A Performance Evaluation of a Fault-tolerant RISC-V with Vector Instruction Support to Space Applications

Carolina Imianosky, Douglas Almeida dos Santos, Douglas Rossi de Melo, Luigi Dilillo, Cesar Albenes Zeferino, Eduardo Augusto Bezerra, Felipe Viel

Title: A Performance Evaluation of a Fault-tolerant RISC-V with Vector Instruction Support to Space Applications

Author(s): Carolina Imianosky, Douglas A. Santos, Douglas R. Melo, Luigi Dilillo, Cesar A. Zeferino, Eduardo A. Bezerra, and Felipe Viel

Document version: Post-print version (Final draft)

Please cite the original version:
C. Imianosky et al., ”A Performance Evaluation of a Fault-tolerant RISC-V with Vector Instruction Support to Space Applications,” 2022 5th IAA Latin American CubeSat Workshop and 3rd IAA Latin American Symposium on Small Satellites (IAA-LA), 2022, pp. 1-9.

A Performance Evaluation of a Fault-tolerant RISC-V with Vector Instruction Support to Space Applications

Carolina Imianosky[(1)], **Douglas A. Santos**[(2)], **Douglas R. Melo**[(1)], **Luigi Dilillo**[(2)], **Cesar A. Zeferino**[(1)], **Eduardo A. Bezerra**[(3)], **Felipe Viel**[(1),(3)]

[(1)]LEDS, University of Vale do Itajaí, Brazil
[(2)]LIRMM, University of Montpellier, CNRS, France
[(3)]SpaceLab, Federal University of Santa Catarina, Brazil

Keywords: *RISC-V, Vector Instructions, Fault tolerance, Space Applications*

Computational systems used in space systems have constantly been evolving in several aspects, ranging from the ability of tolerate faults and failures to process large volumes of data. These aspects mainly affect the characteristics of the processing cores, ranging from microcontrollers to microprocessors, impacting in the computer architecture and organization used. One instruction set architecture under extensive study for application in the space environment is the RISC-V. This architecture has been widely used in space systems because it is simple, open, and modular, enabling the application of techniques that mitigate faults caused in a space environment. However, the application of these techniques affects the performance of the components. Thus, it affects the high-resolution data captured by the sensors, which needs to be processed before being transmitted to Earth. Therefore, it is necessary to apply techniques that accelerate the processing of this data. As a solution to the demand for an increase in processing performance, RISC-V can support vector instructions, which allow operating on a vector of data with only one instruction. This approach allows exploring levels of data parallelism and improving the acceleration of applications. Therefore, we developed support for a subset of the vector extension for an existing functional fault-tolerant RISC-V processor. We analyzed the vector instructions that are relevant to digital signal processing, since it is a time-costly type of processing, to define the instructions that constitute the subset. Thus, we implemented only sequential memory access, addition, and subtraction vector instructions. We evaluated the impact of using these instructions compared to scalar instructions, analyzing the execution time, logical resource utilization, and power consumption. The results showed a performance improvement of up to 4x when using the vector instructions relative to the scalar instructions, but, there was a hardware overhead of 1.5x for consumed Lookup Tables and 1.8x for Flip-Flop. Besides the hardware overhead, this cost is negligible compared to the acceleration offered.

1. Introduction

Space systems are affected by the harsh elements of the space environment, such as radiation, extreme temperatures, and lack of gravity. These elements may lead

Email addresses: `imianosky@edu.univali.br` (**Carolina Imianosky**), `viel@univali.br` (**Felipe Viel**)

to faults that can affect the functioning of computational systems. Thus, it is important to apply fault-tolerance techniques in spacecrafts design to improve the systems reliability [1].

The processor implemented by Santos et al. [2], also known as HARV, is an example of technology for space systems, in which the authors developed a low-cost fault-tolerant RISC-V processor. This work used fault-tolerance techniques to prevent possible errors due to its exposition to harsh environments. Also, in [3], the authors analyzed the impact of Deep Neural Networks (DNNs) for space applications. Therefore, they evaluated the use of DNNs in a RISC-V vector processor and presented a several recommendations to systematically enable OBDM with RISC-V vector processors. Thus, as it can be observed, there is a tendency to use the RISC-V ISA (Instruction Set Architecture) as processing core for space applications.

