Implementation of Vector Instructions in the RISC-V Rocket Processor

Iago Caran Aquino

Relatório Técnico - IC-PFG-23-62
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Implementation of Vector Instructions in the RISC-V Rocket Processor

Iago Caran Aquino [*] Lucas Wanner[*] Sandro Rigo[*]

Abstract

With the consolidation of the vector extension of the RISC-V ISA in 2021, its application to highly parallelizable workloads such as HPC and Machine Learning becomes mandatory. Still, there is currently no open implementation of this extension for the Rocket processor. With this, we aimed to create a proof of concept of a vector unit for this processor, serving as a basis for future developments. The theoretical analysis indicates benefits to be extracted from this approach that differ from the experimental results, due to implementation issues that require optimization. Future efforts should lead to significant performance gains, in addition to execution in an FPGA environment for energy efficiency studies.

Resumo

Com a consolidação da extensão vetorial para a ISA RISC-V em 2021, sua aplicação em cargas de trabalho altamente paralelizáveis, como HPC e Machine Learning, torna-se mais desejável. Ainda assim, não há no momento uma implementação aberta desta extensão para o processador Rocket. Com isso, tivemos como objetivo criar uma prova de conceito de unidade vetorial para este processador, servindo de base para futuros desenvolvimentos. A unidade implementada na linguagem Chisel dentro do framework Chipyard suporta algumas das instruções da especificação, a partir das quais caracterizamos seu desempenho atual. A análise teórica indica benefícios a serem extraídos desta abordagem que diferem dos resultados experimentais, devido a questões na implementação que exigem otimização. Futuros esforços devem levar a ganhos significativos de desempenho, além da execução em ambiente de FPGA para estudos de eficiência energética.

[*]Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

1 Introduction

An instruction set architecture (ISA) is an abstract model of a computer that allows a programmer to write code for a set of processors that implements that ISA. These models are, for the most part, proprietary, created and maintained by large companies such as Intel and ARM, which determine the entire functioning of their ISAs and charge high amounts for licensing and customization. In this context, the RISC-V[1] ISA emerges as an open and free alternative for developing chips and software, created by the University of California, Berkeley, and managed by the RISC-V International.

The RISC-V ISA receives attention from multiple companies for its development and application in the most varied fields of application, such as High-Performance Computing (HPC) and Machine Learning, both benefiting from vectorization for enhancing performance. In the HPC domain, data parallelism is used for increased throughput, allowing for operations on large datasets. Meanwhile, in machine learning, executing large-scale matrix and tensor operations is the basis for training and inference processes, which are highly scalable from efficient parallelization.

The increasing demand in these areas stimulated the development of specialized vector accelerators, exemplified by the Hwacha vector unit[2] created by the Berkeley Architecture Research team, which features a custom ISA extension with particular emphasis on energy efficiency. Hwacha has successfully created a System-On-Chip (SoC)[3], accentuating its practical applicability. Additionally, various commercial implementations adhere to the vector specifications[4] and implementations outside the Chipyard ecosystem, like the RISC-V2[5].

Instruction sets incorporate specific instructions to speed up and standardize the interface of certain operations. Modern processors from companies like Intel and ARM have instructions dedicated to these tasks, named AVX[6] and SVE[7], respectively. In 2021, RISC-V consolidated its "V" extension[8] for vector instructions. Still, as of today, no open implementation exists for the Rocket processor, the reference processor developed by the creators of the ISA. Therefore, this project aims to fill this gap by integrating some of these instructions into the Rocket processor.

2 Methodology

2.1 Chipyard Ecosystem

The Rocket Chip[9] is developed around the Chipyard framework. It assembles multiple tools to enable the design, evaluation, and implementation of an SoC comprising the processor core, memory, and all peripherals necessary for its operation. It also bundles all the software needed for compiling and simulating code with the GNU Compiler Collection (GCC) and the Spike simulator, as well as tools for FPGA-accelerated simulation and the Hammer VLSI flow for physical design. However, it requires access to proprietary EDA tools and process technology libraries.

