UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
NÍVEL MESTRADO PROFISSIONAL

RAFAEL DE FIGUEREDO VIANA

DESIGN AND SIMULATION OF A RISC-V DUAL-CORE LOCKSTEP FOR FAULT TOLERANT SYSTEMS

São Leopoldo
2020

Rafael de Figueredo Viana

Design and Simulation of a RISC-V Dual-Core Lockstep for Fault Tolerant Systems

Dissertation presented as a partial requirement to obtain the title of Master in Electrical Engineering, by the Programa de Pós-Graduação em Engenharia Elétrica da Universidade do Vale do Rio dos Sinos - UNISINOS.

Orientador: Dr. Marcio Rosa da Silva

Co-orientador: Dr. Jorge Luis Victoria Barbosa

São Leopoldo

2020

V614d Viana, Rafael de Figueredo.
Design and simulation of a RISC-V dual-core lockstep for fault tolerant systems / Rafael de Figueredo Viana. – 2020.
61 f. : il. ; 30 cm.

Dissertação (mestrado) – Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Engenharia Elétrica, 2020.
"Orientador: Dr. Marcio Rosa da Silva
Co-orientador: Dr. Jorge Luis Victoria Barbosa."

1. Lockstep. 2. Tolerância a falhas. 3. Injeção de falhas. 4. RISC-V. 5. Dual-core. I. Título.

CDU 621.3

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecária: Amanda Schuster – CRB 10/2517)

ACKNOWLEDGMENT

The development of this work had the help of several people, of which I would like to thank. The supervising teachers, which helped me through the very beginning, the Professor Rodrigo Figueiredo, for showing me functional safety and how you can do so many things with a new idea.

To all my friends, who supported me through the challenges of life, and especially to Rafa, Vilson, and Henrique, a family that I met in this new life stage of mine. I would like to give special thanks to Anderson and Eduardo, who helped me greatly in the technical aspects of this work.

I also want to thanks Amadeu Susin and all folks from LaPSI, who gave me the opportunity to start working with interesting people in a larger project, and all folks from ittChip who received me with great ease.

RESUMO

Processadores embarcados são utilizados cada vez mais na indústria e em aplicações civis, como em dispositivos de aplicações críticas em segurança. O parâmetro crítico dos processadores, anteriormente performance, foi substituído pela necessidade da garantia de confiabilidade do sistema. Esta mudança de paradigma acarreta na utilização de técnicas para desenvolvimento de dispositivos tolerantes a falhas. Aplicações aeroespaciais e, mais recentemente, automotivas, são mais suscetíveis a falhas causadas pela incidência de radiação nos circuitos integrados que compõem os sistemas, devido à redução do tamanho do transistor e aumento da complexidade dos dispositivos. Neste contexto, o uso de FPGA (do inglês *Field Programmable Gate Array*) é atraente à indústria para implementação de sistemas seguros, devido a versatilidade e customização de designs nos dispositivos. Porém FPGA resistentes à radiação possuem alto custo de aquisição, além de serem desenvolvidas com tecnologia de circuitos integrados atrasada em relação a FPGA COTS (*Commercial Off The Shelf*). A fim de aumentar a confiabilidade e segurança de sistemas implementados em FPGA COTS, este trabalho implementa uma arquitetura de *Lockstep dual core* (do termo inglês *Dual Core Lockstep* - DCLS) para processadores de arquitetura *open-source* RISC-V, utilizando o core RI5CY. Acreditamos que este é o primeiro trabalho que implementa uma arquitetura DCLS com CPUs RISC-V, executa uma rotina de injeção de falhas via software e avalia o overhead em software e hardware. Um framework de injeção de falha é proposto e implementado utilizando uma ferramenta aberta de simulação. O sistema é implementado em FPGA e o overhead em hardware do sistema é pequeno, chegando a 5.18% de utilização de área comparado com a área utilizada por um único *core*. O sistema alcança uma redução de frequência de *clock* de 18,5%, ao ser implementado em uma Kintex KC705. Os resultados da injeção de falhas indicam que o sistema é eficaz na detecção de falhas nas saídas de cores, onde todos os erros visíveis foram detectados. Os testes de injeção de falha mostram a discrepância entre a injeção de falha transitória e permanente no *Design Under Test* - DUT devido a diferença de erros visíveis.

Palavras-Chave: *Lockstep*, tolerância a falhas, injeção de falhas, RISC-V, dual-core, processadores embarcados, FPGA.

ABSTRACT

Embedded processors are increasingly being used in every industry and consumer segment, including critical-safety applications. The critical parameter of the processors, previously performance, was replaced by the need to guarantee the reliability of the system. This paradigm shift leads to the use of techniques for the development of fault-tolerant devices. Aerospace and, more recently, automotive applications are more susceptible to failures caused by the incidence of radiation in the integrated circuits that make up the systems, due to the reduction in the size of the transistor and the increase in the complexity of the devices. In this context, the use of FPGA (Field Programmable Gate Array) is attractive to the industry for implementing secure systems, due to the versatility and customization of designs on the devices. However, radiation-resistant FPGA has a high acquisition cost, in addition to being developed with legacy integrated circuit technology if compared with FPGA COTS (Commercial Off the Shelf). To increase the reliability and security of systems implemented in FPGA COTS, this work implements a dual-core Lockstep (DCLS) system for open-source processors architecture RISC-V, using the RI5CY core. We believe that this is the first work that implements a DCLS architecture with RISC-V cores, performs a fault injection routine via software, and evaluates its hardware and software overhead. A fault injection framework is proposed and implemented using an open-source simulation tool. The system is implemented in FPGA and the hardware overhead is small, reaching just over 5.18% compared to a single RI5CY core. The maximum clock frequency reduction achieved by the system implemented in a Xilinx Kintex KC705 reached 18.5%. Fault injection results indicate that the system is effective in detecting faults at the outputs of colors, where all visible errors were detected. Fault injection tests shows the discrepancy between transient and permanent fault injection in the Design Under Test due to the difference between visible errors.

Keywords: Lockstep, fault tolerance, fault injection, RISC-V, dual-core, embedded processors, FPGA.

FIGURE LIST

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ABS	Anti-lock Breaking System
ASIC	Application Specific Integrated Circuit
BIST	Built In Self Test
IC	Integrated Circuit
CEU	Code Emulated Upset
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
CRAM	Configuration Random Acess Memory
DUT	Device Under Test
DMR	Double Modular Redundancy
DSP	Digital Signal Processing
ECC	Error Correction Code
ECU	Electronic Control Unit
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
ICAP	Internal Configuration Access Port
IIR	Infinite Impulse Response
IP	Intellectual Property
LUT	Look Up Table
MMR	Multi Modular Redundancy
RAM	Random Acess Memory
RTL	Register Transfer Level
SEE	Single Event Effect
SEL	Single Event Latchup
SEU	Single Event Upset
SIL	Safety Integrity Level
SIHFT	Software Implemented Hardware Fault Tolerance
SoC	System on Chip
SRAM	Static Random Acess Memory
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver-Transmitter

CONTENTS

1 INTRODUCTION

Embedded systems with programmable logic devices - Field Programmable Gate Array - FPGA are widely used for several critical safety applications. Tasks involving these systems are becoming increasingly complex and critical in terms of safety, such as medical applications, trains, planes (aircrafts), satellites and aerospace vehicles, etc. In this way, embedded systems play a crucial role in ensuring the integrity of data and human security.

These devices' hardware is susceptible to the occurrence of failures due to the incidence of radiation or other environmental factors, such as high voltage spikes, abrupt temperature changes or electromagnetic interference, which may generate logic and control errors in the device. The occurrence of errors caused by radiation is called Single-Event-Effect - SEE, which can cause irreparable errors in the logic implemented in the FPGA. One of the challenges in implementing these secure systems is ensuring functional safety using Commercial-of-The-Shelf - COTS components. Although these components do not have radiation tolerance levels suitable for the use of safe systems in applications exposed to harsh environments, they are cheaper and deliver greater performance compared to hardened devices.

One of the key factors to ensure safety in integrated circuits for critical applications is the ability to detect errors during operations, which can cause catastrophic losses depending on the application used. Central processing units - CPUs are sensitive elements within the system, since they are composed of various elements of memory and control, primarily made of sequential logics, which are more susceptible to failures than combinational ones (Iturbe; Venu; Ozer, 2016). Therefore, there is a need to implement devices capable of detecting failures during the operation in CPUs for secure applications in programmable logic, due to the cost of implementation and flexibility of integration with other systems.

Solutions implemented at the architectural level are used to enable the usage of unsafe cores in security-critical applications (VENU; OZER; ROBINSON, 2016), allowing for a quick integration of hard-cores CPUs with the desired solution in the safe system. These solutions check the output data of cores, as they are the only accessible interfaces within hard-cores. Soft-core CPUs enable wider system customization approaches since it is possible to modify the intellectual property (IP) design with more flexibility, allowing more complex and fully customized designs. This flexibility can be explored in fault-tolerance techniques, where multiple fault-tolerance methodologies can be combined and result in a system with a greater safety level.

1.1 Main Objective and Motivation

A classical fault tolerance technique is the replication of the target hardware, where the same application is executed in multiple instances at the same time, and the outputs are compared to check for divergences during the operation. This technique is called lockstep, where the number of replication is defined by the target application specifications and complexity. In this work, a dual-core lockstep (DCLS) design is presented as fault-tolerant architecture, to increase system reliability and enable complex applications in a harsh environment. Its main objective is to enable COTS FPGA utilization in critical-safe missions, while keeping the area and clock timing overhead at low overhead factor. An open-source

soft-core IP is used as a target hardware replication module. We believe that this is the first work that approaches DCLS architecture with open-source CPU ISA RISC-V as a fault-tolerance design technique, and in this work, the proposed design architecture is shown, both in hardware and software approaches overview. A fault injection methodology is proposed, to evaluate the design's fault-tolerance capability with applications used in the literature. An implementation analysis is also shown, along with the fault injection analysis and hardware and software overhead overview.

1.2 Work Structure

This work is organized as follows: in chapter 2, a background review is shown, where key concepts regarding system design and functional safety are approached. In chapter 3, the literature is revised, showing works that implements fault tolerance design solutions. Chapter 4 shows the proposed dual-core lockstep solution, where the proposed hardware architecture and functional specifications are described. In chapter 5 the fault injection methodology is shown, used to evaluate the proposed DCLS design, explaining the fault injection routines and error detection rules. Chapter 6 presents the results obtained from the proposed fault injection methodology, as well as the implementation and the system reliability analysis. Chapter 7 concludes the dissertation and future works proposals are presented.

2 BACKGROUND REVIEW

This chapter reviews themes related to the dissertation, regarding the central theme of the work. Concepts on the use of embedded systems in industry will be reviewed, as well as the application of devices with programmable logic. Radiation effects on integrated circuits, impact of these phenomena on systems with safety criticality, and fault-tolerant systems design will be discussed. Common failures in CPUs and their effects in applications will be presented.

2.1 Embedded Systems

Advances in processor architectures made possible its usage in applications that require high computational performance and low cost. The available architectures range from processors with a single core to multiple cores within the same chip. With the increase in performance and the amount of resources available in these processors, safe-critical applications could be implemented using these devices, given that more robust programs with a greater number of security elements can be implemented in the systems without compromising time restrictions (FLORIDIA; SANCHEZ, 2018).

The implementation approach of these processors depends on the specifications of the system to be developed and the design restrictions, as processors manufactured on a chip (hard-core), or intellectual property IP (soft-core) can be used, by implementing in FPGA. Although the CPU's hard-core lead to lower integration and testing costs in the system (given that the device meets the manufacturer's specifications), they do not have customization flexibility for specific implementations that require changes in the core, such as modifications to achieve greater system safety, for example. In terms of performance in soft-core, it depends on the technological node of the FPGA to be implemented, which may result in lower performance and higher energy consumption compared to hard-core processors.

2.2 Programmable Logic Devices

As the industry needed to deploy more complex and higher performance systems, FPGA appeared as solutions for integration and flexibility, as they have a lower development cost compared with application-specific integrated circuits - ASIC. Modern FPGAs are composed of programmable logic, as well as embedded components, such as hard-core CPUs, high capacity memories, dedicated processing blocks, bus structures inside the chip, analog interfaces and peripherals for specific applications.

FPGAs have high flexibility, the ability to integrate into complexer systems and allow for rapid project development. However, FPGA Commercial-Off-The-Shelf - COTS, which have higher performance and competitive prices, are not developed for fault-tolerant applications caused by radiation, such as radiation-resistant (rad-hardened) FPGAS. FPGAs use Static Random Access Memories - SRAM for configuring the programmable logic, and are more vulnerable to faults caused by radiation (ITURBE et al., 2016). This factor limits the use of FPGA in applications with safety criticality.

Usually, rad-hardened devices use proprietary technologies to manufacture the devices, such as radiation-resistant standard cells and specialized architectures to mitigate faults. Companies like Xilinx have specialized FPGAs in environments with a high level of radiation (SPACE-GRADE RAD - HARD VIRTEX-5QV FPGA), for missions in space and in applications that require a high level of availability. These devices have a high cost of implementation, being an impacting factor in the decision of projects with safety criticality, besides being some generations outdated with the state-of-the-art technology of COTS devices (ASADI; TAHOORI, 2005). Due to these factors, the industry has been looking for solutions that allow the usage of COTS devices in fault-tolerant systems, to increase the performance and flexibility of these devices (TAMBARA et al., 2015). The next section reviews radiation effects into embedded systems using FPGA.

