Implementation and Reliability Evaluation of a RISC-V Vector Extension Unit

Carolina Imianosky, Douglas Santos, Douglas Melo, Felipe Viel, Luigi Dilillo

Title: Implementation and Reliability Evaluation of a RISC-V Vector Extension Unit

Author(s): Carolina Imianosky, Douglas A. Santos, Douglas R. Melo, Felipe Viel, Luigi Dilillo

Document version: Post-print version (Final draft)

Please cite the original version:
C. Imianosky *et al*., " Implementation and Reliability Evaluation of a RISC-V Vector Extension Unit," 2023 36th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2023.

Implementation and Reliability Evaluation of a RISC-V Vector Extension Unit

Carolina Imianosky*, Douglas A. Santos*, Douglas R. Melo†, Felipe Viel†‡, Luigi Dilillo*
*IES, University of Montpellier, CNRS, Montpellier, France
†LEDS, University of Vale do Itajaí, Itajaí, Brazil
‡SpaceLab, Federal University of Santa Catarina, Florianópolis, Brazil
{carolina.imianosky, douglas.santos}@umontpellier.fr, {drm, viel}@univali.br, luigi.dilillo@umontpellier.fr

Abstract

The space environment challenges the reliable functioning of electronic systems due to radiation exposure and extreme temperatures. To mitigate potential failures, fault tolerance techniques are crucial in these systems. With the implementation of fault tolerance techniques, the performance of the system is affected, often impairing data processing. One way to accelerate data processing is to use vector instructions, which allow operating on a data vector with only one instruction. Therefore, we developed a Vector Extension Unit that supports a subset of the RISC-V Vector Extension for HARV, a RISC-V-based processor. We evaluated the impact of using the Vector Extension instructions over the scalar instructions. We assessed the reliability of the core by simulation based on fault injection campaigns with both baseline and hardened HARV. The vector unit offers substantial performance acceleration of up to 8.1 times when compared to scalar operations, and, despite the hardware overhead, it enhanced the reliability of the executions from 93.9% to 97.9% for the baseline HARV and from 97.9% to 99.3% for the hardened HARV.

Index Terms

RISC-V, System-on-Chip, Vector Instructions, Fault Tolerance, Space Systems

I. Introduction

With radiation exposure and extreme temperatures, the space environment affects the functioning of systems during a mission. The potential occurrence of critical failures requires spacecraft to be designed using fault tolerance techniques to improve their reliability [1].

The RISC-V is an Instruction Set Architecture (ISA) that has gained popularity due to its simplicity, openness, and modularity, i.e., it consists of a basic ISA and optional extensions [2]. Among the ISA extensions defined in the standard, the RISC-V Vector Extension (RVVE) [3] enables systems to execute a single instruction on multiple data elements simultaneously. This extension provides higher data parallelism levels, improving critical factors for space systems, such as processing latency and power consumption. Thus, it is possible to speed up onboard applications such as AI, which are effective in remote sensing, as for image classification and target detection [4].

The HARV (HArdened RISC-V) [5] is a low-cost fault-tolerant RISC-V processor, hardened against SEUs (Single-event Upsets) and SETs (Single-event Transients) using Triple Modular Redundancy (TMR) and Error-Correcting Code (ECC). Furthermore, this processor has been characterized in different irradiation facilities [6], proving suitable for reliable applications.

The use of AI in nano- and pico-satellite is an alternative for compensating for its lack of resources (power, volume, computing) due to the ability to process the data onboard instead of sending them to the ground station [4]. Meanwhile, the faults caused by the space environment can lead to prediction errors. Besides that, combining AI with fault-tolerance techniques increases hardware costs and reduces processor performance [7]. The vector extension can speed up AI processing but is intrinsically prone to radiation-induced failures since it essentially comprises sequential logic elements such as registers. Thus, evaluating the reliability of AI applications that use vector extension is crucial.

In this context, this work presents an extension of the HARV processor to include a Vector Extension Unit (VEU) that supports a subset of the RVVE, reducing the processing latency of the processor, which is affected by the implemented fault tolerance techniques. To validate our implementation, we execute and compare benchmarks using scalar and vector instructions in HARV with baseline and hardened configurations.

