Characterization of a RISC-V Microcontroller Through Fault Injection

Asciolla, Dario; Dilillo, Luigi; Santos, Douglas; Melo, Douglas; Menicucci, Alessandra; Ottavi, Marco

DOI
10.1007/978-3-030-37277-4_11

Publication date
2020

Document Version
Final published version

Published in
Applications in Electronics Pervading Industry, Environment and Society - APPLEPIES 2019

Chapter 11
Characterization of a RISC-V Microcontroller Through Fault Injection

Dario Asciolla, Luigi Dilillo, Douglas Santos, Douglas Melo, Alessandra Menicucci and Marco Ottavi

Abstract This article reports the results of fault injection on a microcontroller based on the RISC-V (Riscy) architecture. The fault injection approach uses fault simulation based on Modelsim and targets a set of 1000 fault injected per microcontroller block and per benchmarck. The chosen benchmarks are the Dhrystone and CoreMark that may represent generic workloads. The results show certain block are more prone to fault than others, as also confirmed by a vulnerability analysis that correlates the number of observed faults and the rate of access to the blocks.

Keywords RISC-V · Fault injection · Micronontroller · Simulation

D. Asciolla
LIRMM, University of Montpellier, Montpellier, France
e-mail: dasciolla@lirmm.fr

L. Dilillo
LIRMM, CNRS, University of Montpellier, Montpellier, France
e-mail: dilillo@lirmm.fr

D. Santos · D. Melo
Laboratory of Embedded and Distributed Systems, University of Vale do Itajaí, Itajaí, Brazil
e-mail: douglasas@edu.univali.br

D. Melo
e-mail: drm@univali.br

A. Menicucci
Department of Space Engineering, Delft University of Technology, Delft, Netherlands
e-mail: A.Menicucci@tudelft.nl

M. Ottavi (✉)
Department of Electronics Engineering, University of Roma Tor Vergata, Rome, Italy
e-mail: ottavi@ing.uniroma2.it

S. Saponara and A. De Gloria (eds.), *Applications in Electronics Pervading Industry, Environment and Society*, Lecture Notes in Electrical Engineering 627,
https://doi.org/10.1007/978-3-030-37277-4_11

11.1 Introduction

The space environment interaction with electronics represents an important challenge for satellite missions. Ionizing particles and electromagnetic radiations affect electronic devices by inducing faults in specific circuit areas that may lead to system failures. These failures can be temporary, with the occurrence of the so-called Soft Errors, or permanent with the occurrence of Hard Errors [1]. Moreover, the exposition of electronics to radiation induces to premature aging, with the deterioration of performance and/or functionality of the systems. For this reason, the space industry resorts to hardening techniques that are generally based on hardware and software redundancy, with the generation of custom devices. This type of solutions, while effective, are energy and hardware greedy and leads to costs several times higher than for conventional COTS (Commercial Off The Shelf) electronics.

Completely programmable hardware platforms such as FPGA make the development of a system extremely flexible but, at the same time, require ad-hoc designing and therefore they are not available to a broad programming community. On the other side, the use of standard processors ISA architectures allows many developers to design applications, but the closeness of the architectures does not allow the designers to make easy and cheap modifications to the underlying hardware.

RISC-V allows developers to combine the advantages of both worlds [2], providing flexibility to both hardware and software. On one side we can modify the architecture to obtain specific applications, while on the other side we can open to applications made by programmers, who are unaware of the underlying hardware. RISC-V is an open ISA born from both the academia and research environment [3]. The RISC-V ISA was originally developed in the Computer Science Division of the EECS Department at the University of California, Berkeley [3]. RISC-V represents a promising platform to experiment different techniques and to design new architectures.

The purpose of this paper is characterizing a RISC-V core, through an extensive simulation-based fault injection campaign with the target of identifying the most critical modules within the core. For this purpose, the study of sequential modules is fundamental, especially for COTS components, because they can suffer from bit flips [4]. The data stored in the memory element can be corrupted, with Single Event Upsets (SEUs), after the interaction with ionizing particles and electromagnetic radiations in general. This paper analyzes the effects of SEU (Single Event Upset) in a RISC-V core. This characterization is useful in perspective to design a fault-tolerant version of a RISC-V core for space applications by applying targeted hardening techniques, shaped on the sensitivity of the different blocks composing the system. The final target is the use of this kind of low cost hardened processor within nanosatellites, like Cubesats, and in other systems where high reliability, flexibility and low cost are required.