2.3 Radiation Issues in Integrated Circuits

The incidence of radioactive particles in integrated circuits leads to unwanted behavior on these devices. In this section, concepts about the causes of radiation in these devices will be reviewed, in addition to their effects.

2.3.1 Causes

Due to the scalability of integrated circuits in recent years, they are more susceptible to faults caused by radiation. The usage of more complex designs in sensitive applications, such as in the aerospace and avionics industry resulted in an increase in the occurrence of faults in the integrated circuits used in these systems (NORMAND, 1996). The effect of ionizing particles on integrated circuits was first discovered in the 60 (Wallmark; Marcus, 1962) decade, being then an important study area for reliability engineering and fault tolerance. Since then, techniques and methods have been developed to mitigate the unwanted effects of radiation on integrated circuits, since these systems have become less reliable and could cause billion-dollar losses to the industry.

The radiation effects present in integrated circuits (ICs) implemented in space can also be observed in terrestrial applications, caused by the incidence of radioactive particles in ICs inciding from space, due to the exposure of radioactive materials or ionized environments (Dodd; Massengill, 2003). Shortly after the discovery of the effects of radiation on integrated circuits implemented in space, several techniques were developed to reduce the impact of radiation on these systems, to increase their reliability, safety and availability. The vulnerability of ICs has become a mainstream product reliability metric across the semiconductor industry, due to the scalability of the number of transistors integrated in the same CI and the need for system availability (SEMICONDUCTOR. . . , 1999). The incidence of ions in ICs generates a nuclear interaction between the particle and the silicon atoms, resulting in the creation of a pulse of transient current, which can be interpreted as altering the logical state of the device and eventually causing system failures (CHIELLE et al., 2016).

2.3.2 Radiation Effect on ICs

The exposure of ICs into radioactive environments causes the bombardment of ionizing particles in these devices. The action of energetic particles on silicon results in the transfer of energy in its tracks. The absorption of this energy creates a charge that can cause a spurious current pulse in the affected device, causing a fault. The current caused by the energy charge effect is defined as Single Event Effect - SEE (BAUMANN, 2005). The main effects of the SEE can be defined in two groups, Single Event Upset - SEU and Single Event Latchup - SEL.

SEU represents the logical change of a memory element value, which can result in unexpected behavior of the device (ASADI; TAHOORI, 2005), from a bit-flip in registers to errors in the program execution control, due to the change in the executed instruction by the CPU. SELs are short-circuits between the power signal and the ground, caused by the effect of SEE on the parasitic thyristors present in CMOS. These events generate hardware faults known as stuck-at, as the signal value remains constant at logic level 1 or 0 (VELAZCO; FAURE, 2007).

In logic circuits, memory cells are more susceptible to logic inversion due to the effects of radiation. Random Access Memory - RAM are more sensitive devices due to the higher density and number of registers, being a crucial factor in the usage of FPGA for systems implementation, as these devices uses Configuration RAM - CRAM as to implement the target design.

CRAMs are used to configure the logical elements - LE available in the FPGAs, to implement the programmed logic. These CRAMs define the signal routing paths, the values stored by the Look Up Table - LUTs, DSP elements, and user RAMs. The impact of a particle on a CRAM causes a permanent error in the design mapped in the FPGA (ASADI; TAHOORI, 2005). This type of fault is called a permanent fault since the only way to eliminate it is by reconfiguring the FPGA. When a SEU occurs at a user registry, the error is eliminated after several interactions. This type of fault is called a transient fault.

Transient faults can cause soft-errors in the system, which in turn are naturally eliminated from the device over time. Permanent faults cause hard-errors in the system, which need more specialized analysis to determine if the error is recoverable or if the device has suffered a fatal error (OZER et al., 2018). The propagation time of faults within the hardware depends on the type of fault (transient or permanent), and the region in which the fault originates. Depending on the fault type, it can take hundreds of thousands of clock cycles to be detected on the system output, being a critical factor for the development of safe systems using ICs. The next section introduces the concept of functional safety, explaining the central idea about the technique, and addressing its relationship with ICs project.

2.4 Functional Safety

The usage of electronic devices in safety-critical applications creates the need to ensure the reliability of these systems. Functional safety is used as a basis for the development of these systems, defined to create rules and metrics to guarantee the absence of unacceptable risks during the operation of the devices (CHONNAD; IACOB; LITOVTCHENKO, 2018). In

this section, definitions on functional safety and safety requirements in integrated circuits are reviewed.

2.4.1 Definitions

Due to the susceptibility of IC faults, techniques were developed to increase the reliability of the systems. Fault tolerance has become an important criterion in the industry, due to the increased use of integrated circuits in safety-critical applications (automotive, avionics, military). Failure of these systems can result in injury to people, equipment and the environment (WALKINGTON; SUGAVANAM; NUNNS, 2013).

There is also a need for applications that require a high level of reliability that pose no risk to humans in the event of failure, but to economic factors, such as high availability servers. To reduce risks and create standards in the industry, safety integrity levels - SIL were created, which are used to measure the level of risk for a safety measure used by the system. SIL levels are defined according to fault coverage within a secure system. Table 1 shows the statistical definitions for the different levels of security integrity, with the security level defined based on the quantification of the probability of failures and risk reduction of the device (IEC, 2020).

Table 1: Safety Integrity level Definitions.

Safety Integrity Level	Safety Level	Probability of Failure on Demand	Risk Reduction Factor
SIL 4	>99.99%	0.001% to 0.01%	100.000 to 10.000
SIL 3	99,9% to 99.99%	0.01% to 0.1%	10.000 to 1.000
SIL 2	99% to 99.9%	0.1% to 1%	1.000 a 100
SIL 1	90% to 99%	1% to 10%	100 a 10

Source: From the author.

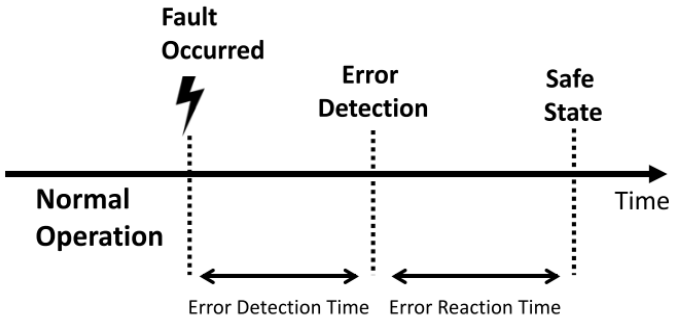
There are SIL definitions for different critical applications (automotive - ASIL, avionics - DAL, rail - SIL, etc.), each having its specifications and functional safety requirements. These definitions classify systems into levels of criticality concerning the potential damage they could cause in the event of failure (CHONNAD; IACOB; LITOVTCHENKO, 2018).

2.4.2 Security Requirements for Error Detection

Any safety mechanism in a critical system must ensure that it reaches a safe state to prevent risks in the event of error detection. More complex systems require greater effort to achieve the desired SIL certification. When a fault manifests as an error, the security mechanism must detect it and report the occurrence to the system. The time interval between the fault occurring and the error detection is defined as the error detection time. Once the error is detected, the security mechanism must ensure that the system reaches a safe state before risks arise. The error reaction time is defined as the time interval between error detection and

change to a safe state, being used as an important metric in critical security systems (OZER et al., 2018). Figure 1 shows the timeline between events in the event of a fault.

Figure 1: Error time reaction.



Source: (OZER et al., 2018).

The safe state must be reached within the system's safe time limit. Delaying the time limit can be fatal, depending on the application. Therefore, the reaction time to errors is defined based on the system's ability to act in the worst case of error, and is a critical parameter of the system, and cannot be violated. However, any reduction in the error reaction time is safe and increases the availability of the system (WALKINGTON; SUGAVANAM; NUNNS, 2013). An example of a system with SIL is the Anti-Lock Braking System - ABS, which uses Fuzzy logic and PID controllers implemented an Electronic Control Unit (ECU). These devices have strict functional safety requirements, as a possible system failure can be fatal. The next section shows fault tolerance design techniques in ICs, used to increase the safety factor of a system (SABBELLA; ARUNACHALAM, 2019).

2.5 Fault-Tolerant Integrated Circuit Design

To guarantee the desired SIL in safe embedded systems, techniques have been developed to ensure that the system reaches the safe state regardless of the type of error that occurs during its operation (OZER et al., 2018). The techniques used to develop safe embedded systems are based on redundancies of software and hardware. The redundancy of hardware implies a higher cost of implementation, due to the overhead necessary for the implementation of these systems, however, they present a higher error detection rate. The technique is transparent to the application, not presenting overhead in software. In the software redundancy approach is expected a lower cost of implementation concerning hardware overhead, however, they may lead to a great system performance penalty.

The strategy to be implemented in the system must take into account the effects of errors in the system, by identifying the most vulnerable components. Based on the information collected, it is possible to define the appropriate methodology to protect the system with the highest critical factor. Solutions based on software and hardware are implemented to determine the occurrence of errors in hardware, and have different problems and specifications. As CPUs are complex circuits, a fault within a can take several clock cycles before manifesting as errors in the device outputs. The time between the occurrence of

the fault and the detection of the error must be minimized to increase the reliability and safety of the device.

In case of fault detection, the secure system can initiate a series of internal tests (Built in Self-Test - BIST) to determine the source of the fault and if possible, correct them. If the error is transient, it is possible to perform the rollback operation of the system to a known state, and restart the application. In the event of permanent fault, the secure system can provide the user or the system manager with a risk warning and initiate a security protocol to ensure the integrity of the system and users. The next section introduces the concept of fault characterization in ICs, used to guide fault-tolerant system designs and techniques.

2.6 Fault Characterization in ICs

To determine the fault tolerance of integrated circuits, it is necessary to apply techniques to predict the fault rate caused by SEUs in these devices. Fault injection is the approach used by researchers and industry to validate the mechanisms for detecting and mitigating failures in the device under test (Device Under Test - DUT), by producing unwanted behaviors. The advancement of semiconductor technology and the increase in the complexity of the devices creates the need to use robust fault injection techniques to guarantee the coverage of the devices to determine their level of security.

Fault injection is the technique for validating dependence on fault tolerance of systems, which consists of controlled experiments in which the behavior of the system is observed under faulty environment (ARLAT, 1990). Fault injections techniques are categorized by (ZIADE; AYOUBI; VELAZCO, 2004):

a) Fault Injection in hardware: Disturbances are inserted into physically implemented devices, with environmental interference (ionizing radiation, electromagnetic fields), through the injection of current and voltage surges in the power distribution tracks of the devices or by application of single values on the IC pins. Additional hardware is used to inject faults in the system hardware. Depending on the faults and their location, fault injections in hardware can be defined as follows:
   - Fault Injection with contact: They are defined as injection with contact, in which the injector has direct physical contact with the system, producing the type of fault desired on the chip;
   - No contact fault injection: The injector has no direct physical contact with the system, using external sources to produce physical disturbances to produce faults in the system under test.

b) Simulation Fault Injection: Faults are injected at the coding level (Verilog, VHDL), making changes in the design of the DUT to determine the fault tolerance of the system. The system's behavior is simulated using specific tools (Modelsim - MentorGraphics, Incisive Simulator - Cadence, etc.) to collect the results obtained by the fault injection, making it possible to analyze the occurrence and distribution of the faults present in the system. This approach is used when only the behavioral model of the system is available.

c) Emulation Fault Injection: FPGA is used to evaluate the design in programmable logic, increasing the speed of the fault simulation. It is possible to study the

behavior of the circuits in the application environment, increasing the effectiveness of the fault tolerance tests of the project. The fault injection device is implemented using programmable logic resources (such as bit inverters or data scramblers), or resources of the FPGA itself, such as primitives that changes implemented logic at run time, such as the internal configuration access port (Internal Configuration Access Port - ICAP) of the Xilinx FPGA devices.

d) Hybrid Fault Injection: Use of mixed techniques for fault injection controlled by software and results analysis via hardware. Hybrid solutions use two or more fault injection techniques to achieve higher levels of DUT excitation, increasing the effectiveness of tests, simplifying parts of the test definition from fault injection via software, in addition to increasing the accuracy from monitoring hardware.

This section reviewed the available fault injection technique types used to perform a failure characterization. The next section explains the faults that may happen in CPUs under harsh environmental conditions.

2.7 Faults in CPUs

As the processor architecture becomes more complex, the susceptibility to errors increases proportionally, requiring more advanced safety mechanisms. The characterization of errors caused by SEU in data elements and control of CPUs is necessary to increase the resilience to errors in these devices. The main effects of soft-errors found in CPU's are presented, based on (CHO, 2018):

a) Silent Data Corruption: The application runs normally without any error indication, however, there are divergences between the faulty program outputs and the error-free program outputs. Due to the complexity of the applications, errors are analyzed at different levels, to facilitate the analysis of error propagation.

b) Unexpected Termination: The device ends the execution of the application indicating the occurrence of an exception in hardware, due to an improper action performed by the program, such as reading protected memory or writing outside the memory region, arithmetic exception or kernel crash.

c) Suspension: The application does not produce any results or forces the application to terminate after the predetermined time limit expires.

