Rodrigo Travessini

LOW OVERHEAD SINGLE EVENT UPSET RELIABILITY IMPROVEMENT FOR SOFT CORE PROCESSORS

Dissertação submetida ao Programa de Pós Graduação em Engenharia Elétrica para a obtenção do Grau de Mestre em Engenharia Elétrica.
Orientador: Prof. Eduardo Augusto Bezerra, PhD

Florianópolis

2018

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Travessini, Rodrigo
Low overhead single event upset reliability improvement for soft core processors / Rodrigo Travessini ; orientador, Eduardo Augusto Bezerra, 2018.
92 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós Graduação em Engenharia Elétrica, Florianópolis, 2018.

Inclui referências.

1. Engenharia Elétrica. 2. Processadores Soft Core. 3. Tolerância a Falhas. 4. Injeção de Falhas. 5. Single Event Upset. I. Bezerra, Eduardo Augusto. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

Rodrigo Travessini

LOW OVERHEAD SINGLE EVENT UPSET RELIABILITY IMPROVEMENT FOR SOFT CORE PROCESSORS

Esta Dissertação foi julgada aprovada para a obtenção do Título de “Mestre em Engenharia Elétrica”, e aprovada em sua forma final pelo Programa de Pós Graduação em Engenharia Elétrica.

Florianópolis, 07 de abril 2018.

[illegible]

Prof. Marcelo Lobo Heldwein, Dr.
Coordenador do Curso
Universidade Federal de Santa Catarina

Este trabalho é dedicado a todos que contribuíram direta ou indiretamente na minha formação acadêmia.

AGRADECIMENTOS

Primeiramente agradeço aos meus pais que nunca mediram esforços para que eu pudesse me dedicar aos estudos e possibilitaram todo o suporte necessário durante essa caminhada. Em igual proporção agradeço ao meu irmão, a quem eu admiro imensamente e por quem nutro grande amizade. Quero agradecer também a minha namorada Soraya pelo companheirismo, amor e compreensão ao longo de todo o mestrado. Expresso meu agradecimento ao Prof. Eduardo Bezerra pela oportunidade de trabalhar nessa área tão fascinante da engenharia. Por fim, estendo meus agradecimentos a todos os meus colegas e amigos Daniel Severo, Eduardo Schmidt, Frank Kair, Giulia Ciprandi, João Tomazoni, Michelle Medeiros, Paulo Villa, Pedro Rigotti, Ramon Balthazar, Renan Goulart, Romano Weirich e a todos os outros que de alguma forma contribuíram nessa jornada.

RESUMO

Os processadores utilizados em aplicações embarcadas são componentes indispensáveis na eletrônica de satélites. Eles são empregados para lidar com uma variedade de tarefas, incluindo processamento de dados, tratamento de comandos, além do próprio controle do satélite. No entanto, como qualquer outro componente eletrônico utilizado no meio espacial, a exposição à radiação ionizante pode induzir diversos efeitos indesejados, dentre os quais as falhas transientes são os mais recorrentes. Para superar esse problema, os processadores utilizados em aplicações espaciais são geralmente fabricados por meio de técnicas não convencionais que oferecem menor suscetibilidade a danos por radiação. Infelizmente, a produção de chips tolerante a radiação requer longos períodos de teste e desenvolvimento, o que os torna mais caros que os processadores comerciais. Frente a esta limitação, processadores soft core surgem como uma abordagem alternativa, apresentando menor custo e a possibilidade de integrar técnicas de tolerância a falha para atingir os níveis de confiabilidade requeridos por aplicações espaciais. Todavia, as abordagens clássicas para prover tolerância a falhas em processadores soft core envolvem penalidades significativas em termos de área e desempenho. Neste contexto, o presente trabalho apresenta uma estratégia de tolerância a falhas de baixo custo, na qual apenas as partes mais vulneráveis do prcessador são protegidas. A identificação destas partes é baseada nos resultados de uma extensa campanha de injeção de falhas. Quando comparada a técnicas do estado da arte, como a de redundância modular tripla (TMR), a estratégia proposta resultou eficiente ao fornecer níveis similares de tolerância a falhas a um custo de área inferior.

Palavras-chave: Processadores Soft Core. Tolerância a Falhas. Injeção de Falhas. *Single Event Upset.* Falhas Transientes. LEON3.

RESUMO EXPANDIDO

1 INTRODUÇÃO

Os processadores embarcados são amplamente utilizados em aplicações espaciais e correspondem a um componente indispensável na eletrônica de satélites. Eles são empregados para lidar com uma variedade de tarefas, incluindo processamento de dados, tratamento de comandos, além do próprio controle do satélite.

No entanto, o uso de qualquer componente eletrônico no rigoroso ambiente espacial exige cuidados especiais. A exposição à radiação ionizante pode ser nociva à dispositivos desprotegidos, causando diversos efeitos indesejáveis. Um dos efeitos induzidos por radiação mais recorrentes são as falhas transientes (*soft errors*), que ocorrem sempre quando um evento radioativo inverte o dado armazenado em uma célula de memória, registrador, *latch*, ou *flip-flop*. A falha é considerada transiente pois o circuito/dispositivo não é danificado permanentemente pela radiação, e caso novos dados sejam escritos no local afetado, o dispositivo irá armazena-los corretamente.

Para mitigar os efeitos radioativos, os processadores utilizados em aplicações espaciais são geralmente fabricados por meio de técnicas não convencionais que oferecem menor suscetibilidade a danos por radiação. Infelizmente, a produção de chips tolerantes a radiação requer longos períodos de teste e desenvolvimento, o que os torna mais caros que os processadores comerciais.

Dadas as limitações de custo e desempenho dos processadores tolerantes a radiação, existe um crescente interesse em utilizar componentes comerciais de prateleira (COTS) em aplicações espaciais. Neste contexto, processadores soft core com código aberto surgem como uma alternativa atraente, apresentando menor custo, além de manter a possibilidade de integrar técnicas de tolerância a falha para atingir os níveis de confiabilidade desejados.

Uma das técnicas de tolerância a falhas mais amplamente empregadas para proteger processadores é a redundância modular tripla (TMR). Quando implementada corretamente, esta técnica possibilita o mascaramento de qualquer erro único, porém com um incremento de área e energia que pode ultrapassar 200%. Considerando as restrições de área e energia, especialmente no caso de nanossatélites, os custos de implementar TMR em todo o processador podem ser proibitivos. Nesse sentido, são de grande interesse técnicas que empreguem a redundância somente nas partes mais vulneráveis do processador (redundância parcial), tornando possível, assim, atender os níveis de confiabilidade exigidos, sem comprometer outras restrições.

Para o melhor emprego de redundância parcial, é essencial compreender como as falhas induzidas pela radiação se manifestam no processador alvo, e de que forma elas se propagam até as suas interfaces. Tal informação pode ser obtida através da realização de experimentos de injeção de falhas, que por sua vez é considerado um método importante para avaliar a confiabilidade de um sistema em teste.

Como estudo de caso é investigado o processador LEON3 da Cobham Gaisler, que é distribuído em código aberto sob a licença GNU GPL, e tem como foco principal aplicações espaciais críticas.

2 OBJETIVOS

O objetivo principal deste trabalho é apresentar o desenvolvimento de uma técnica de tolerância a falhas de baixo custo, visando mitigar falhas transientes em processadores soft core. A técnica é baseada no conceito de redundância parcial, no qual apenas as partes mais vulneráveis do processador são protegidas, obtendo assim um equilíbrio entre confiabilidade e outras restrições de projeto. Como forma de identificar os locais ideais para empregar a redundância, é conduzida uma extensa campanha de injeção de falhas no processador alvo.

3 METODOLOGIA

Em um primeiro momento foi realizado um estudo acerca do processador LEON3, que teve como objetivos: compreender a estrutura do pipeline do processador; definir uma configuração para o processador a ser utilizada neste trabalho (por exemplo, tamanho de cache, inclusão de módulos opcionais); e entender como a implementação VHDL está organizada. Tais informações foram fundamentais para as etapas restantes e direcionaram algumas das escolhas feitas durante a implementação da plataforma de injeção de falhas e da técnica de redundância parcial.

A etapa seguinte abrangeu o desenvolvimento da plataforma de injeção de falhas. Começando pela definição da estratégia de injeção, e seguindo com a implementação propriamente dita. Também fez parte desta etapa, a definição dos modelos de falhas e a descrição dos efeitos de falha esperados.

Depois de desenvolvida a plataforma de injeção de falhas, o próximo passo foi conduzir a campanha de injeção no processador LEON3. A partir desses experimentos foi possível extrair uma lista dos componentes mais vulneráveis no processador. Com base nos resultados desta etapa, fora implementada a técnica de redundância parcial.

Finalmente, o último passo foi avaliar o desempenho da técnica proposta. A avaliação teve dois momentos: quantificar a melhoria na confiabilidade do processador através de uma nova campanha de injeção de falhas e determinar os custos de área e desempenho sintetizando o processador em um FPGA.

4 RESULTADOS E DISCUSSÃO

Para realização da campanha de injeção de falhas decidiu-se utilizar uma técnica baseada em simulação. Essa escolha foi ao encontro de trabalhos anteriores, que também optaram pelo mesmo método. Essa técnica mostrou-se satisfatória, considerando que permitiu a execução da maioria dos testes desejados. A principal limitação encontrada, que já havia sido relatada em outros estudos, foi o tempo de simulação.

Os resultados experimentais da campanha conduzida no LEON3 sem nenhuma técnica de tolerância a falhas empregada revelaram que a maioria das falhas que levaram o processador a um comportamento errôneo estavam limitadas a um pequeno grupo de registradores, entre eles o contador de programa e os operandos da ULA. Além disso, foi descoberto que apenas um terço das falhas injetadas se propagou para as interfaces do processador. Esses resultados apresentaram evidências substanciais de que uma melhoria na confiabilidade era possível protegendo apenas as partes mais vulneráveis do processador.

Posteriormente, com a técnica de redundância parcial implementada no LEON3, uma nova campanha de injeção de falhas mostrou que protegendo apenas 30 registradores de um total de 362, já era possível ter uma redução de seis vezes no número de falhas que levaram a efeitos nocivos. Ao mesmo tempo, os custos em termos de utilização de recursos (por exemplo, registradores e LUTs) do FPGA foram muito inferiores a uma triplicação completa (variado entre 3 e 12 vezes dependendo do recurso em questão). É necessário notar, no entanto, que a técnica proposta não é ideal para todas as aplicações, e de forma alguma substitui a técnica clássica TMR quando índices maiores de tolerância a falha são exigidos.