The project is written in the Chisel hardware description language[10], an extension of the Scala language that describes complex and parameterizable circuits utilizing objectoriented and functional programming features.

2.2 Tools

The following tools were installed to run the project on an Ubuntu 22.04 system. It is important to highlight that the process was done with version 1.8.1 of Chipyard and that some steps may change for other versions.

The code in Figure 1 shows the configuration of environment variables and installation of dependencies done before all other steps.

```
export RISCV=/opt/riscv
export PATH=$PATH:$RISCV/bin

sudo apt update
sudo apt install -y git help2man jq openjdk-8-jdk default-jdk
     default-jre libusb-dev rsync libguestfs-tools expat bc
   libncurses5-dev gengetopt gettext software-properties-
   common cmake device-tree-compiler libpixman-1-dev libgtk
   -3-dev autoconf automake autotools-dev curl python3 libmpc
   -dev libmpfr-dev libgmp-dev gawk build-essential bison
   flex texinfo gperf libtool patchutils bc zlib1g-dev
   libexpat-dev ninja-build scala
```

Figure 1: Environment variable configuration and dependency installation

2.2.1 GNU Compiler Collection

The GNU Compiler Collection (GCC) is a collection of tools produced by the GNU Project supporting different programming languages, hardware architectures, and operating systems. GCC is distributed as free software under the GNU General Public License by the Free Software Foundation.

As part of the riscv-tools, a collection of software toolchains used to develop and execute software on the RISC-V ISA, GCC was also extended to target the RISC-V ISA. Although the official GCC already supports RISC-V, the version distributed with riscv-tools will always have the latest additions to the RISC-V spec, which is why we chose to compile it from source code, following the steps in Figure 2.

2.2.2 Proxy Kernel

The RISC-V Proxy Kernel (pk) is another software included in the riscv-tools that enables support for statically linked RISC-V ELF binaries in systems with limited I/O capability, handling related system calls by proxying them to a host computer. This approach facilitates the execution of code compiled without the modifications needed for bare-metal targets, like the spike emulator.

```
git clone --recursive https://github.com/riscv/riscv-gnu-
    toolchain
cd riscv-gnu-toolchain
git submodule update --recursive
./configure --prefix=$RISCV --enable-multilib --with-arch=
    rv64imafdcv_zicsr_zifencei

sudo make -j`nproc`
sudo make linux -j`nproc`
sudo make install -j`nproc`
```

Figure 2: GCC installation steps

```
git clone https://github.com/riscv/riscv-pk.git
cd riscv-pk
mkdir build && cd build
../configure --prefix=$RISCV --with-arch=
    rv64imafdcv_zicsr_zifencei --host=riscv64-unknown-linux-
    gnu
make -j`nproc`
make install
```

Figure 3: Proxy Kernel installation steps

2.2.3 riscv-tests

A suite of unit tests for RISC-V processors is available under the riscv-tests repository. Although the tests were not used in this project's scope, the compilation scripts were our reference to target bare-metal execution. The steps in Figure 4 illustrate the installation.

```
git clone https://github.com/riscv/riscv-tests
cd riscv-tests
git submodule update --init --recursive
autoconf
./configure --prefix=$RISCV/target
make -j`nproc`
make install
```

Figure 4: riscv-tests installation steps

2.2.4 Spike

Spike is the golden reference functional RISC-V ISA C++ software simulator, providing full system emulation. It is maintained by the same people who maintain the RISC-V ISA and, therefore, is the first simulator to get new instructions.

It is a functional simulator that does not replicate the microarchitecture and timing details. It also serves as a starting point for running software on a RISC-V target and supports most extensions.

The simulator was compiled from the source code and used to verify the behavior of our tests.

```
git clone https://github.com/riscv/riscv-pk.git
cd riscv-pk
mkdir build && cd build
../configure --prefix=$RISCV --host=riscv64-unknown-linux-gnu
make -j`nproc`
make install
```

Figure 5: Spike installation steps

2.2.5 Verilator

Verilator is an open-source Verilog simulator that compiles into an optimized model wrapped inside a multithreaded C++ module, which GCC then compiles.

