilinx IPs; however, host-end modules (UART terminal and SD card) are not available.

2.3.5 FPGA run:

Actually, run the program on an FPGA board. Full I/O support (UART and SD). Programs can be compiled and run in three different modes:

- *Bare metal mode:* Supervisor programs are devoid of input/output access. RTL simulation and behavioral simulation. This mode of operation prevents peripheral support for programs. Only ISA and cache regression tests utilize this mode. An ISA test case's outcome is indicated by a program's return value. Zero indicates a successful outcome, but none-zero indicates a failed situation. Programs built in this way would operate on FPGA simulations or on real FPGAs quietly.
- *Newlib mode:* supervisor programmes that have I/O device access. FPGA run and FPGA simulation. Programmes executed in this mode are single-threaded but have complete control (supervisor priority) over peripherals, which are restricted in simulation. In this mode, bootloaders are executed.
- *Bare metal mode:*
  - Behavioural simulation: RISC-V-gnu-toolchain(newlib); RISC-V-isa-sim; RISC-V-fesvr.
  - RTL simulation: RISC-V-gnu-toolchain(newlib); verilator (built-in)
- *Newlib (supervisor) mode:*

  - FPGA simulation: RISC-V-gnu-toolchain(newlib); vivado.
  - FPGA run: RISC-V-gnu-toolchain(newlib); vivado.

- *Linux (user) mode:*
  - Behavioural simulation: RISC-V-gnu-toolchain(newlib+linux); RISC-V-isa-sim; RISC-V-fesvr; RISC V-pk; vmlinux; root.bin.
  - FPGA run: RISC-V-gnu-toolchain(newlib+linux); vivado; vmlinux; root.bin.

2.3.6 Summary

The instruction set and a few of the current implementations were given special attention as this chapter reviewed the state-of-the-art RISC-V ISA. It started by describing how RISC-V evolved while accounting for their ISA [16]. Many of the currently in use RISC-V core implementations were examined and contrasted. Ultimately, PICO-RV32 has been chosen as the simulation's core, and modifications to it are suggested.

Chapter 3

VPU Design and Implementation

In this chapter, we will be focusing on the implementation of our Vector Processing Unit (VPU) based on the RISC-V architecture. The vector processing unit is designed to enhance computational efficiency by enabling parallel operations on vector elements, making it suitable for applications that involve mathematical computations such as convolution and matrix operations.

Our VPU supports SIMD (Single Instruction, Multiple Data) architecture, which allows it to process multiple data elements with a single instruction. this capability makes it faster and reduces instruction size.

The VPU can load a maximum of 32 elements of a vector from memory into a register file, perform addition operations on two vectors, and store the resulting vector back into memory. This mechanism, enabled by SIMD architecture, accelerates vector operations and enhances overall computational throughput.

Additionally, the VPU includes features for effective memory management and parallel data handling, ensuring that data transfer between memory and the register file is optimized. These capabilities are crucial for high-performance computing tasks that require large data manipulation and parallel processing.

3.1 Motivation

With the increasing demand for high-performance computing (HPC) in fields such as machine learning, artificial intelligence,there is a growing need for processors that can handle large volumes of data efficiently. scalar processors, which process one data element at a time, are often very slow for these requirements. On the other hand, vector processors, which operate on entire vectors of data simultaneously using SIMD architecture, offer a significant performance advantage.

3.2 Design and architecture

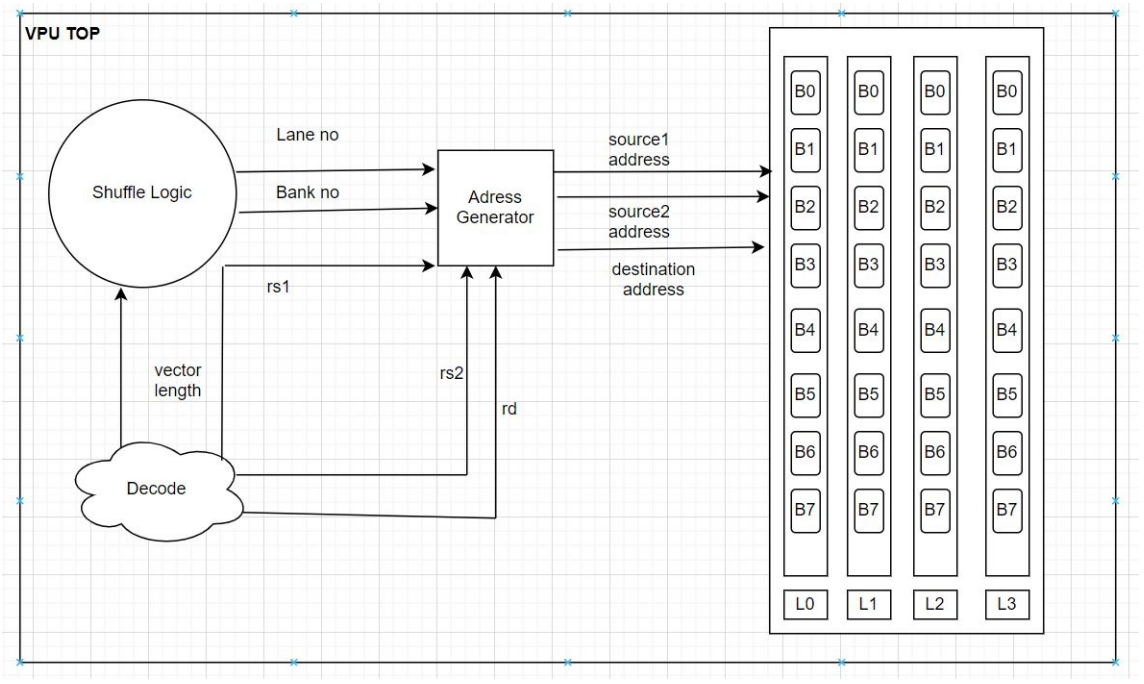

Figure 3.1: Top level representation of the proposed VPU

The design and architecture of our Vector Processing Unit (VPU) are carefully designed to efficiently handle vector operations. The core components of the VPU include an instruction decoder, shuffle logic, and four parallel lanes, each playing a crucial role in ensuring high-speed data processing and efficient memory management. Below is a detailed description of each component and its function. Figure 3.1 shows the complete diagram of

VPU.

3.2.1 Decode Unit

The decode unit processes instructions to extract the source1 and source2 addresses, the destination address, and the vector length. For vector instructions, the opcode bits [6:0] are always 0110011, and funct7 is 0000001. The specific operation is determined by the funct3 bits [14:12] of the instruction. If funct3 is 000, the instruction is an add operation. The source1 address is specified by instr[19:15], the source2 address by instr[24:20], and the destination address by instr[11:7]. For subtraction operations, funct3 is 001. For load operations, funct3 is 010, instr[21:20] specifies the vector length, instr[19:15] specifies source1 and instr[11:7] specifies destination address. For store operations, funct3 is 011. Figure 3.2 shows the custom instruction of our designed vector processor.

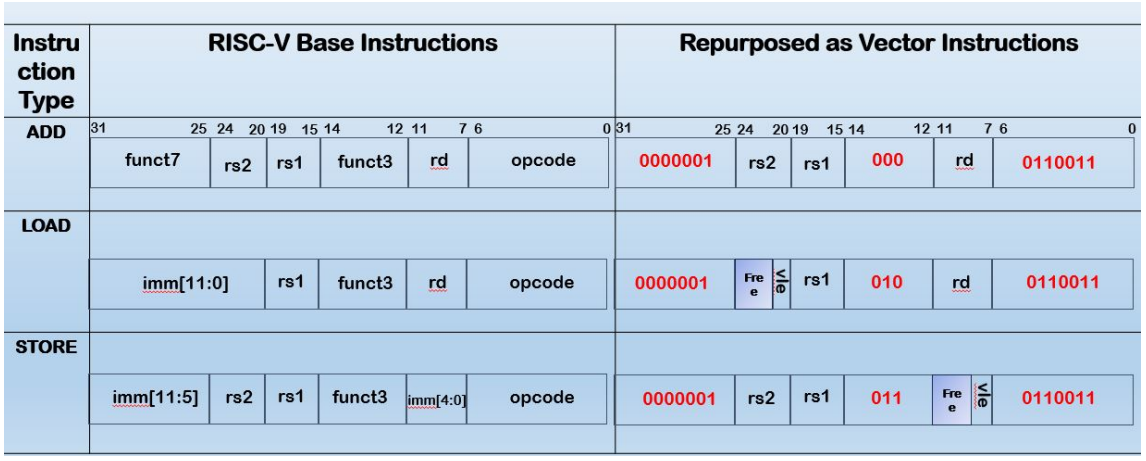

Figure 3.2: VPU custom instruction formats

Figure 3.3 shows the simulation result of VPU Top.