The main contribution of this work is the description of the proposed vector unit and its reliability characterization in the HARV-SoC (HARV System-on-Chip). This characterization enables the improvement's assessment of the core

The results presented in this paper have been obtained in the framework of the EU project RADNEXT, receiving funding from the European Union's Horizon 2020 research and innovation programme, Grant Agreement no. 101008126, the Region d'Occitanie and the École Doctorale I2S from the University of Montpellier (contract nos. 20007368/ALDOCT-000932 and 22009671), the Foundation for Support of Research and Innovation, Santa Catarina (FAPESC-2021TR001907), and the Brazilian National Council for Scientific and Technological Development (CNPq - processes 138179/2021-2, 140368/2021-3 and 350794/2023-5).

regarding its performance and fault tolerance. The reliability assessment of the RVVE in the processor was performed by fault injection simulations. This assessment provides an insightful analysis of the impact of using vector extension for reliable applications.

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

II. Related Work

Some commercial and academic cores currently implement or rely on the RVVE. For example, [8] describe the implementation of an RVVE on the HPP64 NOEL-V platform for space applications. This platform aims to improve the performance and capabilities of space processors while considering the specific constraints and requirements of satellite data systems. Their study compares the performance and resource utilization of the vector processor with different levels of data-level parallelism (DLP) to the baseline scalar processor.

Another example is [9], which integrates two non-binary low-density parity-check decoding algorithms for deep-space communication standards on RISC-V SoCs using vector extensions. The authors discuss the migration of critical computing tasks in communication systems from dedicated devices to general-purpose processors, such as GPUs and Field-Programmable Gate Arrays (FPGAs). The work aimed to increase flexibility, reduce costs, and improve efficiency through constant upgrades in hardware and software designs.

The authors in [10] implemented a vector processing unit based on the RISC-V architecture but focused on incorporating SIMD capabilities. Their goal was to enhance the performance of embedded processing tasks. This unit was integrated into the CV32E40P [11] RISC-V-based processor and was evaluated based on resource utilization, power consumption, and performance for different design configurations. The authors compared the metrics when using only the CV32E40P processor and when using it with the unit.

Both [8] and [9] present RVVE implementations designed for space applications and use hardened processor cores. However, none of them presented evaluations of the reliability of the core with the RVVE.

Given this context, we implemented the VEU for the HARV core to assess the performance gain and the hardware overhead of the implementation. Also, we performed fault injection simulations for reliability analysis.

III. RISC-V Vector Extension

The RISC-V ISA is a Reduced Instruction Set Computer (RISC) architecture designed to be modular. It consists of a basic instruction set and various optional extensions that can be added to form a more robust instruction set [12]. The RVVE is an optional extension that supports vector architectures, which explore Data-Level Parallelism (DLP) concepts. With this, it is possible to operate simultaneously on multiple vector data elements, speeding up computational processing [13].

Vector architectures originated from SIMD processing, which emerged in the 1970s for partitioning large registers (e.g., 128-bit registers) into multiple elements and operating them in parallel. However, in SIMD instructions, the operation code (Opcode) defines the operation and the data width, meaning that increasing the width of the vectors also implies an increase in the instruction set [14]. Implementation of vector architectures, on the other hand, defines the size of the vector rather than the Opcode. Thus, it reduces the size of the instruction set and makes the hardware design more flexible for data parallelization without affecting algorithm development.

The RVVE adds 32 vector registers to the base RISC-V and enables partitioning them to perform multiple operations simultaneously. The size of the elements is configured via seven additional registers called Control and Status Registers (CSRs) [12], and the Configuration-Setting instructions change the values of the CSRs. Vector arithmetic instructions operate on values stored in vector registers, and memory read and write operations transfer data elements between memory and registers. All instructions in this set fit into two existing instruction formats: LOAD-FP/STORE-FP from the floating-point extension and OP-V, an exclusive RVV format [3].