The high-level tool then compiles the FIRRTL code, generating specialized Verilog code for different targets such as FPGA, ASIC, and simulation. The Verilog output is then compiled into a multithreaded C++ code by the Verilator simulator, which generates an executable used to run a cycle-accurate simulation of code execution.

```
git clone https://github.com/verilator/verilator
cd verilator
git checkout stable
autoconf
./configure
make -j`nproc`
sudo make install
```

Figure 6: Verilator installation steps

2.3 RISC-V Instruction Set

The RISC-V ISA does not define a singular set of instructions but a foundation on which other extensions can be incorporated for different specializations. Each distinct instruction set is represented by a letter, with ”I” being the essential one for establishing a compiler target and supporting modern operating system environments. Alongside these extensions, the specification also has two main variants for 32-bit and 64-bit address spaces, further expanding the flexibility of the ISA.

Some of these extensions are:

- M: Integer multiplication and division
- A: Atomic instructions
- F: Single-precision floating point
- D: Double-precision floating point
- C: Compact instructions

Besides those, other extensions define instructions for specific tasks, as is the case with the ”V” extension for vector manipulation, which defines operations such as addition, subtraction, and multiplication of vectors. For this project, the Rocket Core is configured with RV64I support.

2.4 Rocket Core

The Rocket Core is a scalar processor core[11], originally developed at the University of California, Berkeley, and currently supported by SiFive, that implements the RV64I specification and can be customized with various extensions. The core follows a 5-stage in-order pipeline, illustrated in Figure 7, which means each stage can only advance when the next one is ready, simplifying execution flow control and avoiding complex synchronization efforts as the added vector instructions take multiple cycles to complete.

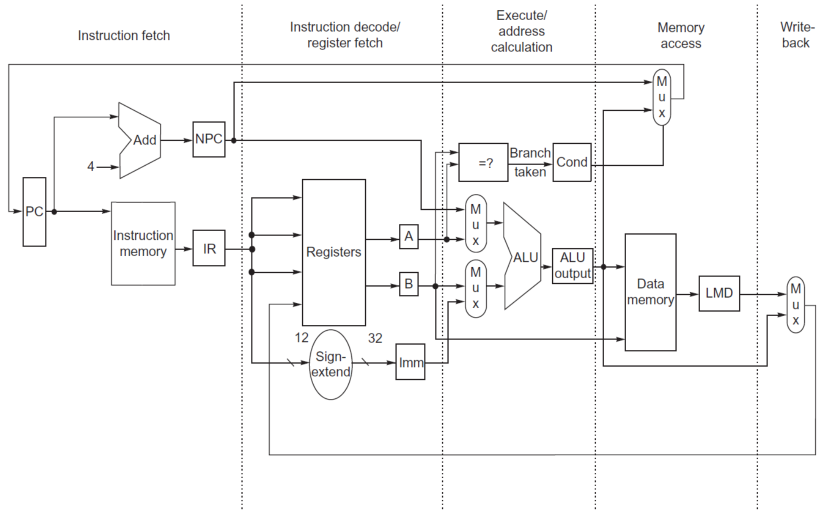

Figure 7: Illustration of the five-stage pipeline[12]

The Rocket Chip generator combines a Rocket Core, a page-table walker, an L1 instruction cache, and an L1 data cache into a RocketTile that may feature accelerator cores connected as coprocessors. The tile is then linked to the SystemBus, where the L2 Cache and other peripherals are connected to complete the SoC.

2.5 RISC-V ”V” Vector Extension

The RISC-V ”V” Vector Extension adds instructions for data-level parallelism using a set of 32 vector registers that can be configured for multiple data and vector sizes. Each register has a fixed-size VLEN and can hold multiple vector elements. The instructions allow for operating over all the elements at the same time, characterizing a Single-Instruction Multiple-Data (SIMD) processing.