3.2.2 Data Shuffle Logic

The shuffle logic is a crucial component of our VPU as it determines the location within the register file from which vector elements are loaded or stored. It takes the vector length as input and outputs the lane number and bank number.The first element of the vector is

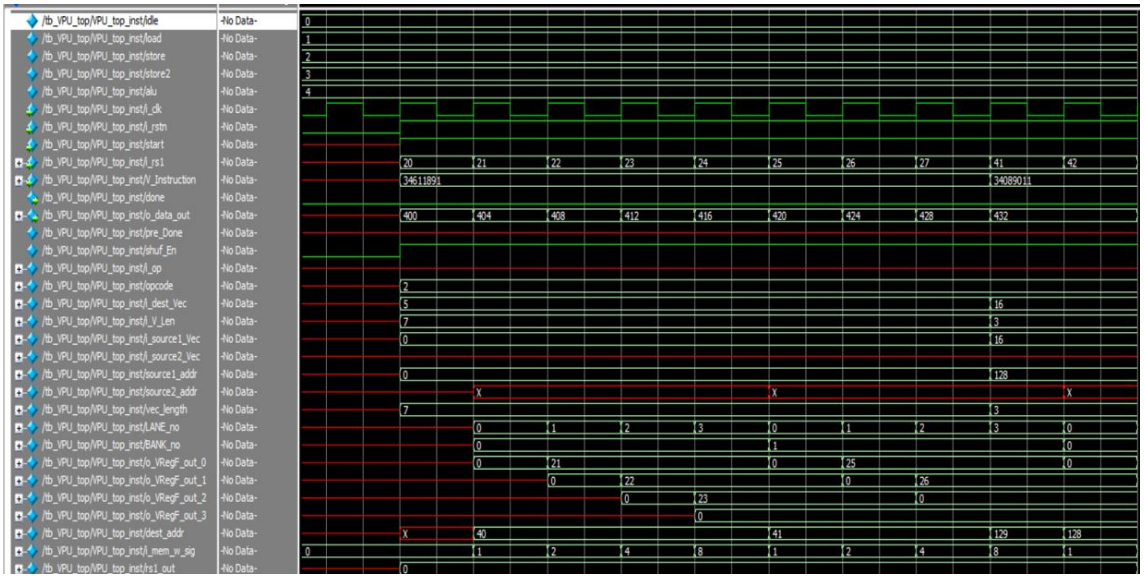

Figure 3.3: VPU Top simulation

placed in the first bank of lane 1, the second element in the first bank of lane 2, the third in the first bank of lane 3, and the fourth in the first bank of lane 4. After every fourth element, the bank number is incremented. The lane number is calculated by taking the modulus of the vector index with four. Figure 3.4 shows how the data is placed in four parallel lanes.

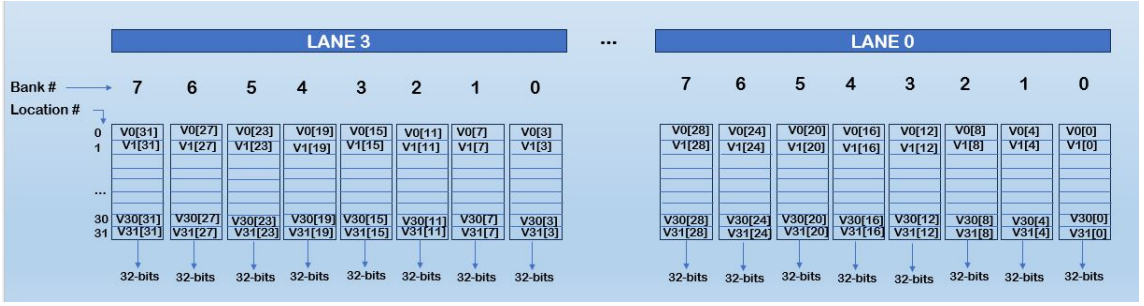

Figure 3.4: Data placement in lanes

Figure 3.5 shows the simulation results of shuffle logic (i.e, how data is write and placed in four parallel lanes and the Alu result.

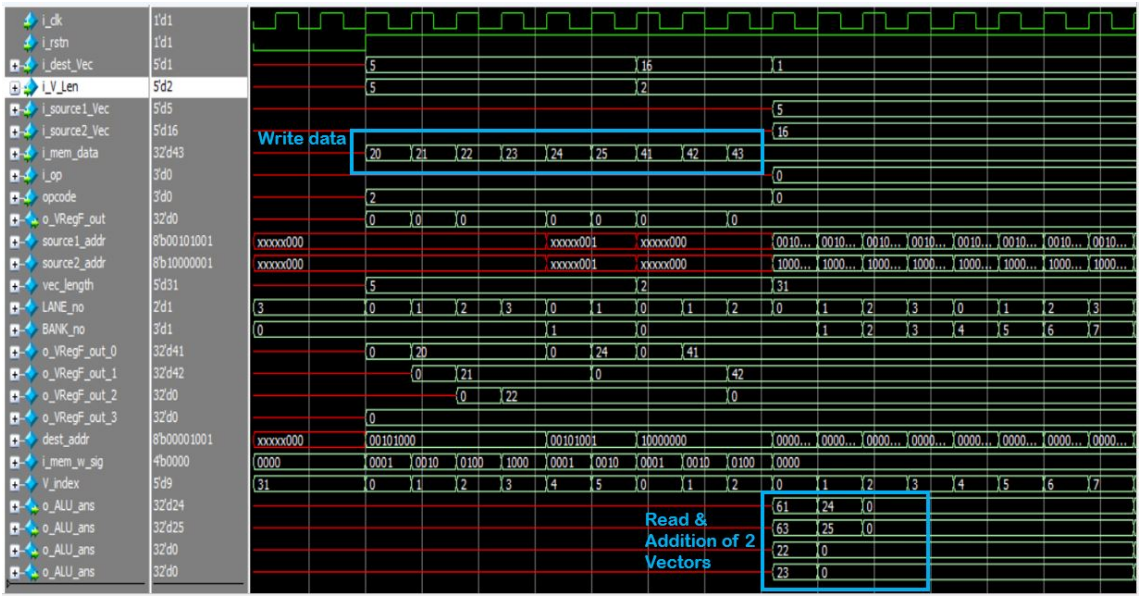

Figure 3.5: Shuffle logic simulation

3.2.3 Processing Lanes

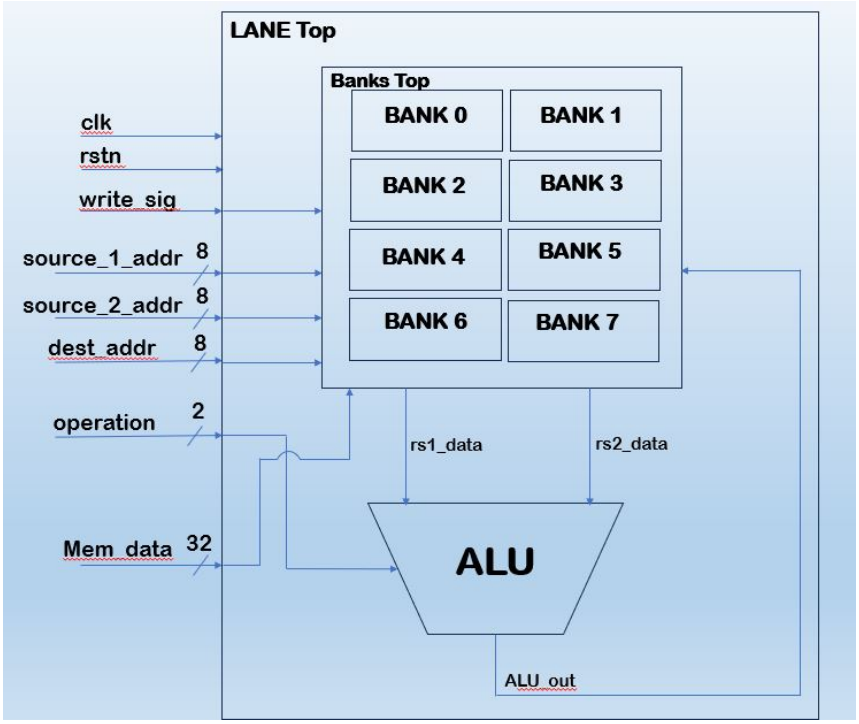

Figure 3.6: Block level representation of Single lane architecture

Lanes are the primary components of the VPU that enable enhanced parallel processing. We have four lanes operating in parallel, each equipped with eight banks and one ALU. It takes an 8-bit address each for source1, source2, and destination addresses. The first 5 bits of the address determine the location number, while the last 3 bits determine the bank number. Additionally, it takes a 2-bit input for ALU operations and a 32-bit data input, which is stored in the appropriate bank based on the address. Figure 3.6 shows the custom instruction of our designed vector processor.