The rest of the paper is organized as follows. Sections 11.2 and 11.3 detail the RISC-V platform and fault injection methodology, respectively. Section 11.4 introduces the chosen algorithmic benchmarks. Sections 11.5 and 11.6 present the simulation results as well as an analysis of the microcontroller vulnerability. Conclusions are given in Sect. 11.7.

11.2 Platform

The chosen RISC-V platform is the Parallel Ultra Low Power (PULP) Platform. It is been designed from Integrated Systems Laboratory (IIS) of ETH Zrich and Energy-efficient Embedded Systems (EEES) group of the University of Bologna [5]. It is an open-source platform and very useful for our purposes, because it is possible to access all parts of the system. In this specific case, from this platform, the Pulpino microcontroller has been chosen. It is built for RISC-V Riscy and zero-riscy core [6]. Pulpino offers a separate memory for instructions and data. It uses AXI (Advanced eXtensible Interface) interface as its main interconnect and a bridge to APB (Advanced Peripheral Bus) for simple peripherals [6]. All architectural details about Pulpino platform [6] are shown in Fig. 11.1.

The choice of Riscy core for this study is based on the following reasons. Firstly, it is a four-stage RISC-V core and it can run most of the typical workloads. It is a 32-bit core and for the chosen configuration, it can manage only integer numbers. It implements the RV32I instruction set. All software that runs over this core has been compiled using the GNU RISC-V Toolchain [7] with an optimization level equal to 3.