In the spirit of a reduced instruction set, the vector extension implements a configuration instruction that allows the programmer to specify the data type represented in the registers, thus reducing the need for specialized instructions for each data type. For example, if VLEN is 128 bits, the unit can be configured for each register to hold two 64-bit values, four 32-bit values, eight 16-bit values, or sixteen 8-bit values. The first three configurations are illustrated in Figure 8. After configuration, instructions may be used for integer, fixed-point, and floating-point arithmetic, vector reduction, masking, and permutation operations.

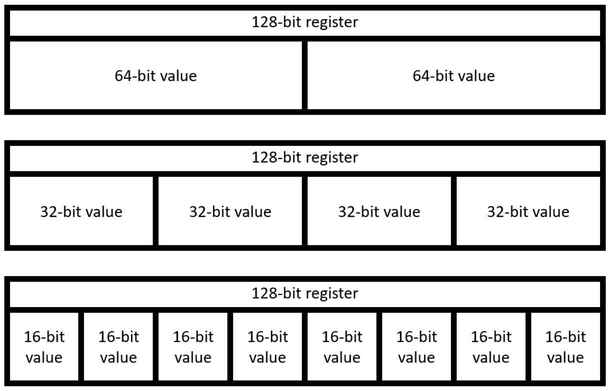

Figure 8: Illustration of 128-bit register usage

3 The Vector Unit

In this project, we designed a vector unit integrated directly into the processor core, similar to the floating-point unit. It is integrated within the core via multiple points in the pipeline: the fetch stage to receive the instructions for processing, assuming control of the execution flow upon identifying a vector instruction; the decode stage to receive the values of the scalar registers; and the writeback stage to write to the scalar registers.

Moreover, the vector unit is connected to the L1 cache to access memory for loading and storing vectors. These connections are depicted in the diagram of Figure 9.

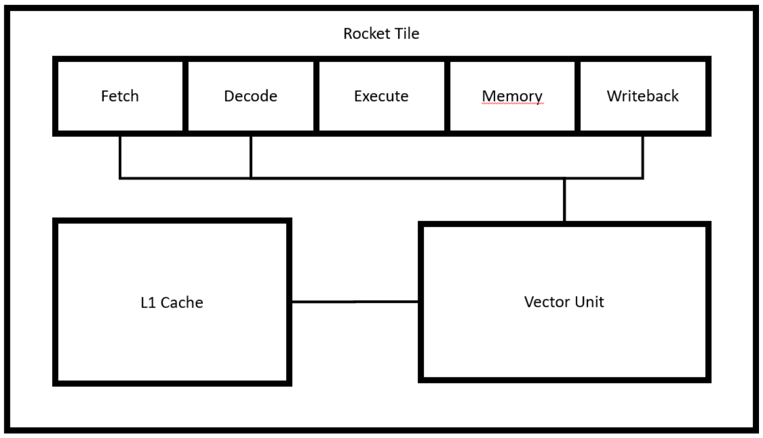

Figure 9: Illustration of vector unit connection to rocket core

The connections exposed by the vector unit are defined in the VPUCoreIO class, as illustrated by the code snippet in Figure 10. The inputs consist of the system clock, a cache interface, the actual instruction, and the two register values, and outputs the writeback data and a signal to stall the core pipeline while processing is not finished. There are also connections to the Control and Status Registers, but their behavior is not implemented yet.

The cache interface works in a request/response model, featuring multiple request interfaces connected to an arbiter. This arbiter selects the next request for processing once the cache becomes available. In contrast, the response interface is shared, requiring the connected modules to verify its identification when a response becomes available. The *HellaCacheIO* interface contains both request and response signals specified by the *HellaCacheReq* and *HellaCacheResp* classes, respectively. The main characteristic of the request class is the *HasCoreMemOp* trait, which defines the memory operation interface.

The code snippet in Figure 11 shows the *HasCoreMemOp* definition where the key signals are: *tag*, serving as an identifier for the request; *addr*, representing the memory address to access; *cmd*, indicating the type of operation (read or write); and *size*, indicating the number of words to be retrieved.