RISC-V is an ISA that has gained popularity within the past few years. This is mostly due to its simplicity, openness, and modularity, i.e., composed by a base ISA and optional extensions [4]. Among the various ISA extensions defined in the standard, there is the vector extension. With this extension, the systems can perform an instruction on multiple data elements simultaneously [5]. Thus, it is possible to explore high data parallelism levels, decreasing processing latency and power consumption. These factors are critical in space systems since they have limitations regarding energy harvesting [6].

Currently, some commercial and academic cores implement or are based on the RISC-V Vector (RVV) extension. For instance, Arrow [7] is a configurable co-processor aimed at edge machine learning inference that implements a subset of RVV version 0.9. Further, Johns and Kazmierski [8] implemented a subset of the RVV v0.8 extension instructions to a RISC-V processor to satisfy microcontrollers area and power consumption requirements. Also, Ara [9] is a parametric in-order high-performance 64-bit vector unit based on RVV version 0.5 that works in tandem with Ariane. However, although these works explore the RVV extension, to the best of our knowledge, there are no works that implement the RVV extension for a fault-tolerant processor.

Given the context above, this work has as its main contribution the support development of a vector instructions subset for a low-cost fault-tolerant RISC-V processor developed in [2]. Our primary focus was to improve the performance of signal processing applications. The experiments comprised the execution of basic arithmetic operations and memory read and write with scalar and vector instructions to compare the impact of using vector instructions.

The remainder of this paper is organized as follows. Section II presents additional concepts on hardware acceleration, RISC-V ISA, and its vector extension. Next, Section III discusses related work, and Section IV describes the materials and methods employed in this study. Following, Section V presents and discusses the experimental results, while Section VI gives the final remarks.

2. Background

2.1. Hardware Acceleration

One way to speed up computational processing is using the Data-Level Parallelism (DLP) concept, which is used to operate on multiple data elements simultaneously [5]. Single Instruction, Multiple Data (SIMD) is one class of processors exploring the DLP. This architecture became popular in the 1970s for partitioning 64-bit registers into multiple 8-, 16- or 32-bit registers and operating them in parallel. This way, a single instruction operates on several data array elements within only one instruction fetch and decode. The Operation Code (Opcode) supplied the data width and the operation [10].

Subsequently, to speed up SIMD architectures, the architects increased the width of the registers to process more elements in parallel. Once data and operation width are indicated in the instructions Opcode, expanding the SIMD registers also implies increasing the instruction set [10]. Thus, SIMD instructions operate in fixed-size registers that are generally set to 128 bits wide [11].

An alternative to SIMD architecture that explores DLP is the vector architecture, represented in Fig. 1. This architecture gathers objects from the main memory, puts them into exclusive vector registers, operates on these registers, and then scatters the results back to the main memory. However, unlike SIMD, the size of vector is determined by the implementation rather than described in the Opcode [10]. Consequently, vector architectures require a smaller instruction set and make hardware design flexible to parallelize data without affecting algorithms development.

