Reliability analysis of a fault-tolerant RISC-V system-on-chip

Douglas Almeida dos Santos, Lucas Matana Luza, Luigi Dilillo, Cesar Albenes Zeferino, Douglas Rossi de Melo

▶ **To cite this version:**

Douglas Almeida dos Santos, Lucas Matana Luza, Luigi Dilillo, Cesar Albenes Zeferino, Douglas Rossi de Melo. Reliability analysis of a fault-tolerant RISC-V system-on-chip. Microelectronics Reliability, 2021, 125, pp.#114346. 10.1016/j.microrel.2021.114346 . lirmm-03358839

HAL Id: lirmm-03358839

https://hal-lirmm.ccsd.cnrs.fr/lirmm-03358839

Submitted on 29 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

This is a self-archived version of an original article.
This reprint may differ from the original in pagination and typographic detail.

Title: Reliability Analysis of a Fault-Tolerant RISC-V System-on-Chip

Author(s): Douglas Almeida Santos, Lucas Matana Luza, Luigi Dilillo, Cesar Albenes Zeferino, Douglas Rossi Melo

DOI: 10.1016/j.microrel.2021.114346

Published: October 2021

Document version: Post-print version (Final draft)

Please cite the original version:
Douglas Almeida Santos, Lucas Matana Luza, Luigi Dilillo, Cesar Albenes Zeferino, Douglas Rossi Melo, "Reliability analysis of a fault-tolerant RISC-V system-on-chip," Microelectronics Reliability, 2021, pp. 1-8, doi: 10.1016/j.microrel.2021.114346.

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorized user.

Reliability Analysis of a Fault-Tolerant RISC-V System-on-Chip

Douglas Almeida Santos*†, Lucas Matana Luza†, Luigi Dilillo†,
Cesar Albenes Zeferino* and Douglas Rossi Melo*
*University of Vale do Itajaí, LEDS, Brazil
†LIRMM, Univ Montpellier, CNRS, Montpellier, France
dalmeidados@lirmm.fr, lucas.matana-luza@lirmm.fr,
dilillo@lirmm.fr, zeferino@univali.br, drm@univali.br,

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent ac posuere magna. Phasellus arcu erat, faucibus mollis convallis et, rutrum et turpis. Curabitur gravida, arcu eget ullamcorper placerat, eros felis sagittis nisl, a feugiat sem ipsum non massa. Integer tristique risus at lectus gravida, in gravida neque tincidunt. Etiam mauris urna, commodo vel faucibus eget, facilisis ac justo. Sed ut augue sit amet nibh iaculis consequat. Integer sit amet ligula ultricies, porta libero cursus, feugiat ex. Vestibulum eu orci elit. Nullam in felis ut mauris tempus viverra ac eu dui. Vestibulum in ex lectus. Nam tincidunt ligula ex, in tristique ligula vulputate at. Sed volutpat tortor vitae lectus dapibus viverra quis eu nisi. Curabitur laoreet sollicitudin purus id efficitur. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nunc ac congue metus, vel malesuada nulla. Nullam rutrum aliquet nisl. Aenean eu diam ac risus aliquam ornare.

Index Terms

RISC-V, Digital Systems, Fault Tolerance

I. Introduction

Space systems operate in harsh environments and are exposed to radiation, extreme temperatures, and vacuum. The hostility of space can produce transient, intermittent, or permanent faults and affect the functioning of computer systems [1]. Single or multiple bit changes, for example, can lead to catastrophic failures in critical systems. Thus, system designers must use fault tolerance techniques to deal with the space environment's characteristics.

Several techniques can increase the reliability of electronic space systems. One of them relies on redundancy, and three classes of redundancy techniques are available, including spatial, temporal, and information [2]. Spatial redundancy consists of multiple instances of the same module and later comparison of their outputs to define the values to be presented to the system. Temporal redundancy relies on multiple executions of the same task, followed by comparing the results of all the executions to define the output. Information redundancy consists of adding bits to the data to detect and even correct the data in the event of an error.

Computing systems used in space applications must use processors that implement redundancy techniques to provide reliability to the system. An emerging processor architecture that is becoming an industry standard is RISC-V [3]. RISC-V's design is quite similar to the one of MIPS processor, having an optimized instruction set architecture (ISA) that aims at simplifying implementation. Although there were several soft-core implementations of RISC-V [4]–[12], and considering that the architecture has been evaluated for use in space applications [13], there are few fault-tolerant RISC-V processors.

Given the context above, in [14], we presented the design of a low-cost fault-tolerant RISC-V processor, aiming at hardening the processor against SEU (Single-event Upset) and SET (Single-event Transient) faults. The proposed solution applies triple modular redundancy (TMR) to protect the controllers and the arithmetic-logic unit (ALU), and the Hamming code for protecting all processor registers. The results obtained demonstrate that the circuitry added to the processor increases its reliability by reducing error propagation. In the present work, we extend the design and analysis carried out in [14]. We present improvements made in the processor's design to enable its use as a microcontroller integrated with peripherals through a shared bus. These improvements enabled further evaluation of the processor reliability based on the error propagation to the system output. Besides, we present a more detailed assessment of the propagation of errors resulting from SEUs by analyzing the output of a UART (Universal Asynchronous Receiver/Transmitter) integrated with the fault-tolerant processor in a System-on-Chip (SoC). From the results obtained, it is possible to verify the cause of the propagated errors. The developed processor has a lower cost than similar solutions described in the literature and has a low propagation of errors resulting from SEUs

and SETs. Thus, the work's main contributions rely on the design and reliability analysis of a low-cost fault-tolerant RISC-V processor for use in space applications.

The remainder of this article is organized as follows. Section II presents the main characteristics of the RISC-V architecture. Following, Section III discusses state-of-the-art works on RISC-V processors and fault tolerance. Next, Section IV describes the design of the proposed resilient processor and its extension to be integrated into an SoC. Section V describes the materials and methods used to implement and evaluate the processor and the system. Concluding, Section VI presents and discusses the results and Section VII gives the final remarks.

II. RISC-V

RISC-V is an open instruction-set architecture (ISA) developed by Waterman et al. [3]. The architecture specification is still in development [15], but the 32-bit ISA (RV32I) is already in its final version.

This architecture has three basic instruction sets: (i) RV32I, a set of 47 instructions complete enough to satisfy the basic requirements of modern operating systems; (ii) RV32E, resembles the previous set, but has only 15 registers; and (iii) RV64I, similar to the RV32I set, differs in the width of the data registers and program counter (PC) [16].