The response interface implements, alongside the *HasCoreMemOp* trait, the *HasCoreData* trait, Figure 12. As implied by the name, it encapsulates the requested data. It's important to note that the response field is declared within a *Valid* context, indicating that alongside the data, there is a valid signal to specify the data is ready to be consumed.

Internally, the vector unit follows the core five-stage pipeline, where the execution and memory stages may span multiple cycles to complete their operations, as is the case for multiplication instructions.

The first step to set up vector operations involves executing the *vsetvli* configuration instruction to define the data type under consideration. Subsequently, the *vle* instruction is employed to load the data into the registers, followed by a combination of operation instructions acting simultaneously on all loaded elements. Finally, the outcome is recovered from the vector unit with the *vse* instruction.

It is important to note that the *vle* and *vse* instructions encode the element size directly

```
class VPUCoreIO(implicit p: Parameters) extends CoreBundle()(
    p) {
  val clock = Input(Clock())
  val mem = new HellaCacheIO
  val killd = Input(Bool())
  val killx = Input(Bool())
  val killm = Input(Bool())
  val inst = Input(Bits(32.W))
  val op1_data = Input(Bits(xLen.W))
  val op2_data = Input(Bits(xLen.W))
  val wb_data = Output(Valid(Bits(xLen.W)))
  val stall_pipeline = Output(Bool())
  val csr = new Bundle {
    val vconfig = Input(new VConfig())
    val vstart = Input(UInt(maxVLMax.log2.W))
    val vxrm = Input(UInt(2.W))
    val set_vs_dirty = Output(Bool())
    val set_vconfig = Flipped(Valid(new VConfig))
    val set_vstart = Flipped(Valid(vstart))
    val set_vxsat = Output(Bool())
  }
  val status = Input(new MStatus)
}
```

Figure 10: VPUCoreIO class implementation

within them and do not rely on the vector unit configuration. This feature reduces the need to call *vsetvli* repeatedly when loading multiple vectors, for mixed-precision operations, and for accommodating memory layouts without affecting the entire vector unit configuration.

As mentioned before, the VLEN parameter defines the size of the vector registers. Combined with the element size configuration, it dictates the number of elements manipulated by one instruction. For instance, one instruction would operate on four data values with VLEN set to 128 bits and elements sized at 32 bits. In contrast, for VLEN set to 256 bits and elements of 16 bits, the same instruction could manipulate as many as sixteen values.

```
trait HasCoreMemOp extends HasL1HellaCacheParameters {
  val addr = UInt(coreMaxAddrBits.W)
  val idx  = (usingVM && untagBits > pgIdxBits).option(UInt(
     coreMaxAddrBits.W))
  val tag  = UInt((coreParams.dcacheReqTagBits + log2Ceil(
     dcacheArbPorts)).W)
  val cmd  = UInt(M_SZ.W)
  val size = UInt(log2Ceil(coreDataBytes.log2 + 1).W)
  val signed = Bool()
  val dprv = UInt(PRV.SZ.W)
  val dv = Bool()
}
```

Figure 11: HasCoreMemOp

```
trait HasCoreData extends HasCoreParameters {
  val data = UInt(coreDataBits.W)
  val mask = UInt(coreDataBytes.W)
}
```

Figure 12: HasCoreData

4 Experimental Results

The following sections will explain some applications of vector operations and show scalar and vector implementations of the algorithms, as well as our results from verilator simulation.