Part of the errors that propagate to the CPU outputs do not impact the behavior of the application, due to its intrinsic masking. However, an error can cause incorrect values to be written to an element in memory. If subsequent instructions perform write operations to the corrupt location before the application uses its data, the error will not result in unexpected behavior or erroneous application execution (SAGGESE et al., 2005). The result of the failed application is equivalent to the result of the application without errors, as the error present during the execution of the application was masked or overwritten during the execution.

At the physical architecture level, faults can generate errors in each step of the processor pipeline, due to changes in the flow control, in addition to the incorrect execution of instructions. Faults in the data write/read stage can lead to erroneous data in the application output. The propagation of faults from their source to the processor outputs can take hundreds of cycles of clock, resulting in greater latency for detecting errors in systems that only check the system outputs.

In general, combinational logic is less susceptible to soft-errors (Seifert et al., 2012) compared to sequential logic, making it necessary to use techniques for detecting errors in memory elements. This chapter presented the key concepts about embedded systems, FPGA utilization in harsh environment, system safety, and IC fault characterization. The next chapter presents the fault tolerance enhancements related works, where multiple techniques are shown.

3 RELATED WORKS

The necessity to guarantee the functional safety of electronic circuits leads to the development of techniques for fault detection and validation of safe systems. In this chapter, related works reviews will be presented about fault-tolerant CPU design techniques, and fault injection methodologies used in the literature, to validate the security requirements on these devices.

3.1 Fault Tolerant CPU Design

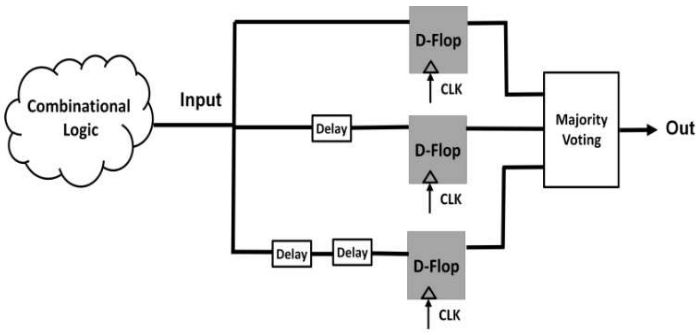
Fault-Tolerant CPU designs are crucial to enable high-performance devices into high risk environments. This section shows the fault tolerance enhancement designs applied in the literature.

3.1.1 Designs Based on Hardware

Fault detection solutions based in hardware are implemented in the product design stage at the physical level, using specific hardware for monitoring the execution of instructions on the processor and memory access, to increase the reliability of the devices and the functional safety factor. Of the techniques used, the most explored in radiation-tolerant devices is the modular redundancy of sensitive elements (usually registers). The levels of redundancy can be double, triple, or multi-modular (Dual Modular Redundancy - DMR, Triple Modular Redundancy - TMR, and Multi Modular Redundancy - MMR, respectively). The redundancy of hardware allows fault detection on similar devices, avoiding critical system failures, and allows for uninterrupted execution (in cases of triple or greater redundancy) until the failure is mitigated or eliminated from the system.

Radiation resistant devices have Register Transfer Level - RTL level redundancy, through the implementation of error detectors, register redundancy, majority voters, and parity checkers (COBHAM, 2019). These additional components result in higher energy consumption, area utilization, and reduced CPU performance (ITURBE et al., 2016). In CPUs, these techniques are applied at multiple levels of abstraction, from RTL (register redundancy) to the architectural level (processor redundancy), where failures are detected by comparing the CPUs outputs at each clock cycle. Hardware redundancies designs are shown in the following. It is possible to obtain greater fault-tolerance factor through the triple redundancy (TMR) of all registers, as demonstrated by (GHAHROODI; OZER; BULL, 2016) in a core ARM CORTEX-R4. In this work, all flip-flops and latches have been replaced with TMR versions in netlist at gate-level. Figure 2 shows the TMR circuit implemented in the processor registers. The clock decreased by 30% compared to the original clock, while the overhead of area and dynamic power increased by 100%.

Figure 2: TMR with flip-flops majority voter.



Source: (GHAHROODI; OZER; BULL, 2016).

In multi-core systems, the design techniques of fault-tolerant systems involve the implementation of redundancies of RAMs, CACHES, controllers, and other elements, in addition to the use of ERROR CORRECTION CODE - ECC, for detecting and mitigating errors on servers with high availability requirements, as demonstrated by (AGGARWAL; RANGANATHAN, 2007). In this work, two-processor clusters are defined for running applications so that possible errors are isolated from the system, ensuring the delivery of the server task.

At the architectural level, one of the solutions used is through the redundancy of CPUs on the same device, in a discreet fashion, or on the same chip (INFINEON, 2012), to increase the reliability of the system. The detection of divergences between redundant CPUs can be performed by comparing the outputs of the devices at each clock, using lockstep structures. These structures will be covered in more detail in the later sections.

3.1.2 Software Based Designs

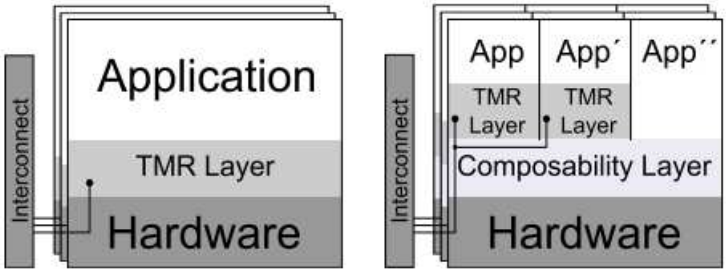
Solutions based on software implies in the redundancy of applications, variables, and memory regions used by the system. These techniques protect the processor in the execution of instructions and the data flow, to guarantee the reliability of the delivered results and detection of execution errors caused by failures in the hardware. These techniques are called Software Implemented Hardware Fault-Tolerant - SIHFT.

These strategies use specific modifications in the source code of the application, with the inclusion of routines that performs memory integrity tests periodically, being possible to obtain the result of the comparison only after the final processing of the variables. Memory signature verification techniques can be used to verify the reliability of program execution. Since the verification occurs in variables that control the execution of the program, these techniques have a higher error detection rate when compared to programs that only verify the program variables (BENSO et al., 2005).

It is possible to develop secure code using source-to-source compilers, as demonstrated by (RENNER, 2003). The author created a compiler capable of generating safe codes, in which variables are kept in memory and verified by monitoring applications, through the insertion of routines in the source code to compare the variables and perform actions in case of error detection.

Another technique used to increase the security of application execution is through the concept of composability, introduced in secure systems by (RESCH; STEININGER; SCHERRER, 2013). The authors introduce the concept of software composibility together with TMR, in which a framework is defined to manage the sets of secure functions, operating in a software triple modular redundancy (TMR) fashion. The control of the execution of the functions and validation of the results of the operations is done by the layer of composibility, being necessary only one instance of hardware for execution of the secure application. Figure 3 shows the architecture in the software proposed in the work.

Figure 3: TMR implementation via software.



Source: (RESCH; STEININGER; SCHERRER, 2013).

In the operating systems safety level, hypervisors are used to manage the interface between the secure application and hardware, where operating systems (OS) run concurrently on the same hardware, so that the existence of the system response checker is transparent to the OS's (BRESSOUD; SCHNEIDER, 1995). Backup's reservations are maintained and verified by the hypervisor to ensure usability in the event of an available OS failure. The advantage of using hypervisors is the reduction of the need to implement secure applications, however, they have a high cost of overhead, as they run multiple OSs on the same system.

3.1.3 Hardware and Software Based Designs

Solutions based on software cannot fully detect errors in the system, due to errors in the control of instruction execution (AZAMBUJA et al., 2011)]. In this way, mixed fault-tolerant system solutions benefit from the speed and accuracy of hardware error detection with the flexibility and configurability of software error detection technique, achieving fault tolerance at all layers of the system. Hardware devices are integrated into applications more efficiently, ensuring greater performance and reliability in the secure system, using frameworks to create sets of rules for interoperability of the layers.

The work presented by (CHENG et al., 2016) proposes a framework for fault detection using resilience techniques, with a design that has a fault tolerance coverage from the physical circuits to the application layer of the system. The hardware layer uses radiation-resistant flip-flops, so that they have greater tolerance to SEE. In the logical layer, parity checkers are used in the inputs and outputs of flip-flops. The heuristic used by the system has no significant impact on the clock frequency of the system, performing the grouping of flip-flops based on time parameters and the creation of logic verification pipelines. At the architecture level,

algorithms were implemented to check data flow and control (Data Flow Checking and Control Flow Checking, respectively) in order to validate the execution of instructions and memory access functions. At the software level, techniques for verifying software signatures were implemented by modifying compilers to verify static control flow diagrams. The impact of the implementation on energy consumption reached 6% in the worst case, while the system obtained a 50x higher error detection rate.

The design presented by (TIWARI et al., 2011) uses a CPU with external architecture for low-level control of hardware, that allows a microkernel to access device properties and information at gate-level. The microkernel is responsible for validating the steps of the system pipeline and running a basic operating system, used by other layers of software used to run the secure application. The external processor architecture allows explicit control of software in the implemented hardware.

The work published by (AZAMBUJA et al., 2011) uses a non-intrusive watchdog to detect variations in the flow control of a soft-core MIPS and decode application instructions, along with rules to generate additional instructions to control the additional hardware module. In this manner, it is possible to manage hardware and software signatures generated by the systems to detect failures in the execution of instructions. The error detection capability reached 100% coverage, with a runtime overhead of up to 153%.

Another strategy is to use hardware redundancies in specific vulnerable locations, to reduce the total technique overhead, along with compilers that perform variable redundancy and control of execution flows, such as demonstrated by (AMUTHA; RAMYA; SUBASHINI, 2012). The proposed architecture reached about 90% error detection coverage.

In this section, fault-tolerance processor design techniques were reviewed, with methodologies based on software and hardware for fault detection. In this work, the failure detection technique based on hardware will be implemented. Table 2 shows a comparison between the techniques covered in this section. The choice of technique during the development of the system must be made considering the cost of implementation and the impact of the technique on the project, both in terms of performance reduction (overhead in software), as well as consumption increase in power and device area (overhead in hardware).

Table 2: CPU robust design techniques.

Design Type	Pros	Cons
Hardware	- High detection rate; - Have no overhead of software.	- It presents overhead in hardware, increasing energy consumption and area; - Increase in the physical device development cost.
Software	- Lower detection rate; - Doesn't have overhead in hardware; - High tolerance design flexibility.	- High overhead in performance; - May not detect failures due to CPU control errors.
Hardware and Software	- High detection rate; - Combining the speed of hardware with the flexibility of software.	- Needs more development time; - High overhead in hardware and software.

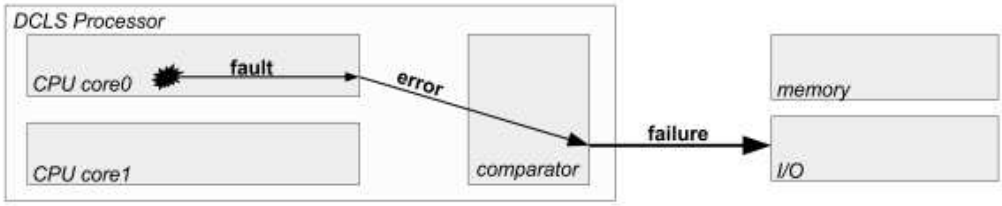
Source: From the author.

This section has shown the literature approach concerning robust CPU design techniques. The next section introduces CPUs fault injection methods, to allow designers to validate their fault-tolerant systems.

3.2 CPUs Fault Injection

Fault injection experiments can be executed to evaluate the fault-tolerance factor of CPUs, checking the design vulnerability, and the error observation rate obtained during fault injection testing. The fault injection is performed to generate observable errors in the outputs of the device, which are possible to be detected through conventional fault detection techniques. Figure 4 shows the propagation of faults within the processor to the outputs of the system, and shows the general architecture of a lockstep, a term that will be covered later. One of the challenges while performing fault injection is to produce observable errors to test the fault detection capacity of locksteps, due to the time to propagate faults from the fault origin to system output.

Figure 4: Fault propagation in a lockstep system.



Source: (Iturbe; Venu; Ozer, 2016).

Access to internal components is limited on hard-core CPUs, due to the device construction and manufacturer specifications, and it is not possible to access the internal CPU registers (VELAZCO; REZGUI; ECOFFET, 2000). Thus, it is necessary to use specific techniques for fault injection, by manipulating elements of external memories and processor interfaces. Soft-core CPUs presents greater flexibility for fault injection, since the availability of the high-level hardware description (HDL) allows the manipulation of internal variables.

Only a fraction of the faults injected into CPUs are observed as errors in the outputs of the systems, in which a large part is masked by the dynamics of the processor itself. The errors that are propagated to the outputs are generally generated in the injection of faults in specific register zones of the processor, which can be exploited to increase the most vulnerable parts of the device, as described by (MANSOUR; VELAZCO, 2012).