ABSTRACT

Embedded processors are an essential component in most satellite electronics. They are employed to handle a variety of functions including data processing, command handling, and satellite control. However, like any other electronic component used in space, the exposure to radiation may induce many undesired effects, such as soft errors. To overcome this issue, the processors used in space applications are usually manufactured through non-conventional techniques that provide reduced susceptibility to radiation damage and thus are known as rad-hard. Unfortunately, the production of rad-hard chips requires extensive development and testing, making them more expensive and slower than commercial parts. Soft core processors appear as an alternative approach, with lower cost, and the possibility to implement hardening by design strategies to achieve the required dependability levels of space applications. Nevertheless, classical approaches for providing fault tolerance in soft core processors involve significant area and performance costs. In this context, the present work introduces a low overhead fault tolerance strategy which protects only the most vulnerable parts of the processor. The identification of these parts is based on the results of an extensive fault injection campaign, also conducted in this work. When compared to state-of-the-art techniques such as triple modular redundancy (TMR), the proposed strategy proved to be quite efficient, providing similar levels of fault tolerance with a much lower area cost.

Keywords: Soft Core Processors. Fault Tolerance. Fault Injection. Single Event Uptset. Soft Error. LEON3.

LIST OF FIGURES

Figure 1 Energetic particle strike in a silicon device. 33
Figure 2 Single Event Upset in SRAM 34
Figure 3 SET latching under different time scenarios. 35
Figure 4 Effects of SEUs and SETs in the FPGA architecture . . . 36
Figure 5 Triple Modular Redundancy concept 38
Figure 6 Triple Modular Redundancy applied to sequential logic 39
Figure 7 Full Triple Modular Redundancy 40
Figure 8 Time Redundancy concept 41
Figure 9 Signal probability computation at the output of a boolean gate 46
Figure 10 Basic flow of the BYU-LANL Partial TMR (BLTmr) tool 47
Figure 11 A standard compiler flow augmented with Shoestring's reliability-aware code generation passes 48
Figure 12 Emulation-based fault injection platform used in (REBAUDENGO et al., 2003) 49
Figure 13 Simulation-based fault injection platform used in (TOULOUPIS et al., 2007) 50
Figure 14 Dependability terminology used in (ITURBE et al., 2016) 51
Figure 15 Methodology 53
Figure 16 LEON3 Architecture with Main Peripherals 55
Figure 17 LEON3 Integer Unit Datapath 57
Figure 18 LEON3 Processor Core (PROC3) 58
Figure 19 Fault Injection Environment 62
Figure 20 Dynamic Instruction Mix 64
Figure 21 TCL Procedure that traverses a root signal and lists all the internal signals bitwise 66
Figure 22 The arrangement used to monitor fault propagation in the PROC3 interfaces 67
Figure 23 TCL Procedure used for generating a bitflip in a flip-flop 69
Figure 24 Example logs generated by the Fault Injection Manager 70
Figure 25 Steps for implementing the PTMR technique 73
Figure 26 Overall PROC3 Performance 76
Figure 27 Overall Performance by Workload 76

Figure 28 Overall Performance by Module........................ 77
Figure 29 IU3 Performance Split by Pipeline Stage............... 78
Figure 30 Individual Register Performance....................... 79
Figure 31 Fault Propagation in PROC3 interfaces................. 80
Figure 32 Boxplot of the Fault Manifestation Time in PROC3 interfaces. The bottom and top of the box corresponds to the first and third quartiles, the line inside the box is the median, the whiskers are at 1.5 IQR, values outside of that range are represented by dots. 81
Figure 33 Expected SBU Tolerance 82
Figure 34 Protected LEON3 Overall Performance 82
Figure 35 Triple Modular Redundancy with SET detection....... 87

LIST OF TABLES

Table 1 LEON3 Configuration 60
Table 2 Design Maximum Frequency 83
Table 3 FPGA Resources Utilization 83
Table 4 FPGA Power Consumption 84

LIST OF ABBREVIATIONS

ADC	Analog-to-digital Converter
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application Specific Integrated Circuit
COTS	Commercial off-the-shelf
CRAM	Configuration Random Access Memory
CSV	Comma-separated Values
DMR	Dual Modular Redundancy
DSP	Digital Signal Processor
DSU	Debug Support Unit
DUT	Design Under Test
DWC	Duplicate with Comparison
ECC	Error-Correcting Code
EDC	Error-Detecting Code
ESA	European Space Agency
ESTEC	European Space Research and Technology Centre
FLI	Foreign Language Interface
FPGA	Field Programmable Gate Array
GL	Gate Level
GRLIB	Gaisler Research IP Library
GUI	Graphical User Interface
HDL	Hardware Description Language
IP	Intellectual Property
LRU	Least Recently Used
LUT	Lookup Table
MBU	Multiple Bit Upset
MMU	Memory Management Unit
PTMR	Partial Triple Modular Redundancy
RTL	Register-Transfer Level
SBU	Single Bit Upset
SEE	Single Event Effect
SET	Single Event Transient
SEU	Single Event Upset

SPARC	Scalable Processor Architecture
SRAM	Static Random Access Memory
STMR	Selective Triple Modular Redundancy
TCL	Tool Command Language
TMR	Triple Modular Redundancy
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit

CONTENTS

1 INTRODUCTION 25
1.1 OBJECTIVES 27
1.1.1 General Objectives 27
1.1.2 Specific Objectives 28
1.2 PUBLICATIONS 28
1.2.1 Journal Paper 28
1.2.2 Conference Papers 28
1.3 DOCUMENT STRUCTURE 29
2 BACKGROUND 31
2.1 FAULTS, ERRORS, AND FAILURES 31
2.2 RADIATION EFFECTS ON ELECTRONIC DEVICES .. 32
2.2.1 Single Event Upset — SEU 33
2.2.2 Single Event Transient — SET 33
2.3 SOFT ERRORS IN FPGAS 35
2.4 FAULT TOLERANCE TECHNIQUES 37
2.4.1 Hardware Redundancy 38
2.4.2 Information Redundancy 40
2.4.3 Time Redundancy 41
2.4.4 Software Redundancy 41
2.5 FAULT INJECTION 42
3 RELATED WORK 45
3.1 LOW OVERHEAD FAULT TOLERANCE TECHNIQUES 45
3.2 SOFT ERROR VULNERABILITY ANALYSIS 49
4 METHODOLOGY AND DEVELOPMENT 53
4.1 METHODOLOGY 53
4.2 LEON3 SOFT CORE PROCESSOR 54
4.2.1 Pipeline Structure 56
4.2.2 VHDL Structure and Implementation 58
4.2.3 Configuration 60
4.3 FAULT INJECTION PLATFORM 61
4.3.1 Fault Model 62
4.3.2 Fault Effects Classification 63
4.3.3 Workload Description 64
4.3.4 Implementation Details 65
4.3.4.1 List Design Flip-Flops 65
4.3.4.2 Fault Propagation Monitoring 66
4.3.4.3 Fault Effect Monitoring 67

4.3.4.4 Fault Injection 68
4.3.4.5 Log Generation 70
4.4 PARTIAL REDUNDANCY TECHNIQUE 71
5 RESULTS AND DISCUSSION 75
5.1 FAULT INJECTION IN THE UNPROTECTED LEON3 . 75
5.1.1 Overall Performance 75
5.1.2 Integer Pipeline Performance 77
5.1.3 Individual Register Performance 78
5.1.4 Fault Propagation 79
5.2 FAULT INJECTION IN THE PROTECTED LEON3 81
5.2.1 Overall Performance 81
5.2.2 Synthesis Area/Performance Overhead 82
6 CONCLUSIONS 85
6.1 FUTURE WORKS 86
REFERENCES 89

1 INTRODUCTION

Embedded processors are extensively used in space applications, and correspond to an essential component in satellite electronics. They are employed to handle a variety of functions including data processing, command handling, and satellite control.

Nevertheless, the use of any electronic component in the harsh space environment demands particular attention. Exposure to ionizing radiation can be harmful to unprotected devices, causing many undesired effects. Especially in the case of modern devices, the ever-shrinking dimensions of manufacturing technologies and the lower supply voltages lead to an increased susceptibility to those effects, expanding their reach even to terrestrial altitudes (GORDON et al., 2004).

One of the most common radiation-induced effects is the soft error (BAUMANN, 2004), which occurs when the radiation event reverses the data state of a memory element (e.g., register, latch, flip-flop). In contrast with hard errors, which represent real circuit errors that force the circuit to fail repeatedly in the same manner, soft errors consist of only data corruption without hardware damage, meaning that the affected memory element can be corrected, by overwriting the cell with the original value.

Even though soft errors do not involve hardware damage, they are no lesser threat if not detected and correctly treated. Many applications are considered safety critical and thus have very rigid dependability requirements. In such applications, any malfunction may have severe adverse effects. For example in the case of space applications, non treated soft errors may lead to a premature end of a satellite mission, resulting in significant economic and scientific losses.

In view of these effects, space applications traditionally only employed radiation hardened (rad-hard) processors. These are components produced through non-conventional processing techniques that provide reduced susceptibility to radiation-induced damage. Unfortunately, due to the extensive development and testing required to produce rad-hard chips, these are expensive parts and lag commercial devices by several technology generations (approx. 10 years) (KEYS et al., 2008).

Given the cost and performance limitations of rad-hard processors, there is a growing interest in using Commercial off-the-shelf (COTS) components in space applications. The motivation is even stronger, in the case of some space programmes, such as the Brazilian, that encounters restrictions in the acquisition process of rad-hard components in view of export license limitations [1].

In this context, soft-core processors appear as an attractive alternative by providing processor-specific customization, the possibility to include custom reliability techniques, and the flexibility to integrate with customized Field Programmable Gate Array (FPGA) logic (TONG et al., 2006). Moreover, the FPGAs themselves have several appealing features, including a significant number of configurable logic, reasonable operating frequencies, and a plethora of embedded hard-blocks (such as Analog-to-digital Converters (ADCs) and Digital Signal Processors (DSPs)). As a result, several satellite missions have been adopting soft core processors implemented in FPGAs (KLETZING et al., 2013; MARKIEWICZ et al., 2004).

However, as most COTS FPGAs are not radiation hardened, fault tolerance strategies must be employed to improve the system reliability in the space environment. In the case of the memory hierarchy (e.g., register file, caches and main memory), there are already well-established techniques, usually based on using Error-Detecting Codes (EDCs) and Error-Correcting Codes (ECCs) (HAMMING, 1950; CHEN; HSIAO, 1984). When dealing with soft errors in computational or control logic, the most common approach is the use of Triple Modular Redundancy (TMR) (LYONS; VANDERKULK, 1962). This approach consists of instantiating three replicas of the original design and a majority voter. When implemented correctly it provides single error masking and double error detection, yet the area and power overhead may reach over 200%.

Considering area and energy constraints, especially in the case of nanosatellites, the costs of implementing a traditional redundancy-based fault tolerance strategy in the entire design could be prohibitive. In this sense, techniques that selectively employ the redundancy in only the most vulnerable areas of a design are of great interest, making it possible to achieve the required dependability levels, without compromising others constraints.