IV. HARV Vector Extension Unit

In this work, we developed a VEU (Fig. 1) for the HARV processor. We implemented the VEU so that the maximum size of the vector can be configurable by the implementation. The VEU comprises the interface units, the Vector Register File (VRF), a vector elements counter, and the vector execution unit. To enable the HARV's support to the VEU, we modified two components of the core: the instruction decode and the execution.

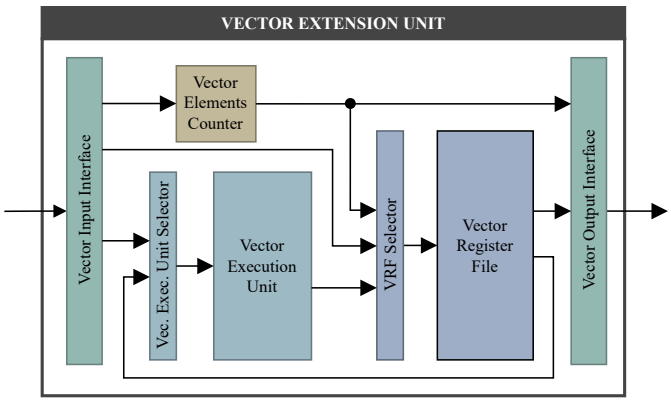

Fig. 1. Vector Extension Unit

The HARV control unit integrates the instruction decoder. The instruction bits 6 to 0 are the input to a combinational logic component that determines the instruction format, according to the RISC-V ISA specification [14]. Afterward, the outputs of the control component are either enabled (1) or disabled (0) according to the instruction's operation. To support the RVVE subset, we added the three instruction formats defined in the RVVE v1.0 specification: Load-FP, Store-FP, and OP-V [3].

In the HARV microarchitecture [5], the main and ALU control units were merged as a single component to simplify the implementation. So we extended it also to control the Vector Execution Unit. Also, the HARV processor reads the source register during the decoding stage and does the same for the VRF.

The HARV's execution unit performs the arithmetic and logic operations. These operations include: add, subtraction, logical shift (left or right), arithmetic shift (right only), set to 1 (on less than), AND, OR, and XOR. To support the vector instructions, this unit was extended to operate on 64-bit operands and is used to complement the Vector Execution Unit, eventually operating in some vector elements.

The Interface Units are responsible for directing the inputs and outputs of the VEU. The inputs are signals from the controller, the addresses to read and write from the VRF, the result from the external ALU, and the values from the vector CSRs. The outputs are one signal indicating if all the necessary elements were processed; and two arrays of data that serve either for memory writing or as input for the external ALU if required.

The VRF contains 32 registers similar to the scalar register file but exclusive to the VEU. Thus, it can perform two readings and one writing simultaneously. The data outputs are defined to have the size of maximum vector size specified by the implementation in a way that all the elements of the vector are read at once. The data input is defined as 64-bit since it is the maximum element size, so for writing, a 64-bit element is handled as two 32-bit elements. The element counter holds the number of vector elements that were already processed. Thus, it defines the following array of data within the whole data vector to be processed.

The vector execution unit comprises four 8-bit simplified ALUs operating on the vector elements and exclusive to arithmetic vector instructions. The simplified ALUs have carry-in and carry-out signals, which are multiplexed to enable simultaneous operations for two 16-bit or one 32-bit element. A diagram of this unit is shown in Fig. 2. In the case of 32- and 64-bit vector elements, the external HARV ALU is also used to operate on one element. With this architecture, it is possible to operate on one 64-bit, two 32-bit, two 16-bit, or four 8-bit elements simultaneously.

The primary goal of this work was to support simple vector instructions to optimize application processing, such as Neural Networks, and assess the behavior of the HARV with the Vector unit in fault injection simulations. Therefore, the subset of instructions was limited to sequential instructions for memory access (word, half-word, and byte) and integer arithmetic operations such as addition and subtraction.