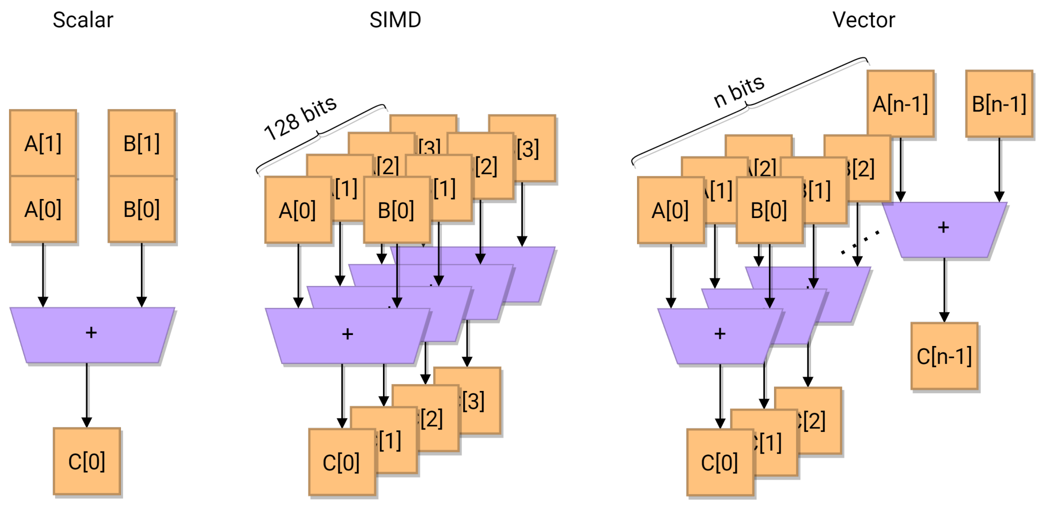

Figure 1: Addition instruction in scalar, SIMD and vector architecture [11].

2.2. RISC-V

The RISC-V ISA was developed at the University of California, Berkeley, based on the Reduced Instruction Set Computer (RISC) architecture [12]. This ISA has a highly regular instruction encoding and simple memory access instructions with a direct memory model. One of the benefits of RISC-V is the minimal size of simple cores, which are much smaller compared to Advanced RISC Machine (ARM) and x86 architectures. However, the difference is not noticeable on higher-capacity cores [13].

The RISC-V structure is modular, i.e., it comprises a base instruction set and a variety of optional extensions. The three main base instruction sets are RV32I, RV32E, and RV64I. The first one is a 32-bit set with 47 instructions, enough to fill basic requirements of the modern operational systems. The other two sets are very similar. The RV32E is a variation with only 15 registers aimed at embedded systems, and the RV64I differs only on the integer and Program Counter (PC) registers width [10]. The optional extensions can be added as needed by each application to form a more robust instruction set [12].

2.2.1. Vector extension

The RISC-V vector extension allows adapting the RISC-V ISA to a vector architecture. It adds 32 vector registers to the base RISC-V and enables its splitting for executing more operations simultaneously, which is configured through the seven additional CSRs [12].

The Configuration-Setting instructions change the values of CSRs. The vector arithmetic instructions operate on values stored in the vector registers, and memory read and write operations transfers a certain number of bits from memory to the vector database and back again. All instructions in this set fit into two existing instruction formats: LOAD-FP/STORE-FP from the floating-point extension, or OP-V, a vector-exclusive format [14].

3. Architecture

HARV [2] is a low-cost fault-tolerant RISC-V processor developed through cooperation between the Laboratory of Embedded and Distributed Systems from the University of Vale do Itajaí (UNIVALI) and the Laboratory of Informatics, Robotics, and Microelectronics of Montpellier (LIRMM) from the University of Montpellier. The authors focused on using the least amount of resources possible. Thus, the processor uses a single-cycle micro-architecture, reducing the required registers.

This processor has five primary units: (i) instruction fetch; (ii) instruction decode; (iii) execution; (iv) memory access; and (v) write-back. We modified the decode, execute and register HARV's units to support the vector instructions subset, highlighted at Figure 2.

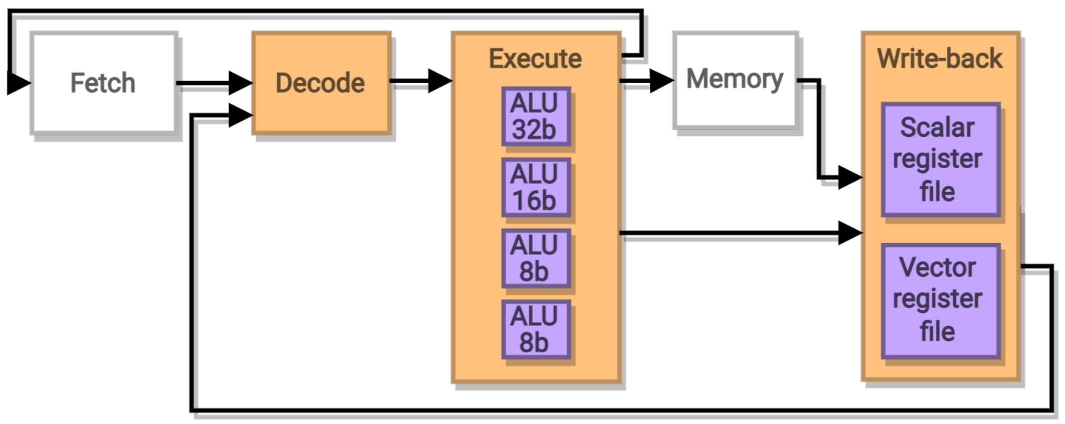

Figure 2: HARV block diagram with the modifications to support vector instructions.