3.2.3.1 Banks

Each lane contains 8 memory banks. Based on the first 5 bits of the addresses, data is taken from each bank. Two 8-to-1 multiplexers are used, with their select lines being the first 3 bits of the address (i.e., the bank number). This way, data is picked from the banks for operation purposes and sent to the ALU and subsequently written back to the destination address. Figure 3.7 shows the memory banks of single lane and how the data is picked and write to these banks.

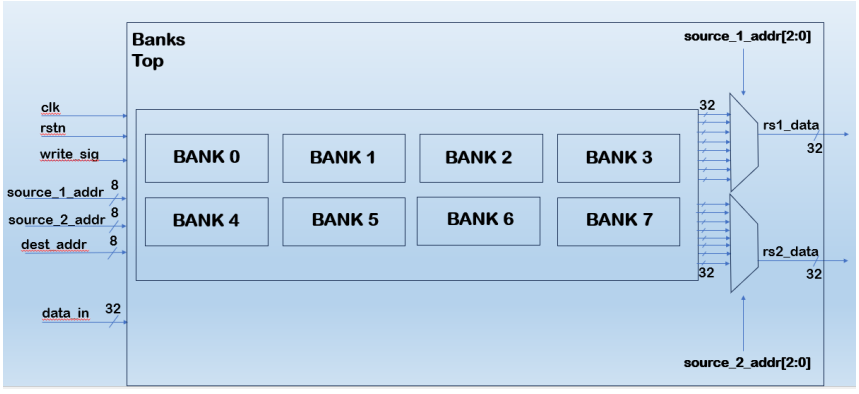

Figure 3.7: Memory Banks of single lane

3.2.3.2 Single Bank

Each bank has 32 locations, each 32 bits wide. The first 5 bits of the address are used to select a location for reading or writing data on each rising clock edge. Each bank has its own write enable signal and is dual-port, allowing for two data outputs. Figure 3.8 shows

the memory banks of single banks and how the data is picked and write to these banks.

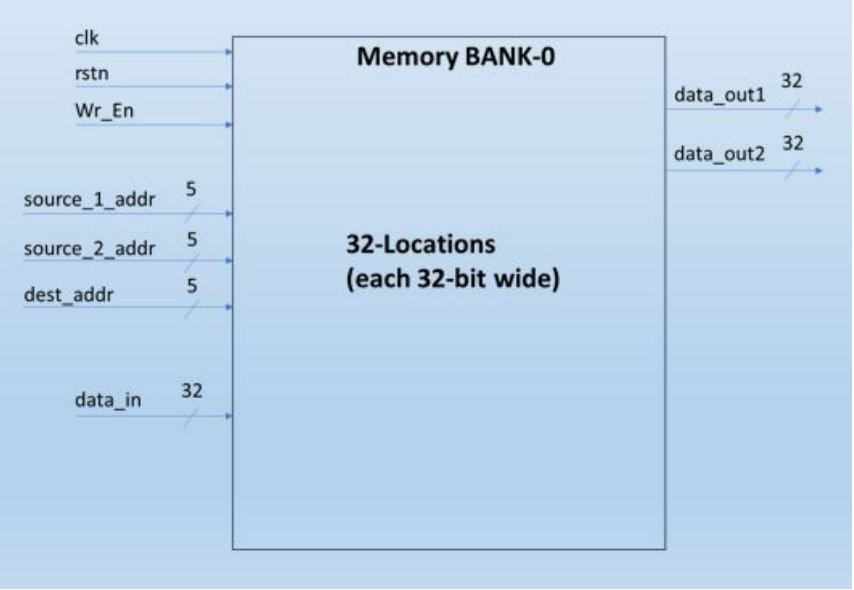

Figure 3.8: Single memory bank

3.3 Summary

The vector processing unit (VPU) is designed to enhance computational efficiency by enabling parallel operations on vector elements, making it highly suitable for applications involving complex mathematical computations such as convolution and matrix operations. Based on the RISC-V architecture, our VPU supports the SIMD architecture to process vector elements with a single instruction, increasing processing speed and reducing instruction size.

Key features of the VPU include the ability to load up to 32 vector elements from memory into a register file, perform addition operations on two vectors, and store the resulting vector back into memory. This parallel processing capability significantly accelerates vector operations and improves overall computational throughput.

Overall, the VPU design and implementation provide a powerful and efficient solution for applications involving convalution or vector operation.

Chapter 4

Integration of VPU with RISC-V Core

4.1 Introduction

In this chapter, we will focus on the integration of the designed Vector Processing Unit (VPU) with the scalar RISC-V core. After researching different scalar RISC-V cores, we have selected the PicoRV32, which provides the Pico Co-Processor Interface (PCPI) to integrate co-processors. These co-processors can include AI accelerators, DSP accelerators, cryptographic co-processors, or any other type for faster processing. PicoRV32 also has built-in multiplication and division cores, that can be enabled if needed.

We integrate our VPU with the PicoRV32 using the Pico Co-Processor Interface (PCPI). PCPI has various data and control signals for communication between the PicoRV32 and our designed VPU.

4.2 Design and architecture

The VPU-integrated RISC-V processor operates such that the scalar processor executes scalar instructions according to the core's defined mechanism, while the Vector Processing Unit (VPU) handles vector instructions.

4.2.1 Overview of PCPI Interface

ENABLE-PCPI is set to 1 to facilitate communication between the PicoRV32 and the VPU. The Pico Co-Processor Interface provides the following signals for communication: pcpi-valid, pcpi-insn, pcpi-rs1, and pcpi-rs2 are output signals from the PCPI, while pcpi-wait, pcpi-ready, pcpi-rd, and pcpi-wr are input signals to the PCPI. Figure 4.1 shows the Integration of RISC-V core with our designed VPU. In Figure 4.1 red lines shows the control signals while green lines denotes data signals.

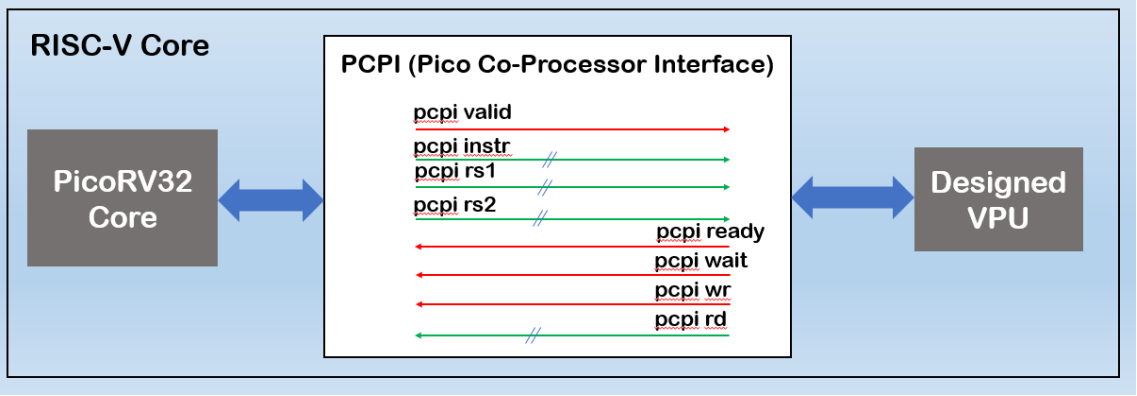

Figure 4.1: Integration of VPU with RISC-V Processor

When an unsupported instruction comes, pcpi-valid is asserted. The instruction is transferred to the co-processor if pcpi-ready is asserted. pcpi-valid and pcpi-ready are the handshaking signals between the core processor and the co-processor. The instruction is transferred to the co-processor using pcpi-insn, and the source registers rs1 and rs2 are decoded, with their values sent to pcpi-rs1 and pcpi-rs2, respectively. pcpi-wait is set to 1 and pcpi-ready is set to 0 until the execution is completed.