In this session, a bunch of fault injection in CPUs works will be reviewed, which approaches tolerance tests and fault vulnerability of soft and hard-core CPUs, in addition to the creation of techniques and systems for fault detection and error mitigation. Fault injection techniques via CEU and radiation exposure are implemented in the work developed by (VELAZCO; REZGUI; ECOFFET, 2000), in which the authors compared fault injection techniques in hard-core devices, such as the 80C51 microcontroller and module DSP 320C50, making changes to the contents of the memories external to the components and causing unwanted interruptions to observe the effect of these experiments on the system responses. The tests carried out indicated that the rate of observable errors in the system outputs was

higher during fault injection via CEU, demonstrating the need to use memory protections in safe applications exposed to radiation.

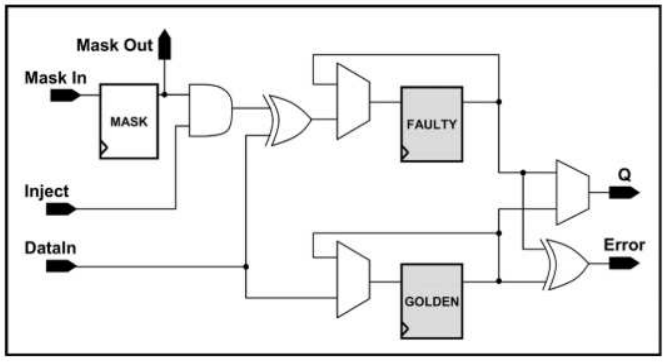
The technique proposed by (WANG et al., 2004) performs fault injection via simulation on AMD Athlon and Alpha 21264 commercial processors. The analyzed experiment indicated that approximately 15% of the injected faults generated visible errors in the CPUs outputs. Applying low-cost redundancy techniques in these known regions, there was a 75% reduction in failures compared to the initial tests.

Another technique for fault injection through emulation in FPGA's is presented by (KASTENSMIDT; FILHO; CARRO, 2006), in which a tool for fault injection was developed by modifying the FPGA's configuration file (.bitstream), being possible define the fault injection region directly through the application. In this way, it was possible to change the routing configuration of the implemented logic and user registers.

Fault injection can be performed by modifying LUT's from the FPGA, as shown in (KENTERLIS et al., 2006). The approach performs the analysis of the LUT's used to map the design in programmable logic, modifying its configuration to result in execution failures in the system output.

The work done by (VALDERAS et al., 2007) implements complementary fault injection techniques by emulating code upsets and by emulating LEON processors on FPGA. Faults injected via code fault emulation (as known as Code Emulated Upset - CEU) were performed via alteration of memory elements external to the CPU (such as SRAM, for example), generating the execution of erroneous codes and asynchronous activation of interrupts. The result of the operations generated corrupted data in the memory elements of the processor. Faults injected via FPGA were implemented by replacing the original flip-flops with versions with fault injection control, as described in Figure 5. The application was initially executed with golden flip-flops to collect the fault-free results. Then, the faults were injected by changing the mux in the output of the flip-flops to produce faulty results. The responses were then analyzed using a computer. Failures injected via CEU resulted in a greater number of device failures observed at the system outputs.

Figure 5: Hardware Fault Injection.



Source: (VALDERAS et al., 2007).

The effects of SEUs on CPUs memories soft-core are emulated in (MANSOUR; VELAZCO, 2012), using the Direct Fault Injection - DFI technique, changing the architecture of the internal registers to inject faults during the clock cycle time execution. Additional

hardware is added in the CPU implementation, where muxes and selectors are used to control the fault injection.

There are also techniques based on the injection of faults through heavy ions to assess the sensitivity to radiation of the system components. (TAMBARA et al., 2015) performs tests with a radioactive ion injector on a Zynq-7000, to measure the effects of radiation on the elements of the FPGA, such as configuration RAMs, RAM blocks available to the user and CPU cache memories hard-core ARM Cortex-A9. It is possible to perform fault injection using logic reconfiguration modules, available in some FPGA's on the market.

The work performed in (KESH; ASAMI, 2018) performs fault injection using the ICAP module in a FPGA KINTEX 7. The module is controlled remotely through an application developed in MATLAB, which performs communication with the module through an interface UART. The work proposes a methodology to optimize the failure injection time, in which only the essential design configuration bits are changed, in contrast to the traditional methods, which perform fault injection in all configuration bits, including those not used for the design mapping in the FPGA.

In this section, works that developed and implemented techniques for the injection of failures in processors were reviewed. Table 3 compiles the main techniques explored. The choice of the fault injection technique must be made taking into account the time available for developing and applying the tests, the complexity of the system to be tested and the resources available. In this work, fault injection techniques will be used via simulation, for initial system testing, and fault injection via emulation, making changes to the design configuration in the FPGA, to carry out the final validation of the device's effectiveness. The next section shows the lockstep technique, with its definitions and literature examples.

Table 3: Fault injection techniques.

Injection Type	Pros	Cons
CEU	- Rapid implementation strategy, can be performed without the need to change the hardware and obtain good coverage of failures; - Faster implementation methodology only needs to change the instruction memory to implement the failures.	- It does not have the flexibility to inject faults in specific regions of the core; - Limitation on the types of injectable failures.
Simulation	- It has greater flexibility for implementing failures and system tests; - Greater practicality of implementation, due to the use of resources from the simulation tools.	- Consumes more time for validation compared to other techniques, due to the need for computational capacity;
Emulation – Alteration of design	- It has greater speed and error validation range, making it possible to carry out internal modifications to the devices; - The fault injection test can be performed on any device;	- Needs more development time, due to the need to change the design; - It can result in overhead in hardware of the device, limiting the practical application;
Emulation - Changing the design configuration on the FPGA	- It has a high speed of error checking; - Uses internal modules of the programmable logic device, without the need to change the design;	- Needs the implementation of control logic for fault injection; - Fault injection is exclusive to devices, not having portability.

Source: From the author.

3.3 Lockstep

Lockstep is an error detection technique, by running copies of programs on redundant hardware, comparing devices at each cycle of clock. Applications with lockstep are popular on ultra-availability servers (JEFFERY; FIGUEIREDO, 2012) and secure systems due to their ease of implementation, high error coverage, transparency in software and performance (OZER et al., 2018). There are three main types of implementation of lockstep: systemic, sub systemic and CPU only:

a) Systemic: CPU, CACHE and RAM control signals and buses;

b) Sub-systemic: CPU and CACHE control buses and signals;

c) CPU: Only the CPU buses will be analyzed by the lockstep.

Systems with application criticality may benefit from CPU lockstep implementation, due to greater error coverage and lower implementation cost compared to the systemic level lockstep. In lockstep of CPUs with dual modular redundancy (DMR), it is not possible to indicate which CPU originated the error, since lockstep detects discrepancies only in the outputs of the devices, making the hardware reliability analysis necessary. In triple or multiple redundancies (TMR or MMR, respectively), the lockstep can identify the CPU that originated the error, using majority voting logic. In this way, specific tests can be performed on the faulty device. Although it is possible to detect the faulty core in TMR or MMR techniques, its downside is the increased hardware overhead compared to the DMR technique. Thus, a trade-off between fault origin detection and area and timing overhead is presented on choosing the appropriate fault tolerance technique. Depending on the application, the CPU with errors can be disabled until the identified error is corrected, increasing the availability of the system (GHAHROODI; OZER; BULL, 2016). In any case, the checker cannot predict whether the error type is transient or permanent, and it is necessary to alert the system controller and the user in case of hardware failures.

Most of the lockstep architectures use only the external buses to the core (non-intrusive method), to make the system more generic, facilitating its integration in more complex systems with greater ease. However, the time to propagate failures from their source to the CPU outputs can be magnitude of thousands of cycles of clock (OZER et al., 2018). Since error detection in lockstep is not immediate due to fault propagation, a permanent fault can spread to multiple CPU outputs compared to transient faults. Thus, the use of signals internal to the core (intrusive method) allows a higher rate of failure detection and an increase in the speed of action of the safe system in the event of errors in the device.

When an error is detected, lockstep signals the secure system controller that the event has occurred and puts the processor in a safe state to reduce hazards. The system controller can then initiate a series of internal tests (BIST) to identify the source and type of the error, whether it is transient or permanent. In the event of a transient error, a rollback can be performed to reset the current state of the processor to a known safe state. In case of fatal errors, the system controller can change the system to a definite safe state and indicate a fatal system failure. The challenge of implementing the lockstep technique is to ensure that all redundant CPUs produce the same output in each execution cycle without errors, since all internal CPU variables must be initialized in the same way at system reset, as described by (OZER et al., 2018). CPUs registers initialized with different values can lead to divergences during normal operation, resulting in false positives during error detection. In this way,

Lockstep implementations need a meticulous design to guarantee the equivalence of the CPUs internal state in the system reset. In the following, some works from the literature on implementing lockstep are shown.

3.3.1 Lockstep Implementation Review

A non-invasive architecture of lockstep is developed by (ABATE et al., 2009), implementing the design in two IBM Power PC 405 hard-core embedded in a Xilinx Virtex II Pro FPGA, so that the verification of the processor's execution parity is carried out after the execution of the application. In cases of divergence, checkpoints are used to perform a rollback of the previous processor context, saving time in clock cycles and increasing system availability. In the fault injection experiments, about 70% did not result in changes in the outputs of the processors, exemplifying the theory that ASIC's are more resistant to radiation compared to programmable logic, and such must have more complex mechanisms for fault detection and mitigation.

The lockstep developed by (Iturbe; Venu; Ozer, 2016) has dual CPU redundancy architecture ARM CORTEX-R5. The CPUs share cache memory and peripheral access ports, performing redundancy only at the CPU level. The clock of the primary CPU operates two cycles ahead of the secondary, to minimize the possibility of transient errors at the same time. Signals from the internal stages of the CPU pipeline were used to increase the speed of fault detection in the system and the response time to the error, to mitigate the propagation of errors to elements external to the CPUs. The work categorizes the most critical parts to core failures, injecting failures in the regions, and accounting for the propagation of visible errors in the outputs. In the tests performed, 70% of the errors observed in the outputs were caused by failures in about 10% of the core registers. These metrics demonstrate the sensitivity of the sequential elements inside the processor, allowing the implementation of redundancy techniques in a smaller number of elements, increasing the overall reliability of the system.

The lockstep architecture shown in (VENU; OZER; ROBINSON, 2016) implements a triple redundancy in soft-cores ARM CORTEX-R for detecting and correcting individual errors. The structure was developed with a flexible design to be used in any ARM CPU, to reduce costs and implementation time. The error detection and correction systems are isolated from the CPUs' power and clocks nets, in addition to using a robust coding technique to protect the verification logic, defined in ISO-26262. The implementation of lockstep resulted in a 30% reduction in clock compared to the system operating without lockstep, with a single CPU.

The work carried out in (ITURBE et al., 2016) builds a non-intrusive triple-core lockstep architecture with ARM CORTEX-R5 soft-cores for critical security applications, where each CPU has its clock tree and share instruction and data caches, which are protected by error correction codes - ECC. A majority voting structure was implemented, to allow the secure code execution in case of failure of only one CPU, increasing the availability of the system. The error detection logic works along with the CPUs re-synchronization logic in case of transient failure. The CPUs re-synchronization time was completed in less than 2 us, using a 1 GHz clock.

An invasive lockstep architecture in two soft-cores ARM CORTEX-R5 is implemented in (OZER et al., 2018), using internal CPU signals to create a divergence status map between CPU outputs. At work, 62 categories of signals internal to the CPUs used were

compared, to enable the prediction of the source of the error injected into the system. The predicted error site was then used as a basis for choosing internal tests for detecting transient or permanent failures. The manifestation of transient errors in the outputs of the system was about 5%, while the rate of permanent errors was 40%. The prediction of error location reached 86% for transient errors, while the accuracy of the prediction of permanent failures was equal to 49%.

In the work shown in (RODRIGUES et al., 2019), a lockstep framework called xLockstep was proposed, embedded in a Zync FPGA. The system is composed of a hard-core ARM Cortex-A9 built in the hard processor portion of the FPGA and a RISC-V soft-core, implemented in the programmable logic. In the proposed work, the application is executed in a loosely-coupled fashion in both cores, and an IP implemented in the programmable logic checks for divergences between code execution and synchronize its operation. The lockstep design modules were connected via an AXI bus. The proposed xLockstep resource usage was extremely lightweight, with an order of nearly 2% area consumption in comparison with the RISC-V soft-core implementation.

In this section, techniques for fault detection in CPUs were presented. Table 4 presents the main articles used as the basis for this research. They are used to implement lockstep of processors for secure systems.

Table 4: Review of major works.