V. Materials and Methods

We implemented the vector instruction support using VHDL (VHSIC Hardware Description Language) since HARV-SoC was also implemented using this HDL. In this work, we propose an improvement of the architecture introduced in [15] that enables 64-bit width elements and has a different VEU organization. Furthermore, this enhanced architecture is used to assess HARV-SoC's reliability with the VEU through fault injection simulations.

We used the Xilinx Vivado 2020.2 Design Suite tool and the Zynq ZC7020 FPGA SoC device to collect the synthesis data. Thus, we analyzed the number of Look-up Tables (LUTs) and Flip-Flops (FFs), the maximum operating frequency (F_{max}), and the dynamic power dissipation (P_{dyn}) for the HARV-SoC.

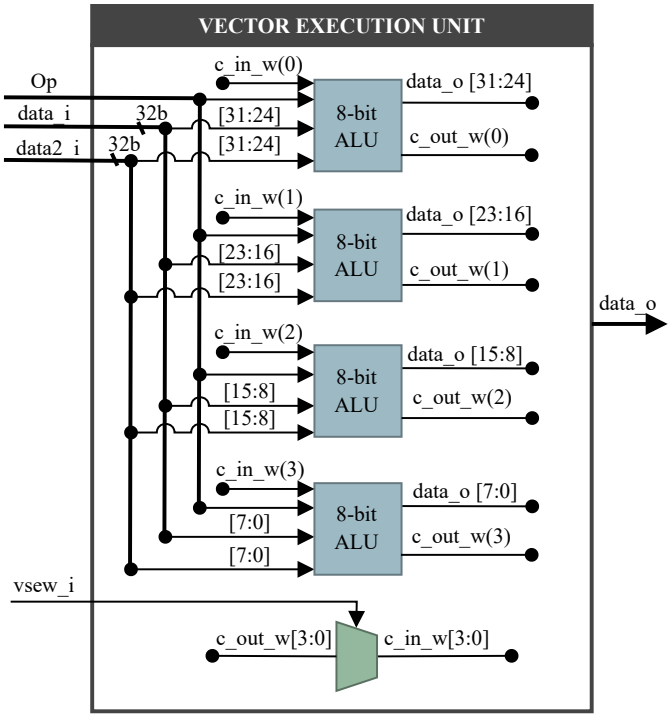

Fig. 2. Vector Execution Unit

The implemented applications for all algorithms were compiled using GNU Compiler Collection (GCC) for RISC-V [16], and its vector operations were implemented in Assembly vector instructions. The following subsections detail the setups used for the performance and reliability evaluation.

A. Performance evaluation

To estimate the performance gain, we implemented an Assembly algorithm that reads two vectors of the memory, adds each of their elements, and stores the result in the memory. The maximum vector length was set to 256 bits. With this algorithm, we varied the vector length from 32- to 256-bit and the data width of the vector elements from 8- to 64-bit. The algorithm was executed using the baseline HARV and the HARV with VEU.

We measured the processing latency in both executions by reading the cycle counter register (*mcycle*) at the start and the end of the algorithm execution. The difference between these two values is the number of cycles the processor spends to execute the program. The acceleration metric was obtained by dividing the number of cycles for execution in the baseline HARV by the number of processor cycles used for the executions with the VEU.

B. Reliability evaluation

We evaluated reliability through fault injection simulations using TCL (Tool Command Language) and Python scripts with the ModelSim SE-64 2019.01 software. At first, a golden run is executed, which runs the application without fault injections. Then, we extract baseline indicators to run the simulations with fault injection, such as system FFs, execution time, and UART (Universal Asynchronous Receiver/Transmitter) output. After the golden run, we run the fault injection simulations.

First, all register signals are fetched, and the simulation time is set to zero. Then, for each simulation, it selects random locations and simulation times to inject bit flips, based on neutron-characterized flip-flop FIT_{NYC} (Failure In Time for a billion hours in the New York City's neutron flux at sea level) of 248 and a neutron flux of $2.5 \times 10^{10} n/cm^2/s$. We calculate the neutron fluence since the last incremented time for every randomly generated simulation time and use the cross-section to calculate the FF error rate from a given fluence. We iterate through them to decide if we should inject an error on a given FF and randomly choose if the injection should occur. All the FFs of the SoC and the processor are susceptible to bit-flip.