When the co-processor completes execution, the result is written to pcpi-rd and pcpi-wr is asserted, indicating that the co-processor wants to write something.

The PCPI waits for 16 clock cycles. If no PCPI core acknowledges the instruction within this time, an illegal instruction exception is raised. Every instruction of the PicoRV32, whether it is a memory instruction or an ALU instruction, takes 4 clock cycles to execute. If a PCPI instruction takes more than four cycles, pcpi-wait is set to 1 until the instruction

is executed. This signal prevents the PicoRV32 core from raising an illegal instruction exception. The main processor remains in an idle state when the co-processor is running.

4.2.2 PicoRV32 and Memory Intergration

PicoRV32 doesn't have the internal data and instruction memory. We have made the top-level wrapper of PicoRV32 named Final-Top and make a memory following Von-Neumman architecture, combined data and instruction memory. The size of memory is 1KB, every location is 32 bit wide and memory depth is 256.

To efficiently handle the transfer of data and instruction between memory and main processor, PicoRV32 provides different data and control signals for this purpose. These signals are asserted and de-asserted according to requirements.

These are the signals between memory and PicoRV32. Mem-valid, mem-instr, mem-addr, mem-wstrb, mem-wdata are the output signals from PicoRv32 and input for the memory, whereas mem-ready and mem-rdata are output signals from the memory and input to the PicoRV32. Figure 4.2 shows the interface of memory with PicoRV32.

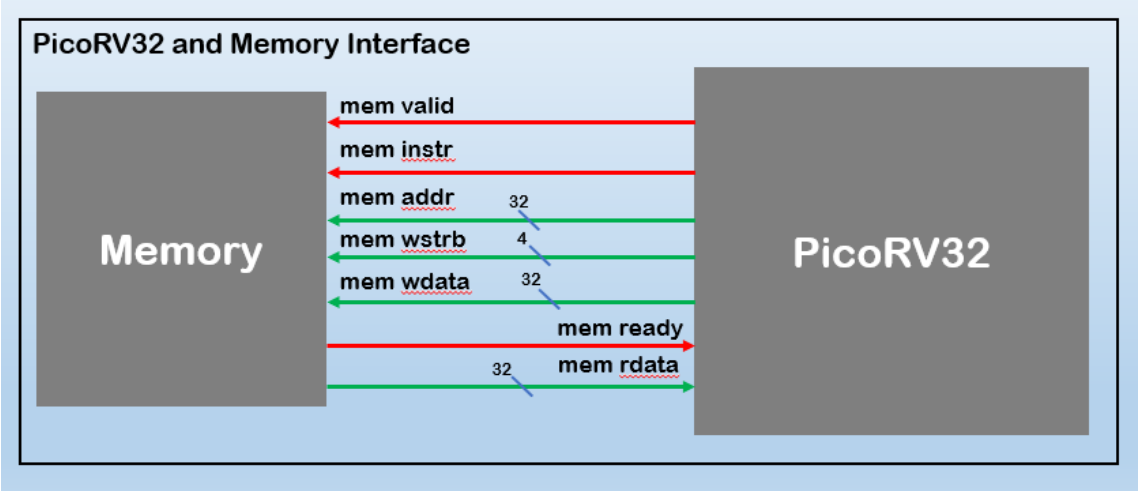

Figure 4.2: PicoRV32 and memory interface

Mem-valid and mem-ready act as handshaking signals between the memory and the main core. When mem-valid is high, the mem-ready signal is set to 1, and the instruction is loaded from the instruction memory at the address mem-addr and placed in mem-rdata.

For store instructions, data is written to the memory at the address mem-addr from mem-wdata, with mem-wstrb set to all 1s.

4.2.3 VPU and Memory Integration

The VPU does not have its own memory; it uses the PicoRV32 memory to load data and instructions and to store data back into memory after performing ALU operations. The VPU can access the memory for load and store purposes. However, the main processor and the co-processor cannot access memory at the same time. To prevent simultaneous access, we have implemented a 2-to-1 multiplexer controlled by the vpu-wait signal. When vpu-wait is high, the VPU accesses the memory; when it is low, the main core accesses the memory. This ensures that only one processor accesses the memory at any given time. Figure 4.3 integration of memory and VPU.

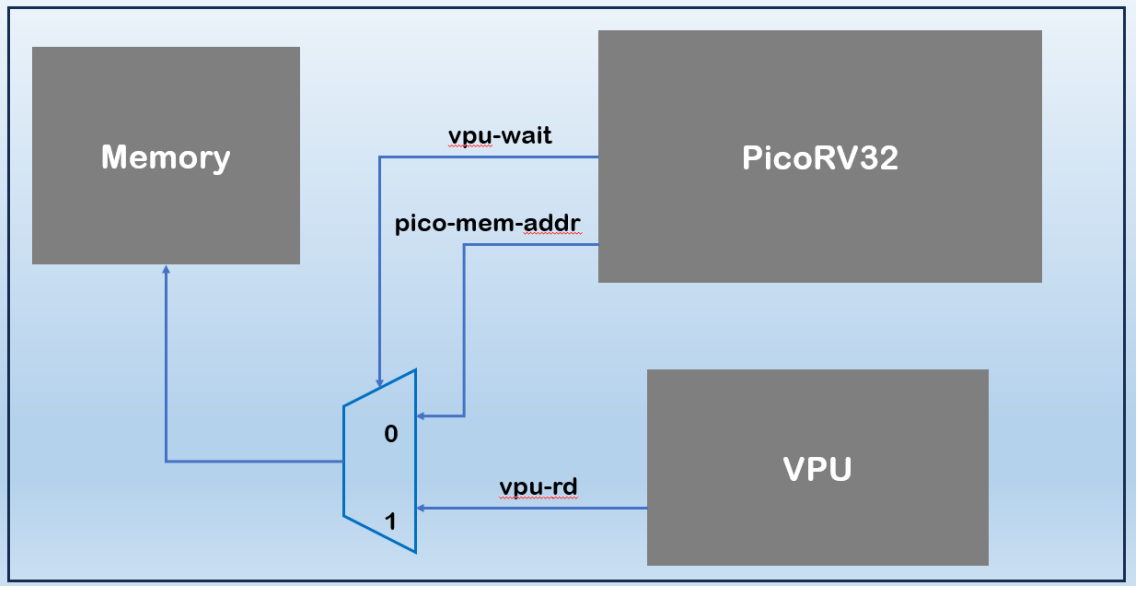

Figure 4.3: VPU and memory integration

The VPU-wait operates based on the done signal from the co-processor, remaining high until the co-processor finishes working and asserts the done signal. So, the memory is controlled using the vpu-wait control signal.

4.2.4 VPU Top-Wrapper FSM

We have made a Finite Set Machine (FSM) to generate the control signals for different vector instructions. VPU signals are asserted and de-asserted depending on the operation. Figure 4.4 shows the FSM.

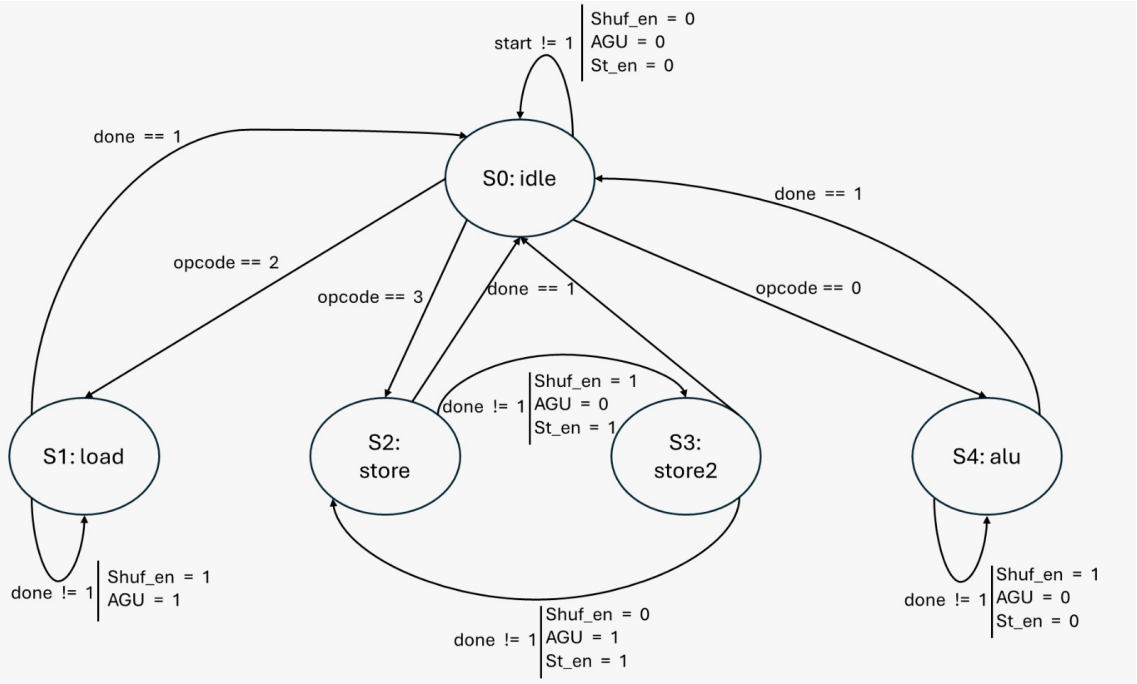

Figure 4.4: VPU and memory integration

The FSM we have created consists of 5 states: idle, load, store, store2, and alu. Each state has its own set of control signals. In the idle state, the VPU does nothing; the other states are explained one by one