Authors	Description
(ABATE et al., 2009)	Authors used a IBM Power PC 405 hard core in a Xilinx Virtex II Pro FPGA to implement a DCLS system.
(Iturbe; Venu; Ozer, 2016)	Work shown a DCLS implementation with ARM CORTEX-R5 soft-core CPUs. The most critical parts of the core are characterized, in a way to deploy a design with higher reliability.
(VENU; OZER; ROBINSON, 2016)	TCLS architecture using an ARM CORTEX-R soft-core triple redundancy scheme for ISO-26262 compliance and flexible design.
(ITURBE et al., 2016)	TCLS with ARM CORTEX-R5 soft-core in non-intrusive design, for safety critical applications. The design includes EEC and presents a low latency re-synchronization.
(OZER et al., 2018)	DCLS with ARM CORTEX-R5 soft-core focused on estimating the source region of the failure, increasing the error prediction rate in order to decrease error correction time and increase system availability.
(RODRIGUES et al., 2019)	Heterogeneous DCLS with an ARM CORTEX-A9 hard-core and a RISC-V soft-core. The authors shows a programming framework called LOCK-V, which is intended to deliver a quick solution for both ARM and RISC-V application development.

Source: From the author.

Although there are major works relating to lockstep techniques, both in hard and soft-core implementations, we have not yet seen a work that approaches a fault-tolerance design exclusively with RISC-V soft-cores. Thus, we believe that this is the first work that explores a dual-core lockstep architecture with 32-bit RISC-V soft-core processors as a fault-tolerance technique. In this work, the DCLS is shown, where its architecture, functional specification, and design flow is presented. The next chapter shows the presented DCLS architecture.

4 PROPOSED DUAL-CORE LOCKSTEP SOLUTION

The DCLS solution proposed in this work is presented in this chapter. The lockstep is a hardware and software technique aimed to increase a system's fault tolerance. In the hardware part, both core's outputs are compared at each clock cycle, and any divergence between its output indicates an error in the system. In the software part, the core context may be saved and restored when a correctable error is detected. The technique downside is an overhead in the system, both in the hardware part due to the core replication and additional modules to control and detect the lockstep operation, and in the software part due to the necessity to handle the core backup/restore operation. This chapter explores the proposed DCLS hardware architecture and functional specifications.

4.1 DCLS Hardware Architecture

The DCLS hardware architecture is composed of multiple specific modules and the core replication. In this work, a 32-bit 4 stage RISC-V core (named RI5CY) is used as the target CPU replication. It is a soft-core developed by ETH Zurich, used in the PULP project (CONTI et al., 2016). Additional hardware components were used to evaluate the DCLS, such as instruction and data memory, interconnection buses, and peripherals. Figure 6 shows the proposed DCLS system architecture and its components. It features 32 KB instruction, data, and secure data memory, used to store the application's execution context. A custom DMA handles the secure data memory read/write operation, and it is accessible by the user.

An AXI bus is used to enable the interconnection between system components. A Dedicated bus in the DCLS control unit handles the interface with the external system control, allowing DCLS command assertions. Lastly, a GPIO module enables I/O in the system. All DCLS sub-modules are designed using a one-hot registers technique and uses mostly combinational logic, due to the susceptibility of registers to failures. The DCLS sub-modules functionality description are listed below:

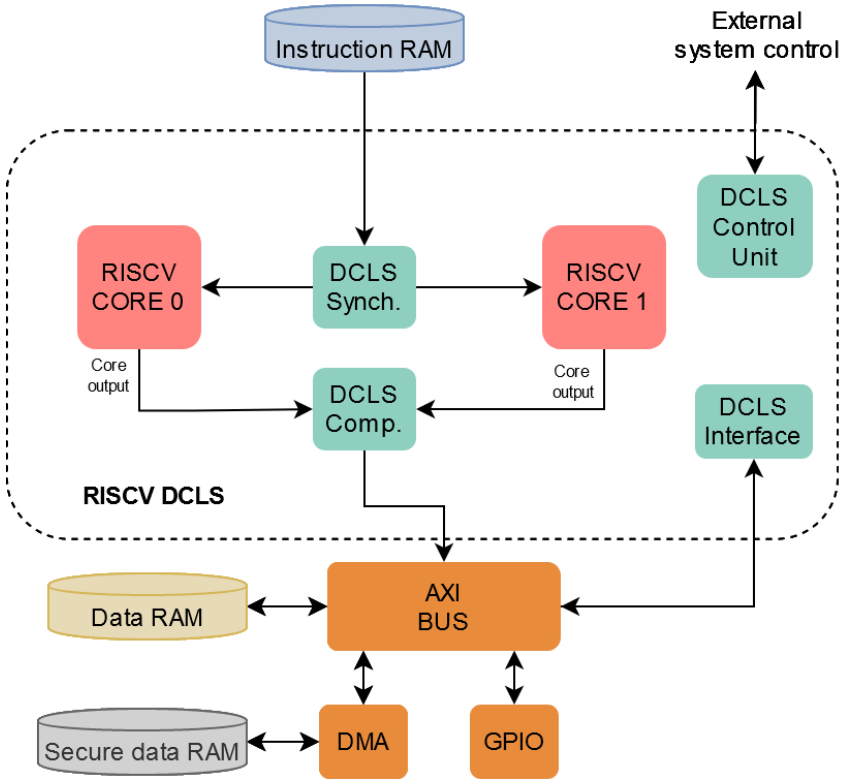
a) **DCLS control unit**: Controls DCLS modules operation at hardware level, and is it slave of the external system control. Users may pause and resume applications in a transparent fashion using the DCLS system;

b) **Synchronizer**: Injects core signal controls and define its execution status;

c) **Comparator**: Compare cores output buses at clock cycle; forwards valid signals to the AXI bus. It is always enabled to avoid mismatch under-reporting;

d) **DCLS interface**: It's accessed as a peripheral; allow users to trigger a context backup.

Figure 6: DCLS architecture.



Source: From the author.

4.2 DCLS Functional Specification

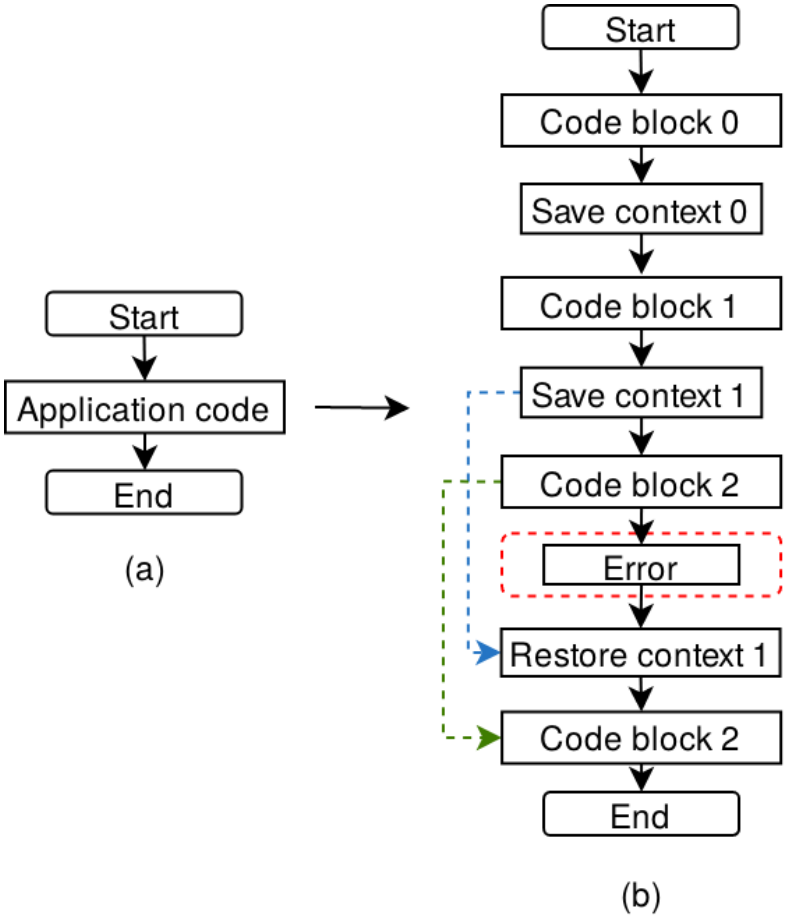
The DCLS system implements both software and hardware solutions. The hardware behavior must follow a flow according to the software requirements, and vice-versa. In the hardware portion, the main features of the DCLS system is the bus comparator to detect errors. In the software portion, it is the context backup and restore the main addition of the DCLS system. Thus an execution flow is designed to integrate both hardware and software solutions. By using a RISC-V soft-core, it is possible to change the design as needed, enabling a flexiblity factor in the design process.

4.2.1 Overview

Figure 7 shows the DCLS execution flow. Figure 7 (a) shows the unhardened execution flow of an application, and Figure 7 (b) shows the fault-tolerant application execution, with the addition of an error occurrence to illustrate the execution flow. In the unhardened flow, the application is executed without error detection and context backup/restore, whereas the hardened flow has additional features, such as segmented application code execution and context save, to increase the application fault-tolerance level. In case of an error, as illustrated by Figure 7 (b), the last valid context is loaded back into the cores and the application is resumed. The safe application must include routines and be

segmented conveniently to allow context backup operations during its execution. The safe application is defined as a time-finite application, that has a beginning and an end, as illustrated by Figure 7 (b). All context backup and restore operations are executed within its time execution window. The DCLS can be used in loop-applications, but this case will not be evaluated in this work.

Figure 7: DCLS application execution flow.



Source: From the author.

The main objective of the DCLS is to detect errors and be able to perform a rollback of the core's context to a prior know safe state. During the execution of the safe application, the user may trigger a context save operation to allow a rollback operation when an error is detected. The safe state is composed of the following components:

a) **General purpose registers** - GPR: All application data handled by the core are stored in the GPRs, including stack pointer, return address, temporary and saved registers, function arguments, and return values;

b) **Control and status registers** - CSR: Stores core configuration values, such as machine and user program counter, interruption vector configuration, and other control parameters;

c) **Data memory**: It's the application RAM, used to store program data and the stack, where the core context is saved.

The context save routine is triggered at the user level by accessing the DCLS interface peripheral, using an interruption request. The user must include the checkpoints in the application code to execute the context backup operations. The data memory content management must be evaluated during the interrupt handling. Only the external system control may trigger the context restore, due to the necessity to evaluate the error type (transient, permanent, correctable, uncorrectable). The context backup flow is described as follows:

1. User application write into DCLS interface register;
2. An interruption request is generated at gate-level;
3. The specified interruption handler copies core's context into the stack;
4. The whole data memory RAM is copied via DMA to a secure memory;
5. When the copy operation is done, the application may resume its execution.

After detecting an error and evaluating whether it's a correctable one, the DCLS application context may be restored. Its context restoration flow is described below:

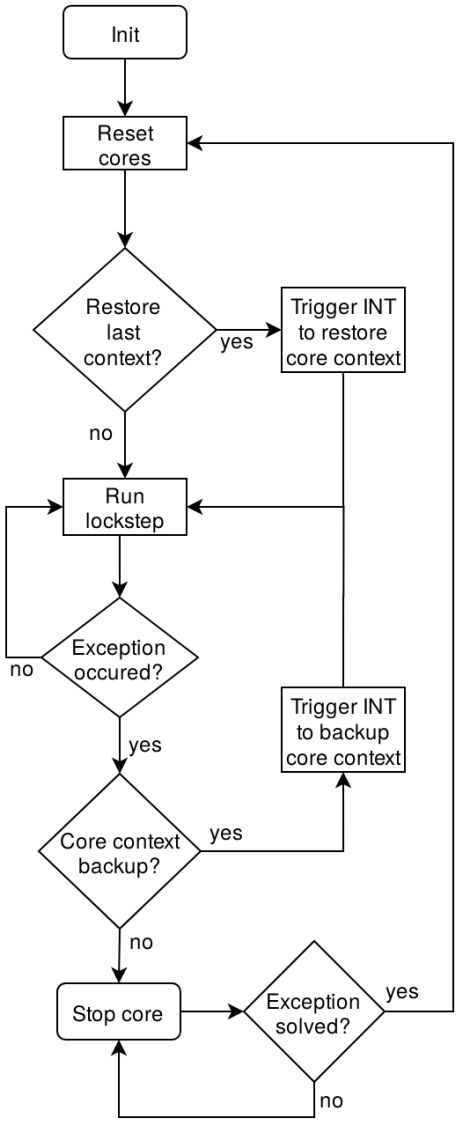
1. The external system control asserts a reset and context restore flag into the DCLS control unit;
2. Both cores are re-initialized;
3. An interruption request is generated at gate-level;
4. The secure memory content is copied to the data memory via DMA;
5. The specified interruption handler restore core's context from the stack;
6. Resume application execution.

4.2.2 DCLS Control Unit

The DCLS features a control unit, shown in Figure 6, which is responsible for controlling other modules during the execution flow at the hardware level. It is controlled via a dedicated bus, which only the external system control may access. It is transparent to the user application, and it has the following functionality:

a) Enable/stall cores;
b) Internal core reset;
c) DCLS Modules enable control.

Figure 8: DCLS control unit execution flow.



Source: From the author.

Its state machine execution flow is designed to handle the context restore and system execution control. Figure 8 shows the DCLS control unit execution flow. After both cores are initialized, the DCLS system may perform a context restore. If there is an available context to be restored, the DCLS may restore it and resume the application from the point where it was saved. If not, the application will be initialized and executed.