There are already some works in the literature, such as (SAMUDRALA et al., 2004; PRATT et al., 2006), that propose partial redundancy approaches, also known as selective redundancy approaches, as an affordable fault tolerance strategy for resource-constrained designs.

The control logic of embedded processors provide an excellent

[1]For more information see the International Traffic in Arms Regulations (ITAR) (22 C.F.R. Parts 120–130) and the Export Administration Regulations (EAR) (15 C.F.R. Parts 730–774)

opportunity to explore the use of partial redundancy since they usually are very elaborate designs that implement hundreds of instructions, with many of them never being executed, or executed very sparsely in common workloads. Therefore possibly indicating that large parts of the design are underutilized and may not require full redundancy in some applications.

In order to take advantage of partial redundancy, more in-depth knowledge of how radiation-induced faults manifest in the target processor, and how they propagate to its boundaries is fundamental. Such information is obtainable through fault injection, which in turn is considered an important method for assessing the dependability of a system under test (ZIADE et al., 2004).

ARM Research did a very broad study (ITURBE et al., 2016) in this regard, conducting an intensive fault injection campaign in the ARM Cortex-R5 CPU core. They concluded that only 10% of the sequential elements in the Cortex-R5, accounts for more than 70% of the errors, suggesting that an important reliability improvement can be obtained by protecting just these most sensitive components.

As a case study, this work investigates the LEON3 soft core processor from Cobham Gaisler (COBHAM GAISLER AB, 2017), which was developed targeting critical space applications, supported by the European Space Agency (ESA). The LEON3 is described using the VHDL description language, and its source code is freely available for research and educational use under the GNU GPL license.

1.1 OBJECTIVES

1.1.1 General Objectives

Given the preceding, the primary objective of this work is to present the development of a low overhead fault tolerance technique aimed at mitigating soft errors in the control logic of soft-core processors. The technique is based on the concept of partial redundancy, where only the most vulnerable parts of the processor are protected, hence obtaining a balance between dependability and other design constraints. As a means to identify the optimal locations for employing the redundancy, an extensive fault injection campaign is conducted in the target processor.

1.1.2 Specific Objectives

In addition, the following specific objectives were established for this work:

- Investigate the radiation effects on electronic components;
- Investigate the LEON3 micro-architecture and its VHDL implementation;
- Evaluate the different fault injection techniques present in the literature, and specify one to be applied in the context of this work;
- Through fault injection, assess the soft error vulnerability of the LEON3 processor and identify its most sensitive components;
- Specify and implement a strategy to improve the reliability of the most vulnerable parts in the LEON3 processor;
- Evaluate the performance of the proposed approach.

1.2 PUBLICATIONS

During the master's studies, emerged the opportunity to participate and contribute to the development of different fault tolerance strategies for space applications. The outcome is the following publications submitted to conferences and a journal paper accepted.

1.2.1 Journal Paper

- A Dynamic Partial Reconfiguration Design Flow for Permanent Faults Mitigation in FPGAs. In: Microelectronics Reliability, 2018. *Published.* (MARTINS et al., 2018)

1.2.2 Conference Papers

- Processor Checkpoint Recovery for Transient Faults in Critical Applications. In: 19th IEEE Latin-American Test Symposium (LATS), 2018. *Unpublished.* (VILLA et al., 2018)

- Processor Core Profiling for SEU Effect Analysis. In: 19th IEEE Latin-American Test Symposium (LATS), 2018. *Unpublished.* (TRAVESSINI et al., 2018)

1.3 DOCUMENT STRUCTURE

The remaining of this document is organized as follows:

- **Chapter 2: Background.** Provides an introduction to some relevant topics in the context of this work, including a review of the radiation effects on electronic devices, basic concepts of fault injection, and traditional fault tolerance strategies used on processors.
- **Chapter 3: Related Work.** Presents a literature review, covering studies in the field of soft error vulnerability assessment, as well as studies that propose low overhead fault tolerance techniques.
- **Chapter 4: Methodology and Development.** Begins presenting the methodology used in the course of development and follows with a detailed description of each development stage. The main topics covered are the structure of the fault injection platform and the implementation of the proposed fault tolerance technique.
- **Chapter 5: Experimental Results.** Presents the results of the fault injection campaign, along with the evaluation of the proposed fault tolerance technique.
- **Chapter 6: Conclusions.** Finally summarizes the accomplishments of this work, discusses its importance and limitations, and presents future work perspectives.

2 BACKGROUND

This chapter provides an introduction to some basic concepts that are required to understand the remaining of this work. It begins with a review of the radiation effects on electronic devices, describing the primary mechanisms generators of soft errors. Following is a discussion of the predominant vulnerabilities to FPGAs devices. Next, it continues with the presentation of traditional fault tolerance strategies, particularly the ones based on redundancy. Finally, it presents the concept of fault injection and some of the techniques used for this purpose.

2.1 FAULTS, ERRORS, AND FAILURES

The taxonomy used in the area of dependable systems is extensive and may vary between different authors. The definitions adopted in this work are based on (AVIZIENIS et al., 2004).

The most fundamental concepts are those of fault, error, and failures. A fault may have many different causes. There are the natural faults, which are physical (hardware) faults that are caused by natural phenomena without human participation, and there is also the human-made faults, for instance, the ones originated by mistakes during the system design. A fault is said active when it causes an error; otherwise, it is dormant. An error, in turn, is defined as a divergence in part of the total state of the system that may lead to a subsequent service failure. An error is detected if an error message or error signal indicates its presence. Errors that are present but not detected are latent errors. Lastly, a service failure (often abbreviated to failure) is defined as a deviation in the service delivered by the system from the correct service expected by a user or another system. One important aspect to take into consideration is the placement of the system boundaries, which determines the limits of the system being analyzed. Thus, if an error remains internal to the system boundaries and does not cause the system service to deviate, then no failure occurs.

An example can be used to illustrate these concepts. Take for instance that an energetic particle hits a processor's arithmetic and logic unit (ALU) and temporarily changes the value of an internal wire, characterizing a fault. If that wire is in the adder unit, for example, and the executing instruction is not using this unit, then no error occurs. On the contrary, if an add instruction is in execution when the fault occurs and it causes a register to receive an erroneous value, then an error takes place. Finally, if this error does not cause the service delivered by this program to deviate, then the system does not present a failure. If the service differs, a failure occurs.

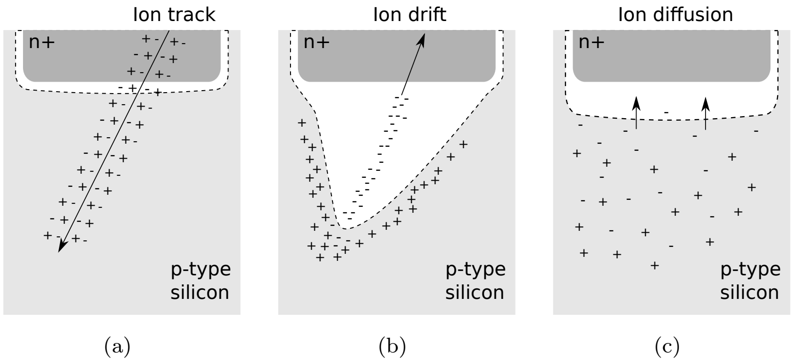
2.2 RADIATION EFFECTS ON ELECTRONIC DEVICES

When operating in harsh radiation environments, electronic devices can be directly struck by several different particle types (e.g., photons, electrons, protons, neutrons or heavy ions), altering their electrical properties and possibly leading to a component failure. The situation is further worsened on newer devices, as the miniaturization and the lower operating voltages reduce the energy necessary to a radiation event induce an error. Therefore, lower energy particle strikes that were harmless in previous generations can now be considered a threat.

The reverse-biased juction is the most charge-sensitive part of circuits, particularly if the juction is floating or weakly driven. Figure 1 illustrates the sequence of events that may occur once an energetic particle hits the substrate of a silicon device. At first, the radiation event creates an ionization track with free electron-hole pairs (a). Then, when the resultant ionization track traverses or comes close to the depletion region, carriers are rapidly collected by the electric field creating a large current/voltage transient at that node (b). In another phase, diffusion dominates the collection process, until all excess carriers have been collected, recombined or diffused away from the junction area (c). In case the total collected charge (Q_{coll}) exceeds a critical charge (Q_{crit}), which in turn depends on node characteristics (e.g., total capacitance in the sensitive volume of the transistor and operating voltage), the event may induce an error (BAUMANN, 2005a).

The term used to refer to events caused by a single particle strike is Single Event Effect (SEE). A SEE may be either destructive (hard errors), when it results in permanent damage to the device (e.g., burnout, gate rupture, frozen bits), or non-destructive (soft errors), when the event generates only a temporary data corruption (e.g., unwanted bitflips in memory cells and registers, that can be resolved by overwriting the affected bit). Of primary interest to this work, the following sections describe the main kinds of soft errors.

Figure 1 – Energetic particle strike in a silicon device.



Source: Adapted from (BAUMANN, 2005a).

2.2.1 Single Event Upset — SEU

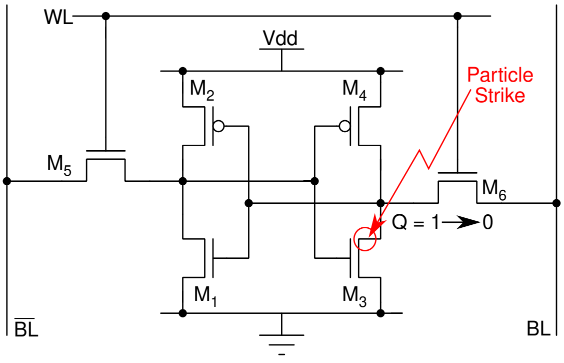
A Single Event Upset (SEU) happens every time an energetic particle directly hits a storage element (e.g., memory cell, register, latch, and flip-flop) causing enough charge disturbance to modify the stored value (BAUMANN, 2005a). In the most common scenario, the radiation event affects only a single bit, called Single Bit Upset (SBU). Higher energy radiation events may cause multiple bits to be modified, leading to a Multiple Bit Upset (MBU).

Figure 2 depicts how a radiation event may induce an SEU in a six-transistor Static Random Access Memory (SRAM). At first, before the radiation event, the cell is storing the value '1'. When the ion hits the drain of the NMOS transistor M_3, it causes a transient change in the output voltage of the right inverter, which is directly connected to the input of the left inverter. In case the voltage in the input of the left inverter falls below the switching threshold, the stored value in the cell is changed and becomes '0', resulting in a SEU.