The RISC-V architecture was designed to simplify the implementation of processors on silicon. Instruction encoding is highly regular because the memory model is straightforward and has no complex instructions for accessing the data memory.

RISC-V implementations have small-size cores, usually much smaller than those of the Advanced RISC Machine (ARM) and x86 architectures, and several of them are available for use. Some examples include the Ibex [4] and PicoRV32 [5] processors, which are focused on the low use of logic resources, and the Ariane [6], Berkeley Out-of-Order Machine (BOOM) [7] and RI5CY [8] processors, which are more complete implementations of RISC-V and target devices with superior computing power.

Our implementation comprises only instructions from the RV32I instruction-set. The RISC-V organization has already frozen this instruction-set, which means that it will not have further modifications. This instruction-set enables our implementation to run the specified benchmark algorithms while keeping minimal resource usage.

III. Related Work

There are several processors designed for use in the space environment, mainly synthesizable soft-cores. The designs of these processors apply fault tolerance techniques to protect different parts of the circuitry. LEON4-FT [17] employs error correction codes (ECC) to protect the RAM blocks against single-event upsets. Configurable Fault-Tolerant Processor (CFTP) [18] uses TMR throughout the processor core. MIPS Crypto [19] applies TMR and matrix encoding techniques to protect the memory buffer units. In [20], the authors used the Hamming code to protect registers and TMR for providing fault tolerance for the message passing interface. The work of [21] covered the application of TMR and the shuffling of memory elements in a LEON3 processor. In [22], the authors attached a module to a LEON3 processor to perform an error detection and correction technique, called Parity per Byte and Duplication (PBD), to protect memory data. These examples illustrate how fault tolerance techniques were employed in SPARC- and MIPS-based processors. Below, we discuss some state-of-the-art works, emphasizing solutions for providing fault tolerance for RISC-V processors.

Heida et al. [23] started with the soft-core processor developed by Technolution BV (which implements the RV32I RISC-V instruction-set) and applied the Hamming and NMR (N-modular Redundancy) techniques in registers and the pipeline in a hybrid way. The fault injection was performed using several instances of the saboteur module placed in random positions of the processor's circuitry. These modules are placed just before decoders and voters and inject faults into the components' output with the highest probability of errors. The benchmarks used for emulation with fault injection are the Basic SoC test, ISA verification set, Hello world, Dhrystone, and Coremark. All benchmarks are used only for validation and take into account the values of the registers. The authors do not report an assessment of the propagation of errors in the system.

In [24], the authors introduced a technique to protect the register file of the lowRISC RISC-V processor against SEUs. The technique consists of duplicating the entire register file and storing parity. The second register file mirrors the former and performs all writing operations simultaneously. Error masking occurs when a register is read, and the parity checking mechanism detects an error. In this case, the mirrored register is used. The authors evaluated the reliability of the solution proposed through the injection of faults in FPGA and applied an injection campaign by inverting random bits of the register file at random times. As a result, the work reduced error propagation from 10.17% in the original version to zero in the modified version. Moreover, the occupancy of logical resources was lower than that of TMR-based solutions.

In [25], the authors present the use of the BL-TMR software to strengthen critical circuits in the Taiga RISC-V processor netlist. The solution utilizes the Xilinx Vivado tool's netlist to analyze and triple the critical nodes. The

authors tested the processor using neutron radiation in an experiment that involved 20 conventional Taiga processors and 20 TMR processors. The processors were configured on two different Xilinx Kintex Ultrascale KU040 FPGAs and ran the Dhrystone benchmark. The number of errors propagated was evaluated based on the UART output. As a result, the fault tolerance technique improved the mean work to failure by 24 times due to the reduction of the neutron cross-section by 33 times. However, the occupancy of logic resources increased by 5.6 times, and the operating frequency decreased by 27%.

Ramos et al. [26] presented a methodology to protect the processor ALU from faults in the FPGA configuration memory. The authors argue that it is not necessary to protect all ALU operations, but only the most common ones. These operations may differ according to the currently running algorithm, but SRAM-based FPGAs can be easily reconfigured for each application. The evaluation experiments used Dijkstra, Fibonacci, matrix multiplication, and quicksort algorithms. The authors protected only two operations of each algorithm using the TMR technique, and this partial TMR approach improved ALU's reliability with low overhead.

Aranda et al. [27] analyzed the critical bits of a RISC-V processor implemented in an SRAM-based FPGA. The authors carried out fault injection campaigns to analyze the viability of the processor for space applications. Two RISC-V implementations were evaluated based on the Rocket RISC-V, one unprotected and the other protected. The unprotected version consisted of the original Rocket processor, while the protected one was designed to improve the reliability of the FPGA configuration memory by applying distributed TMR (DTMR) to the netlist. The experiments carried out showed a 4% improvement in the reliability of a RISC-V processor implemented in an SRAM-based FPGA. As the unprotected one already had a high number of correct results (96%), the small increase obtained was sufficient to reach 99.9% of correct results.

A. Discussion

In [25] and [27], the authors explore the automated insertion of TMR at the netlist level and do not deal directly with the processor microarchitecture. Whereas these solutions seek to protect the FPGA's configuration memory, other critical components of the processor may remain exposed. The authors of [24] present a technique applied at the processor microarchitecture level, protecting the register file through a combination of dual modular redundancy (DMR) and parity checking techniques. On the other hand, the authors of [26] developed a specific technique to protect the processor against SEUs in the FPGA configuration memory, specifically in ALU operations.

In [23], the authors presented a hybrid combination of NMR and Hamming code techniques to protect the entire processor from taking advantage of the low impact of the NMR technique on the operating frequency and low resource overhead of the Hamming code technique.

As shown in Table I, all analyzed works use the UART output or internal registers as a criterion for evaluating the propagation of errors, referring to the output of the benchmark algorithms executed in the processor. Only in [23], [24], and [26], the authors took into account the processor registers and presented a classification of the types of errors obtained. The methods used for the fault injection campaign in most works were based on simulation and emulation because they are low-cost approaches. The only work that carries out the campaign with particles is [25], which performed the tests using a beam of neutron particles.

In this work, we apply TMR to the ALU and control logic and Hamming code to the register file and program counter. TMR is intended to protect the circuitry against some single-event effects (SEEs) and faults in the FPGA configuration memory. On the other hand, Hamming protects only registers against single bit upsets and has a lower logical resource overhead than TMR. We perform verification and testing through simulations, which is the cheapest method and has the most straightforward implementation. For evaluation, we assess both the processor output and the register values.