Regarding the benchmark algorithm for the fault injection simulations, we implemented an algorithm in C language and the Assembly for the vector instructions. This algorithm adds two vectors with 8192 16-bit elements each. We executed the algorithms with the baseline processor (RV32I) and with the vector extension (RV32IV). Without the VEU, each vector element is read and operated sequentially in a loop. For the execution with the VEU, the vectors

are subdivided into 256-bit vectors of 16-bit elements and operated using OP-V instructions. Thus, the VEU reads the two 256-bit vectors from the data memory, adds the elements, and writes the result to the memory.

We ran 1024 experiments for each of the four scenarios: baseline HARV-SoC, baseline HARV-SoC with the VEU, hardened HARV-SoC, and hardened HARV-SoC with the VEU. This way, we could assess the impact of the VEU in terms of fault tolerance for each SoC version.

We used the internal registers of the processor and the UART output to evaluate the reliability of the SoC. Thus, we classified the executions into four categories: (*i*) *correct*, when the application was executed correctly; (*ii*) *benchmark error*, when the application is executed but has incorrect results; (*iii*) *hang*, when the application does not end the execution nor produces any result; and (*iv*) *exception*, when an unscheduled event disrupts the program execution [17]. The exceptions available were: instruction address misaligned, instruction access fault, illegal instructions, load access fault, and store access fault.

The classification of the executions was performed using a Python script that analyses the files that contain the UART outputs of each execution. Thus, we could determine the number of executions for each category. This script also reports the average number of injected faults and the average execution time.

VI. Results

The following subsections present the results of the VEU implementation. First, we describe the synthesis results, such as the number of FFs and LUTs, maximum operating frequency, and dynamic power dissipation. After, we show the acceleration obtained in comparison to the baseline processor. Finally, we show the results of the fault injection simulations.

A. Synthesis

The plot in Fig. 3 reports the number of LUTs and FFs for each implementation. The plot shows the resource usage by the baseline HARV-SoC and the HARV-SoC with 32-, 64-, 128-, and 256-bit VEU.

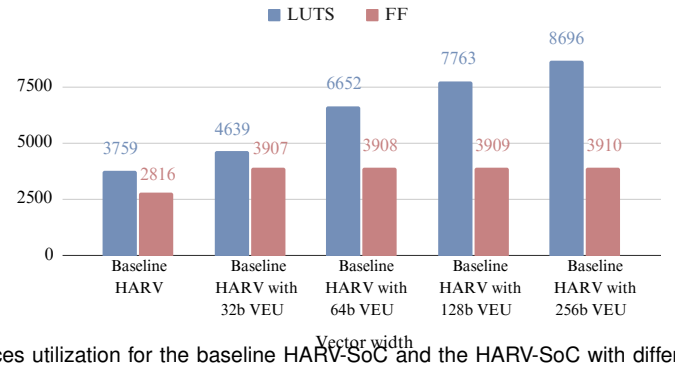

Fig. 3. Resources utilization for the baseline HARV-SoC and the HARV-SoC with different widths VEUs.

The number of the LUTs and FFs increases according to the vector length. The baseline processor needs 3759 LUTs, whereas the version with the 32-, 64-, 128-, and 256-bit VEU requires 1.23, 1.77, 2.07, and 2.31 times the amount of LUTs, respectively. For FFs, it requires 2816, and the versions with the 32-, 64-, 128-, and 256-bit VEU requires 1.39 times more LUTs than baseline HARV-SoC.

For the hardened versions of the implementations, the number of LUTs and FFs is represented in Fig. 4. In the hardened versions, the baseline processor needs 6172 LUTs, whereas the version with the 32-, 64-, 128-, and 256-bit VEU requires 1.16, 1.67, 1.85, and 2.01 times more LUTs, respectively. The number of necessary FFs is 3864 and increases 1.28 times for all the versions with different sizes of VEU.