4.2.4.1 S1: Load State

When an instruction with opcode 2 is received, the current state changes to 'load'. In the 'load' state, Shuf-En is set to 1 to place data in the memory banks at addresses generated by the shuffle logic. The Address Generator (AG) block is enabled to generate the source addresses for memory. The AG increments the address by 1 after each cycle, sending it through pcpi-rd. AGU is set to 1 based on the vector load elements' length, which ranges

from 4 to 32. 'Done' is controlled by AGU; when the address generator reaches the vector length, 'done' is asserted, and the current state is set back to 'idle.

4.2.4.2 S2: Store State and S3: Store2 State

The operation of the store instruction differs from that of the load and ALU instructions. When an instruction with opcode 3 is received, the current state alternates between the 'store' and 'store2' states. As shown in Figure 4.1, there is only one 32-bit bus as input to the Pico Co-Processor Interface (PCPI). Completing a store operation for one vector element requires two cycles: in the first cycle, the write address is sent to the main processor's memory, and in the second cycle, the write data is sent and written to the address provided by the VPU in the previous cycle. Thus, the current state is 'store' for one cycle and 'store2' for the next. AGU and Shuf-En are enabled and disabled between these states. The Address Generator (AG) increments the address by 1 after each cycle, sending it through pcpi-rd. AGU is set to 1 based on the vector store elements' length, which ranges from 4 to 32. 'Done' is controlled by AGU; when the address generator reaches the vector length, 'done' is asserted, and the current state is set back to 'idle'.

4.2.4.3 S4: ALU State

Our vector ALU performs only the addition operation. When an instruction with opcode 0 is received, the current state is set to 'alu,' which is the final state of the FSM. Since we have four lanes working in parallel, the addition operation is completed in 8 cycles. The number of cycles for the addition operation is fixed, regardless of the vector length. In the case of an addition instruction, AGU and St-en are de-asserted, and only Shuf-en is set to 1 to retrieve sr1 and sr2 and store the result in rd.

4.2.5 Working VPU Integrated RISC-V Processor

In this section, we discuss the operational flow of memory and ALU instructions. Our integrated RISC-V processor can handle both scalar and vector instructions seamlessly.

If the instruction is scalar, it executes normally on the main processor. To clarify how the main processor identifies a vector instruction, the PicoRV32 uses fixed opcode and func7 bits of the R-type instruction in the RISC-V ISA. The fixed bits for the opcode are "0110011", and for func7 are "0000001". When an instruction with this format is fetched, pcpi-valid is asserted, indicating to the system that a vector instruction is ready to be processed. This assertion triggers the vector processor to begin execution. While the vector processing unit is working, the main processor remains idle.

We have repurposed the RISC-V instruction format for our Vector Processing Unit (VPU) as in the Figure 3.2

4.2.5.1 Instruction Fetch

PicoRV32 fetches an instruction from memory every four cycles and executes each scalar instruction in four cycles. In the case of a vector instruction, the execution time depends on the vector length.

4.2.5.2 Working of Load Instruction

If the instruction is a scalar load, the main core loads the data and keeps it in the register file before processing. In the case of a vector load instruction, it is sent to the co-processor through pcpi-insn. The co-processor sends the load address to the main processor via pcpi-rs1, incrementing by one each cycle. For vector load operations, a maximum of 32 elements can be loaded with a single instruction, with the length varying based on the vlen field in the repurposed RISC-V instruction. If the vlen bits are "00", it loads 4 elements; for "01", it loads 8 elements; and for "10", it loads 32 elements. The loaded vector elements are kept in the memory banks of the vector processor. Figure 4.6 shows the working of the vector load instruction.

4.2.5.3 Working of ALU Instruction

Scalar ALU instructions are executed by the main processor. In the case of a vector instruction, it is sent to the VPU through the Pico Co-Processor Interface (PCPI) on pcpi-insn. Since we have four lanes in the VPU working in parallel, it adds four corresponding vector elements per cycle. Therefore, for the addition of 32 elements, the VPU takes 8 cycles to complete the operation. The processed data is then placed in the memory banks. Figure 4.5 shows the working of vadd instruction.

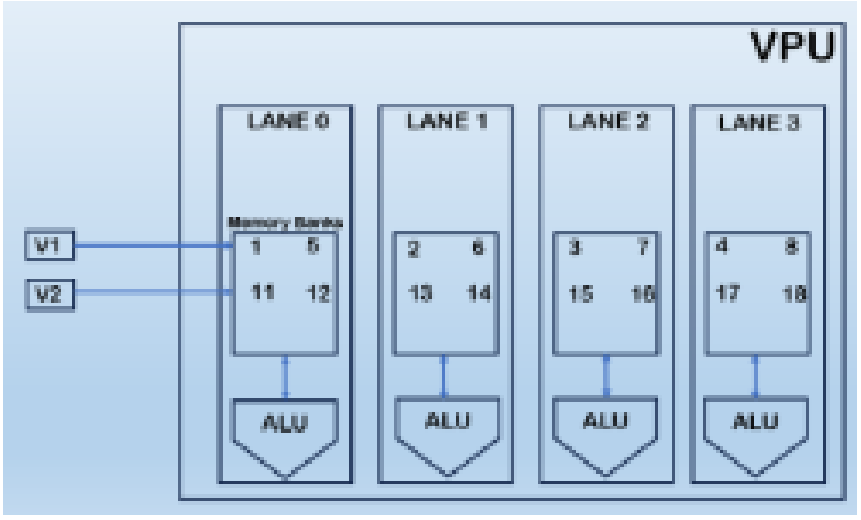

Figure 4.5: Working of vadd instruction

4.2.5.4 Working of Store Instruction

Store instructions are executed differently from load and ALU operations. Scalar store instructions are processed by the main processor, while vector store instructions are sent by the main processor to the VPU through pcpi-insn, initiating the store operation. Since store instructions also require a store address, the store operation is completed in two cycles for each element. To store 32 elements in memory after processing, it will take 64 cycles to complete the operation.

Since we are using the same bus pcpi-rd for writing addresses and write data. In the first cycle write address is sent to memory through pcpi-rd and in next cycle write data is sent to memory through pcpi-rd. How memory get to know which is the write address

and which is the write data? For this purpose we have used the PCPI pcpi-wr signal to differentiate the between write address and write data. If the pcpi-wr is 0, pcpi-rd has write address, if it is 1 then pcpi-wr is set to 1.

We have variable-length store elements, specified in the repurposed instruction format as in Figure 3.2 using bits [8:7] for store element length. If the value of vlen is "00," it stores 4 elements in memory from the banks; if vlen is "01," it stores 8 elements; if vlen is "10," it stores 16 elements; and if vlen is "11," it stores 32 elements. This procedure is followed for every vector store operation. Figure 4.6 shows the working of store instruction.