This work aimed at implementing only the main vector instructions for digital signal processing to optimize these applications. Therefore, among the several possible instructions to be implemented, the subset was limited to instructions of sequential memory read and write (word, half-word, and byte) and integer arithmetic operations, such as addition and subtraction, for 32-bit architecture.

3.1. Instruction Fetch

The instruction fetch unit consists of a PC register, a 32-bit adder, and the required logic circuits for jumps and conditional branches. The adder increments the value stored in the PC register in case of sequential execution. Otherwise, in the case of executing a conditional branch, it adds the address offset to the PC register.

3.2. Instruction Decode

In HARV, the control unit integrates the instruction decoder, responsible for identifying the operation that will be performed. The 6 to 0 instruction bits are used as input for a combinational logic that asserts the control signals for the data path. Thus, the control unit identifies the format of the instructions according to the RISC-V ISA specification [10]. To support the vector instructions subset, we added the three instructions format defined in version 1.0 of the vector extension specification: Load-FP, Store-FP, and OP-V [14]. Subsequently, according to the format of the instructions, the output of the control unit related to the operation performed are enabled (1) or not (0).

In order to simplify the implementation, Santos et al. [2] developed the main and ALU control units as a single component. Thus, the control unit also has an output informing the ALU which operation to perform. For scalar instructions, the field *funct3* from the instruction (bits 14 to 12) defines the operation. Whereas for vector arithmetic instructions, the ALU operation is defined by the instruction field *funct6* (bits 31 to 26).

As in HARV, the processor performs the source registers reading in the decoding stage. This is done based on the addresses informed in the fields *rs1_i* and *rs2_i* of the instruction, in case of scalar registers, or *vs1_i* and *vs2_i*, in case of vector registers. For writing, the addresses fields are *rd_i*, for scalar registers and *vd_i* for vector registers.

The scalar register file was kept identical to the base processor. Thus, it can perform two registers read, and one register write simultaneously. Although, in order to write in the register, the input *write_en_i* must be enabled. The register writing is clock-synchronous, while the reading is independent of the clock, meaning that the data written in the register on the previous cycle can be read in the current cycle. The vector register file works similarly to the scalar register file, and the difference between both is that the vector register file is exclusive to vectors.

3.3. Execution

The execution unit performs operations with two 32-bit data vectors. Therefore, this unit has two data inputs, as shown in Figure 3, *data1_i* e *data2_i*. In addition, the *ALUOp_i* input, derived from the control, signals which operation to perform according to the instruction.

To develop this unit, we base on the implementation of [8]. For this reason, we used four ALUs. One of these ALUs is identical to the 32-bit ALU implemented in HARV. The arithmetic and logical operations supported by this ALU are: add, shift (left/right logical and right arithmetic), set on less than, AND, OR, and XOR. The other three are one 16-bit and two 8-bit ALUs. However, we simplified these ALUs to support only the operations used in the vector instructions subset of this work since these are useful only for vector operations.

With the width of the input vectors set to 32 bits, in just one cycle (as long as there is no pipeline and it is single-cycle), it can run either one 32-bit, two 16-bit, or four 8-bit operations. Therefore, the level of parallelization of operations depends on the vector size of elements. The inputs and outputs are mapped to their respective ALUs according to the width setting of the vector elements. The elements that are being operated on wider ALUs are zero-extended. The 32-bit ALU is selected as the only output for scalar instructions to decrease the hardware needed for vector extension.