IV. FAULT-TOLERANT RISC-V

Our processor proposal is based on the non-privileged specification RISC-V [28]. We implemented the RV32I instruction set, except for the synchronization instructions and environment calls, and driven the design to minimize the logical resources necessary to implement the processor. For this reason, our baseline processor uses a single-cycle microarchitecture (organization) [16], which enables reducing the number of registers to be protected. Thus, the developed processor has five primary units:(i) instruction fetch; (ii) instruction decode; (iii) execution; (iv) memory access; and (v) write-back, as shown in Fig. 1.

TABLE I: Related works comparison

Work	Core	Techniques	Hardened Components	Verification and test	Evaluation
[23]	Technolution core	NMR and Hamming	Entire architecture	Emulation	Processor's registers
[24]	lowRISC	DMR and parity check	Register file	Emulation	Processor's output (UART) and registers
[25]	Taiga	BL-TMR	Configuration memory (automated)	Neutron radiation	Processor's output (UART)
[26]	lowRISC	Application-adapted TMR	ALU	Simulation	Processor's output (UART) and registers
[27]	Rocket	DTMR	Configuration memory (automated)	Simulation and Emulation	Processor's output (UART)
This work	Our core	TMR and Hamming	Register file, PC, ALU, and Control logic	Simulation	Processor's output (UART) and registers

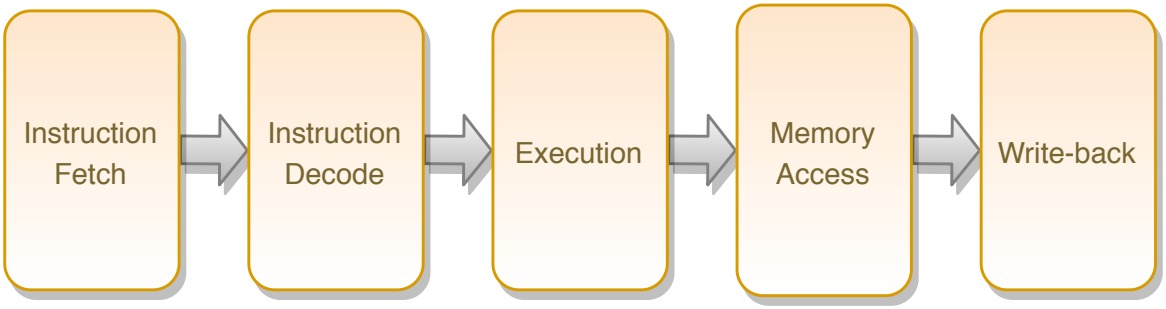


Fig. 1: Primary units of the proposed processor.

A. Hardware Design

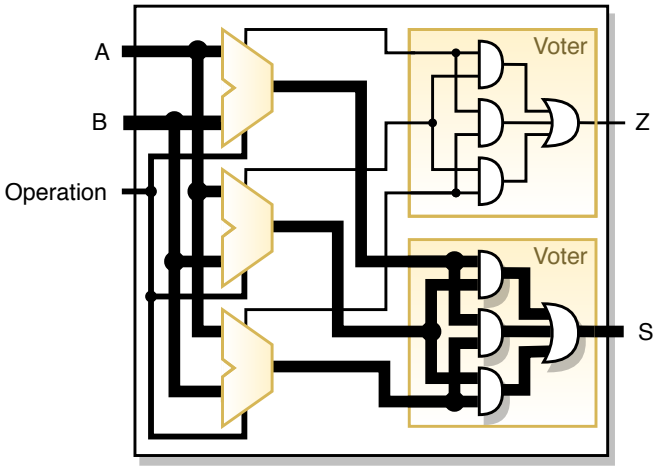
The instruction fetch unit comprises the program counter (PC), a 32-bit adder, and the circuitry necessary for computing jumps and conditional branches. The adder increments the PC in sequential execution or adds the offset in conditional branches.

The control unit integrates the instruction decoder responsible for identifying the operation to be performed. It decodes the instruction opcode and asserts/deasserts the control signals for the data path. We have implemented the main and ALU control units as a single component to facilitate the implementation of the fault tolerance techniques.

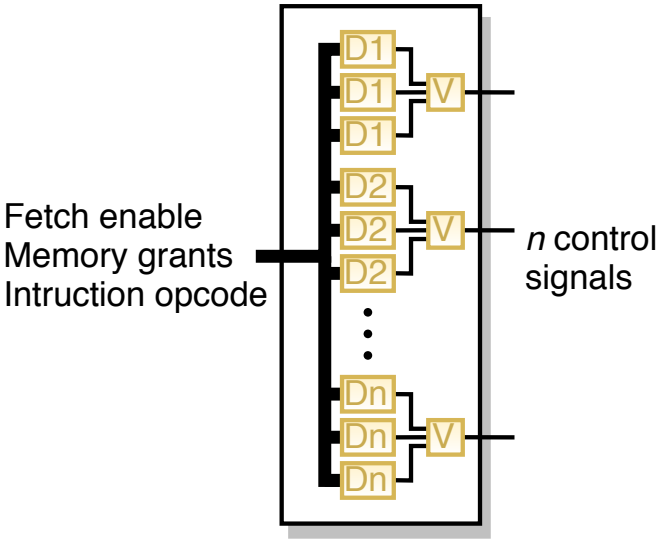
The execution unit performs the arithmetic and logical operations necessary for different instructions. These operations include: add, subtraction, logical shift (left or right), arithmetic shift (right only), set to 1 (on less than), and, or, and xor.

The memory access unit performs read and write operations to/from the data memory. The RISC-V specification describes that data access can be performed over 8-, 16-, and 32-bit data widths. However, all readings return 32-bit data. According to the type of instruction, the sign can be extended or not.

Writing to the register file is done by the write-back unit. This unit selects the value to be written to the specified register, which can be the ALU output, a variable read from the data memory, or an immediate operand of the instruction.



(a) ALU TMR



(b) Control TMR

Fig. 2: TMR architecture.

B. Fault Tolerance Application

We implemented fault tolerance techniques at the microarchitectural level, enabling the programmer to use the same instructions without worrying about the processor implementation. Our design focused only on soft-errors in the processor organization.

The ALU and the control unit were protected using the TMR technique. Thus, these components were tripled, and each of their outputs goes through a simple majority voting circuit. Fig. 2 illustrates how the TMR technique was applied to protect the ALU and the control logic. The ALU has one voter for each of the two outputs: zero and result. Concerning the control logic, the figure generically illustrates its implementation, showing a decoding/voting circuit for each of the n output signals generated by this block.