Table I provides an overview of the values for the (F_{max}) and (P_{dyn}). The HARV-SoC achieves a F_{max} of 57.9 MHz and consumes 129 mW of dynamic power for the baseline scenario. The addition of the 32-bit VEU exhibits a slightly lower F_{max} of 56.8 MHz while consuming 131 mW of power. The 64-bit VEU reduces the F_{max} to 49.2 MHz and increases the power consumption to 146 mW. Similarly, the 128-bit VEU shows a decrease in F_{max} to 47.8 MHz while maintaining the same power consumption of 146 mW. Lastly, the 256-bit VEU causes a slight improvement in F_{max} to 50.3 MHz with a power consumption of 140 mW.

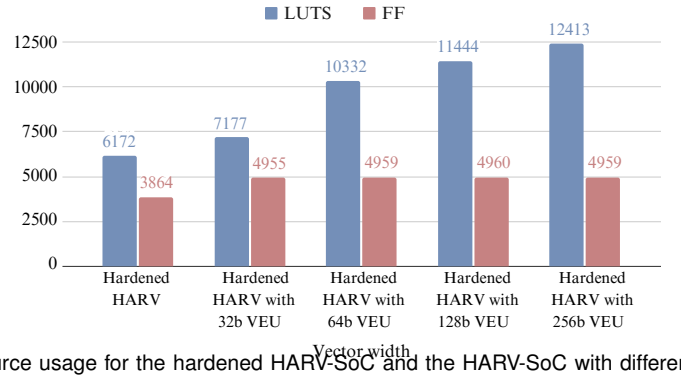

Fig. 4. Resource usage for the hardened HARV-SoC and the HARV-SoC with different VEU widths.

Considering the hardened scenarios, the HARV-SoC achieves a F_{max} of 38.8 MHz and dissipates 158 mW. With 32-bit VEU, the F_{max} decreases to 34.1 MHz while dissipating 188 mW. The 64-bit VEU reduces the F_{max} to 34.3 MHz and increases the P_{dyn} to 188 mW. Similarly, the 128-bit VEU decreases the F_{max} to 33.7 MHz with a P_{dyn} of 183 mW. Lastly, the 256-bit VEU has a F_{max} of 34.8 MHz with a P_{dyn} of 187 mW.

TABLE I
MAXIMUM OPERATING FREQUENCY AND DYNAMIC POWER DISSIPATION RESULTS.

HARV-SoC	VEU Width	F_{max} (MHz)	P_{dyn} (mW)
Baseline	-	57.9	129
	32 bits	56.8	131
	64 bits	49.2	146
	128 bits	47.8	146
	256 bits	50.3	140
Hardened	-	35.8	158
	32 bits	34.1	165
	64 bits	34.3	188
	128 bits	33.7	183
	256 bits	34.8	187

B. Performance

The plot in Fig. 5 presents the acceleration values achieved through the execution of vector operations compared to scalar operations. It shows the performance improvements obtained through different vector configurations, varying between 32-, 64-, 128-, and 256-bit vectors, along with 8-, 16-, 32-, and 64-bit elements.

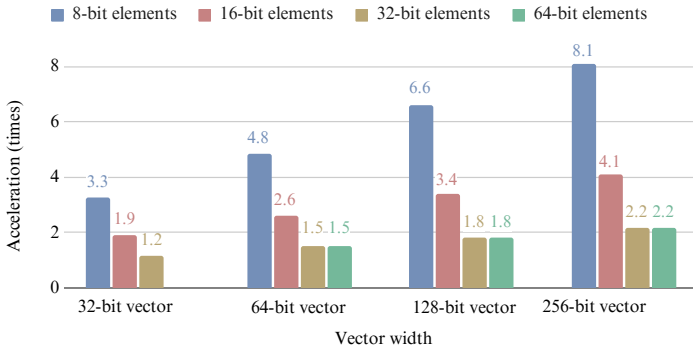

Fig. 5. Execution acceleration for different VEU widths.