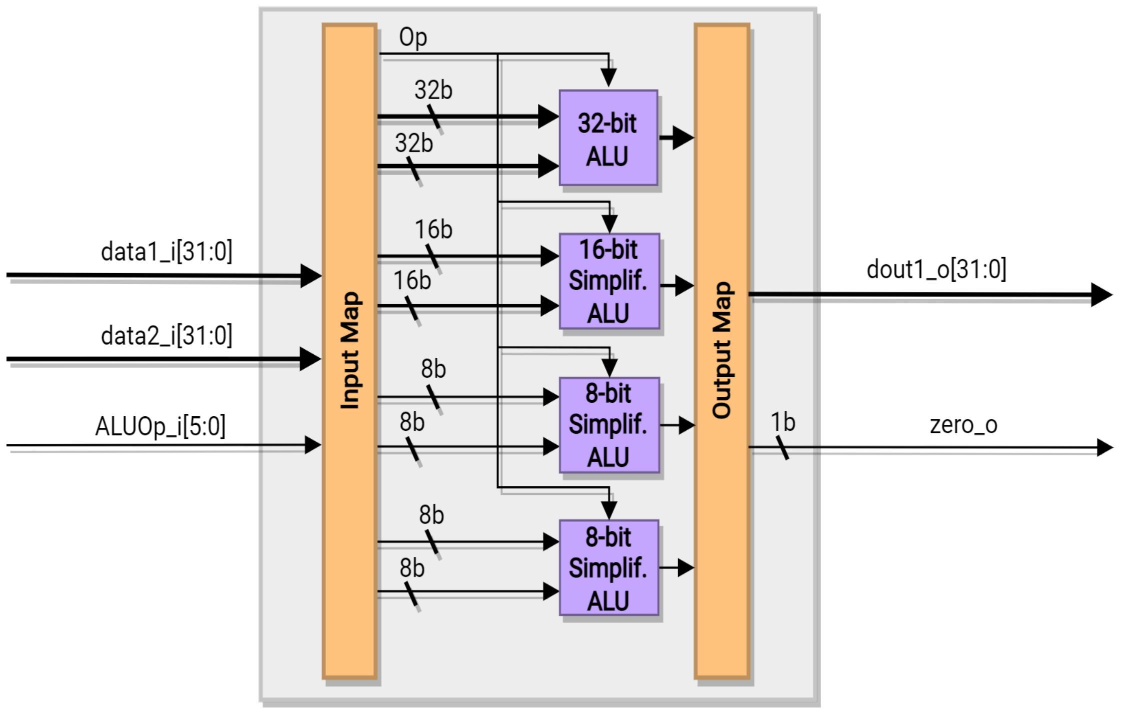

Figure 3: Execution Unit.

3.4. Memory Access

The memory access unit performs read and write operations to/from the data memory. The RISC-V specification describes that accesses with 8-, 16-, and 32-bit word widths can be performed, and all readings result in 32-bit width data. According to the instruction executed, the data signal can be extended or not. In RISC-V vector specification, there are three types of vector memory access: sequential, stride and indexed. However, we used only the sequential method in this work, which is identical to the scalar memory access implemented in HARV.

3.5. Write-back

The register write is only executed when processing instructions that write into the register file. This step is responsible for selecting the data to be written in the vector or scalar register file. The Figure 4 presents an overview of how the register writing is executed. This representation describes this operation in a simplified way. Therefore, we have abstracted the multiplexers used for other operations and the signals referring to the control outputs, immediate selection, and ALU input signals.

The inputs of the register file are connected to multiplexers, commanded by the control, and have as input the memory read and the execution results. In the case of the scalar register files, there are two additional multiplexers, also driven by the control, which inputs are the PC value and the immediate field of the instruction. The value is stored in the register at address *rd_i*, for the scalar register file, or *vd_i*, for the vector ones. These addresses are specified in the instruction, and writing to each register file is only performed if the respective write control signal is active.