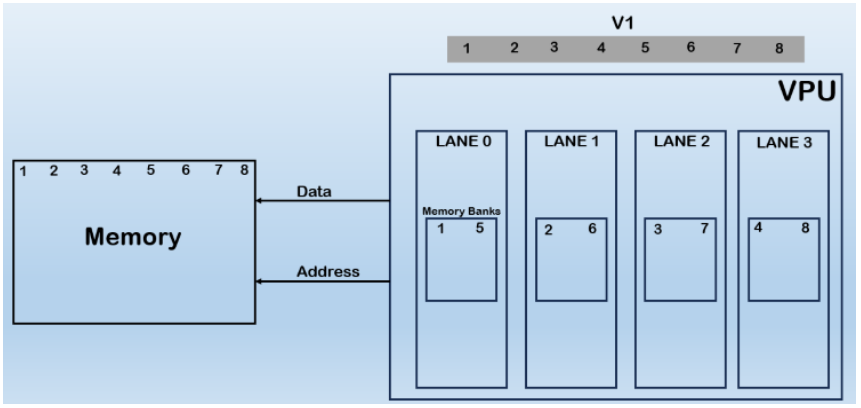

Figure 4.6: Working of vstore instruction

4.2.5.5 Final Waveform

We have verified the operation of the VPU-integrated RISC-V processor through simulation. Figure 4.7 shows the simulated results of our VPU-integrated RISC-V processor.

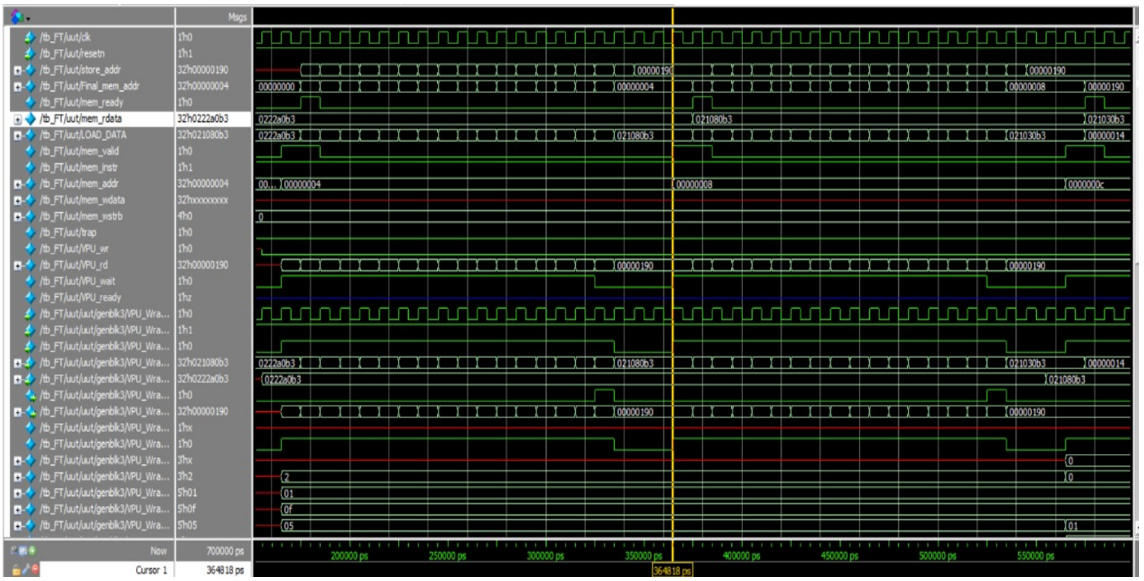

Figure 4.7: Simulation results of VPU integrated RISC-V processor

4.3 Summary

In this chapter, we analyze the integration of our designed Vector Processing Unit (VPU) with the scalar RISC-V core, specifically using the PicoRV32 core. The integration advantages the Pico Co-Processor Interface (PCPI) to execute both scalar and vector instructions. The PCPI interface, with its different control and data signals, ensures that unsupported instructions are efficiently sent to the co-processor for execution.

We explained the design and architecture of the VPU-integrated RISC-V processor, showing how scalar instructions are executed by the main processor while vector instructions are managed by the VPU. The integration process also involved creating a top-level wrapper for the PicoRV32 to include a combined data and instruction memory, following the Von Neumann architecture. This setup ensures efficient data transfer between memory and the main processor.

The chapter also covered the detailed working of load, ALU, and store instructions. Scalar instructions are handled directly by the main processor, while vector instructions are processed by the VPU. The VPU's capability to process multiple vector elements in parallel

significantly increases performance, especially for operations involving large data sets.

Additionally, we implemented a control mechanism using the vpu-wait signal to prevent simultaneous memory access by the main processor and the VPU. This ensures smooth and conflict-free operation when accessing shared memory resources.

Overall, the successful integration of the VPU with the PicoRV32 core show the feasibility and efficiency of combining scalar and vector processing within a scalar RISC-V processor architecture. This integration not only enhances processing capabilities but also opens up possibilities for implementing various co-processors to handle specialized tasks, thereby improving overall system performance.

Chapter 5

Cross-Compilation Toolchain and Analysis

5.1 Introduction

In this chapter, we will discuss the cross-compilation toolchain, which is essentially a C compiler for RISC-V. Since our target machine is RISC-V, we need a compiler that can convert our C code into RISC-V assembly code. This is achieved by configuring GCC with the RISC-V ISA, which allows it to generate a disassembly file. This file can then be simulated and debugged using a simulator called Spike and the proxy kernel (pk). Through this process, we can examine the instruction size and measure the compilation time.

5.2 Motivation

The cross-compilation toolchain will help us in benchmarking our VPU-integrated core against a scalar RISC-V core. As we focus on SIMD architecture, the toolchain will allow us to evaluate how much the instruction size is reduced by comparing the number of lines

in the disassembly file of the RISC-V scalar core with that of our VPU-integrated RISC-V core.

5.3 Toolchain Setup

The cross-compilation toolchain setup on Linux is accomplished using the official repository of the RISC-V cross-compiler toolchain[35]. This setup involves the following steps:

- Clone the toolchain directory into the local machine from the official repository of RISC-V.
- Configure environment variables to ensure proper functioning of the toolchain.
- Build and install the RISC-V toolchain to compile code for the target architecture.
- Install and configure Spike and PK using the repository to enable simulation and debugging capabilities.

Figure 5.1 shows the successful installation and version detail of cross compiler toolchain.

```
ahsanali@DESKTOP-QVVBNOA:~$ riscv32-unknown-elf-gcc --version
riscv32-unknown-elf-gcc () 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Figure 5.1: Cross compiler toolchian

5.4 Components

The toolchain consists of the following components:

- **RISC-V GNU Compiler**: Converts C code into RISC-V assembly code.
- **Spike Simulator**: Simulates and debug the disassembly file.
- **Proxy Kernel (pk)**: Provides a runtime environment for the simulation.

The detailed explanation of the tools is given in the subsequent sections.

5.4.1 RISC-V GNU Compiler

The RISC-V GNU Compiler consists of the following key components:

- **GCC Compiler**: Configured with the RISC-V ISA, this compiler translates C code into RISC-V assembly code.
- **Binutils**: A collection of binary tools including the linker and assembler, which are generating executable files from the compiled code.
- **Necessary Libraries**: These libraries provide additional functionality required during the compilation and linking process.

Together, these components enable the conversion of C code into RISC-V assembly code.Figure 5.2 shows the compilation of C code using GNU toolchain.

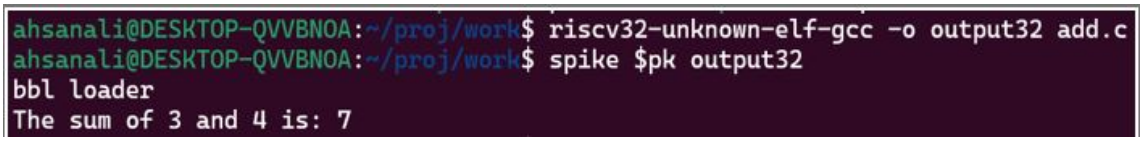

Figure 5.2: Compilation of C code

5.4.2 Spike

Spike is essentially a simulator for RISC-V. With the help of Spike, we can simulate and debug our disassembly file. It allows us to check register values and observe how they change at any point during execution. Additionally, we can check the output of our program using Spike, making it an invaluable tool for debugging, performance analysis, and verification.

With the help of spike and proxy kernel (pk) we can we view the output of the C or C++ on the terminal. Figure 5.3 shows the simulation using Spike.

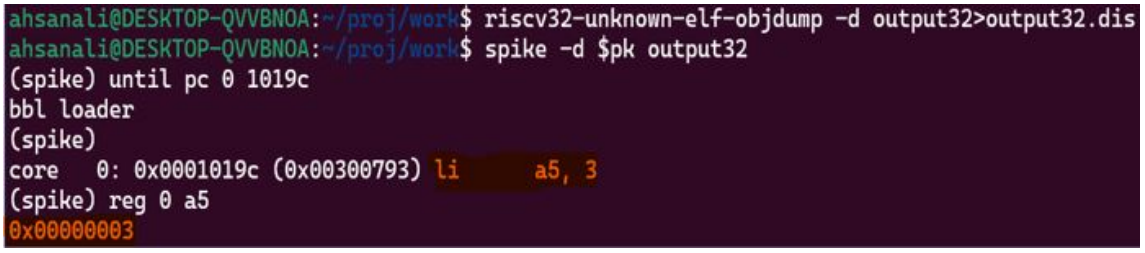

Figure 5.3: Simulation using Spike

5.4.3 Proxy kernel(PK)

The Proxy Kernel (pk) serves as a runtime environment for the simulation. It acts as a software bridge for RISC-V, managing memory and basic input-output operations. Pk act as an operating system for RISC-V, providing services to facilitate program execution within the simulation environment. Figure 5.4 shows the inputs and outputs operation using spike and pk.

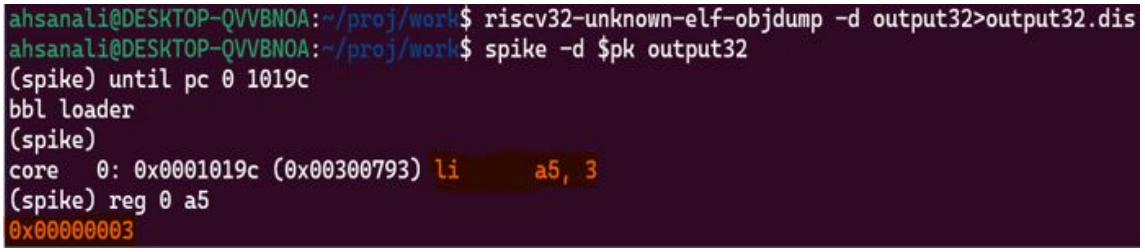

Figure 5.4: Inputs and outputs operation using spike and PK

5.5 Analysis of the final core

As mentioned earlier, the toolchain will assist us in benchmarking. To begin with benchmarking, we first run a simple program containing only a main function, where an array is declared along with the necessary libraries. We then check how many lines of disassembly are generated. This program produces 2108 lines of disassembly. Figure 5.5 shows the C code in which we are declaring and initialize only first element of an array and Figure 5.6 shows the corresponding disassembly lines of code.

Now, when we load one element of the array, we observe that the number of instructions