There are two main exceptions in the DCLS state machine context: one is the user request to save the current execution context, and the other one is error detection. The user request to save the execution context is performed by accessing the DCLS interface, which is available as an APB peripheral, as shown in Figure 8. After accessing the peripheral and writing into the control register, an interruption request is generated to both cores, and after the context backup, the application resumes its execution. In case of a detected error, the DCLS system enters a safe state, where both cores are stopped and the external system control is alerted. If the exception is solved, the external system control must assert a solved flag into

the DCLS, as the system may perform a core reset, load the last saved context and resume the application execution.

4.3 DCLS Solution Overview

The DCLS is designed to detect divergences between core's output at any execution point, and indicate to an external system control that an error has been found. It must synchronize both core's input with timely-correct instructions, to ensure its tightly-coupled execution, and a local controller supervises its operation and ensures that every module is working properly. User may configure target application to include checkpoints, which triggers context save operations, that can be restored after an error is detected and cleared.

The external control system is the master of the DCLS system, and may perform context restore operations at will. The context backup/restore scheme allows an application to be paused, saved into a safe memory, have the whole system reset/reconfigured, and resume the application execution after performing a context restore operation. The downside of the technique are the software and hardware overhead, needed to deploy the fault- tolerance addition to the system. As there is no extra core to check which core propagated the detected error, it is necessary to perform an external system check, and take the needed action to ensure the system reliability. The next section explains the built fault injection methodology, to evaluate the proposed DCLS design.

5 FAULT-INJECTION METHODOLOGY

A fault-injection methodology is needed to evaluate the DCLS error detection capability and backup/restore features. This chapter describes the fault injection framework, error type, and the tool used to evaluate the fault injections.

5.1 Introduction

Fault injection methodologies are applied to test and evaluate a Design Under Test – DUT fault-tolerance capabilities. A majority of techniques are shown in the literature, as explained in section 3.2. All methodologies pursue the objective of applying controlled fault injections to the system and expect output changes due to the injected fault. The fault injection method is selected based on the project's budget, resource availability, and desired fault coverage. In this work, a simulated fault injection framework is presented to evaluate the DCLS system fault-tolerance features. Its details are presented in section 5.2.

An open-source simulation tool, called Verilator is proposed as the fault injection implementation software. The Verilator is a SystemVerilog simulation tool, that works by transforming the HDL files into object-oriented C++ code. It allows the construction of complex testbenches using the C++ power, as well as simulating multiple threads. It is available in the industry since the '90s and is used by the largest silicon industry companies, such as AMD, Intel, ARM, NXP, and others. It is also used in the academy as simulation tool (AHMAD; CIESIELSKI, 2014), (PETRISKO et al., 2020). It allows users to probe inside a design's registers using high-level functions, allowing a high fault injection coverage without additional hardware modification. Also, it allows multiple simulation instances executions, without the need for additional licenses and tool support.

5.2 Fault Injection Framework

A fault injection framework was developed to inject faults into the DCLS architecture shown in 6. The main objective of the fault injector is to successfully flip bits at the DUT, according to the injected fault type. The flipping of bits inside the core may produce output errors at the core's output buses, representing an error. There are two fault injections type:

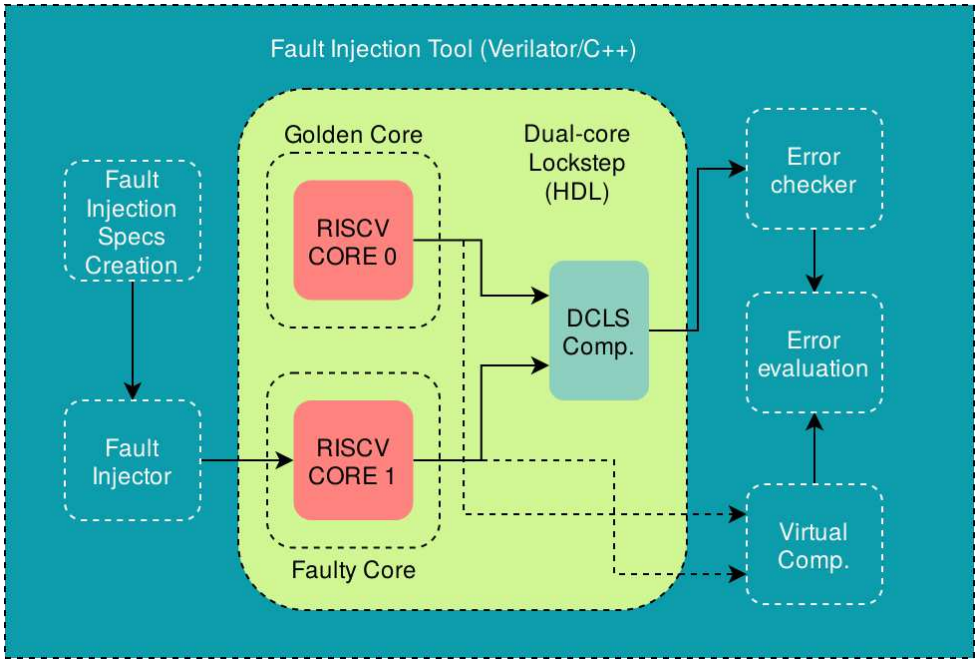
a) Transient fault injection: A bit is inverted during one clock cycle;

b) Permanent fault injection: A bit is stuck at a logical value (HIGH or LOW) during the whole execution run.

The fault injection diagram is shown in Figure 9. The DCLS architecture shown in Figure 6 is fully implemented in the Verilator, and the Figure 9 is simplified due to illustration purposes. The fault injector framework accesses the faulty core internal registers, allowing it to read and modify its values at any simulation execution time. The fault injector framework controls the DCLS system simulation, using the Verilator's programming interface. It uses the DCLS system I/O to determine the application execution phase, to correctly inject faults, and observe the system's response. The fault injector framework also works as the external system

controller, as shown in the Figure 6. It manages both fault injection and safe state management of the DCLS system.

Figure 9: Fault Injector Framework.

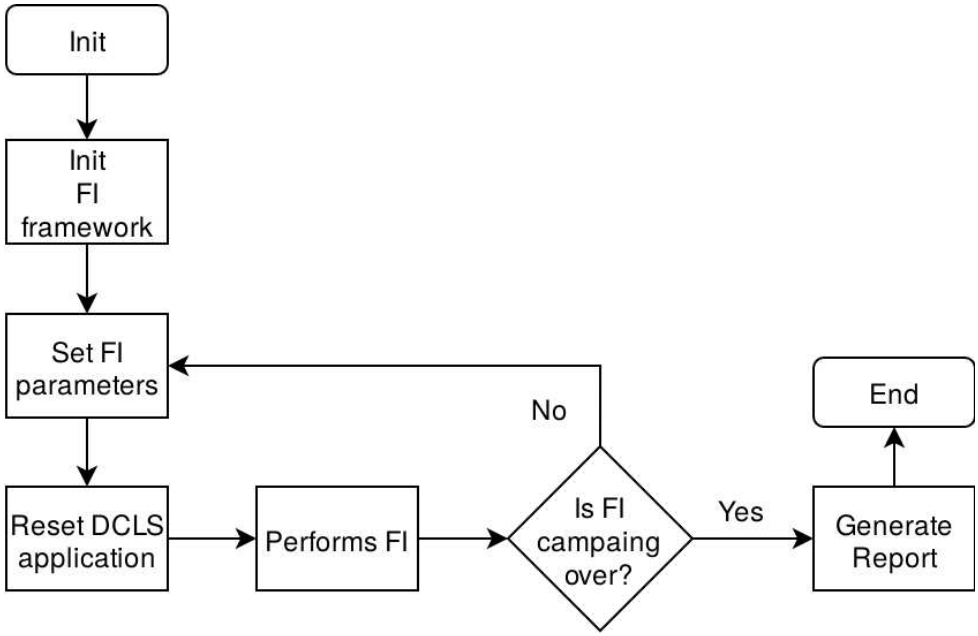


Source: From the author.

A single core is the main target of the fault injector, called faulty core, and all its pipeline's internal registers are prone to fault injection. The fault injection is implemented using an exhaustive fashion, where all internal registers experience a bit flip during the fault injection campaign. This may result in an expanded fault coverage, considering that not all internal modules are used in the application (i.e. debugging unit and performance checker) and would not be excited during a regular application execution. All fault injection runs flips only a single bit at a time, meaning that a single n-bit wise register may result in multiple fault injection candidates, as each bit is considered an independent variable. The remaining core is considered the golden one, where it runs fault-free. There are a total of 2857 registers bits inside the RI5CY pipeline, and a fault injection campaign is defined as the execution of fault injection in all available bits inside the core within target application execution.

The objective of the fault injection is to produce errors at the faulty core, to evaluate the DCLS error detection capability and the system's safe state management. The target application is executed multiple times, where a single fault is injected at each execution run. The fault injection framework controls the fault injection routine and the results processing. The fault injection flow is shown in Figure 10.

Figure 10: Fault Injector Execution Flow.



Source: From the author.

At each FI run, new parameters must be defined, such as target fault injection type, injection time, and selected registers. These parameters are listed in Table 5:

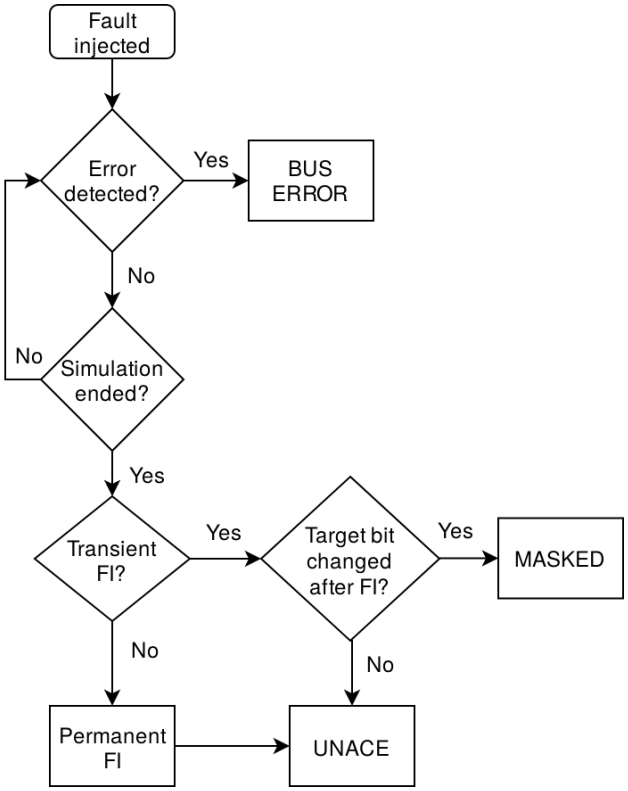
Table 5: Fault Injection Parameters.

Parameter	Description
Fault injection type	Defines FI implementation. It can be a soft-fault (transient) or a hard fault (permanent).
Fault injection time	Defines FI time. It is defined randomly.
Fault injection register bit	Defines target FI bit. As the FI uses an exhaustive fashion, a bit is flipped only once given a FI type.

Source: From the author.

After the injection of the fault, the fault injector framework watches for any divergence in the DCLS cores via a virtual comparer, shown in Figure 9. If an output mismatch is observed by the virtual comparer but not corresponded by the DCLS comparer, an error flag is asserted and the simulation is stopped. If both virtual and DCLS comparer detects an error, its type is processed and logged for report generation. The Verilator can modify any bit value at any time, but its value may be updated by the bit's netlist during the simulation run. To avoid incorrect fault injections, a bit watcher is implemented in the fault injector framework, where it is always evaluated before the simulation's clock update. In transient fault injection runs, it checks if the target bit register was updated after the fault injection operation. In a permanent fault injection run, it re-injects the target fault if necessary. The error reports are generated after executing the fault injection campaign. It follows a set of rules shown in Figure 11.

Figure 11: Error Type Rules.



Source: From the author.

Table 6 shows further explanations about error types, indicating each peculiarity and explaining its practical effects on the Design Under Test during the fault-injection campaign.

Table 6: Error Types Explanation.

Error Type	Description
Bus error	Is true whenever the DCLS comparator detects an error. It may be at any core's output buses.
UNACE	Is true when the injected fault was on a register Unnecessary for Architecturally Correct Execution — UNACE.
Masked	Is true when an injected transient fault did not propagate to the outputs and was over written during application execution.

Source: From the author.

This section exhibited the proposed fault injection framework. It consists of an application built on the Verilator simulation tool, and uses object-oriented programming language C++ to create, evaluate and generate reports upon the fault injection routines simulations. The next section explains the target applications used to evaluate the fault tolerance capability of the DCLS system during the proposed fault-injection campaign.

5.3 Application

This section shows the target applications for evaluating the DCLS system. The applications are intended to use as much as possible of the core's processing power, to increase the register utilization and therefore increasing its susceptibility to faults. DSP applications are selected as test cases, as they are used widely in the literature. A matrix multiplication and an IIR filter are used to evaluate the DCLS system. Different application sizes are used in the fault injection campaigns. As the application size differs, the susceptibility and result of fault injections may change. The matrix multiplication is evaluated using a 7x7 and a 3x3 matrix sizes, both using a single-precision data type. The compiler was configured to emulate all float points operations, as the target core doesn't include a hardware float point unit. The IIR filter is evaluated with a 10 and 100-point input vector, and uses a fixed point C library to perform its operations. Both DSP operations are heavy CPU consuming, as it both emulates float and fixed-point operations. This increases even further the register file utilization, increasing the system susceptibility to faults. A total of 3 contexts backup checkpoint are added to all applications, to evaluate the DCLS software overhead and allow context restoration after an error detection. Table 7 resumes each application case used to evaluate the DCLS system.