2.2.2 Single Event Transient — SET

When an energetic particle hits a combinational logic circuit, instead of a storage element, the collection of a sufficient radiation-

Figure 2 – Single Event Upset in SRAM

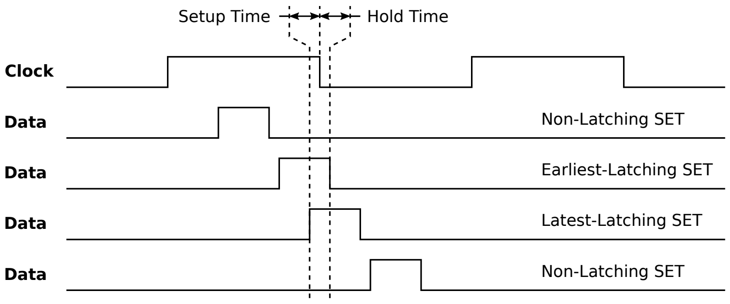


Source: Adapted from (SAJID et al., 2017)

induced charge will generate a transient pulse (i.e., glitch) in the output, known as Single Event Transient (SET) (BAUMANN, 2005b). In case the transient pulse is propagated and stored into a memory element, the SET will generate a soft error.

In contrast with SEUs, which have an error rate independent of the circuit clock frequency, a SET may only generate a soft error if the transient pulse arrives at the input of the memory element during the latching edge of the clock. Due to this behavior, the probability of latching a transient pulse increase with higher clock frequencies, as well the longer the pulse width (GADLAGE et al., 2004). Figure 3 shows SET pulses arriving at the input of a memory element at different time instants. Note that the considered design is falling edge-triggered, requiring a SET pulse timed with the falling edge of the clock to induce an error. In both the top and the bottom data waves, the pulse is either too early or too late and thus are non-latching SETs. On the other hand, in the two middle data waves, the SET pulses are aligned with the clock and are latched into the memory element. Notice that if the pulse was long enough (i.e., at least as long as the clock period), it would always be latched.

Figure 3 – SET latching under different time scenarios.



Source: Adapted from (BENEDETTO et al., 2006).

2.3 SOFT ERRORS IN FPGAS

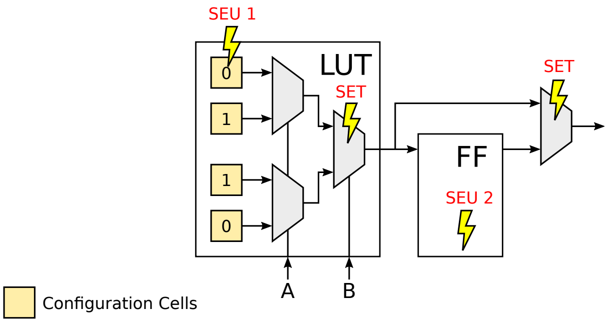
In order to propose efficient fault tolerance strategies aimed at mitigating soft errors in FPGA-Based soft core processors, it is important to understand the particularities of the FPGAs devices regarding this type of error, specifically how SETs and SEUs may disturb them.

First, in the case of SETs, they correspond to only a small fraction of the overall failure rate in FPGAs, being much less frequent than in Application Specific Integrated Circuit (ASIC) devices (HUSSEIN; SWIFT, 2015; LESEA et al., 2005). This high tolerance is mostly due to the large capacitive loading present in the signal paths inside the FPGA structure, consequently increasing the critical charge and requiring higher-energy particle strikes to induce an error. Besides, the highest design operating frequencies found in FPGAs are typically much smaller than what is required for SETs to be a significant contributor to the soft error rate (ALTERA CORPORATION, 2013).

Now, from an SEU standpoint, there are two ways in which an FPGA can be affected: by upsets in its configuration memory (e.g., CRAM), as well by upsets in non configuration bits (e.g., registers and embedded memory). Figure 4 illustrates both cases, as $SEU1$ and $SEU2$ respectively.

FPGAs rely on their configuration memory to store the configuration bitstream, which is responsible for specifying the behavior of every logic element inside the device along with the interconnection net-

Figure 4 – Effects of SEUs and SETs in the FPGA architecture



Source: Author.

work between them. Therefore, bitflips in the configuration memory may deteriorate the original circuit functionality, remaining erroneous until a new configuration is downloaded into the device.

The probability of an SEU happening in the configuration memory, consequently the error rate, is highly dependent on the FPGA technology. SRAM-based FPGAs, which currently dominate the market, are particularly sensitive to this kind of error (WANG et al., 1999). In this technology, the configuration memory is implemented with volatile SRAM switches, which do not retain the configuration when the power is removed. Non-volatile alternatives, such as flash-based and anti-fuse FPGAs, provide an improved radiation tolerance at the cost of usually being one generation (i.e., technology node) behind (WANG, 2003; POIVEY et al., 2011). In particular, anti-fuse cells are immune to SEUs, however with the disadvantage of not being reprogrammable.

In order to be used in harsh radiation environments, FPGA devices that do not have SEU-immune configuration cells (e.g., SRAM-based), must make use of specific fault tolerance strategies aimed at mitigating fault effects in this particular memory. The most often employed technique is called configuration scrubbing. This approach involves a periodic refresh of the FPGA's configuration memory while the FPGA is operating. The scrubbing prevents the build-up of multiple configuration faults and reduces the time in which an invalid circuit is allowed to operate. For further discussion on this strategy see (CARMICHAEL; TSENG, 2009).

When it comes to SEUs in non-configuration bits, they have similar effects to what is observed in ASICs. Therefore, traditional techniques based on redundancy may be applied. Section 2.4 provides further details regarding this form of fault tolerance.

2.4 FAULT TOLERANCE TECHNIQUES

As a means to mitigate radiation-induced faults on electronic devices, several fault tolerance techniques have been proposed and experimented. Most works focus on two main classes of approaches. The first possibility is the development of SEU-hardened components, which can be realized either through modifications in the fabrication process or through custom transistor designs. The other alternative is to protect the device with redundancy, i.e., extra functionalities with the sole purpose of detecting and correcting errors, that would not be necessary in a fault-free environment.

In the case of FPGAs, redundancy-based strategies are often preferred, as they can be used with COTS components. This way, not requiring the development and fabrication of custom devices, consequently providing a lower cost and better time to market (KASTENSMIDT; REIS, 2007).

Redundancy-based strategies are also very flexible. Implementations can be made at different levels of the circuit design flow, with varying degrees of protection. For instance, one approach can be made simple enough only to provide error detection, while other may provide both detection and correction, but possibly with a higher performance impact. This great flexibility translates into a wide variety of techniques present in the literature. Most of the developed strategies fall into four main categories (GOLOUBEVA et al., 2006): hardware redundancy, information redundancy, time redundancy, and software redundancy.

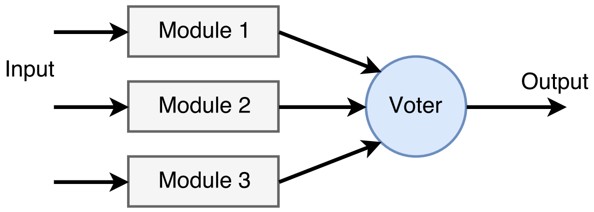
Before proceeding with a discussion of the different forms of redundancy, it is important to note that any redundancy scheme implies some penalty to be paid. Each technique contains a combination of area overhead, performance degradation, and power dissipation increase. The circuit designer must find the best trade-off, in order to meet the area, time and power constraints, as well the soft error hardness required (KASTENSMIDT; REIS, 2007).

2.4.1 Hardware Redundancy

Hardware redundancy consists of the physical replication of hardware components and paths, which allow the design to continue operation even when some parts fail (KASTENSMIDT; CARRO, et al., 2006). A straightforward implementation is a technique called Duplicate with Comparison (DWC) or Dual Modular Redundancy (DMR). In this approach, the module to be protected is replicated, and the outputs from both replicas are continuously compared. Any identified mismatch would signalize the presence of errors. Note that in this technique the comparator should also be protected since it can be itself a source of errors. Additionally notice, that this simple approach only provides the detection of errors, but it cannot identify which module is presenting the wrong output.

To be able to detect errors and also determine the correct output, more than two replicas are necessary. Given this, in 1956 Von Neumann proposed the Triple Modular Redundancy (TMR) scheme (NEUMANN, 1956), in which three identical modules perform the same task concurrently, and a voter compares their outputs, agreeing with the majority. Figure 5 presents a conceptual representation of this scheme. In this approach, if only one of the modules presents an incorrect output, the voter could still select the correct output from the other two replicas, thus entirely masking the error. However, if more than one module contain errors, the voter cannot identify the correct output, and the TMR system fails. In other words, the TMR technique can tolerate at most one module containing errors. For improved fault tolerance, this technique may be generalized with n identical modules, where n is typically odd, named N-modular redundancy.

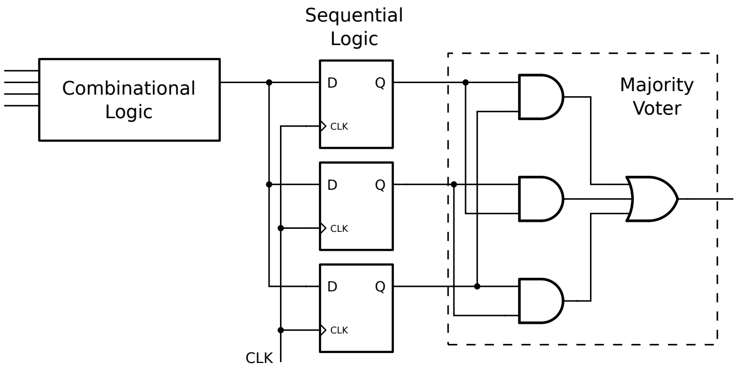
Figure 5 – Triple Modular Redundancy concept



Source: Adapted from (GOLOUBEVA et al., 2006).

Although conceptually simple, several factors must be taken into account when implementing modular redundancy. For instance, if only SEU mitigation is required, and soft errors due to SETs are infrequent, the solution provided in figure 6 is suitable. This approach only replicates sequential elements (e.g., flip-flops), i.e., elements vulnerable to SEU.

Figure 6 – Triple Modular Redundancy applied to sequential logic

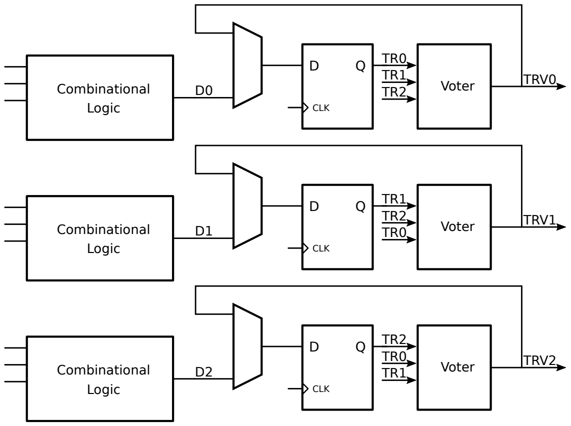


Source: Adapted from (KASTENSMIDT; CARRO, et al., 2006).