We then increased the width of all registers by six bits and added the Hamming encoder and decoder. The encoder uses XOR logic gates to compute the parity bits for the data to be written into the register. The decoder is responsible for verifying the parity of the encoded data and correcting the data when it detects an error. Correction is performed by inverting the wrong bit.

Fig. 3 and Fig. 4 show the Hamming code technique applied to the instruction fetch unit and the register file, respectively. In the former, we protected the register of the program counter with an encoder and a decoder. The register file was protected with a single encoder at its input port and a pair of decoders at its two output ports.

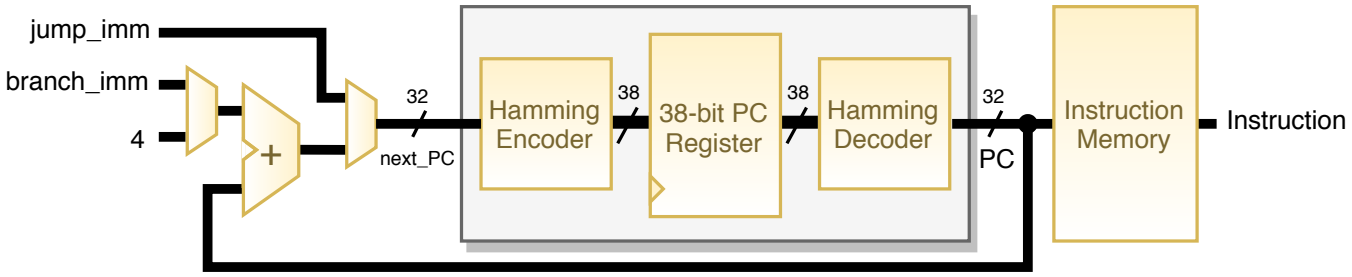


Fig. 3: Instruction Fetch unit using Hamming.

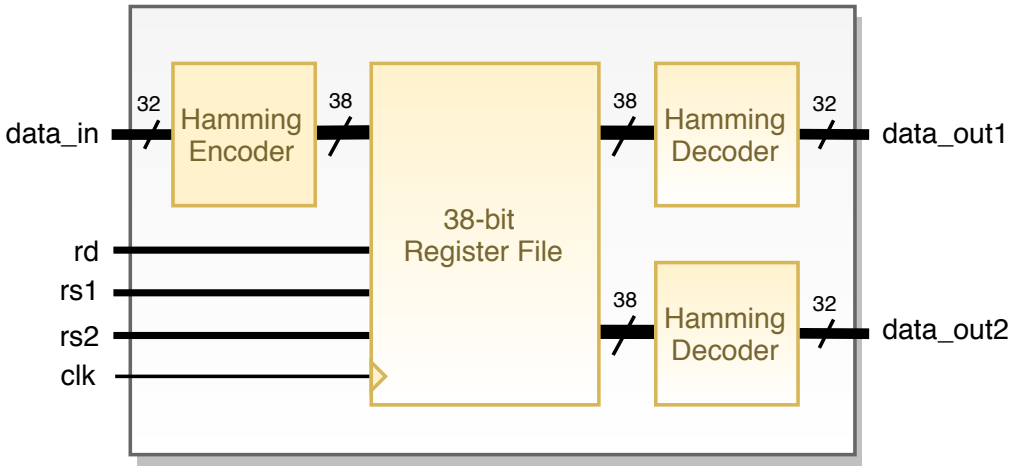


Fig. 4: Register file using Hamming.

V. Materials and Methods

We implemented a baseline RISC-V soft-core to measure resource overhead and serve as a golden model to assess the enhanced resilience achieved with the fault tolerance techniques. We described the soft-core in VHDL and used a platform-independent approach; no vendor-specific intellectual property (IP) blocks were used. This approach facilitates the processor's reuse using devices from different manufacturers.

Subsequently, we improved the design of the processor to integrate it into a System-on-Chip. The SoC served as an experimental platform and aimed at prototyping an FPGA device for future physical tests.

We used the Xilinx Vivado Design Suite 2019.1 and the Zynq ZC7020 SoC device to collect the synthesis results. The costs and performance metrics include the number of Look-up Tables (LUTs) and flip-flops (FFs), the dynamic power dissipation (P_{dyn}) with processors running at 50 MHz, and the maximum operating frequency (F_{max}).

We also used the Mentor Graphics ModelSim simulator to verify the developed processor and assess its resilience by applying the scripts proposed in [29] to perform fault injection. The codes used in the execution of all the algorithms were compiled with a GNU Compiler Collection (GCC) adapted for the RISC-V architecture [30].

A. Design verification

We evaluated the fault tolerance techniques through simulations by using scripts based on the work of [29]. Our scripts consist of a first fault-free execution used as a reference (golden signature) and several other executions with fault injections representing the actual experiment. The logic elements targeted by the fault injection depend on the type of fault model applied. In particular, we injected two types of fault: SEU (affecting sequential logic) and SET (affecting combinational logic). The fault injection executions mimic the high and low flux of ionizing particles striking the circuit. For this purpose, we injected one or ten bit-flips in each simulation run. Each bit-flip targets a random node (SET) or register (SEU) at a random moment within the simulation period. Please, note that these faults are injected in different moments of the run with ten bit-flips in different moments and do not represent concurrent multiple bit upsets. The bit-flips target only the processor's architecture and does not affect the instruction and data memories.

The fault injection simulation uses different methodologies for each type of fault. We consider that the SET events affect combinational logic. Therefore, to simulate an error caused by a SET event, the entire system's external and internal signals were filtered out. The injection of this fault consists of inverting and freezing a random bit's signal at a random moment. In contrast, we consider that the SEU events affect data stored in registers. For this reason, each SEU fault injection was made by inverting one bit of the data stored in a random register of the system.

The script was executed in four processor implementations, which apply fault tolerance in different combinations: (*i*) non-hardened, without fault-tolerance. (*ii*) Hamming, with hamming applied in the registers. (*iii*) TMR, with TMR applied in the control and ALU. and (*iv*) TMR and hamming, which applies hamming in the registers and TMR in the control and ALU.

The results are evaluated by checking the error in two ways. The first one is made by the check of internal registers with respect on the golden signature. The second consists in checking the response of the full circuit in the primary output, that in our case is made through the UART output. The primary output check is made only on the SoC version of the processor. This approach enabled obtaining an analysis of the most critical components for the functioning of the processor.