For the scalar execution, the 64-bit vectors are treated as two 32-bit, due to the core architecture (RV32I) and its register width, making the execution time in this case similar to the two 32-bit elements. With the VEU, it is possible to operate in just one 64-bit element per instruction, and for 32-bit elements, it is possible to operate in two elements simultaneously. Thus, due to the architecture of the implementation, the acceleration values from scalar to vector executions for 32-bit and 64-bit elements are similar.

The acceleration for all the vector lengths is more significant for 8-bit elements and decreases as the element width increases. For 32-bit vectors, an element width of 8 bits results in a speedup of 3.3 times and increases 1.5 times for 64-bit vectors. The 128-bit vector shows an increase of 1.8 times compared to the 64-bit. For the 256-bit, it has an acceleration increase of 1.5 times compared to the 128-bit vector.

The 16-bit elements show an increase of approximately 1.7 times as the size of the vector increases. For 32- and 64-bit elements, the acceleration is similar in all cases. It is worth noting that 32-bit vectors have no executions with 64-bit elements. The acceleration increase is 0.3 times from the 64- to the 32-bit vectors and the 128- to 64-bit. From the 256- to the 128-bit, the difference in acceleration is 0.4 times.

C. Fault Injection Simulation

We evaluated the output of the UART of each execution using the Python script and the four categories of results mentioned in Section V-B. The experimental results are presented in Table II, which compares the results of the baseline scalar and baseline vector approaches and the hardened scalar and hardened vector approaches. Thus, it presents the number of correct executions, benchmark errors, hangs, and exception executions achieved by each approach for the 1024 executions.

In the baseline processor, the vector approach showed an increase of 40 correct executions. Within the hardened category, the vector approach improved with 15 more correct executions. The scalar approach had one less execution with benchmark errors than the vector in the baseline processor. Meanwhile, the hardened scenario presented one more benchmark error than the vector. For the hang executions, the vector approach showed an improvement of 25 executions in the baseline scenario. In the hardened design, the vector approach has a difference of 14 executions, where it did not show any case of hang executions. Regarding the exceptions, the baseline processor with the vector approach had 16 instances less than the scalar. The hardened scalar scenario had two more exceptions than the vector scenario.

TABLE II
FAULT INJECTION EXECUTIONS

Execution classification	**Baseline**		**Hardened**	
	Scalar	**Vector**	**Scalar**	**Vector**
Correct	962	1002	1002	1017
Benchmark error	6	7	3	4
Hang	28	3	14	0
Exception	28	12	5	3

Table III compares the average number of fault injections and execution times of the baseline and hardened scenarios for both scalar and vector operations to assess the impact of the execution time on the number of injected faults. In both baseline and hardened systems, the average execution time of the vector approach was approximately half of the scalar one. The difference in the execution time directly impacts the number of injections since the longer it takes to execute, the more faults are likely to be injected. In the baseline scenarios, the scalar execution had an average of 2.1 fault injections, while the vector had just one. In the hardened scenarios, the scalar operation had an average of 4.44 injected faults, and the vector operation had 2.05.

TABLE III
INJECTIONS ANALYSIS

Metric	**Baseline**		**Hardened**	
	Scalar	**Vector**	**Scalar**	**Vector**
Average execution time (ms)	99	48	147	70
Average number of injections	2.1	1.0	4.44	2.05

D. Discussion

On the synthesis side, the comparison between the baseline processor and the processor with the VEU reveals that the VEU implementation incurs additional LUTs and FFs compared to the baseline processor, with the number increasing as the VEU width expands. Specifically, the wider VEUs require more resources due to the multiplexing logic to access the VRF. Adding a VEU slightly reduces the F_{max} and increases power consumption. The impact on F_{max} and power consumption varies with VEU sizes. In conclusion, incorporating a VEU introduces hardware overhead

regarding LUTs and FFs. However, it is crucial to consider the trade-off between the hardware overhead and the acceleration improvement.

Regarding performance, the VEU has significantly improved processing latency compared to scalar operations. We observed substantial speedups in all tested scenarios by analyzing different vector lengths (32-, 64-, 128-, and 256-bit) and element widths (8-, 16-, 32-, and 64-bit). As the vector length increased, we noticed that the accelerations improved. Additionally, we observed a correlation between the element width and the achieved acceleration, with narrower element widths, such as 8 bits, exhibiting higher accelerations than wider ones. This suggests that narrower element widths benefit more from vectorization, allowing them to exploit the increased parallelism vector operations offer.