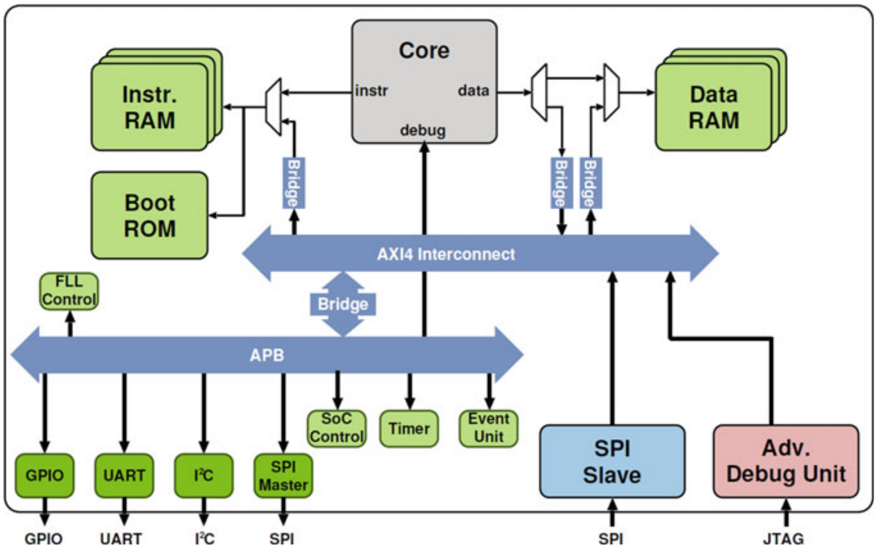


Fig. 11.1 RISC-V PULPino platform architecture overview

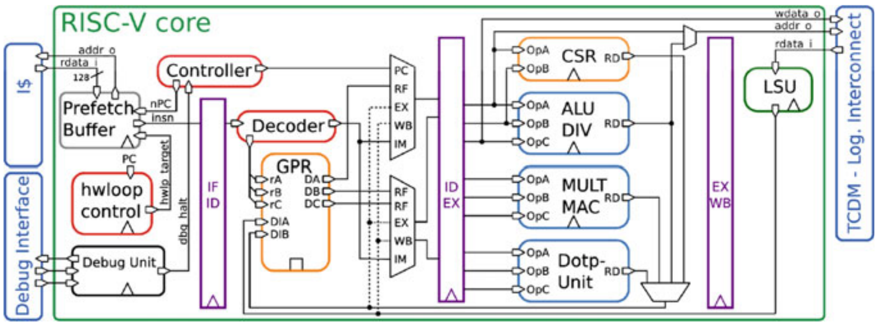


Fig. 11.2 RISC-V Riscy core architecture overview

Riscy core architecture overview [6] is shown in Fig. 11.2. For the characterization campaign, we are focusing over all sequential modules inside the core. In Riscy core there are four stages: Instruction Fetch, Instruction Decode, Execution and the Write Back. All stages are separated by an interface register. In the Instruction Fetch stage, sequential parts are inside the prefetch buffer, the hardware loop control module and inside the instruction fetch top-level registers. In the Instruction Decode stage, sequential modules are inside the register file and the controller unit. In the Execution stage, the control-state registers and the multiplier are both sequential modules. In the write-back stage, the load-store unit is the only sequential module. The only architectural modification that was made consists of the redefinition of state machines inside sequential modules by using the binary codification instead of labels. This modification was introduced to simplify the fault injection procedure.

11.3 Fault Injection Environment

For replicating the typical effects of space radiation environment on electronics [8], a simulation-based fault injection technique was chosen. This technique allows full access to the entire processor without any architectural modifications. One of the main disadvantages of this technique is that, being simulation based, required a long time to run compared to execution on hardware emulators such as the FPGA based ones [9]. This simulation-based strategy is based on TCL (Tool Command Language) scripts, that allow the manipulation of signals for fault injection and observe fault effects. The HDL (Hardware description language) simulator used for the system simulation and to run TCL scripts was the Modelsim [10] from Mentor Graphics.

11.3.1 Fault Model

The used fault model is based on SEU occurrence in sequential logic blocks. In each simulation, a single fault is injected to cause a bit flip inside the chosen sequential block. Other effects could be Multiple Bit Upset (MBUs), Multi Cell Upsets Single Event Latchups, but they are out of the scope of this study.

11.3.2 Simulation Procedure

The first step of the procedure, a golden simulation, with no injected fault , is performed to obtain the reference data to be used for the detection of mismatches caused by fault injections. The Riscy core fault injection is performed for all sequential subsystems. For each sequential subsystem, 1000 simulations are performed, 1 fault injected per simulation run. The following steps [11] summarize the tasks executed for each simulation:

(1) Selection of a flip-flop, in a certain sequential subsystem, where the fault will be injected. This is done selecting, in a random way, from a list that contains all signals, in the VHDL code, which implement registers. Each signal corresponds to each bit of the register.
(2) Selection of a random instant when the fault will be injected. In order to avoid a fault during the logging process, the fault is injected before the reporting process.
(3) Simulation runs until the chosen injection instant.
(4) Injection of the fault by forcing a bit flip in the target sequential element.
(5) Simulation runs until the end of the algorithmic benchmark.
(6) Making a copy of the register file content.
(7) Storing the print out of the program results.

If exceptions are generated during the execution, they are stored in a file and whether the core doesn't respond after a threshold time a relative log is generated.

11.3.3 Fault Effects Classification

Data obtained during the simulation campaign are used to classify fault effects that can be summarized in five categories [11] that are listed below:

- *No Effect*—The simulation finishes obtaining the correct result from the program and the content of the register file is equal to the reference one.
- *Latent*—The simulation finishes obtaining the right result, but the content of the register file is not equal to the reference.
- *Wrong result*—The system has a failure and the simulation finishes obtaining the wrong program result.
- *Timed out*—The simulation takes an abnormal amount of time to finish the program execution compared to the reference.
- *Exceptions*—The core generates exceptions during the simulation.

Latent errors potentially can propagate and lead to a system failure in the future, but these errors may also be masked by the normal core functioning.

11.4 Chosen Benchmarks

Benchmarks usually are used to evaluate performances of microcontrollers, microprocessors and computing systems in general. In this characterization campaign, they are used because they offer a generic workload that can cover almost all operations that a core can execute. Benchmarks perform a large number of different operations such as logic, numeric and string operations. The chosen benchmarks, for this campaign, are Dhrystone and CoreMark, described below.