We used the following algorithms to verify and evaluate the processor implementations:

- Sum of vectors: an algorithm that adds two 300-element unidimensional arrays and stores the result in a third array.
- CCSDS-123: a hyperspectral image compression algorithm standardized by the Consultative Committee for Spatial Data Systems (CCSDS) and based on the implementation of [31].
- Coremark benchmark: tests various processor components and is considered efficient for fault tolerance testing [32].

For the baseline version of the processor, we ran 100 experiments. However, for the SoC version, we decided to improve our sample statistics by increasing the number of experiments from 100 to 1,000 runs for the SoC verification. The increase in the number of simulations, combined with the larger circuit, substantially increased the execution time, making only the sum of vectors algorithm feasible in our SoC evaluation. However, we still used all the three algorithms to evaluate the baseline processor.

For the SoC evaluation, we considered the processor's internal registers and the UART output. This approach enables to classify errors based on the categories proposed by [33]. We use the following categories: (i) match, when the application ran correctly, regardless of the execution time; (ii) mismatch, when the application runs but with incorrect output values; and (iii) hang, when the application does not produce any results or does not end the execution.

B. Experimental platform

1) Processor core extension: We integrated the processor with other components in the form of a basic System-on-Chip (SoC) to enable the proposed experiments. However, an SoC implemented in FPGA must use real memory, which is not as fast as a register. Also, an SoC must provide the processor with access to peripherals that add other functionality (e.g., communication). Therefore, bearing in mind that a single-cycle processor is not viable in a real system, we extended the original core by implementing the control using the finite state machine (FSM) illustrated in Fig. 5. The possible states are Reset, Idle, ReqInstr (requesting instruction from memory—the instruction fetch), Run (instruction execution), DMemStall (data memory stall), and UpdatePC (updating the program counter).

The control starts in the Reset state and then goes to the Idle state, in which it waits for the assertion of the input signal $fetch\_enable$. When this signal is asserted, the instruction fetch begins, and the FSM remains in the ReqInstr state. After instruction reading, the state machine advances to the Run state to execute the instruction's operation. If the instruction does not access the data memory, the control updates the program counter (UpdatePC) and goes back to ReqInstr state. However, whether the instruction performs a data load or store, the FSM advances to the DMemStall state, which asserts the request signal for the data memory or bus and waits for the grant signal to update PC and fetch the next instruction. This implementation made the core a multi-cycle, even though most instructions execute just one clock cycle (e.g., for I- and R-type instructions).

2) System architecture: The complete implementation of an SoC consists of adding components and peripherals that enable prototyping the processor running real applications on FPGA. The UART programming interface enables writing the application code into the instruction memory and loading contents to data memory. Moreover, it is worth noting that we described the processor's memories to take advantage of the FPGA's memory blocks (BRAM).

The programming interface works using 32-bit messages that signal the operation (the eight most significant bits) and data (the 24 least significant bits). The commands are:

1) Write in a specific register to reset the processor.

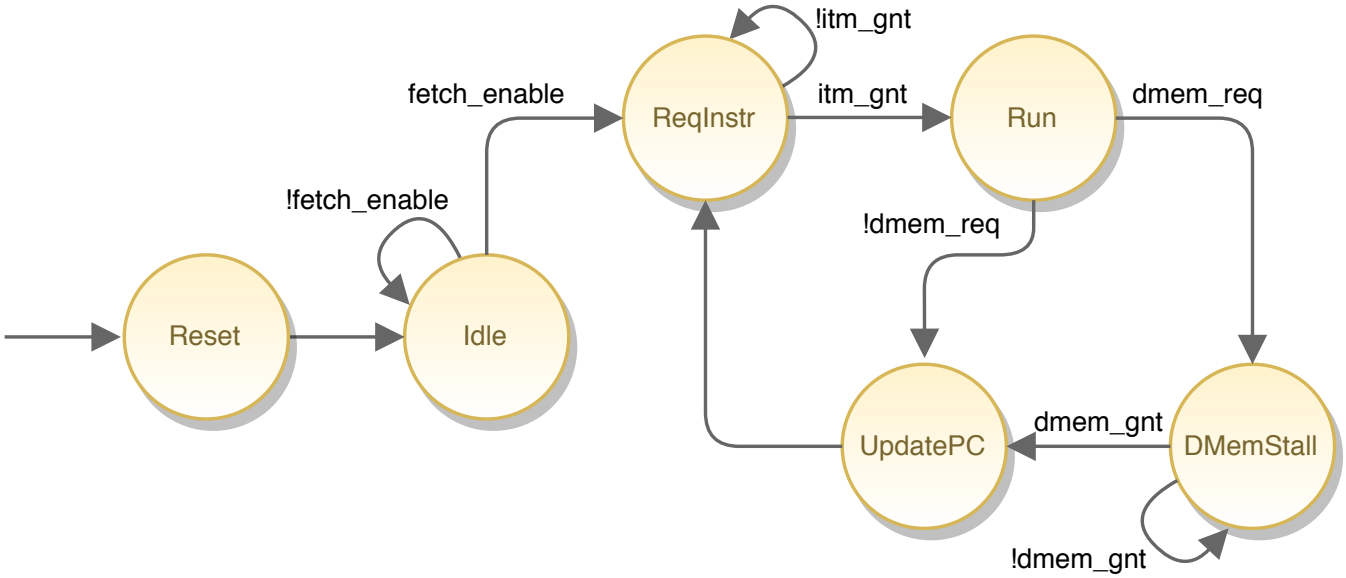


Fig. 5: Control finite state machine.

2) Write the least significant bit of data to the $fetch_{enable}$ register.
3) Write data in memory, where the 24 bits of the message represent the address of the cell to be written, and the 32 bits of the next message are the data to be written into the cell.
4) Read data from memory, where the message content is the address to be read.

The implemented bus follows the AMBA AXI4-lite protocol and implements the processor as the only master. To choose the access between the bus and the data memory in load and store instructions, the processor uses the 15_{th} bit, which will define in the multiplexer whether the access will be to memory or peripherals. Although the implemented bus supports other peripherals, the only one we have implemented and connected is the UART IP. Fig. 6 shows the diagram block of the implemented SoC.