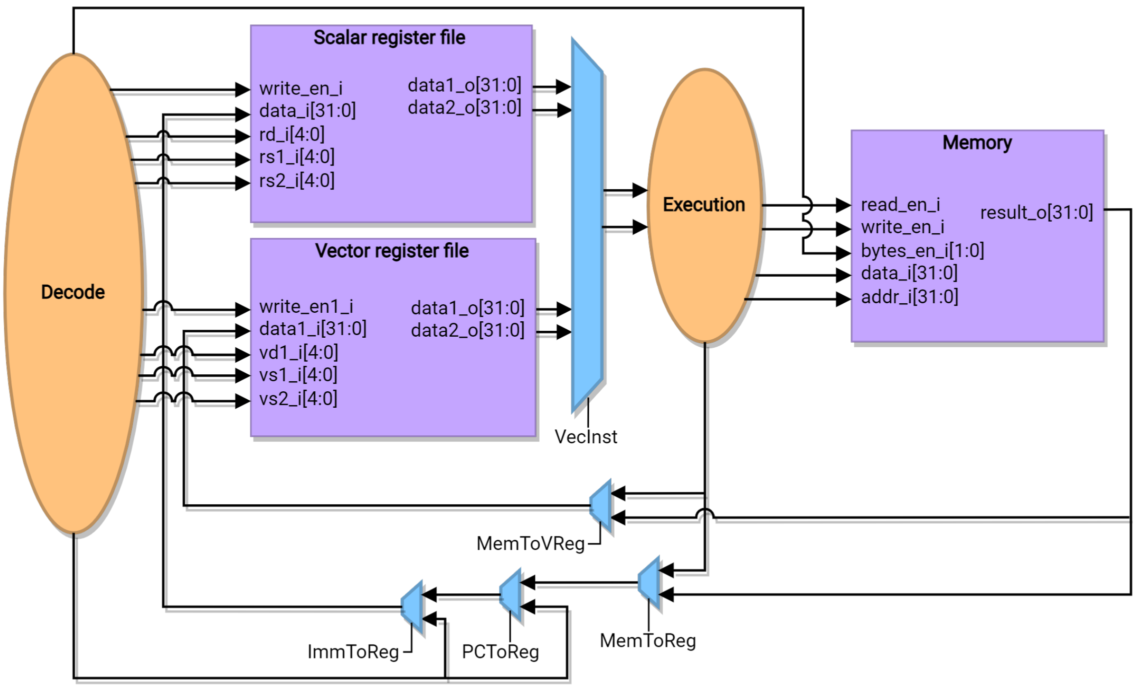

Figure 4: Write-back unit.

4. Results

4.1. Methodology

Since HARV was implemented using VHSIC Hardware Description Language (VHDL), we developed the vector instruction support using VHDL as well. We used the Xilinx Vivado 2020.2 Design Suite tool to collect the synthesis data and the Zynq ZC7020 SoC device. The metrics analyzed include the number of Look-up Tables (LUTs) and Flip-Flops (FFs), maximum operating frequency, and the estimated power dissipation.

The benchmark algorithm is a vector addition that reads two vectors from the data memory, adds the elements, and writes the result to the memory. The vectors used are 32-bits wide, but the element's width was varied among all the possibilities. This algorithm was executed in two scenarios: using only scalar instructions and using the implemented vector instructions. The programs were written in the RISC-V assembly language and optimized for vector architecture.

Therefore, it was possible to compare the read, write, and operating latency of vector elements in both scenarios with 50MHz of clock. We explored spatial memory locality in the scenario with vector instructions by storing the whole vector as a word in the memory. By this, it was possible to increase even more the performance gain.

4.2. Benchmark Performance

Table 1 presents a comparison between the number of cycles needed to execute memory access and arithmetic instructions for scalar and vector architectures. Reading and writing 16- and 8-bit memory elements with scalar instructions take two and four separate memory accesses, respectively. With vector instructions, on the other hand, it is possible to read all the elements in only one memory access, since the number of instructions required to perform the same operations is reduced. Thus, to execute memory loads and stores, the vector instructions required four and two times fewer cycles when operating on 8- and 16-bit elements, respectively.

The same happens for the arithmetic instructions since, with scalar instructions, it is necessary to operate on the elements individually. Whereas with vector instructions, the elements are operated on simultaneously. This way, when processing 8-bit data elements, as is the case of image pixels, using vector instructions decreases the number of cycles by up to four times. For 16-bit elements, the required number of cycles is two times smaller.

Table 1: Number of cycles to execute scalar and vector instructions for each possibility of vector element width.

	Operating latency (number of cycles)					
Vector elements	**Four 8-bit**		**Two 16-bit**		**One 32-bit**	
Instruction type	Scalar	Vector	Scalar	Vector	Scalar	Vector
Load / Store	32	8	16	8	8	8
Arith	20	5	10	5	5	5

4.3. Synthesis Result

The synthesis results for the original HARV processor and HARV with the vector extension are shown in Table 2. The maximum operating frequency increased 4.6%, and the power dissipation increased 6.1%.