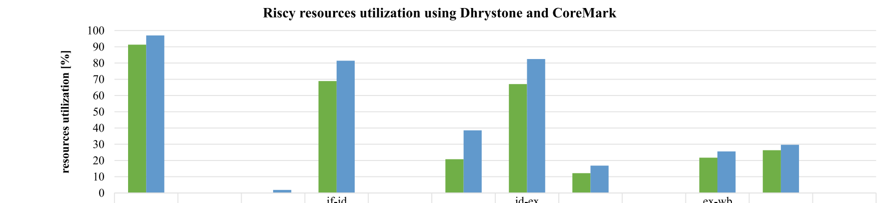
- *Dhrystone benchmark* provides a measure of integer performance and no floating point instructions. Here, it has been used the 2.1 C version that avoids over optimization problems [12] encountered with the first version. Dhrystone benchmark workload [13] can be categorized in: ALU operations for 42% of the instructions; 20% load instructions; 15% store instructions; 21% branch instructions; 2% shifting instructions.
- *CoreMark benchmark* from EEMBC [14] (Embedded Microprocessor Benchmark Consortium) is specifically designed for embedded systems and it can be used to measure microcontrollers and microprocessors performances. It is considered the next version of Dhrystone [14]. It implements numbers of algorithms like find, sort, matrix manipulation, state machine and crc. The crc is used both to provide a typical workload for an embedded application and to check the results of the operations. This benchmark is designed to be independent of the compiler optimization options and this is one of the improvements respect to Dhrystone [14] .

11.5 Simulation Results

This section presents and discusses the results of the fault injection campaign and the measure of the utilization of sequential modules. These simulations are useful for vulnerability estimation.

11.5.1 Resources Utilization Using Dhrystone and Coremark

The resources utilization has been measured using a Modelsim simulation running a TCL script. Coremark is configured to perform 1 cycle while Dhrystone 1000 cycles. In this study, the focus is over the sequential parts inside modules that are the target



	prefetch unit	if top level registers	hw loop	if-id pipeline register	controller	register file	id-ex pipeline register	control-state registers	multiplier registers	ex-wb pipeline register	load store unit	interrupt controller
🟩 r.u. - Dhrystone [%]	91.31	0.00013	0.00	68.89	0.00150	20.77	67.04	12.18	0.00	21.70	26.31	0.00
🟦 r.u. - CoreMark [%]	97.01	0.09246	1.88	81.45	0.00179	38.56	82.49	16.81	0.00	25.58	29.67	0.00

Fig. 11.3 RISC-V Riscy sequential resources utilization

of the characterization through fault injection. A simulation both for Dhrystone and CoreMark is performed obtaining the resources utilization for the entire simulation time. In this simulation, the measure of the resource utilization is performed counting how many times the value, stored in a given register, changes from the beginning to the end of the simulation. This was made using a TCL script that runs in Modelsim.

The workload is similar for both simulations, as shown in Fig. 11.3, for Dhrystone and CoreMark. The most used modules are the prefetch unit, the instruction fetch-instruction decode pipeline register and the instruction decode-execute pipeline register. There are modules that are never used during the benchmark execution like the hardware loop for Dhrystone and the multiplier registers and the interrupt controller for both benchmarks (in the run simulations).

11.5.2 Characterization Through Fault Injection Using Dhrystone Benchmark as Workload

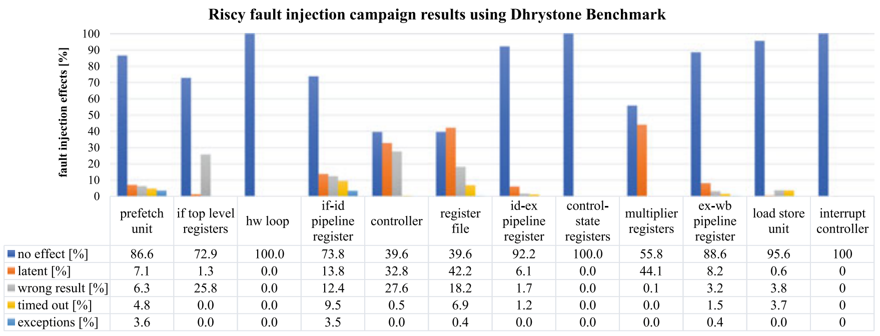
In Fig. 11.4 are shown the results of the simulation campaign. As mentioned above, for each microcontroller block, 1000 simulations were performed, with a fault has been injected for each run.