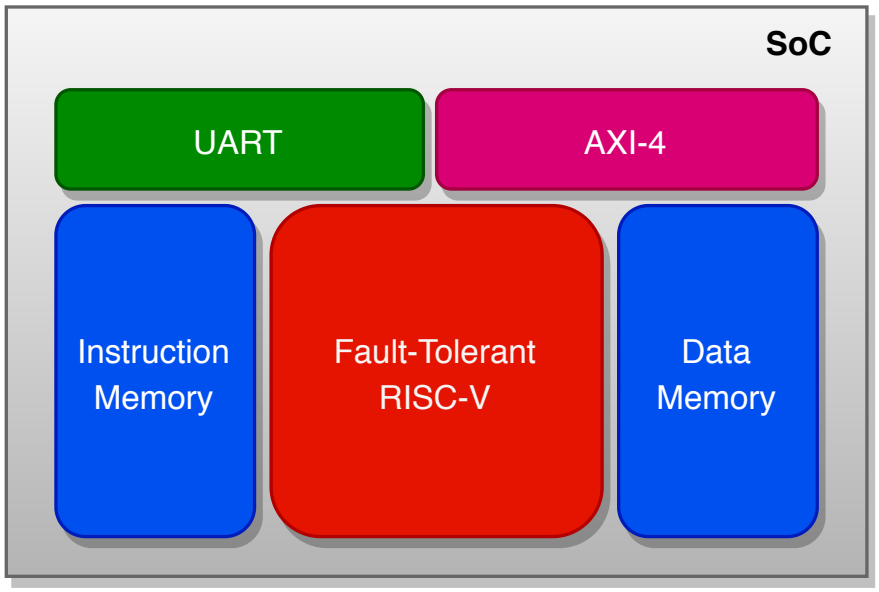


Fig. 6: SoC structure.

VI. Experimental Results

This section presents the synthesis and simulation results of the Fault-Tolerant RISC-V System-on-Chip developed. We first present the experimental results obtained from providing fault tolerance to the baseline processor and compare them with the literature. Subsequently, we present and analyze the results obtained at the system level.

A. Processing core results

The use of fault tolerance techniques increases the number of logical resources used by the processor. Table II compares four different processor implementations. In the implementation that applies only TMR to the ALU and the control unit, the number of LUTs is almost 50% greater than in the non-hardened processor. The occupancy of FFs remains the same because these circuits are purely combinational. In the implementation applying the Hamming code technique only, the number of FFs increases due to the increase of the width of the processor's registers (6 bits are added to each register). The extra LUTs are due to the encoding and decoding blocks and the increase in the output multiplexers' channel width.

TABLE II: Core synthesis results

Implementation	**LUTs**	**FFs**	**F_{max} (MHz)**	**P_{dyn} (mW)***
Non-hardened	1 613	1 024	74.65	146
TMR	2 387	1 024	66.50	151
Hamming	1 854	1 216	54.77	162
TMR and Hamming	2 748	1 216	49.99	171

*Power dissipation when running at 50 MHz.

The maximum operating frequency of the processor decreases as the critical path increases. In the non-hardened implementation, the processor can operate at 74.65 MHz. When applying TMR, a drop of approximately 10% in the maximum frequency occurs. When applying only the Hamming code technique, the degradation increases to 26% compared to the non-hardened processor. Finally, combining the two techniques, the maximum operating frequency is around 50 MHz, 33% less than that of the non-hardened implementation.

The power dissipation increase ranges from 3% in applying the TMR to 17% when combining the two techniques. In this experiment, all processors are operating at 50 MHz, the maximum operating frequency of the slowest implementation.

Table III shows the costs in terms of resources related to the developed processor and the other low-cost RISC-V soft-cores without fault tolerance mechanisms. We disregarded their status and control registers (CSR) for a fairer comparison with these processors because our implementation does not address these registers.

TABLE III: Comparison with low-cost RISC-V processors

Soft-core	**LUTs**	**FFs**	**P_{dyn} (mW) @ 50 MHz**
Ibex [4]	1 680	1 339	135
PicoRV32 [5]	1 580	1 432	119
mRISCV [11]	2 460	2 014	116
DarkRISCV [12]	2 470	2 262	145
This Work	1 613	1 024	146

Given that the processor proposed in this work was developed with a focus on low resource utilization, mainly for sequential logic, its number of LUTs is lower than those of most of the other processors, and the use of FFs is smaller than all the other RISC-V implementations. With all the soft-cores running at 50 MHz, our implementation is among those with the highest power dissipation.

Among the fault-tolerant RISC-V cores, our implementation has the lowest overhead, as shown in Table IV. While the cores presented in [23] and [25] had an LUT overhead higher than five times, our implementation has an overhead of only 70%. Concerning the sequential logic, the works above obtained overheads three times higher than their baseline processors, whereas our implementation increased the number of FFs by only 19%. Our approach enabled reducing the number of elements to be protected, resulting in a lower silicon overhead.

Table V shows the error propagation rate when injecting SEU faults that affect sequential logic. The values presented in the table were obtained by computing the number of runs that had errors propagated in registers. For example, 64 of the 100 runs of the sum of vectors algorithm propagated faults in registers for single-fault injection (i.e., 64%).

TABLE IV: Comparison with fault-tolerant RISC-V processors

Processor core	LUTs overhead ratio	FFs overhead ratio	F_{max} (MHz)
[23]	5.96×	3.70×	36.10
[25]	5.64×	3.00×	227.20
TMR and Hamming	1.70×	1.19×	49.99

TABLE V: Error Propagation for SEU

Algorithm	1-fault injection		10-fault injection	
	Non-hardened	Hardened	Non-hardened	Hardened
Sum of vectors	64%	0%	99%	35%
CCSDS 123	71%	0%	98%	3%
Coremark	73%	0%	100%	6%

In the first experiment, a fault was injected into each run at moments and registers selected randomly. The hardened processor masked the fault in all tests because the Hamming code technique can correct all individual errors. When injecting ten faults in random moments and registers, all algorithms had an error rate close to 100% in the non-hardened processor. However, in the hardened implementation of the processor, Hamming was able to mask 65% of errors when performing the sum of vectors application and more than 90% of errors when executing the other algorithms. These experiments did not take into account FPGA configuration memory, focusing only on the processor design.

Table VI presents the results of the injection of faults in combinational logic. When simulating the occurrence of a single fault, we noticed that the sum of vectors algorithm obtained an error propagation rate reduced by approximately 60% when the processor is hardened. When using the CCSDS-123 algorithm, the error rate was reduced by 84%, and when using Coremark, the error rate decreased by 78%. When simulating the injection of 10 faults, the execution of all algorithms obtained a high error propagation rate, even using the processor's hardened implementation. The reason is that, in our simulation model, SET has a higher probability of affecting signals critical to the execution than SEU. This probability increases because, while the values stored in registers are not crucial at all times, combinational logic entities, such as TMR voters, control logic, ALU, and multiplexers, are always connected to critical parts of the circuits, even when applying TMR. Also, the propagated errors are accumulated in registers currently being used, affecting essential values in the register file. Nevertheless, the hardened processor still reduced the error propagation between 13% and 21% compared to the processor without fault tolerance techniques.