Regarding reliability, the fault injection simulation results have shown that using the VEU significantly reduces error propagation. One of the primary benefits of the VEU is its ability to execute operations more quickly than a baseline processor. The results demonstrate that the vector approach enhances system responsiveness, contributing to better fault tolerance. By performing operations at a higher speed, the window of vulnerability to potential faults is minimized, reducing the average number of injected faults.

The VEU showed better results than the scalar approach in both baseline and hardened scenarios, exhibiting higher rates of correct executions and enhanced resilience against benchmark errors, hangs, and exceptions. This increase makes the VEU a promising solution for applications where system dependability is critical, such as AI applications for space systems.

VII. Conclusion

This work describes the implementation of a VEU that supports a subset of the RVVE for the HARV processor. The main goals were to accelerate data processing applications by exploring data-level parallelism and to assess its reliability.

We evaluated the impact of vector instructions over scalar processing to measure speedup. The results showed that the VEU proved to be a valuable addition to the processor, delivering significant performance acceleration of up to 8.1 times over scalar operations due to exploiting parallelism. Although it introduces hardware overhead, its ability to enhance reliability makes it an attractive choice for demanding applications that require high performance and system dependability.

In future work, we intend to extend the fault tolerance techniques to the VEU and expand it to support multiplication operations, allowing us to test AI applications and assess their performance and reliability improvement. We also intend to submit this architecture to a fault injection campaign in particle accelerator facilities, enabling comparing and validating the obtained results with actual irradiation tests.

References

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

[2] S. D. Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone, "Leveraging the openness and modularity of RISC-V in space," *Journal of Aerospace Information Systems*, vol. 16, no. 11, pp. 454–472, 2019.

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

[4] G. Furano, A. Tavoularis, and M. Rovatti, "Ai in space: applications examples and challenges," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2020, pp. 1–6.

[5] D. A. Santos, L. M. Luza, C. A. Zeferino, L. Dilillo, and 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, 2020, pp. 1–5.

[6] D. A. Santos, A. M. P. Mattos, D. R. Melo, and L. Dilillo, "Enhancing fault awareness and reliability of a fault-tolerant RISC-V system-on-chip," *Electronics*, vol. 12, no. 12, 2023.

[7] V. Duddu, D. V. Rao, and V. E. Balas, "Towards enhancing fault tolerance in neural networks," 2021.

[8] S. Di Mascio, A. Menicucci, E. Gill, and C. Monteleone, "Extending the noel-v platform with a risc-v vector processor for space applications," *Journal of Aerospace Information Systems*, pp. 1–10, 2023.

[9] Y.-M. Kuo, M. F. Flanagan, F. Garcia-Herrero, O. Ruano, and J. A. Maestro, "Integration of a real-time ccsds 410.0-b-32 error-correction decoder on fpga-based risc-v socs using risc-v vector extension," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 1–12, 2023.

[10] M. Ali, M. von Ameln, and D. Goehringer, "Vector processing unit: A risc-v based simd co-processor for embedded processing," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 30–34.

[11] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

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

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam: Elsevier Science, 2011.

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

[15] C. Imianosky, D. A. Santos, D. R. Melo, L. Dilillo, C. A. Zeferino, E. A. Bezerra, and F. Viel, "A Performance Evaluation of a Fault-tolerant RISC-V with Vector Instruction Support to Space Application," LASSS/LACW 2022 - Joint 3rd IAA Latin American Symposium on Small Satellites and 5th IAA Latin American CubeSat Workshop, Nov. 2022.

[16] RISC-V:Organization, "Gnu toolchain for risc-v, including gcc," https://github.com/riscv-collab/riscv-gnu-toolchain, 2023, (Accessed on 06/28/2023).

[17] J. L. Hennessy and D. A. Patterson, "Computer organization and design RISC-V edition: The hardware software interface," 2017.