However, if the system also requires protection from SETs, the approach from figure 6 has limitations. First, by not replicating the combinational logic, all the flip-flops in the sequential logic receive the same input. As a consequence, in case a SET happens in the combinational logic, it may be latched by all three flip-flops, rendering the majority voter useless and resulting in wrong output. Second, a SET may also happen in the majority voter, likewise leading to wrong output.

To manage both situations properly, another TMR scheme is necessary. Figure 7 presents the full TMR strategy, which was proposed by (CARMICHAEL, 2001). Apart from the sequential logic, this approach also replicates the majority voters and the combinational logic. This new configuration has a slightly different operation than the previous one. For instance, if an SEU occurs in one of the flip-flops (i.e., sequential logic), all three majority voters will select the correct output, and at the next clock cycle, the correct output can be loaded to the flip-flop, effectively clearing the SEU. Now, if a SET occurs in one of the combinational logic blocks, the SET may be only latched by one of the redundant flip-flops, and again the majority voter will be able to select the correct output. Finally, if a SET occurs in one of the majority voters, the voter output will show the transient for a short period of time, but since all the circuit is replicated, only one redundant module is affected and the SET will be voted out at the next majority voter.

Figure 7 – Full Triple Modular Redundancy



Source: Adapted from (KASTENSMIDT; CARRO, et al., 2006).

2.4.2 Information Redundancy

Information redundancy resides on adding redundant information to data in order to allow error detection and possibly correction (PRADHAN, 1996). The techniques that comprise this form of redundancy are the EDCs and the ECCs.

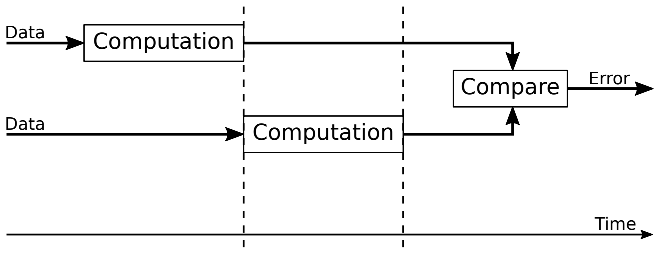
One example is the parity code. In this scheme, one bit is added to a string of binary data to ensure that the total number of ones is odd or even. This parity bit is usually computed when the data string is generated and stored in memory. Every time the data is read, the parity bit is recalculated and compared with the previously stored. If there is a mismatch, it indicates that an error has occurred. Due to its simplicity, this technique can only detect single bit errors and has no correction capability. If greater robustness is required, several other techniques may be used, such as CRC, Hamming and Reed-Solomon codes (LABEL; GATES, 1996).

Note that information redundancy is mainly used for protecting memory arrays (e.g., cache, register file, main memory), and is commonly regarded as difficult to apply to computational or control logic functions (KIM; SOMANI, 2001).

2.4.3 Time Redundancy

Time redundancy consists of the repetition of the same task two or more times, followed by a comparison of the results, to identify possible errors. In case there is a mismatch between the executions, another run may be performed to verify if the error is still present or has disappeared. Figure 8 illustrates the concept. Note that the idea is similar to hardware redundancy, but instead of using extra hardware to perform redundant operations in parallel, time redundancy involves the repetition of tasks in a sequential manner. It is not uncommon to have hybrid techniques that seek to minimize both the impact on area and performance (WU; KARRI, 2004; JOHNSON et al., 1988).

Figure 8 – Time Redundancy concept



Source: Adapted from (GOLOUBEVA et al., 2006).

2.4.4 Software Redundancy

Software redundancy covers an extensive research field. Besides mitigation of radiation-induced faults, there are various others motivations for employing techniques based on this concept. Another relevant domain of application is the protection against design and specification faults, which are not uncommon in software as they are often composed of a substantial number of states, with little regularity.

Software fault-tolerance techniques can be classified into two groups: single-version and multi-version. In the single-version approach, a single software module is modified to include fault detection, containment, and recovery mechanisms. In the case of multi-version techniques, multiple versions of the same software module are developed, frequently using different teams, coding languages or algorithms, with the goal of minimizing the probability that all versions share a common fault. For further discussion on this topic see (GOLOUBEVA et al., 2006; DUBROVA, 2013).

2.5 FAULT INJECTION

Fault injection is a validation technique used for assessing the dependability of fault-tolerant systems. It consists of inserting (i.e., injecting) faults into a system and monitoring the system behavior in response to a fault. There are several different fault injection strategies proposed in the literature. According to (ZIADE et al., 2004), they can be classified into five main categories:

- Hardware-based fault injection — It relies on disturbing the actual hardware system through the use of external physical sources. Some examples are heavy-ion radiation, electromagnetic interference, and laser fault injection. This type of technique requires special-purpose hardware to run the experiments. Furthermore, it involves a high risk of damaging the assessed system.

- Software-based fault injection — This type of technique consists of reproducing at the software level, the errors that would arise in the occurrence of faults in either software or hardware. The injection can be accomplished by corrupting memory contents, or through the mutation of the application software (i.e., by modifying existing lines of code so that they contain faults).

- Simulation-based fault injection — Consists of using simulation tools for injecting faults in a model representation of the system (usually VHDL models). This technique can be used with models at different abstraction levels (e.g., RTL and GL). A significant drawback of this technique is the frequently long simulation times.

- Emulation-based fault injection — It is conceptually similar to simulation-based techniques, however, it makes use of FPGAs for speeding-up fault simulation.
- Hybrid fault injection — This approach is a mix of two or more techniques, with the goal of combining the best features of each of them.

When developing or choosing a fault injection technique, different properties must be considered (ARLAT et al., 2003):

- Reachability — It consists of the ability of the technique to reach the possible fault locations in the system being considered. For instance, hardware-based techniques such as heavy-ion radiation provide high reachability, as the faults are injected directly into the actual hardware system. In contrast, software-based techniques are limited to the locations accessible through software and thus have lower reachability.
- Controllability — It consists of the ability of the technique to control which of the reachable fault locations are actually injected, as well controlling the instant when faults are injected. For instance, heavy-ion radiation has a low controllability, as it is difficult to focus the injection in specific components, furthermore, the exact injection instant cannot be governed. On the other hand, simulation-based techniques provide very high controllability, as any signal value can be corrupted at any specific time.
- Repeatability — It refers to the ability to repeat injection experiments with a very high degree of accuracy, both spatial and temporal. In that sense, to have high repeatability, the technique must also have high controllability. Consequently, heavy-ion injection has low repeatability. Simulation-based techniques provide high repeatability since the experiment can be repeated in the same signal at the same instant.
- Reproducibility — It refers to the ability to reproduce previous results when using the same set-up. Usually, high reproducibility is achievable if the technique has high repeatability. However, high repeatability is not a requirement for high reproducibility.
- Non-intrusiveness — It relates to the ability of the technique to avoid or minimize any undesirable impact on the normal operation of the target system. For instance, software-based techniques

frequently have low non-intrusiveness as the injection mechanism must run on the same system as the software being tested. In contrast, simulation-based techniques usually do not require any change in the target system, providing high non-intrusiveness.

- Time measurement — It consists of the ability to obtain time information associated with the monitored events (e.g., measurement of error detection and propagation latency). Usually, for this purpose, is used a reference model of the system (also known as a golden model) operating synchronously. Thus, allowing the identification of the exact time instants in which both models start to diverge concerning outputs and results.

- Efficacy — It consists of the ability of the technique to reduce the number of non-significant experiments, that is, faults injected in components not accessed or used throughout the experiment.

3 RELATED WORK

Given the increasing interest in using COTS components in space applications, several studies have been carried out in the last decades to assess the vulnerability of these components in the space environment, and also to propose efficient fault tolerance techniques targeting them. In this context, this chapter presents some of these works, which were both a source of motivation for this research, as well guided the development of the proposed technique.

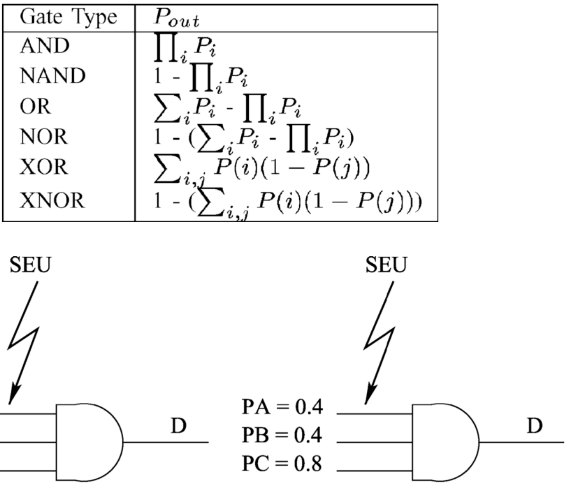
The chapter is organized in two sections. The first one reports studies that propose low overhead fault tolerance techniques centering on soft error mitigation. The second one presents studies that investigate the effects of faults in microprocessors. Both areas are of great interest in this work, due to the characteristics of the proposed technique, which will be presented in detail in the upcoming chapters.

3.1 LOW OVERHEAD FAULT TOLERANCE TECHNIQUES

Fault tolerance is an extensive research field. The literature contains techniques with distinct purposes, using varying forms of redundancy, and consequently with different costs concerning area, energy, and performance. For this section will be presented some techniques that are in more conformity with the proposal of this work, in the sense of being oriented to soft errors, and also for aiming at low overhead.

In (SAMUDRALA et al., 2004) the authors proposed a technique for hardening combinational logic circuits mapped onto Xilinx Virtex FPGAs against SEUs. The strategy is named Selective Triple Modular Redundancy (STMR) and is based on the traditional TMR approach; however, to obtain reduced area overhead, they selectively employ the redundancy in only the most sensitive circuits. The identification of the most sensitive circuits is made by analyzing the signal probabilities of each logic gate within the circuit, i.e., a gate is sensitive if an SEU on any of the inputs is likely to be propagated to the output of the gate. Figure 9 presents the equations used to determine the signal probability for each gate type. Since the technique may require an increase in the number of majority voters than a traditional TMR, the authors suggested the use of tri-state buffers present in the Virtex FPGA to construct SEU immune majority voters. The experimental results show that the technique can provide immunity against SEUs comparable to the full TMR when used along with other mitigation features of the Virtex FPGA. The area overhead of the STMR strategy may reach 60–70% that of TMR.