```c
#include <stdio.h>

int main() {
    int array[5] = {0};
    //  array[0] = 5;
    //  array[1] = 3;
    //  array[2] = 5;
}
```

Figure 5.5: C code for an initialize an array element

```
2101     115b0:   2021              jal   115b8 <__errno>
2102     115b2:   c100              sw s0,0(a0)
2103     115b4:   547d              li s0,-1
2104     115b6:   b7f5              j  115a2 <_write+0x14>
2105
2106 000115b8 <__errno>:
2107     115b8:   d3c1a503          lw a0,-708(gp) # 12b0c <_impure_ptr>
2108     115bc:   8082              ret
2109
```

Figure 5.6: Corresponding disassembly lines

increases by 2. Similarly, when we load two elements, the instructions increase by 4, going from 2108 to 2112. This indicates that loading an element of an array requires 2 instructions. Consequently, if we load 32 elements of an array, the instructions will increase by 64. Figure 5.7 shows the C code in which we are loading two elements of an array and Figure 5.8 shows the corresponding disassembly lines of code for two loads.

```c
#include <stdio.h>

int main() {
    int array[5] = {0};
        array[0] = 5;
        array[1] = 3;
    //  array[2] = 5;
}
```

Figure 5.7: C code to load two elements of an array

Our VPU-integrated core can load an entire vector with just one instruction. Typically, loading a single element of a vector requires 2 instructions, so loading a vector consisting of 32 elements would take 64 instructions. However, our VPU-integrated core reduces this to only one instruction. The instruction for loading a complete vector is: vload rd,

```
2104     115b8:   40800433            neg   s0,s0
2105     115bc:   2021                jal   115c4 <__errno>
2106     115be:   c100                sw s0,0(a0)
2107     115c0:   547d                li s0,-1
2108     115c2:   b7f5                j  115ae <_write+0x14>
2109
2110   000115c4 <__errno>:
2111     115c4:   d3c1a503            lw a0,-708(gp) # 12b1c <_impure_ptr>
2112     115c8:   8082                ret
2113
```

Figure 5.8: Corresponding disassembly lines for two loads

mem[rs1[0]].

Similarly, for the add operation, without performing any add operations and only declaring our variables in main, the disassembly lines of code total 2103. Figure 5.9 shows the C code in which we are declaring three variables only and Figure 5.10 shows the corresponding disassembly lines of code.

```c
#include <math.h>
#include <stdio.h>

int main()
{
  int a, b , c;
  //a = 3 + 1;
}
```

Figure 5.9: C code for declaring variables

```
2096     1159c:   2021                jal   115a4 <__errno>
2097     1159e:   c100                sw s0,0(a0)
2098     115a0:   547d                li s0,-1
2099     115a2:   b7f5                j  1158e <_write+0x14>
2100
2101   000115a4 <__errno>:
2102     115a4:   d3c1a503            lw a0,-708(gp) # 12afc <_impure_ptr>
2103     115a8:   8082                ret
2104
```

Figure 5.10: Corresponding disassembly lines for declaring variables

Now, when we perform one add operation, the disassembly lines increase by 2, going from 2103 to 2105. When performing two add operations, it becomes 2107, and with three add operations, it becomes 2109. This indicates that each add operation requires

two instructions. The figure 5.11 shows the C code to perform three add operations and the Figure 5.12 shows corresponding disassembly lines of code.

```c
#include <math.h>
#include <stdio.h>

int main()
{
  int a, b , c;
  a = 3 + 2;
  b = 4 + 5;
  c = 7 + 9;
}
```

Figure 5.11: C code for three add operation

```
2103      115b0:   c100              sw s0,0(a0)
2104      115b2:   547d              li s0,-1
2105      115b4:   b7f5              j  115a0 <_write+0x14>
2106
2107    000115b6 <__errno>:
2108      115b6:   d3c1a503          lw a0,-708(gp) # 12b0c <_impure_ptr>
2109      115ba:   8082              ret
2110
```

Figure 5.12: Corresponding disassembly lines for three add operations

On the other hand, our VPU-integrated core adds corresponding elements of two vectors using a single add instruction. In contrast, if we want to add two vectors, each containing 32 elements, the scalar core would require 64 instructions for addition. However, our VPU-integrated core accomplishes the addition of two vectors with just one instruction. The instruction for adding corresponding elements of two vectors is: vadd rd, rs1, rs2

Similarly, when we store one element of an array, the disassembly lines increase by one. This indicates that every element of an array requires one instruction for storage.

```c
#include <stdio.h>

int main() {
    int array[5] = {0};
   // array[0] = 42;
   // array[1] = 12;
   // array[1] = 12;
}
```

Figure 5.13: C code for just declaring an array

```
2102      115b2:   c100              sw s0,0(a0)
2103      115b4:   547d              li s0,-1
2104      115b6:   b7f5              j  115a2 <_write+0x14>
2105
2106  ⌄ 000115b8 <__errno>:
2107      115b8:   d3c1a503          lw a0,-708(gp) # 12b0c <_impure_ptr>
2108      115bc:   8082              ret
2109
```

Figure 5.14: Corresponding disassembly lines

The Figure 5.13 shows the C code without any store operation 5.14 shows corresponding disassembly lines of code.

The Figure 5.15 shows the disassembly lines of code to store three array elements.