This procedure is repeated for each block, obtaining the results showed in the plot.

The graph shows that the most critical sequential modules are the controller and the register file, with injected faults that cause a large number of latent errors and wrong results. Despite the fact that the controller is used with a lower frequency than the register file, it causes a large number of failures when it undergoes to fault injection.

Exceptions are generated from modules inside the instruction fetch stage, in the instruction decoder stage and in the execution-write back pipeline register. In this core, exceptions are used to report a wrong instruction operation code.

The hardware loop module, the interrupt controller and the control-state registers don’t cause any failure when faults are injected.



	prefetch unit	if top level registers	hw loop	if-id pipeline register	controller	register file	id-ex pipeline register	control-state registers	multiplier registers	ex-wb pipeline register	load store unit	interrupt controller
🟦 no effect [%]	86.6	72.9	100.0	73.8	39.6	39.6	92.2	100.0	55.8	88.6	95.6	100
🟧 latent [%]	7.1	1.3	0.0	13.8	32.8	42.2	6.1	0.0	44.1	8.2	0.6	0
⬜ wrong result [%]	6.3	25.8	0.0	12.4	27.6	18.2	1.7	0.0	0.1	3.2	3.8	0
🟨 timed out [%]	4.8	0.0	0.0	9.5	0.5	6.9	1.2	0.0	0.0	1.5	3.7	0
🟦 exceptions [%]	3.6	0.0	0.0	3.5	0.0	0.4	0.0	0.0	0.0	0.4	0.0	0

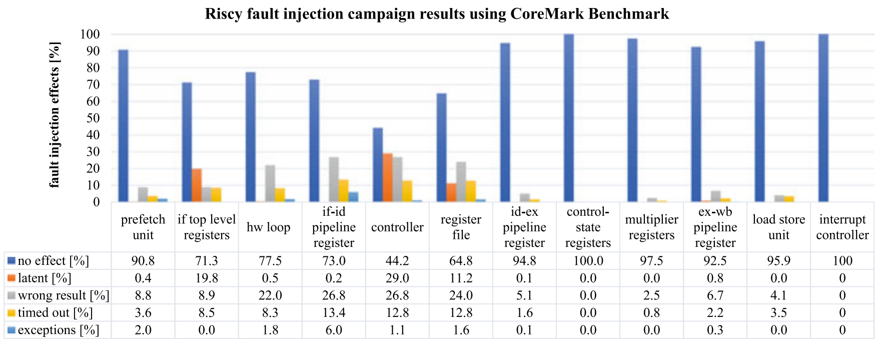
Fig. 11.4 Fault injection campaign results using Dhrystone Benchmark

A particular behavior to be noticed is about the multiplier module. It is never used during the program execution but it causes failures when faults are injected, because due to its implementation, faults can propagate in other modules.

11.5.3 Characterization Through Fault Injection Using CoreMark Benchmark as Workload

Figure 11.5 shows the results of the simulation campaign with the same procedure used above.

Like in the analysis concerning the other benchmark, it can be noticed that the most critical modules are the controller and the register file, which present a large number of latent errors and wrong results. The controller is again accessed with lower frequency w.r.t. the register file, but it displays high vulnerability.