TABLE VI: Error Propagation for SET

Algorithm	1-fault injection		10-fault injection	
	Non-hardened	Hardened	Non-hardened	Hardened
Sum of vectors	39%	16%	99%	78%
CCSDS 123	96%	15%	100%	87%
Coremark	72%	16%	100%	87%

The authors of [20] presented a similar evaluation. Their fault injection model uses bit-flips for registers and freezing signals for combinational parts, injecting a determined number of faults in each entity. In total, 104 faults are injected into the entire system. Compared with the non-hardened implementation, their hardened version reduced the average propagation error by 15.1% with an area overhead of 1.01%. On the other hand, our hardened implementation resulted in a higher reduction in the error propagation rate (i.e., 50.4%) at the price of a higher overhead (i.e., 18.75% FFs and 70.3% LUTs). This result is due to the different algorithms and fault models evaluated in each work, which cannot be directly compared. The system implemented by [20] is dual-core, and the algorithms are optimized to use both cores, which changes the system's behavior and, consequently, the critical parts of the processor. It is worth mentioning that the authors of [25] did a test using a neutron beam and reported an improvement of 24 times in the average work to failure. Although we do not measure the average work to failure, our simulation of 10 fault injections estimates an improvement in SEU error propagation of approximately 17 times when we use the hardened processor and the Coremark algorithm.

B. Experimental platform results

The experimental platform has a higher number of features and functionalities than the core, which results in additional costs. Table VII shows the number of logic resources used by the SoC synthesized in the Zynq-7000 device, considering the different processor implementations (Non-hardened and TMR and Hamming). Comparing the non-hardened implementations, we note that the SoC's additional features implied an increase of 12.8% in the logical resources occupied by the core, as we also observe the degradation of performance in 32.9%. The maximum operating frequency is limited by the $auipc$ (Add Upper Immediate to Program Counter) instruction, which is performed almost entirely using a full combinational circuit with a lengthy critical path. In the hardened implementation, the increase in the number of LUTs equals 20.7%. Also, the operating frequency degradation equals 30.4%, similar to that of the non-hardened implementation. We noticed that the SoC dissipates less dynamic power than the core only because it has fewer I/O ports. Besides, we observe that the use of fault tolerance techniques almost doubled the dissipation of dynamic power.

TABLE VII: SoC synthesis results

Implementation	**LUTs**	**FFs**	**F_{max} (MHz)**	**P_{dyn} (mW)**
Non-hardened	1 819	1 253	50,11	58
TMR and Hamming	3 317	1 457	34.81	100

Table VIII presents error propagation due to SEU injection to the experimental platform, including the processor's internal registers and the UART output. Error propagation was reduced 35 times in the processor and 13 times in the UART output when the hardened implementation is subject to only one SEU. When injecting ten SEUs, error propagation is much higher. However, using hardened implementation still reduces the number of errors propagated. It is worth noting that some faults injected propagate error in the hardened implementation due to unprotected registers in the other components of the SoC.

In total, the fault injection may target 1,255 and 1,453 FFs of the non-hardened and hardened implementations, respectively. This number of FFs may vary depending on the system's parameters, and not all of them are hardened.

TABLE VIII: SEU error propagation to registers and UART

SoC	**1-fault injection**		**10-fault injection**	
	Non-hardened	**Hardened**	**Non-hardened**	**Hardened**
Registers	45.2%	1.3%	99.6%	34.3%
UART output	9.1%	0.7%	48.2%	6.1%

We evaluated the output of the UART considering the three categories of results mentioned in Subsection V-A (i.e., match, mismatch, and hang). Table IX presents the experimental results. In the injection of only one SEU, we found that the use of fault tolerance reduced the incidence of output mismatch errors nine times, while the number of executions with hang errors was reduced by 18 times. When we simulated the injection of ten SEUs, the non-hardened implementation propagated the error to the UART output in 48.2% of executions, while the hardened implementation propagated in only 6.1% of the runs. The reduction in output mismatches and hangs with the hardened processor was 6.2 and 14.1 times, respectively.

TABLE IX: Categorization of error propagation to UART

Outcome Category	**1-fault injection**		**10-fault injection**	
	Non-hardened	**Hardened**	**Non-hardened**	**Hardened**
Match	90.9%	99.3%	51.8%	93.9%
Mismatch	3.6%	0.4%	29.8%	4.8%
Hang	5.5%	0.3%	18.4%	1.3%

The application of fault tolerance made the number of SEU faults propagated to the system output to be minimal, even considering the worst case that propagated errors in 6.1% among all executions. Most executions with errors are output mismatches and can be easily mitigated at the software level. The incidence of hang errors occurred in only 1.3% of the total. These errors represent non-correctable failures that require the use of additional components for their identification and correction.

Table X presents a comparison with two related works that use similar metrics. As each of them uses different algorithms for evaluation, we selected the result based on the one more similar to the algorithm used in our work (i.e., sum of vectors). From [26], we selected the results the authors obtained running the Fibonacci algorithm. From [23], we selected the results from the execution of the Systest algorithm. As we can see, our work obtained similar results in comparison with the other works, with more than 99% of correct runs (matches).

TABLE X: Comparison of error propagation to UART

Outcome Category	Ramos et al. [26]		Aranda et al. [27]		This work	
	None	Full TMR	None	DTMR	None	Hardened
Match	93.94%[1]	99.67%[1]	97.4%	99.94%	90.9%	99.3%
Mismatch	3.21%[2]	0.1%[2]	0.81%	0.0%	3.6%	0.4%
Hang	2.85%	0.25%	1.79%	0.06%	5.5%	0.3%

[1]Considering all correct results, even with internal architecture failures.
[2]Considering simple mismatches and execution exceptions.

VII. Conclusions

In this work, we assessed a fault-tolerant low-cost RISC-V processor and extended it to enable its use as a microcontroller SoC. The results showed how the use of TMR and Hamming code techniques increases costs and that it is possible to verify the cause of the propagated errors. The developed processor has a lower cost than similar solutions and presents a low propagation of errors resulting from SEUs and SETs faults.

As future work, we intend to test the processor at the ISIS Neutron research center using the ChipIr microelectronic irradiation instrument, which enables performing SEE tests with neutron beams. This test will represent the first stage of qualifying the processor for use in future computing systems embedded in nano-satellites. We also intend to provide full support to the Coremark score to improve our metrics comparison. The source code of the processor core is available at [34].