```
2104      115ba:   40800433          neg   s0,s0
2105      115be:   2021              jal   115c6 <__errno>
2106      115c0:   c100              sw s0,0(a0)
2107      115c2:   547d              li s0,-1
2108      115c4:   b7f5              j  115b0 <_write+0x14>
2109
2110  ⌄ 000115c6 <__errno>:
2111      115c6:   d3c1a503          lw a0,-708(gp) # 12b1c <_impure_ptr>
2112      115ca:   8082              ret
2113
```

Figure 5.15: Disassembly to store two array elements

On the other hand, our VPU-integrated core can store all elements of an array with just a single instruction. This means that if a vector consists of 32 elements, the scalar core would require 32 instructions to store the complete vector, whereas our VPU-integrated core needs only one instruction.

5.6 Summary

Our benchmarking analysis reveals significant efficiency improvements with our VPU-integrated core compared to the scalar core. When loading elements, the scalar core requires 2 instructions per element, resulting in 64 instructions to load a vector with 32 elements. In contrast, our VPU-integrated core achieves this with a single instruction. Similarly, for add operations, the scalar core needs 2 instructions per add operation, totaling 64 instructions for adding two vectors with 32 elements each. However, our VPU-integrated

core performs this operation with just one instruction. Furthermore, storing elements follows the same pattern; the scalar core requires 1 instruction per element, leading to 32 instructions for storing a complete vector, whereas our VPU-integrated core accomplishes this with a single instruction. This substantial reduction in the number of instructions demonstrates the enhanced computational performance and reduce the instruction size by a large number. Summary of the results in shown in Table 5.1.

Table 5.1: Comparative results for Base processor vs VPU enabled processor

Features	**Base RISC-V Core**	**Vector RISC-V Core**		
		Add	**Load**	**Store**
Latency	32	8	32	64
Lines of Code	32	1	1	1

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we began by researching the RISC-V architecture and its various cores. After evaluating several options, we selected the PicoRV32 core for integrating our VPU. We then detailed the design and implementation of the VPU, which enhances computational efficiency through SIMD (Single Instruction, Multiple Data) architecture, enabling parallel operations on vector elements. This approach is particularly beneficial for applications involving vector operations, such as convolution and matrix computations.

Our benchmarking analysis highlights the advantages of the VPU-integrated core over a traditional scalar core. The ability to load, add, and store entire vectors with single instructions significantly reduces the number of required instructions. For instance, while a scalar core requires 64 instructions to load a vector with 32 elements, our VPU-integrated core completes this task with a single instruction. Similarly, vector addition and storage operations also see substantial reductions in instruction count, resulting in improved computational throughput and efficiency.

The cross-compilation toolchain, essential for the development and testing of our VPU, was set up using the official RISC-V repository. This toolchain, which includes the RISC-

V GNU compiler, the Spike simulator, and the Proxy Kernel (PK), allowed us to benchmark our VPU-integrated core effectively. We demonstrated how the toolchain facilitates the comparison of instruction sizes and compilation times between the scalar core and our VPU-integrated core, underscoring the benefits of our approach.

We have four lanes operating in parallel, processing 4 elements of a vector at a time. For example, if we add two vectors, the scalar core will complete the addition in 32 cycles, whereas the VPU-integrated PicoRV32 will complete it in just 8 cycles. The number of cycles required for load and store operations remains the same.

In summary, the integration of the VPU with the RISC-V core provides a powerful enhancement to parallel processing capabilities, significantly reducing instruction counts and improving overall performance for vector operations. This work lays a strong foundation for future research and development in the field of vector processing, offering promising avenues for optimizing computational efficiency in various applications.

6.2 Future Work

In our future work, we plan to enhance the capabilities of our VPU by adding vector multiplication operations. Additionally, we aim to implement UVM (Universal Verification Methodology) based verification for the VPU. This will ensure thorough testing and validation of the VPU's functionality and performance, helping to identify and rectify any potential issues. These enhancements will further improve the computational efficiency and reliability of our VPU-integrated PicoRV32 core, making it even more suitable for complex vector-based computations.

Moreover, the Floating Point Unit (FPU) can be added to perform arithmetic operations on decimal numbers. Our VPU follows a 32-bit architecture but can also be extended to 64-bit architecture depending on the requirements

References

[1] Joao Filipe Monteiro Rodrigues. *Configurable RISC-V softcore processor for FPGA implementation*. PhD thesis, Master's thesis, Instituto Superior Técnico, 2019.

[2] Niklas Bruns, Vladimir Herdt, and Rolf Drechsler. Processor verification using symbolic execution: A risc-v case-study. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[3] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. An agile approach to building risc-v microprocessors. *ieee Micro*, 36(2):8–20, 2016.

[4] Tyler McGrew, Eric Schonauer, and Peter Jamieson. Framework and tools for undergraduates designing risc-v processors on an fpga in computer architecture education. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 778–781. IEEE, 2019.

[5] Benjamin W Mezger, Douglas A Santos, Luigi Dilillo, Cesar A Zeferino, and Douglas R Melo. A survey of the risc-v architecture software support. *IEEE Access*, 10:51394–51411, 2022.

[6] Semico Research. RISC-V Market Analysis: The New Kid on the Block.

[7] Joao Filipe Monteiro Rodrigues. *Configurable RISC-V softcore processor for FPGA implementation*. PhD thesis, Master's thesis, Instituto Superior Técnico, 2019.

[8] Enrique Torres-Sánchez, Jesús Alastruey-Benedé, and Enrique Torres-Moreno. Developing an ai iot application with open software on a risc-v soc. In *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2020.

[9] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8. IEEE, 2017.

[10] Shuenn-Yuh Lee, Yi-Wen Hung, Yao-Tse Chang, Chou-Ching Lin, and Gia-Shing Shieh. Risc-v cnn coprocessor for real-time epilepsy detection in wearable application. *IEEE transactions on biomedical circuits and systems*, 15(4):679–691, 2021.

[11] Alexander Dörflinger, Yejun Guan, Sören Michalik, Sönke Michalik, Jamin Naghmouchi, and Harald Michalik. Ecc memory for fault tolerant risc-v processors. In *Architecture of Computing Systems–ARCS 2020: 33rd International Conference, Aachen, Germany, May 25–28, 2020, Proceedings 33*, pages 44–55. Springer, 2020.

[12] Stefano Di Mascio, Alessandra Menicucci, Eberhard Gill, Gianluca Furano, and Claudio Monteleone. Leveraging the openness and modularity of risc-v in space. *Journal of Aerospace Information Systems*, 16(11):454–472, 2019.

[13] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. *Philosophical Transactions of the Royal Society A*, 378(2164):20190155, 2020.

[14] Manupoti Sreenivasulu and T Meenpal. Efficient hardware implementation of 2d convolution on fpga for image processing application. In *2019 IEEE International*

Conference on Electrical, Computer and Communication Technologies (ICECCT), pages 1–5. IEEE, 2019.

[15] Sylvain Paris, Samuel W Hasinoff, and Jan Kautz. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. *ACM Trans. Graph.*, 30(4):68, 2011.

[16] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(2):530–543, 2019.

[17] Marium Masood, Yame Asfia, and Sajid Gul Khawaja. Security challenges faced by risc-v open-source processors and its security features: A survey. In *2023 25th International Multitopic Conference (INMIC)*, pages 1–7. IEEE, 2023.

[18] Gareth Halfacree. Sifive's risc-v cores launch in two ssd families, 2018.

[19] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijjan Chen, et al. Xuantie-910: Innovating cloud and edge computing by risc-v. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–19. IEEE Computer Society, 2020.

[20] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, et al. Blackparrot: An agile open-source risc-v multicore for accelerator socs. *IEEE Micro*, 40(4):93–102, 2020.

[21] Rwan Mahmoud, Tasneem Yousuf, Fadi Aloul, and Imran Zualkernan. Internet of things (iot) security: Current status, challenges and prospective measures.

[22] Clifford Wolf et al. Picorv32-a size-optimized risc-v cpu. *Accessed: Dec*, 19, 2019.

[23] Radio frequency fingerprint-based intelligent mobile edge computing for internet of things authentication. *Sensors (Switzerland)*, 19, 8 2019.

[24] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4:6–2, 2016.

[25] Christopher Celio, David Patterson, and Krste Asanovic. The berkeley out-of-order machine (boom) design specification. *University of California, Berkeley*, 2016.

[26] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. A comparative survey of open-source application-class risc-v processor implementations. In *Proceedings of the 18th ACM international conference on computing frontiers*, pages 12–20, 2021.

[27] Joao Filipe Monteiro Rodrigues. *Configurable RISC-V softcore processor for FPGA implementation*. PhD thesis, Master's thesis, Instituto Superior Técnico, 2019.

[28] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[29] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[30] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116:1–32, 2011.

[31] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2:1–79, 2014.

[32] Andrew Waterman, Krste Asanovic, et al. The risc-v instruction set manual volume i: Unprivileged isa. *Document Version*, 20191213:1–4, 2019.

[33] Andrew Waterman, Krste Asanovic, et al. The risc-v instruction set manual volume i: Unprivileged isa. *Document Version*, 20191213:1–4, 2019.

[34] RISC-V Foundation. Release scalar cryptography v1.0.0-rc2, 2023. Accessed: 2023-04-03.

[35] Towards 6g networks: Use cases and technologies. 3 2019.