	prefetch unit	if top level registers	hw loop	if-id pipeline register	controller	register file	id-ex pipeline register	control-state registers	multiplier registers	ex-wb pipeline register	load store unit	interrupt controller
🟦 no effect [%]	90.8	71.3	77.5	73.0	44.2	64.8	94.8	100.0	97.5	92.5	95.9	100
🟧 latent [%]	0.4	19.8	0.5	0.2	29.0	11.2	0.1	0.0	0.0	0.8	0.0	0
⬜ wrong result [%]	8.8	8.9	22.0	26.8	26.8	24.0	5.1	0.0	2.5	6.7	4.1	0
🟨 timed out [%]	3.6	8.5	8.3	13.4	12.8	12.8	1.6	0.0	0.8	2.2	3.5	0
🟦 exceptions [%]	2.0	0.0	1.8	6.0	1.1	1.6	0.1	0.0	0.0	0.3	0.0	0

Fig. 11.5 Fault injection campaign results using CoreMark Benchmark

Exceptions are generated from modules inside the instruction fetch stage, in the instruction decoder stage and in the execution-write back pipeline register. In this core, exceptions are used to report a wrong operation code of the instruction.

The interrupt controller and the control-state registers don't cause any failure when faults are injected. In this case, CoreMark stimulates the usage of the hardware loop module and we noticed system failures caused by faults injected inside this module.

In this case, the multiplier module shows less latent errors respect the Dhrystone workload.

11.6 Vulnerability Estimation

Results obtained from the fault injection campaigns present similar trends for the two workloads. In this section, we try to calculate the vulnerability of each block of the RISC-V Riscy core, by introducing the following equation:

$$v = \begin{cases} \frac{f}{u} \times c, & \text{if } u > 0 \\ 0, & \text{otherwise} \end{cases} \tag{11.1}$$

- v resource vulnerability.
- f failure rate. It is equal to the number of wrong results normalized to the number of simulations;
- u resource utilization. For each module, it is equal to the number of clock cycles of activity over the total number of clock cycles.
- c normalization constant equal to 100.

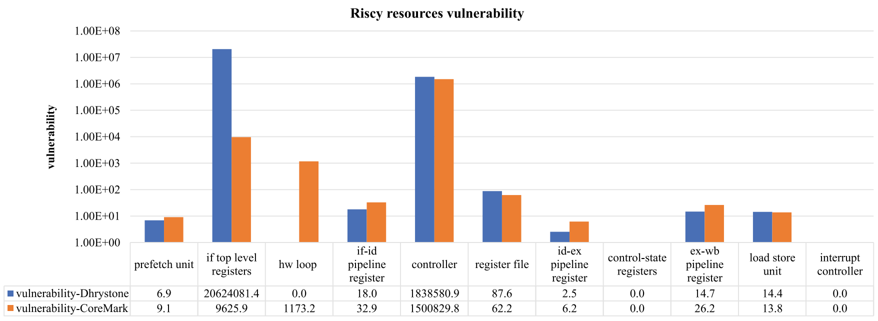
This approach allows to extrapolate a general evaluation of the block vulnerability that is independent of the used benchmark algorithm. Since it is based on the correlation between the amount of detected failures and the actual use of the blocks of the microcontroller. Figure 11.6 shows the results of the vulnerability associated with each block.

The plot uses a logarithmic axis to easily visualize the results.

It can be noticed that there is a correlation between the vulnerability calculated from both campaigns. The subsystems that show a high vulnerability are the instruction fetch registers, the controller and the register file.

Lower but significant vulnerability magnitude is showed for the prefetch unit, instruction fetch-instruction decode pipeline register, the instruction decode-execution pipeline register, ex-wb pipeline register and the load store unit.

Vulnerability is normalized to the resource utilization for the chosen benchmark. There is information about the vulnerability for the hardware loop only from the campaign using CoreMark. For Dhrystone this module wasn't used and it didn't generate system failures.



	prefetch unit	if top level registers	hw loop	if-id pipeline register	controller	register file	id-ex pipeline register	control-state registers	ex-wb pipeline register	load store unit	interrupt controller
■ vulnerability-Dhrystone	6.9	20624081.4	0.0	18.0	1838580.9	87.6	2.5	0.0	14.7	14.4	0.0
■ vulnerability-CoreMark	9.1	9625.9	1173.2	32.9	1500829.8	62.2	6.2	0.0	26.2	13.8	0.0

Fig. 11.6 RISC-V Riscy sequential resources vulnerability

The limitation of the adopted vulnerability model is due to the occurrence of system failures that are observed also in blocks that are not supposed to be used by the algorithm. This is the case for the multiplier registers that generate system failures during the campaign, whether this block is supposed to be never used. For this reason, the multiplier module vulnerability is not shown in Fig. 11.6. These occurrences, which are treated as exceptions, are caused by the propagation of a fault injected in other blocks. These events depend on how the module is designed and can be avoided with modifications in the system design.