Funding

This work was financed in part by CNPq – the Brazilian National Council for Scientific and Technological Development – Processes 315287/2018-7 and 436982/2018-8, Region d'Occitanie – Contracts 20007368/ALDOCT-000932 and UM-181386, VAN ALLEN Foundation – Contract UM-181387, and École Doctorale I2S de l'Université de Montpellier.

References

[1] M. Yang, G. Hua, Y. Feng, and J. Gong, *Fault-tolerance techniques for spacecraft control computers*, 1st ed. Wiley Publishing, 2017.
[2] D. J. Sorin, "Fault tolerant computer architecture," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–104, 2009.
[3] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: Base user-level ISA," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-62, May 2011. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html
[4] P. D. Schiavone *et al.*, "Slow and steady wins the race? a comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *2017 27th International Symp. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sep. 2017, pp. 1–8.
[5] C. Wolf, "PicoRV32 - a size-optimized RISC-V CPU," 2019. [Online]. Available: https://github.com/cliffordwolf/picorv32
[6] P. platform, "Ariane documentation 2019," 2019. [Online]. Available: https://github.com/lowRISC/ariane/
[7] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanović, "BOOMv2: an open-source out-of-order RISC-V core," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html
[8] A. Traber, M. Gautschi, and P. D. Schiavone, "RI5CY: User manual," Tech. Rep., 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," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[10] E. Matthews and L. Shannon, "TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features," in *2017 27th International Conf. on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–4.
[11] C. Duran, L. Rueda, G. Castillo, A. Agudelo, C. Rojas, L. Chaparro, H. Hurtado, J. Romero, W. Ramirez, H. Gomez, H. Hernandez, J. Amaya, and E. Roa, "A 32-bit microcontroller featuring a RISC-V core," 2016. [Online]. Available: https://github.com/onchipuis/mriscv
[12] M. Samsoniuk, "Opensource RISC-V DarkRISCV," 2019. [Online]. Available: https://github.com/darklife/darkriscv
[13] S. Di Mascio, A. Menicucci, G. Furano, C. Monteleone, and M. Ottavi, "The case for RISC-V in space," *Lecture Notes in Electrical Engineering*, vol. 550, no. 9783030119720, pp. 319–325, 2019, international Conference on Applications in Electronics Pervading Industry, Environment and Society, APPLEPIES 2018 ; Conference date: 26-09-2018 Through 27-09-2018.
[14] 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)*, April 2020, pp. 1–5.
[15] A. Waterman and K. Asanović, "The RISC-V instruction set manual volume I: Unprivileged ISA," *RISC-V Foundation (June 2019)*, 2019.

[16] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.
[17] M. Hijorth *et al.*, "GR740: Rad-hard quad-core LEON4FT system-on-chip," in *DASIA 2015-DAta Systems in Aerospace*, vol. 732, 2015.
[18] C. A. Hulme, H. H. Loomis, A. A. Ross, and R. Yuan, "Configurable fault-tolerant processor (CFTP) for spacecraft onboard processing," in *2004 IEEE Aerospace Conf. Proc. (IEEE Cat. No.04TH8720)*, vol. 4, March 2004, pp. 2269–2276 Vol.4.
[19] B. Ustaoglu and B. O. Yalcin, "Fault tolerant register file design for MIPS AES-crypto microprocessor," in *2015 IEEE International Conf. on Electronics, Circuits, and Systems (ICECS)*, Dec 2015, pp. 442–445.
[20] M. Didehban, S. Khosjbakht, H. R. Zarandi, and S. Pourmozaffari, "Reducing of soft error effects on a MIPS-based dual-core processor," in *2010 15th CSI International Symp. on Computer Architecture and Digital Systems*, Sep. 2010, pp. 151–152.
[21] M. J. Wirthlin, A. M. Keller, C. McCloskey, P. Ridd, D. Lee, and J. Draper, "SEU mitigation and validation of the LEON3 soft processor using triple modular redundancy for space processing," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 205–214.
[22] R. C. Goerl, P. R. Villa, L. B. Poehls, E. A. Bezerra, and F. L. Vargas, "An efficient EDAC approach for handling multiple bit upsets in memory array," *Microelectronics Reliability*, vol. 88-90, pp. 214 – 218, 2018, 29th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis (ESREF 2018).
[23] W. F. Heida, "Towards a fault tolerant RISC-V softcore," Ph.D. dissertation, Delft Univ. of Technology, 2016, http://resolver.tudelft.nl/uuid:cee5e97b-d023-4e27-8cb6-75522528e62d.
[24] A. Ramos, A. Ullah, P. Reviriego, and J. A. Maestro, "Efficient protection of the register file in soft-processors implemented on Xilinx FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 299–304, 2018.
[25] A. E. Wilson and M. Wirthlin, "Neutron radiation testing of fault tolerant RISC-V soft processor on Xilinx SRAM-based FPGAs," in *2019 IEEE Space Computing Conf. (SCC)*, July 2019, pp. 25–32.
[26] A. Ramos, R. G. Toral, P. Reviriego, and J. A. Maestro, "An ALU protection methodology for soft processors on SRAM-based FPGAs," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1404–1410, 2019.
[27] L. A. Aranda, N.-J. Wessman, L. Santos, A. Sánchez-Macián, J. Andersson, R. Weigand, and J. A. Maestro, "Analysis of the critical bits of a RISC-V processor implemented in an SRAM-based FPGA for space applications," *Electronics*, vol. 9, no. 1, p. 175, Jan 2020.
[28] A. Waterman and K. Asanović, "The RISC-V instruction set manual-volume I: User-level ISA-document version 2.2," *RISC-V Foundation (May 2017)*, 2017.
[29] R. Travessini, P. R. C. Villa, F. L. Vargas, and E. A. Bezerra, "Processor core profiling for SEU effect analysis," in *2018 IEEE 19th Latin-American Test Symp. (LATS)*, March 2018, pp. 1–6.
[30] RISC-V:Organization, "Gnu toolchain for RISC-V, including GCC," 2019. [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain
[31] L. M. V. Pereira, D. A. Santos, C. A. Zeferino, and D. R. Melo, "A low-cost hardware accelerator for CCSDS 123 predictor in FPGA," in *2019 IEEE International Symp. on Circuits and Systems (ISCAS)*, May 2019, pp. 1–5.
[32] H. Quinn *et al.*, "Using benchmarks for radiation testing of microprocessors and FPGAs," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2547–2554, Dec 2015.
[33] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: Association for Computing Machinery, 2013.
[34] HARV, "HArdened Risc-V," 2020. [Online]. Available: http://xarc.org/harv