Figure 9 – Signal probability computation at the output of a boolean gate

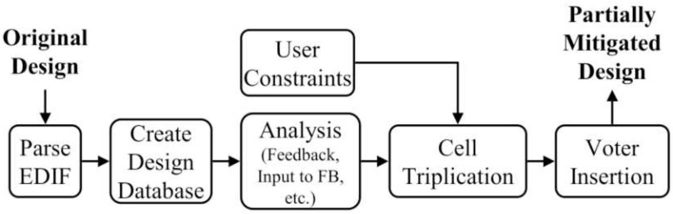


Gate Type	P_{out}
AND	$\prod_i P_i$
NAND	$1 - \prod_i P_i$
OR	$\sum_i P_i - \prod_i P_i$
NOR	$1 - (\sum_i P_i - \prod_i P_i)$
XOR	$\sum_{i,j} P(i)(1 - P(j))$
XNOR	$1 - (\sum_{i,j} P(i)(1 - P(j)))$

Source: (SAMUDRALA et al., 2004).

The idea of protecting only the most critical sections of a design is also explored in (PRATT et al., 2006). However, instead of protecting the combinational logic, this study aims at mitigating the effects of SEUs in the configuration memory of the FPGA. The authors label the configuration bits used by the design mapped onto the FPGA as *sensitive* bits and suggest that the sensitive bits can be split into two categories called *persistent* and *non-persistent*. A non-persistent configuration bit is a sensitive configuration bit that may introduce functional errors when upset by radiation; however, it can be repaired with configuration scrubbing, and the functional errors disappear. In contrast, a persistent configuration bit is a sensitive configuration bit, which even after configuration scrubbing, the introduced functional errors are not repaired. Hence, the study suggests that TMR could be applied only to the circuit structures which correspond to persistent configuration bits. The study also introduces a software tool named BYU-LANL Partial TMR (BLTmr) which automatically classifies circuit structures based on this concept and applies TMR selectively depending on the classification. Figure 10 illustrates the basic flow of the developed tool. The experimental results showed that for a specific design the number of faults in the configuration bits that led to non-repairable functional errors reduced in two orders of magnitude with a hardware cost of 40% over the unmitigated design, which is much lower than a full TMR approach.

Figure 10 – Basic flow of the BYU-LANL Partial TMR (BLTmr) tool



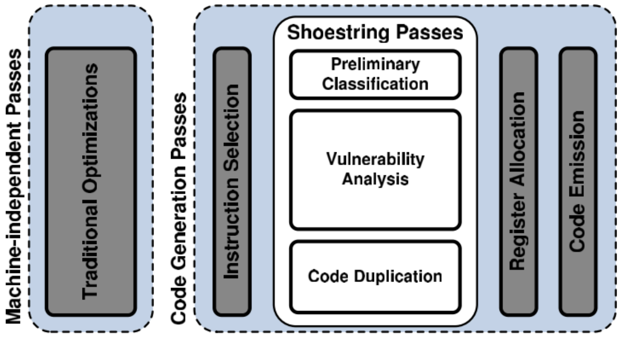
Source: (PRATT et al., 2006).

More recently, the work in (GOMES et al., 2015) explores the concept of approximate logic circuits to reduce the area overhead of the TMR technique. The idea consists of only using approximate logic modules to compose the redundant modules of the TMR. The approximate circuits are modified versions of the original circuit with a smaller area, and that differs its output from the original circuit for a small set of input vectors. Nevertheless, this technique imposes a condition on the approximate circuits: only one of the modules can differ from the original circuit at each input vector scenario, therefore allowing the majority voters to still select two match outputs out of three for any input vector. For computing the approximate functions, the authors used a Boolean factorization method, which was also used to select the best composition of approximate logic. The experimental results show that for a 4-bit ripple carry adder the fault coverage could go up to 93% with 136% of area overhead and 96% with 168%.

Another recent work that explores approximate logic is presented in (SANCHEZ-CLEMENTE et al., 2016). However, in this study, the technique is aimed at FPGA implementations. In this sense, the valuable logic approximations are the ones that reduce the number of Lookup Tables (LUTs) in the circuit, either by eliminating LUTs or by merging contiguous LUTs. Otherwise, the result is a degradation of the logic function of the circuit without achieving any benefits in terms of resource utilization.

The techniques presented so far were all based on hardware redundancy. A different approach is seen in (FENG et al., 2010), where is described a software-based approach named Shoestring, that provides probabilistic soft error reliability. The purpose of the Shoestring technique is to present high soft error coverage with very little overhead. In order to achieve these goals, the authors proposed a strategy which combines low-weight symptom-based fault detection schemes with software-based instruction duplication. Symptom-based detection schemes recognize that applications often exhibit anomalous behavior (e.g., memory access exceptions, mispredicted branches, and cache misses) in the presence of soft errors. Although symptom-based detection is inexpensive, it has a limited fault coverage, requiring the use of other techniques concurrently. In this context, the main contribution of the Shoestring technique is to efficiently select between relying on symptoms or applying instruction duplication to each part of the program code. The selection analysis runs at compile time, by introducing additional reliability-aware code generation passes into the standard compiler flow, as shown in Figure 11. In the experiments conducted by the authors, the Shoestring technique achieved an overall user-visible failure rate of 1.6%, with a performance overhead of 15.8%.

Figure 11 – A standard compiler flow augmented with Shoestring's reliability-aware code generation passes



Source: (FENG et al., 2010).

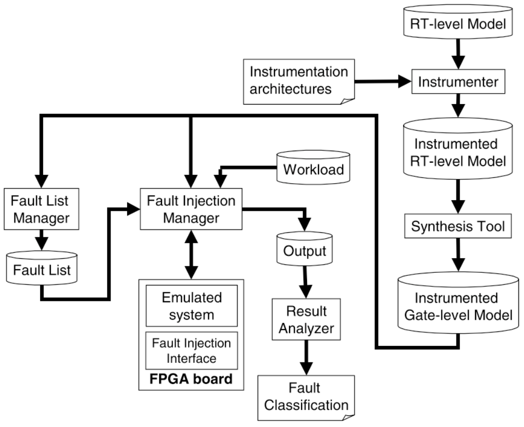
Finally, note that none of the presented techniques offers "five-nines" reliability, making them inappropriate for most critical scenarios. Instead, the techniques focus on less demanding applications, demonstrating that through partial protection it is possible to have significant reliability improvements without incurring substantial performance and area penalties.

3.2 SOFT ERROR VULNERABILITY ANALYSIS

Understanding the fault effect on systems is an important step towards new efficient mitigation techniques. In this perspective, this section will present some studies concerning soft error vulnerability assessment specifically focused on embedded processors. Familiarity with the methodologies used in these studies played a fundamental role in the development of this work.

In (REBAUDENGO et al., 2003) is presented a study that analyzes the effects of SEUs in the first processor in the LEON family. Besides investigating the LEON behavior in the presence of faults, the authors wanted to compare two alternative fault injection techniques: a software-based approach and an emulation-based approach.

Figure 12 – Emulation-based fault injection platform used in (REBAUDENGO et al., 2003)



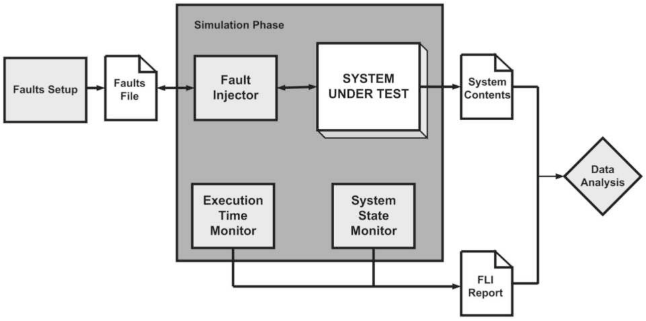
Source: (REBAUDENGO et al., 2003).

Figure 12 shows the emulation-based platform. In a pre-injection step, the processor VHDL model is instrumented, synthesized and mapped to an FPGA device. For running the injection experiments, a host computer works as Fault Injection Manager and communicates with the FPGA board, orchestrating the selection of a new fault, its injection in the system, and the observation of the resulting behavior.

In the results, the authors highlighted the increased reachability obtainable with a simulation/emulation technique when compared to software approaches, considering that the latter cannot inject fault in non-programmer-visible locations such as the pipeline registers. The authors also state that the emulation technique had an accuracy much higher than the software one (up to 13 times higher), concluding that the software approach may lead to significant errors during the error rate estimation.

A similar study is conducted in (TOULOUPIS et al., 2007) using the LEON2 processor. In this paper, the authors adopted a simulation-based fault injection approach, as shown in Figure 13. In the technique, the primary fault injection support is implemented through a non-synthesizable VHDL entity that has access to all registers and can alter their contents at specified times based on the idea of "saboteurs". The entire system is simulated in a commercial VHDL simulator (ModelSim), extensively using the simulator's Foreign Language Interface (FLI) to implement various entities that deal with the fault injection, monitoring, and data collection.

Figure 13 – Simulation-based fault injection platform used in (TOULOUPIS et al., 2007)

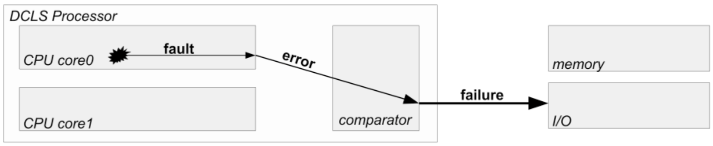


Source: (TOULOUPIS et al., 2007).

In the results, the authors present a detailed analysis of the fault effects in the LEON2 processor, particularly in the pipeline unit. Some important conclusions are the inefficacy of the LEON2 exception mechanism for detecting injected faults, as well the dependence between the observed fault effects and the processor's workload.

More recently, a study conducted by ARM research employees (ITURBE et al., 2016) investigated the effects of soft errors on the ARM Cortex-R5 CPU core. As in the previous work, the authors adopted a simulation-based technique to inject faults in the processor and monitor their effects. The simulator employed was the Synopsis VCS. This study focused on safety-critical applications and in this context, the authors chose to use a somewhat conservative and pessimistic approach to categorize the effects of the injected faults. In this approach, two identical processors are simulated running the same workload in a lock-step fashion, and each time a mismatch is detected between the ports of both processor, a failure is recorded. Figure 14 illustrates the concept. The approach is deemed pessimistic because it considers only logic gate masking and micro-architecture level masking as mechanisms that prevent the faults from generating unwanted effects. However, it does not consider other important mechanisms such as software masking. The experimental results showed that less than 10% of the sequential elements in the Cortex-R5 CPU account for more than 70% of the errors, and these errors are manifested in only 65% of the CPU output ports.

Figure 14 – Dependability terminology used in (ITURBE et al., 2016)



Source: (ITURBE et al., 2016).

Finally, observe among the presented studies that although they are directed in different processors with distinct focuses, there is a tendency to adopt simulation-based and emulation-based fault injection techniques for performing this type of investigation. Nonetheless, all studies point to the limitation concerning simulation time and the difficulty of using more representative workloads.