Table 7: Application Cases.

Application Case	Description
Matrix 7 x 7	Matrix multiplication application using emulated floating point operations. Each matrix is a 7x7 square one.
Matrix 3 x 3	Matrix multiplication application using emulated floating point operations. Each matrix is a 3x3 square one.
IIR 100	IIR filter application that uses fixed-point lib. Filters 100 points of data.
IIR 10	IIR filter application that uses fixed point lib. Filters 10 points of data.

Source: From the author.

This chapter explained the fault injection specifications, as the target applications to evaluate the DCLS fault-tolerance capability. In the next chapter, the results of the proposed system are presented.

6 RESULTS

This chapter shows the obtained results of the DCLS system, in terms of hardware utilization, fault injection results and software overhead. The DCLS system hardware description was designed and validated using ModelSim, a commercial Verilog simulator. The DCLS system then was implemented in the Verilator simulation tool, building the fault injection framework. The system then was re-validated to ensure all expected behavior and hardware interfaces worked correctly, and that the design features work as intended. The Verilator implementation was seamless and didn't require any HDL modification.

6.1 Implementation Results

The DCLS enhances a system's reliability and fault tolerance at a hardware and software with an overhead cost. This section shows the implementation results of the DCLS, both in terms of hardware consumption in an FPGA synthesis and on additional clock cycles needed to implement its software features.

6.1.1 FPGA Synthesis

The DCLS design was implemented in a Xilinx Kintex 7 KC705 FPGA evaluation board, using its programmable part to evaluate the system's area, and timing performance. The Vivado Design Suite was used to synthesize the design. As this work's objective is to evaluate the DCLS solution overhead, only the core replication and additional DCLS modules are analyzed. All other SoC components shown in Figure 6 are not evaluated in this section analysis, as they are considered external components, and can be customized according to the user need.

The total DCLS resource consumption (duplicated core + additional DCLS modules) uses 4492 Flip-Flops (FFs) and 12317 Look-Up Tables (LUTs). Considering that a single RI5CY core unit consumes 2241 FFs and 6003 LUTs, the hardware overhead to implement the additional DCLS modules is at 0.44% in FF and 5.18% of LUT utilization compared to a single RI5CY core. The DCLS additional modules reach a summed resource utilization of only 10 FFs and 311 LUTs.

In terms of clock speed, the unhardened system reached a maximum of 80 MHz, where the clock frequency of the DCLS system archived a maximum of 65 MHz. This represents a reduction of 18.75% of its clock performance, and the resulting performance of the system may decrease even further when adding the context save operation, which may take several clock cycles to execute. The frequency reduction is due to the net delay in both clock and data path, and the critical path is inside the second core. The total power utilization of the DCLS system in the target FPGA board reached 0.363 W, where the unhardened system reached a total of 0.356 W. The power difference is not relevant, and the similar values can be explained primarily due to the FPGA's board base power usage. An ASIC implementation may increase the power consumption difference.

In contrast with other authors with different implementation architecture and results, in (OLIVEIRA et al., 2018), the authors implemented an ARM Cortex-A9 hardcore dual-core lockstep in a Xilinx Zync FPGA, and reached a hardware overhead reached up to 275%. Iturbe et al. (ITURBE et al., 2016) present an ARM Cortex-R5 triple-core lockstep (TCLS), and its TCLS assist unit consumes nearly 18% of the ARM Cortex-R5 area. A lockstep framework using RISC-V is proposed by (RODRIGUES et al., 2019). The proposed lockstep resource usage was extremely lightweight, with an order of nearly 2% area consumption if compared with the RISC-V area utilization.

6.1.2 Software Overhead

The software overhead is the additional execution time needed to perform context backup operations. The greater the number of context backups in an application, the greater the software performance penalty. The RAM size used in the application also has a major impact on the execution time overhead, and must be taken into account when projecting the system. In this work, a 32 KB RAM was used as main memory data, and all its content is copied during a context backup operation. Table 8 shows application cases execution time in clock cycles, both on unhardened and DCLS execution results. The application may suffer a great performance penalty if is not balanced, of it its application size is small if compared with the time needed to perform a context backup operation. Both cases Matrix 3 x 3 and IIR 10 have a great overhead if compared with its counterparts' applications. Besides the hardware overhead, which may increase system power consumption and reduce clock frequency as shown in the subsection 6.1.1, it may also increase the needed clock cycles to execute the application. Although the performance penalty is greater in smaller applications, no memory management optimization was performed to reduce such penalty, opening a gap for future works.

Table 8: Software Overhead.

Application Case	Total clock cycles unhardened	Total clock cycles DCLS with 3 checkpoints	Overhead
Matrix 7 x 7	287,791	511,837	77.85 %
Matrix 3 x 3	14,065	238,111	1592,93 %
IIR 100	206,106	430,152	108,7 %
IIR 10	25,566	249,614	876,35 %

Source: From the author.

This section exhibited the implementation results, both in terms of hardware implementation and software overhead analysis. The next section presents the fault injection experiments executed in the DCLS system.

6.2 Fault Injection Experiments

This section shows the results of the fault injection methodology presented in section 5.2. The fault injection campaigns are performed with the application cases explained in section 5.3. The main objective of the fault injection in this work is to provoke errors at the faulty core outputs, to evaluate the DCLS error detection capability. Each application case was used in an exhaustive fault injection campaign, where each core's pipeline register experienced a soft and a hard fault injection.

6.2.1 Fault Injection Results Overview

Table 9 shows a resumed fault injection statistic. In the same application case, soft faults were harder to propagate to the core's output as an error, whereas hard faults produced more errors during the fault injection campaign. The similar statistics in the error manifestation results may appear from the exhaustive fault injection approach. As hard faults are forced until the end of the simulation run, its corrupted data value can be kept underused for a long time, resulting in a later error propagation, and produce a later detection time as seen in the Table 9.

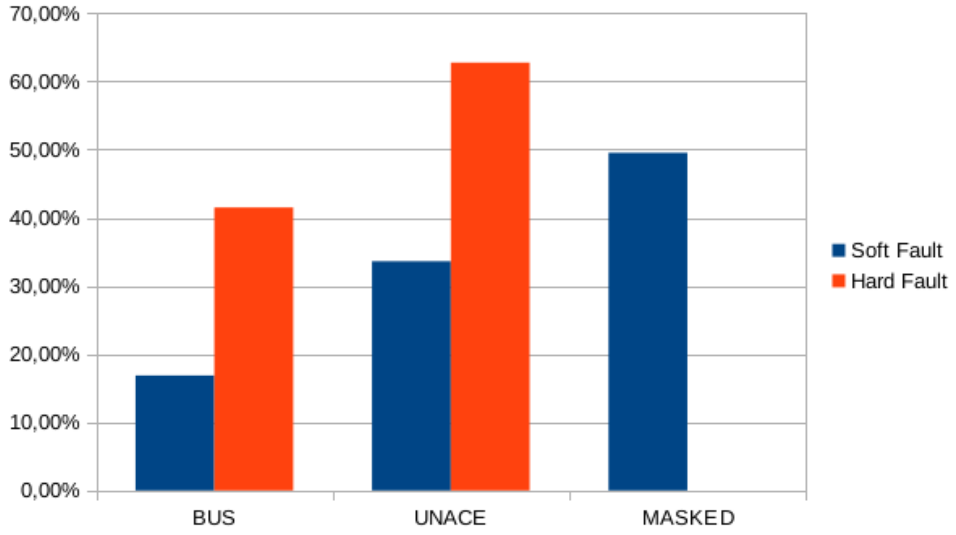
Table 9: Fault Injection Global Statistics.

Statistic	Min	Mean	Max
Hard Error detection time (clock cycles)	1	92	56678
Soft Error detection time (clock cycles)	1	37	305
Hard Error detection manifestation	39,59 %	41,65 %	44,31 %
Soft Error detection manifestation	16,56 %	16,85 %	17,64 %

Source: From the author.

Figure 13 shows the statistic results of error type detected during the fault injection campaign. As hard fault injections cannot produce masked faults, due to its physical nature, it is not present in the figure. Although there are no masked faults in hard faults, errors manifestation at the bus output is considerably superior than the soft fault counterpart. The majority of the injected soft faults didn't produce errors at the core's output, and the corrupt data was overwritten during the fault injection run, masking the fault.

Figure 12: Fault Injection Error Types Statistics.

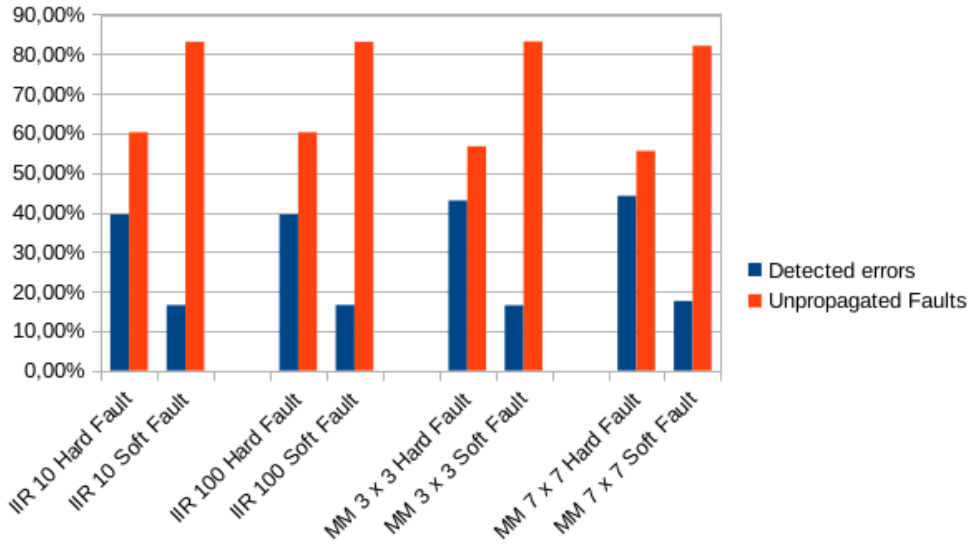


Source: From the author.

6.2.2 Application Cases Analysis

The results collected from fault injection campaigns in the application cases is show in this subsection. Figure 13 a compilation of all application cases used to validate the DCLS system. The overall results are similar, despite being shown in distinct application types and size. All soft fault injection campaigns presented a small error propagation, and therefore error detection by the DCLS system, if compared with the hard fault injection type.

Figure 13: Fault Injection Application Cases Results.



Source: From the author.

As the proposed DCLS architecture in this work is designed in a tightly-coupled fashion, all faults that propagate to the core's output buses will be detected and the system will enter a safe state. All detected errors during the fault injection run resulted in the stopping of the DCLS operation and wait until the external system control signal to trigger a context backup or application reset. The system can be considered safe due to the detection of all visible errors, as shown in the results in this section, and in addition to the fact that all modules where designed with one-hot codification, increasing the system fault tolerance factor. Though affected registers by SEE that do not propagate, they will either be present in the system until a masking or a signal propagation occurs.

Other authors implement DCLS systems in a loosely-coupled fashion, where the system's output is compared asynchronously, thus opening space for posterior error detection. In the work done in (OLIVEIRA et al., 2018), an ARM Cortex-A9 hardcore dual-core lockstep in a Xilinx Zync FPGA is implemented, using a loosely-coupled fashion. The lockstep system compared the system's output only after both cores executed the target application. Up to 78% of propagated errors were detected by the system during fault-injection emulation runs. The design fault-tolerance increased by 80%. The results shown by Abate et al. (OZER et al., 2018), which implemented a dual-core lockstep with an ARM CORTEX-R5, indicates that the design detected 100% of the observable errors during the fault-injection campaign.

The discrepancy between soft and hard faults manifestation is similar to the literature. In (OZER et al., 2018), the manifestation of transient errors in the outputs of the system was about 5%, while the rate of permanent errors was 40%. In this work, the transient error manifestation was about 16%, while the permanent errors manifested at a rate of about 40%. Although the discrepancy was greater in the related work, it shows the distinct quality between transient and permanent errors manifestation.

Another important topic is the common-mode error, where both cores experience the same fault at the same register at the same time, propagating the common error and producing a false negative at the DCLS comparator interface (FLORIDIA; SANCHEZ, 2018). This issue can be solved by adding delay registers in the design, in different core's inputs and outputs, or changing the physical implementation of the design in the ASIC or FPGA, changing then the common exposed area. To increase further the system reliability, a watchdog may be added in the design, and may be configured to trigger a context restore if the DCLS does not respond within the target time limit.

7 CONCLUSION

In this work, a DCLS system was presented. An architecture, functional specification was shown. The DCLS is presented as a technique to increase a system's fault tolerance capability, offering a lower hardware overhead if compared with other fault-tolerant solutions, such as triple modular redundancy. Although it doesn't indicate which core originated the fault, making necessary another fault correction technique, its low overhead and cost can be a great addition to a project, depending on its specification and budget. The usage of open soft-cores opens a plethora of possibilities, due to its capability of creating customized applications, which can be integrated into safer applications, and increase the system's value. Although the system was designed with the RI5CY as target core, it can be adapter to other soft-core implementations, opening areas of study for future researchers.