4.1 Vector addition

A direct application of vector instructions is a vector addition, denoted as the element-wise sum of two vectors $\vec{A}$ and $\vec{B}$, each comprising n elements. The individual components of the resultant vector $\vec{C} = \vec{A} + \vec{B}$ are defined as:

$$c_i = a_i + b_i \tag{1}$$

for $i = 1, ..., n$

Implementing a vector addition is not as simple as applying one instruction because the vector registers are limited in size. Therefore, we need to iterate over several elements that fit in the registers. Let's call this parameter "number done" ($t0$). The code snippet in Figure 13, extracted from the examples provided in the RISC-V "V" Specification, shows an implementation of the algorithm.

```
vvaddint32:
  vsetvli t0, a0, e32, ta, ma
  vle32.v v0, (a1)
    sub a0, a0, t0
    slli t0, t0, 2
    add a1, a1, t0
  vle32.v v1, (a2)
    add a2, a2, t0
  vadd.vv v2, v0, v1
  vse32.v v2, (a3)
    add a3, a3, t0
    bnez a0, vvaddint32
    ret
```

Figure 13: Vectorized implementation of vector addition

Comparing this implementation with the scalar counterpart depicted in Figure 14, it is evident that the "number done" ($t0$) being higher than one makes the loop counter increment faster than in the scalar implementation, leading to a lower number of iterations.

```
vvaddint:
  lw t0, 0(a1)
  lw t1, 0(a2)
  add t2, t0, t1
  sw t2, 0(a3)
  addi a1, a1, 4
  addi a2, a2, 4
  addi a3, a3, 4
  addi t3, t3, 1
  bnez a0, vvaddint
```

Figure 14: Scalar implementation of vector addition

Table 1 compares the number of instructions used by both codes for different input sizes and, for the vectorized version, VLEN sizes with 32-bit elements.

The parallel processing capability of vector operations substantially reduces the number of iterations executed for the vector addition, amounting to less than half the number of instructions executed compared to the scalar counterpart. This reduction doesn't translate directly into reducing the number of cycles, as some instructions take more cycles to be executed, like the memory operations that will take analogous time, since the amount of data fetched is comparable.

Some memory limitations are mitigated when the element size is reduced since one vector memory step can fetch multiple elements in the same request, as opposed to the

	Number of Instructions			
VLEN	-	128	256	512
Input size	Scalar Code	Vector Code		
8	72	22	11	11
16	144	44	22	11
32	288	88	44	22
64	576	176	88	44
128	1152	352	176	88

Table 1: Comparison of the number of instructions for vector size addition

single-element behavior of the scalar code.

Another limitation shown in Table 1 is the parallelism limitation. As VLEN grows, the number of instructions for the same input size decreases. Still, the vector registers are underused for very small input sizes, and therefore, the number of instructions stagnates.

Table 2 compares the number of cycles taken between the two approaches using Verilator simulation, meaning microarchitecture, cache, and timings are considered. The tests used 64-bit elements and a VLEN size of 128 bits, which do not benefit as much from parallelism yet simplifies the memory interface requests.

	Number of Cycles	
Input size	8	16
Scalar Code	168	255
Vector Code	143	254

Table 2: Comparison of number of cycles for vector size addition

The initial implementation does not show a great benefit of the vector approach. The unexpected result occurs due to a pipeline interlock problem in the memory access stage, resulting in four extra cycles per memory instruction. Furthermore, an untreated data hazard occurs between a vector load and a vector operation forces the addition of a *nop* instruction after each vector loading, increasing the cycle count. Further developments of the vector unit can mitigate these problems leading to significant performance gains in future versions.

4.2 Dot Product

Another application of vector instructions is the dot product or scalar product, denoted as the sum of the element-wise multiplication of two vectors $\vec{A}$ and $\vec{B}$, each comprising n elements. The resulting value $C = \vec{A} \cdot \vec{B}$ is defined as:

$$C = \sum_{i=1}^{n} a_i \cdot b_i \tag{2}$$

for $i = 1, ..., n$

The implementation illustrated in Figure 15 shows the use of two other instructions: *vmul.vv*, an element-wise multiplication of the elements of both vectors; and *vredsum.vs*, a reduction of the elements of a vector into a scalar by summing. These instructions can potentially speed up the process with parallelism, as opposed to the multiple instructions used in the scalar implementation in Figure 16.

```
dotprodint64:
  vsetvli t0, a0, e64, ta, ma
  vle64.v v0, (a1)
    sub a0, a0, t0
    slli t0, t0, 3
    add a1, a1, t0
  vle64.v v1, (a2)
    add a2, a2, t0
  vmul.vv v2, v0, v1
  vredsum.vs v3, v2, v3
    bnez a0, dotprodint64
  vse64.v v3, (a3)
```