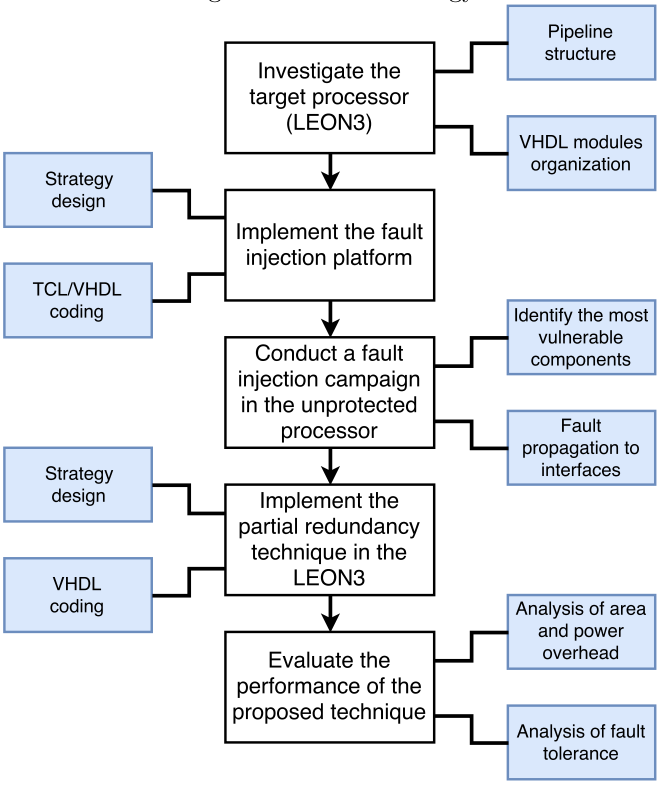
4 METHODOLOGY AND DEVELOPMENT

This chapter opens with the presentation of the methodology used during the development and evaluation of the proposed technique. Next, the chapter continues with a discussion of each development stage.

4.1 METHODOLOGY

Figure 15 presents the adopted methodology in this work.

Figure 15 – Methodology


Investigate the target processor (LEON3)
Pipeline structure
VHDL modules organization
Strategy design
TCL/VHDL coding
Implement the fault injection platform
Conduct a fault injection campaign in the unprotected processor
Identify the most vulnerable components
Fault propagation to interfaces
Strategy design
VHDL coding
Implement the partial redundancy technique in the LEON3
Evaluate the performance of the proposed technique
Analysis of area and power overhead
Analysis of fault tolerance

Source: Author.

The first step, presented in Section 4.2 consists in carrying out a study concerning the target processor of this work, which aims to: comprehend the LEON3 pipeline structure; specify the processor configuration (e.g., cache parameters); and understand how the VHDL implementation is structured. Such information is fundamental to the remaining steps and will direct some choices during the implementation of both the fault injection platform and the partial redundancy technique.

The following phase, presented in Section 4.3, covers the development of the fault injection platform. Starting by the definition of the injection strategy, and continuing with the implementation itself (e.g., development of algorithms). Also belong to this step, the definition of fault models, and expected fault effects in the LEON3.

After developing the fault injection platform, the next step is to carry out the injection experiments in the LEON3 without any fault tolerance strategy applied (i.e., unprotected). From these experiments, it is possible to obtain several relevant statistics, such as fault propagation paths and fault manifestation times. Most importantly, these experiments will provide a list of the LEON3 most vulnerable components. Section 5.1 presents and discusses the results of the fault injection campaign.

From the results of the previous step, the partial redundancy technique is implemented, selectively protecting just the most vulnerable components. Section 4.4 will discuss implementation-related details. Note that although this step is in practice executed after the fault injection, in this text it is presented before. This choice does not detract from understanding, since the process of implementing the technique is the same independent of the results, only changing the locations in which the technique is applied.

Finally, the last step is to evaluate the performance of the proposed technique. The evaluation has two main components: quantify the improvement in fault tolerance through a new fault injection campaign, and determine the area and performance penalty by synthesizing the design.

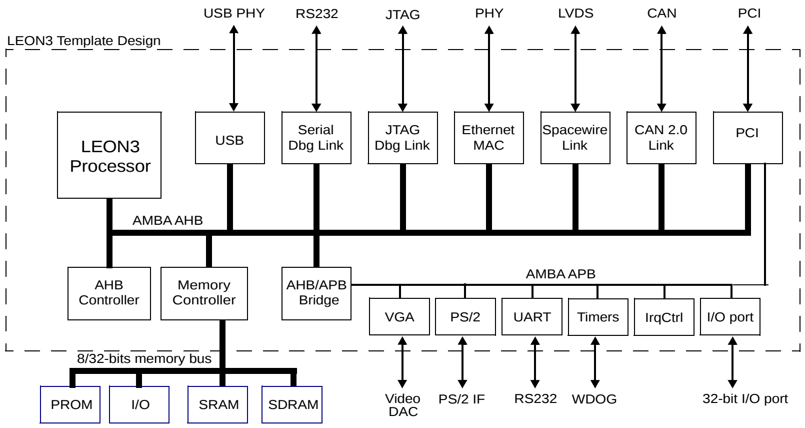
4.2 LEON3 SOFT CORE PROCESSOR

The LEON3 is a 32-bit processor compliant with the SPARC V8 instruction set architecture. It was initially developed by the European Space Research and Technology Centre (ESTEC), part of the

ESA, and subsequently by the Swedish company Cobham Gaisler AB. Its design focuses on embedded applications, by combining high performance, low complexity, and low power consumption. The processor is described using VHDL and can be completely customized depending on the application requirements. Furthermore, the source code is available under the GNU GPL license (COBHAM GAISLER AB, 2017).

One of the LEON3's main features is the seven-stage pipeline with separate instruction and data caches (Harvard Architecture). The processor also has several optional modules, including a memory management unit, hardware multiplier and divider, as well as a debugging support unit. Moreover, the presence of an AMBA-2.0 bus interface, allows the processor to be integrated with many other Intellectual Propertys (IPs) from the Gaisler Research IP Library (GRLIB) (GAISLER et al., 2017). Figure 16 presents a representation of the LEON3 architecture with the main peripherals, demonstrating the numerous configuration possibilities.

Figure 16 – LEON3 Architecture with Main Peripherals



Source: Adapted from (GAISLER et al., 2017).

Beyond the default LEON3 implementation, Cobham Gaisler also offers a fault tolerant version (LEON3FT), which provides protection against SEU errors. According to the *GRLIB IP Core User's Manual* (GAISLER et al., 2017), the LEON3FT fault tolerance is focused on the protection of the on-chip RAM blocks, which are used to implement the register file and the cache memories, and therefore does not comprehend the processor control logic, which in turn is investigated in this work.

4.2.1 Pipeline Structure

The LEON3 processor contains a single-issue pipeline, i.e., at most one instruction is fetched from memory in each clock cycle. Furthermore, the execution of instructions follows the same order in which they were organized by the compiler (statically scheduled). The pipeline is divided in seven stages (GAISLER et al., 2017):

1. FE (Instruction Fetch) — If the instruction cache is enabled, the instruction is fetched from the instruction cache. Otherwise, the fetch is forwarded to the AHB bus. The instruction is valid at the end of this stage and is latched inside the IU.

2. DE (Decode) — The instruction is decoded and the CALL and Branch target addresses are generated.

3. RA (Register Access) — Operands are read from the register file or from internal data bypasses.

4. EX (Execute) — ALU, logical, and shift operations are performed. For memory operations (e.g., LD) and for JMPL/RETT, the address is generated.

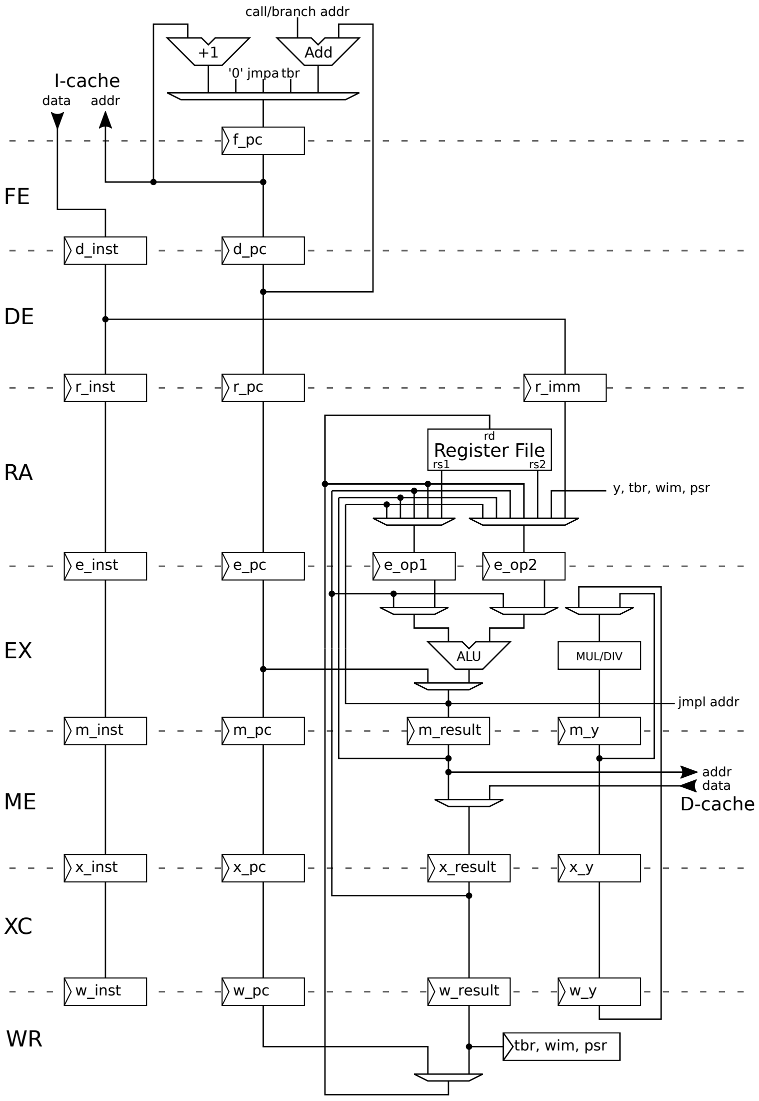
5. ME (Memory) — Data cache is read or written at this time.

6. XC (Exception) — Traps and interrupts are resolved. For cache reads, the data is aligned as appropriate.

7. WR (Write) — The result of any ALU, logical, shift, or cache operations are written back to the register file.

Figure 17 presents a block diagram of the LEON3 datapath, illustrating the key operations and components involved in each pipeline stage. This figure is also important because it contains the main internal pipeline registers, which are part of the components evaluated during the fault injection campaign.