A fault injection framework was proposed, to allow system validation and check the DCLS fault tolerance detection and response. A fault injection flow and an error type definition to create metrics upon the injected fault also was presented. An open-source tool was proposed as the simulation environment both for the DCLS architecture as the proposed fault injection framework. The DCLS system was tested under a fault injection environment with DSP applications, a matrix multiplication, and an IIR filter. The fault injection was evaluated and its results explored, showing the discrepancy between fault propagation between soft and hard faults.

The DCLS system proved capable of detecting faults that propagate at the core's output as errors, and responded with a good time response at every observable error. The design implementation has a very small area overhead, reaching up to 5.18% of LUT utilization compared to a single RI5CY core. Though its maximum frequency decreased 18.5% compared to the unhardened design, and the addition of checkpoints to perform context backups may result in performance penalty, depending on the number of added checkpoints and application size.

The proposed DCLS system reached its objective defined in the beginning of this work, as the system was capable of detecting the observable errors at the system output, as validated through simulation and fault injections. It is capable to enable a fault-tolerant system within a FPGA or ASIC design implementation. The area overhead is lightweight, and its clock timing reduction reached a tolerable factor. The software overhead was analyzed, where the application size and checkpoints number is a critical factor to determine overall performance penalty.

7.1 Future Work

As future works, the assessment of the fault susceptibility of the RI5CY core can be evaluated, exploring the architecture processing capability, using the proposed DCLS architecture and the fault injection framework. Future researches may also include RTOS fault tolerance analysis using the proposed system in this work, including routines, multiple threads and operating system features that may increase the system's vulnerability. A memory management may also be implemented as future work, to reduce the software performance penalty when executing smaller applications. A more complete statistical analyses of fault

injection may also be performed in future works, showing statistical distribution and parameters with a more insightful investigation.

Little modification is necessarily to use 64-bit versions of the RISCV ISA in the DCLS architecture designed in this work, and a fault-tolerance comparison can be made with the 32-bit version. Future researches may develop a system reconfiguration routine module for FPGA applications, and integrate it with the DLCS system shown in this work. A TMR version with the same core as used in this work may also be done, to compare the performance and overhead penalties. A fault injection using radiation with an implemented DCLS system in an FPGA can be performed and its results compared with the ones shown in this work.

In the application part, a software built-in-self-test (SBIST) design is proposed as future work, where the target SBIST may detect which core originated the fault, by forcing all its internal registers to be exposed to a system controller. The fault correction may then be evaluated re-implementing only the detected faulty core, and its performance can be analyzed compared to the traditional fault correction approach, where the whole system is reconfigured.

REFERENCES

ABATE, F. et al. **New techniques for improving the performance of the lockstep architecture for SEEs mitigation in FPGA embedded processors**. IEEE Transactions on Nuclear Science, v. 56, n. 4, p. 1992–2000, 2009.

AGGARWAL, N.; RANGANATHAN, P. **Configurable isolation: building high vailability systems with commodity multi-core processors**. ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture, p. 470–481, 2007.

AHMAD, T.; CIESIELSKI, M. **Parallel multi-core verilog hdl simulation using domain partitioning**. 2014 IEEE Computer Society Annual Symposium on VLSI, p. 619–624, 2014.

AMUTHA, C.; RAMYA, M.; SUBASHINI, C. **A novel co-design approach for soft error mitigation for embedded system.** International Conference on Emerging Trends in Electrical Engineering and Energy Management, v. 58, n. 3, p. 267–270, 2012.

ARLAT, J. **Validation de la sûreté de fonctionnement par injection de fautes: méthode, mise en oeuvre, application**. 190 f. p. Tese (Doutorado), 1990. Thèse de doctorat dirigée par Laprie, Jean-Claude Informatique Toulouse, INPT 1990. Disponível em: <http://www.theses.fr/1990INPT094H>. Acesso em: 07 aug. 2019.

ASADI, G.; TAHOORI, M. B. **Soft error rate estimation and mitigation for SRAM-based FPGAs**. Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, p. 149-160, 2005.

AZAMBUJA, J. R. et al. **Detecting SEEs in microprocessors through a non-intrusive hybrid technique**. IEEE Transactions on Nuclear Science, v. 58, n. 3 PART 2, p. 993–1000, 2011.

BAUMANN, R. C. **Radiation-induced soft errors in advanced semiconductor technologies**. IEEE Transactions on Device and Materials Reliability, v. 5, n. 3, p. 305–315, 2005.

BENSO, A. et al. **Software dependability techniques validated via fault injection experiments**. Radiation and Its Effects on Components and Systems, v. 6, p. 269–274, 2005.

BRESSOUD, T. C.; SCHNEIDER, F. B. **Hypervisor-based Fult-torelance**. Proceedings of the fifteenth ACM symposium on Operating systems principles, p. 1-11, 1995.

CHENG, E. et al. CLEAR: **Crosslayer exploration for architecting resilience combining hardware and software techniques to tolerate soft errors in processor cores**. Proceedings – Design Automation Conference, v. 53, p. 1-6, 2016.

CHIELLE, E. et al. **Hybrid soft error mitigation techniques for COTS processor-based systems**. LATS 2016 - 17th IEEE Latin-American Test Symposium, IEEE, p. 99–104, 2016. Citado na página 14.

CHO, H. **Impact of Microarchitectural Differences of RISC-V Processor Cores on Soft Error Effects**. IEEE Access, IEEE, v. 6, p. 41302–41313, 2018.

CHONNAD, S.; IACOB, R.; LITOVTCHENKO, V. **A quantitative approach to soc functional safety analysis**. In: 2018 31st IEEE International System-on-Chip Conference (SOCC). [S.l.: s.n.]. p. 197–202, 2018.

COBHAM. **UT699E 32 - bit Fault-Tolerant SPARC TM V8/LEON 3FT Processor. 2019**. Disponível em: <https://www.cobhamaes.com/pagesproduct/datasheets/leon/UT699E_LEON3FT_Datasheet.pdf>. Acesso em: 07 aug. 2019.

CONTI, F. et al. **A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision**. Journal of Signal Processing Systems volume, v. 84, p. 339–354, 2016.

DODD, P. E.; MASSENGILL, L. W. **Basic mechanisms and modeling of single-event upset in digital microelectronics**. IEEE Transactions on Nuclear Science, v. 50, n. 3, p. 583–602, June 2003.

FLORIDIA, A.; SANCHEZ, E**. Hybrid on-line self-test strategy for dual-core lockstep processors**. 2018 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), p. 1–6, 2018.

GHAHROODI, M. M.; OZER, E.; BULL, D. **SEU and SET-tolerant ARM Cortex-R4 CPU for Space and Avionics Applications**. Second Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale. p. 4–7, 2013.

IEC. **IEC 61508 Functional Safety Standard**. 2020. Disponível em: <https://www.iec.ch/functionalsafety/>. Acesso em: 01 jul 2020.

INFINEON. **Infineon Tricore: Highly Integrated and Performance Optimized 32-bit Microcontrollers for Automotive and Industrial Applications**. Disponível em: <https://www.infineon.com/dgdl?fileId=5546d46153ac54890153c1a41ffe0000>. Acesso em: 07 ago. 2019. Citado na página 21.

ITURBE, X.; VENU, B.; OZER, E. **Soft error vulnerability assessment of the real-time safety-related arm cortex-r5 cpu**. p. 91–96, Sep. 2016.

ITURBE, X. et al. **A Triple Core Lock-Step (TCLS) ARM R Cortex R-R5 Processor for Safety-Critical and Ultra-Reliable Applications**. Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN-W 2016, n. October, p. 246–249, 2016.

JEFFERY, C. M.; FIGUEIREDO, R. J. **A flexible approach to improving system reliability with virtual lockstep**. IEEE Transactions on Dependable and Secure Computing, IEEE, v. 9, n. 1, p. 2–15, 2012.

KASTENSMIDT, F. L.; FILHO, C. K.; CARRO, L. **Improving reliability of sram-based fpgas by inserting redundant routing**. IEEE Transactions on Nuclear Science, v. 53, n. 4, p. 2060–2068, 2006.

KENTERLIS, P. et al. **A low-cost SEU fault emulation platform for SRAM-based FPGAs**. In: Proceedings - IOLTS 2006: 12th IEEE International On-Line Testing Symposium. [S.l.: s.n.], p. 235–241, 2006.

KESH, M. E.; ASAMI, K. **Fault injection in dynamic partial reconfiguration**. v. 11, n. 8, p. 25–33, June 2018.

MANSOUR, W.; VELAZCO, R. **Seu fault-injection in vhdl-based processors: A case study**. 2012 13th Latin American Test Workshop (LATW), p. 1–5, 04 2012.

NORMAND, E. **Single-event effects in avionics**. IEEE Transactions on Nuclear Science, v. 43, n. 2 PART 1, p. 461–474, 1996.

OLIVEIRA Ádria B. de et al. **Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors**. IEEE Transactions on Nuclear Science, v. 65, n. 8, p. 1783–1790, 2018.

OZER, E. et al. **Error correlation prediction in lockstep processors for safety-critical systems**. In: Proceedings of the Annual International Symposium on Microarchitecture, p. 737–748, 2018.

PETRISKO, D. et al. **Blackparrot: An agile open-source risc-v multicore for accelerator socs**. IEEE Micro, v. 40, n. 4, p. 93–102, 2020.

RENNER, C. **Software implemented hardware fault tolerance increasing software dependability with recco**: **An evaluation.** 2003.

RESCH, S.; STEININGER, A.; SCHERRER, C. **Software composability and mixed criticality for triple modular redundant architectures**. SAFECOMP 2013 - Workshop SASSUR (Next Generation of System Assurance Approaches for Safety-Critical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security, 2013.

RODRIGUES, C. et al. **Towards a heterogeneous fault-tolerance architecture based on arm and risc-v processors**. In: IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society. [S.l.: s.n.], 2019. v. 1, p. 3112–3117.

SABBELLA, R.; ARUNACHALAM, M. **Functional safety development of motor control unit for electric vehicles**. In: 2019 IEEE Transportation Electrification Conference (ITEC-India). [S.l.: s.n.], p. 1–6, 2019.

SAGGESE, G. et al. **An experimental study of soft errors in microprocessors**. IEEE Micro, v. 25, p. 30–39, 11 2005.

SEIFERT, N. et al. **Soft error susceptibilities of 22 nm tri-gate devices**. IEEE Transactions on Nuclear Science, v. 59, n. 6, p. 2666–2673, Dec 2012.

SEMICONDUCTOR Research Corp. Nat. **Technology Roadmap**. 1999.

TAMBARA, L. A. et al. **Heavy ions induced single event upsets testing of the 28 nm Xilinx Zynq-7000 all programmable SoC**. IEEE Radiation Effects Data Workshop, IEEE, v. 2015-Novem, p. 1–6, 2015.

TIWARI, M. et al. **Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security**. ACM SIGARCH Computer Architecture News, v. 39, n. 3, p. 189, 2011. ISSN 01635964. Citado na página 23.

VALDERAS, M. G. et al. **Two complementary approaches for studying the effects of SEUs on digital processors**. IEEE Transactions on Nuclear Science, v. 54, n. 4, p. 924–928, 2007.

VELAZCO, R.; FAURE, F. **Error rate prediction of digital architectures: Test methodology and tools**. Radiation Effects on Embedded Systems, 01 2007.

VELAZCO, R.; REZGUI, S.; ECOFFET, R. **Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code Emulating Upsets) injection**. IEEE Transactions on Nuclear Science, v. 47, n. 6 III, p. 2405–2411, 2000.

VENU, B.; OZER, E.; ROBINSON, X. I. A. **A Fail-Functional Automotive CPU Subsystem Architecture for Mitigating Single Point of Failures**. IEEE International Workshop on Automotive Reliability & Test, 2016.

WALKINGTON, J.; SUGAVANAM, S.; NUNNS, S. **One approach to functional safety assurance and safety lifecycle compliance**. In: 8th IET International System Safety Conference incorporating the Cyber Security Conference 2013. [S.l.: s.n.] v. 1, p. 1007, 2007.

WALLMARK, J. T.; MARCUS, S. M. **Minimum size and maximum packing density of nonredundant semiconductor devices**. Proceedings of the IRE, v. 50, n. 3, p. 286–298, March 1962.

WANG, N. et al. **Characterizing the effects of transient faults on a high-performance processor pipeline**. International Conference on Dependable Systems and Networks. p. 61–70, 2004.

ZIADE, H.; AYOUBI, R.; VELAZCO, R. **A survey on fault injection techniques**. Int. Arab J. Inf. Technol., v. 1, p. 171–186, 01 2004.

APPENDIX A — PUBLICATIONS

VIANA, R.; SILVA, M.; BARBOSA, J. L.V. **Design of a RISC-V Dual-core Lockstep for Fault-tolerant Systems.** Workshop on Circuits and Systems Design (WCAS), n. 10, 2020. (Accepted)