Figure 15: Vectorized implementation of dot product

```
dotprod:
addi t3, zero, 0
loop:
  lw t0, 0(a1)
  lw t1, 0(a2)
  mul t2, t0, t1
  add t3, t3, t2
  addi a1, a1, 8
  addi a2, a2, 8
  addi a0, a0, -1
  bnez a0, loop
addi a3, t3, 0
```

Figure 16: Scalar implementation of dot product

Table 3 compares the number of instructions used by both codes for different input sizes and, for the vectorized version, VLEN sizes with 32-bit elements. Once again, there is good evidence that data-level parallelism can considerably speed up the application.

Table 4 compares the number of cycles taken between both approaches using the implemented vector unit, with the same parameters as before, 64-bit elements, and a VLEN size of 128-bits.

	Number of Instructions			
VLEN	-	128	256	512
Input size	Scalar Code	Vector Code		
8	66	21	11	11
16	130	41	21	11
32	258	81	41	21
64	514	161	81	41
128	1026	321	161	81

Table 3: Comparison of the number of instructions for the dot product

	Number of Cycles	
Input size	8	16
Scalar Code	154	257
Vector Code	145	271

Table 4: Comparison of number of cycles for dot product

Despite the signs of possible improvements, we again fell into an implementation limitation, requiring further optimization to achieve the desired results.

5 Conclusion

The primary objective of this study was to implement a proof-of-concept vector unit integrated into the Rocket Core and evaluate its impact on executing code that benefits from data parallelization.

This work serves as a first step toward establishing a standard implementation of vector instructions for the Rocket processor, and future endeavors may expand upon the current results to cover the complete vector extension specification. Further research may deploy the vector unit on an FPGA platform and perform a power analysis characterization to investigate efficiency in different applications.

The experimental results along with the analysis of the examples demonstrated considerable reductions in instruction count, but not as much for cycle counts. After the addition of more instructions and optimizations, that should lead to significant performance gains in future versions.

For those interested in delving deeper into the details of our work, the source code for the Rocket Core with Vector Unit is openly accessible in the following repository: https://github.com/iagocaran/rocket-chip. More information about the tests and tooling is available in the following repository: https://github.com/iagocaran/PFG-23-62. We encourage the community to explore, contribute, and build upon this foundation.

References

[1] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. A. Patterson, and K. Asanovic, "The risc-v instruction set.," in *Hot Chips Symposium*, p. 1, 2013.

[2] C. Schmidt, A. Ou, and K. Asanović, "Hwacha v4: Decoupled data parallel custom extension," in *Proc. Inaugural RISC-V Summit*, pp. 1–40, 2018.

[3] J. C. Wright, C. Schmidt, B. Keller, D. P. Dabbelt, J. Kwak, V. Iyer, N. Mehta, P.-F. Chiu, S. Bailey, K. Asanović, and B. Nikolić, "A dual-core risc-v vector processor with on-chip fine-grain power management in 28-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 12, pp. 2721–2725, 2020.

[4] J. K. L. Lee, M. Jamieson, N. Brown, and R. Jesus, "Test-driving risc-v vector hardware for hpc," 2023.

[5] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos, "Risc-v2: A scalable risc-v vector processor," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2020.

[6] C. Lomont, "Introduction to intel advanced vector extensions," *Intel white paper*, vol. 23, pp. 1–21, 2011.

[7] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, *et al.*, "The arm scalable vector extension," *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.

[8] Y. Kimura, T. Kikuchi, K. Ootsu, and T. Yokota, "Proposal of scalable vector extension for embedded risc-v soft-core processor," in *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*, pp. 435–439, 2019.

[9] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), p. 1216–1225, Association for Computing Machinery, 2012.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 6th ed., 2017.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 6th ed., 2017.