11.7 Conclusion

This paper introduces a detailed analysis of the SEU effects in the RISC-V Riscy core. The results are based on data obtained from the fault injection campaigns based on simulation-based injection technique. The workload is similar both for Dhrystone and Coremark benchmarks as representative of generic applications.

From simulation results, showed in Fig. 11.4 for Dhrystone workload and in Fig. 11.5 for CoreMark workload, the most critical sequential modules are the controller and the register file. Despite the fact that the controller is used with lower frequency than the register file, it causes a large number of failures when it undergoes to fault injection. Exceptions are caused by faults injected in modules inside the instruction fetch stage, in the instruction decoder stage and in the execution-write back pipeline register. CoreMark stimulates the usage of the hardware loop module and we noticed a relevant system failures caused by faults injected inside this module.

The interrupt controller and the control-state registers don't cause any failure when faults are injected.

The most used resources are the prefetch unit, the instruction fetch-instruction decode pipeline register and the instruction decode-execute pipeline register. There are modules that are never used during the benchmark execution like the hardware loop for Dhrystone and the multiplier registers and the interrupt controller for both benchmarks (in the run simulations).

From the study of vulnerability, showed in Fig. 11.6, the most critical module result to be the controller module.

The Vulnerability can be used to estimate, for each module, the system failure rate when executing other software. This can be done simply making the product between the tabled vulnerability values and the utilization value measured for the given application. These represents an important information for the design of fault-tolerant Risc-V core, since it can be used to evaluate the best redundancy techniques in terms of time usage and hardening impact for each composing block, on the base of its vulnerability.

References

1. Calligaro C, Gatti U (2018) Rad-hard semiconductor memories. Series in Electronic materials and devices
2. Di Mascio S, Menicucci A, Furano G, Monteleone C, Ottavi M (2019) The case for RISC-V in space. In: Saponara S, De Gloria A (eds) Applications in electronics pervading industry, environment and society. Springer International Publishing, Cham, pp 319–325
3. "About the RISC-V Foundation," [Online]. Available: https://riscv.org/risc-v-foundation/. Accessed May 24, 2019
4. Dilillo L, Tsiligiannis G, Gupta V, Bosser A, Saign F, Wrobel F (2016) Soft errors in commercial off-the-shelf static random access memories. J Semicond Sci Technol 32
5. Pulp-Platform (June 2017) "Project info," [Online]. Available: https://pulp-platform.org/projectinfo.html
6. Pulp-Platform (2017 June) "PULPino: Datasheet," PULPino: Datasheet
7. "GNU RISC-V Toolchain," [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain. Accessed May 09, 2019
8. Gupta V, Bosser A, Wrobel F, Saigne F, Dusseau L, Zadeh A, Dilillo L (2016) MTCube project: SEE ground-test results and in-orbit error rate prediction. The 4S symposium, small satellites systems and services symposium, Valletta, Malta
9. Cho H (2018) Impact of microarchitectural differences of RISC-V processor cores on soft error effects. IEEE Access, 6:41302–41313
10. "Mentor Graphics," [Online]. Available: https://www.mentor.com/company/higher_ed/modelsim-student-edition. Accessed May 09, 2019
11. Travessini R, Villa PRC, Vargas FL, Bezerra EA (2018) Processor core profiling for SEU effect analysis. In: Test symposium (LATS)
12. "Roy Longbottom's PC Benchmark collection," [Online]. Available: http://www.roylongbottom.org.uk/dhrystoneresults.htm. Accessed May 09, 2019
13. Price WJ (1989) A benchmark tutorial. IEEE Micro, pp 28–43
14. "Embedded microprocessor benchmark consortium," [Online]. Available: https://www.eembc.org/coremark/. Accessed May 16, 2019