The implementation of the extension increased the logical resources used by the processor. With the vector extension, the number of LUTs is 1.5 times higher than the original HARV, mainly due to adding the vector register file and the three ALUs. At the same time, the number of FFs is 1.8 times higher because of the 32 vector registers. However, the overhead of hardware and power are acceptable compared to the gained acceleration.

Table 2: Synthesis results.

Configuration	**LUTs**	**FFs**	**F$_{max}$(MHz)**	**P(mW)**
Scalar HARV	1,988	1,575	52.15	146
Vector HARV	3,158	2,826	54.72	155

5. Conclusion

This work provided support of a vector instructions subset from the RISC-V vector extension for HARV [2], which is a fault-tolerant processor. The main goal of this implementation was to accelerate signal processing applications by exploring data-level parallelism.

We evaluated the impact of using the vector instructions compared to the scalar ones in order to measure the processing acceleration. The results showed that, with the vector extension, it is possible to read and write elements from the data memory and process them four times faster.

Therefore, in future works, we intend to increase the supported vector instruction set by adding multiplication instructions. With this, it will be possible to expand the benchmark algorithms algorithm and evaluate convolution operations running with vector instructions. Also, we aim to support 64-bit width vectors to further accelerate applications.

Acknowledgments

The authors would like to thank the funding from the Foundation for Support of Research and Innovation, Santa Catarina (FAPESC-2021TR001907), Conselho Nacional de Desenvolvimento Científico e Tecnologico (CNPq) Process 140368/2021-3, and from the Region d'Occitanie and the École Doctorale I2S from the University of Montpellier (contract no. 20007368/ALDOCT-000932). The authors also thank for the support of the University Space Center of Montpellier, France.

References

[1] M. Yang, G. Hua, Y. Feng, J. Gong, Fault-tolerance techniques for spacecraft control computers, John Wiley & Sons, Singapore, 2017.

[2] D. A. Santos, L. M. Luza, C. A. Zeferino, L. Dilillo, D. R. Melo, A low-cost fault-tolerant RISC-V processor for space systems, in: 2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS), Marrakesh, pp. 1–5.

[3] S. D. Mascio, A. Menicucci, E. Gill, G. Furano, C. Monteleone, On-board decision making in space with deep neural networks and RISC-V vector processors, Journal of Aerospace Information Systems 18 (2021) 553–570.

[4] S. D. Mascio, A. Menicucci, E. Gill, G. Furano, C. Monteleone, Leveraging the openness and modularity of RISC-V in space, Journal of Aerospace Information Systems 16 (2019) 454–472.

[5] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, ISSN, Elsevier Science, Amsterdam, 2011.

[6] D. Selva, D. Krejci, A survey and assessment of the capabilities of cubesats for earth observation, Acta Astronautica 74 (2012) 50–68.

[7] I. A. Assir, M. E. Iskandarani, H. R. A. Sandid, M. A. Saghir, Arrow: A RISC-V vector accelerator for machine learning inference, arXiv preprint arXiv:2107.07169 (2021).

[8] M. Johns, T. J. Kazmierski, A minimal RISC-V vector processor for embedded systems, in: 2020 Forum for Specification and Design Languages (FDL), IEEE, Kiel, pp. 1–4.

[9] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, L. Benini, Ara: A 1-ghz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm fd-soi, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 28 (2019) 530–543.

[10] D. Patterson, A. Waterman, The RISC-V Reader: an open architecture Atlas, Strawberry Canyon, San Francisco, 2017.

[11] J. Gebis, D. Patterson, Embracing and extending 20th-century instruction set architectures, Computer 40 (2007) 68–75.

[12] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, The RISC-V instruction set manual, Volume I: User-Level ISA', version 2 (2014).

[13] D. Kanter, RISC-V offers simple, modular ISA, Microprocessor Report (2016).

[14] RISC-V Organization, RISC-V "V" Vector Extension, `https://github.com/riscv/riscv-v-spec`, 2022. [Online; accessed 22 May 2022].