Figure 17 – LEON3 Integer Unit Datapath

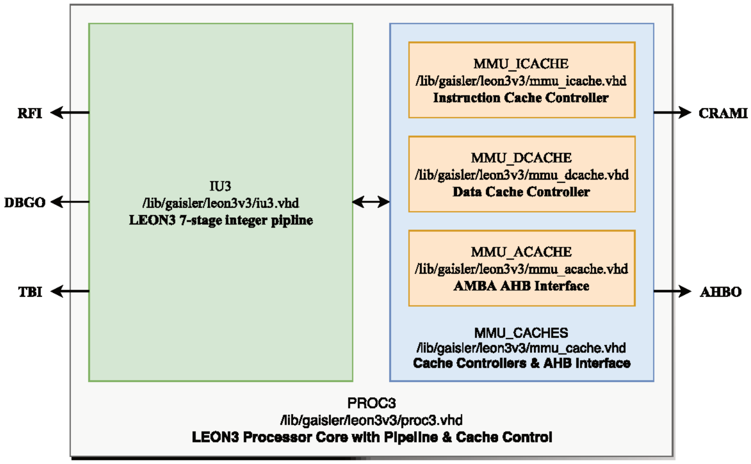


Source: Adapted from (GAISLER et al., 2017).

4.2.2 VHDL Structure and Implementation

Among the various modules that compose the LEON3 processor, this work will investigate the one named Processor Core (PROC3), where most of the processor control logic is implemented. The PROC3 is also composed of two submodules. The larger one, the integer unit (IU3), implements the entire processor seven stage pipeline. The other one is the cache controllers (MMU_CACHES), which can be divided in the instruction cache controller (MMU_ICACHE), the data cache controller (MMU_DCACHE), and the interface between the cache controllers and the AMBA AHB bus (MMU_ACACHE). Figure 18 shows a block diagram of the PROC3 and indicates the path for the VHDL files that implement each module.

Figure 18 LEON3 Processor Core (PROC3)


MMU_ICACHE
/lib/gaisler/leon3v3/mmu_icache.vhd
Instruction Cache Controller
RFI
CRAMI
MMU_DCACHE
/lib/gaisler/leon3v3/mmu_dcache.vhd
Data Cache Controller
IU3
/lib/gaisler/leon3v3/iu3.vhd
LEON3 7-stage integer pipline
DBGO
MMU_ACACHE
/lib/gaisler/leon3v3/mmu_acache.vhd
AMBA AHB Interface
TBI
AHBO
MMU_CACHES
/lib/gaisler/leon3v3/mmu_cache.vhd
Cache Controllers & AHB Interface
PROC3
/lib/gaisler/leon3v3/proc3.vhd
LEON3 Processor Core with Pipeline & Cache Control

Source: Author.

Also presented in figure 13 are the PROC3 main output interfaces (i.e., ports), which will be observed during the fault injection campaign to identify the fault propagation paths and obtain relevant statistics such as fault manifestation times. The interfaces connected to the IU3 are the integer register file (RFI), the processor debug support unit (DBGO), and the instruction trace buffer (TBI). Moreover, the ones connected to the cache controller, are the cache memory array

(CRAMI), and the AMBA AHB bus (AHBO).

Note that in the LEON3 implementation, the cache controllers are present even if the design configuration does not contain caches. The reason for this is that, besides managing every access to memory, whether it is in cache or main memory, the controllers also have the function of ensuring the synchronization of these accesses with the pipeline operation.

Also note, that the PROC3 does not comprise the memory hierarchy. The register file, the cache memory array, and the main memory are all modules outside de PROC3. In this sense, the technique implemented in this work will not cover these modules. As stated before, ECCs are frequently employed as an efficient method to protect memory arrays and correspond to one of the available protection schemes in the LEON3FT.

Concerning the VHDL implementation of each discussed module, the LEON3 designers adopted a structured VHDL design methodology (GAISLER, 2011). The goals of this approach are, among others, to improve readability, decrease simulation time, and provide a uniform algorithm encoding. Among the measures suggested in this methodology and employed in the LEON3, the most relevant for this work are:

- **Using two processes per entity** — In this approach, each VHDL entity has only two processes: one process that contains all combinational (asynchronous) logic, and one process that contains all sequential logic (registers). In this structure, the complete algorithm can be coded in sequential (non-concurrent) statements in the combinational process while the sequential process only contains registers, i.e., the state. For example, in the IU3 implementation, the entire pipeline logic is described through non-concurrent statements in the combinational process. The sequential process is only responsible for updating the pipeline internal registers every clock cycle. The objective of this approach is to improve readability. By reducing the number of concurrent statements, the VHDL code resembles more standard programming languages such as C.

- **Using record types** — The VHDL record type is similar to structures in C. It allows declaring composite objects whose elements can be of different types, including other records. For modules with hundreds of ports and signals, the possibility of grouping them according to functionality may considerably improve readability. For example, in the IU3 implementation, there is a record for each pipeline stage that groups all the registers belonging to that stage, in addition, the records of each stage are also grouped in another record.

- **Using subprograms** — Subprograms (procedures and functions) allow large algorithms to be split into distinct code segments, being a powerful method to improve code readability. Within the context of using two processes per entity, subprograms may be used to hide the complexity from the combinational process. For example, in the LEON3 pipeline implementation, most of the logic is broken into subprograms, which are called from the combinational process, making the pipeline operation flow much more readable.

4.2.3 Configuration

As previously mentioned, the LEON3 processor can be extensively configured, depending on the application requirements and available resources. Table 1 summarizes the main attributes from the configuration employed in this work. Note that the modules that interface with the PROC3 (e.g., Debug Support Unit, Instruction Trace Buffer) were enabled in order to monitor the fault propagation paths.

Table 1 – LEON3 Configuration

	Attribute	Value
Processor	No of processors	1
	SPARC registers windows	8
Integer Unit	MUL/DIV instructions	yes
Debug Support Unit	Enable DSU	yes
	Instruction Trace Buffer	yes
Cache System	Enable Caches	yes
	Associativity	1
	Set size	4 kbytes
	Line size	32 bytes
MMU	Enable MMU	no

Source: Author.

4.3 FAULT INJECTION PLATFORM

In order to choose the fault injection method that best fits the purposes of this work (i.e., assess the soft error vulnerability of the LEON3 processor core and identify its most sensitive components), each property listed in Section 2.5 was revisited:

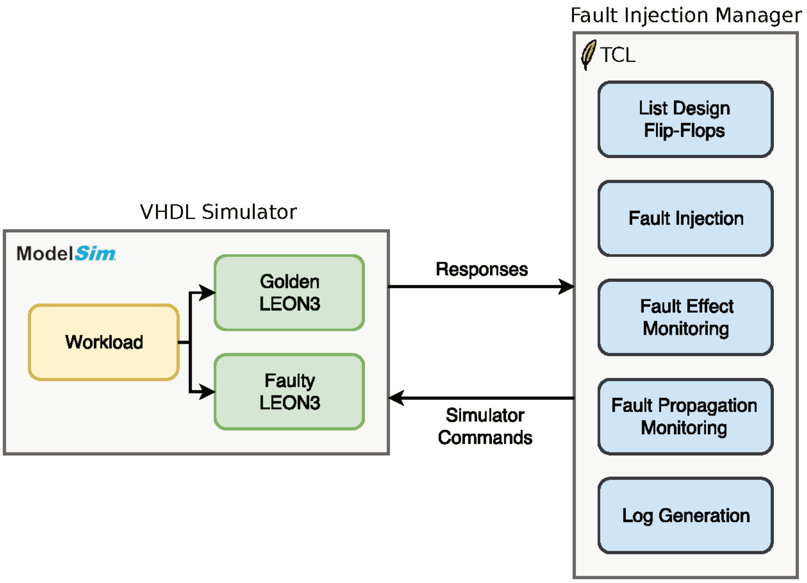
- Reachability — The technique must be able to inject faults on any component inside the PROC3.
- Repeatability — The technique must allow the repetition of experiments. Necessary to compare the performances before and after the implementation of the partial redundancy technique.
- Controllability — The technique must be able to inject faults in each component inside the PROC3 at any specific time instant, a requirement for repeatability.
- Reproducibility — The results must be reproducible, to ensure the credibility of the experiments.
- Non-intrusiveness — The technique must not generate any interference in the LEON3 operation flow.
- Time measurement — Although not a requirement for the main objective of this work, the ability to perform time measurement allows a more in-depth analysis of the effects of the injected faults.
- Efficacy — Not required in this work. All the injection experiments, even in components not accessed or used, correspond to useful information regarding that component impact in the LEON3 operation.

From the available methods presented in Section 2.5, the one found to fit those requirements best is the simulation-based fault injection. In this sense, the HDL simulator chosen for the experiments was the ModelSim (MENTOR GRAPHICS, n.d.) from Mentor Graphics.

The ModelSim software contains both a Graphical User Interface (GUI) for easy access and operation as well as a console to run scripts with Tool Command Language (TCL). These scripts can make use of various built-in commands which provide interaction with the simulation engine, allowing read and write access to any logic signal during any time of the simulation. Thus, through the scripts, it is possible to do both the fault injection as the observation of the fault effects.

Figure 19 depicts the fault injection environment, which is composed of the VHDL Simulator (ModelSim) and the Fault Injection Manager (TCL Script). Note that the simulated design contains two replicas of the LEON3, one for reference (Golden LEON3) and another where the faults are injected (LEON3).

Figure 19 Fault Injection Environment



Source: Author.

Before proceeding with the implementation details of each part of the TCL script, the next sections will cover the fault model, the fault effects classification, and the workload used in this work, given that these definitions have a direct impact on the implementation.

4.3.1 Fault Model

The adopted fault model for the fault injection campaign is the SBU. In each experiment, a single fault is injected by inverting the logic value of the target signal. Multiple faults due to a single radiation event (MBU) are not addressed in this work. In order to accurately model this effect, it is necessary to have information regarding the final design layout and the register neighborhood, which are not available during the HDL simulation.

Furthermore, SETs are also not studied in this work, given their low contribution to the total soft error rate in FPGAs, as stated in Section 2.3. In this context, the fault injection experiments will target only sequential elements (e.g., registers and flip-flops).

4.3.2 Fault Effects Classification

After the end of each fault injection experiment, the data obtained during the simulation is used to classify the fault effects in five categories. The classification, which is based on the one used in (REBAUDENGO et al., 2003) and (TOULOUPIS et al., 2007), is the following:

- *No Effect* — The software running in the processor finishes execution normally, with correct results, and the content of the processor core registers match with the golden run.

- *Latent* — The software running in the processor finishes execution normally, with correct results, but the contents of the processor core registers do not match with the golden run.

- *Wrong Result* — The software running in the processor finishes execution, but with incorrect results.

- *Timed Out* — The software running in the processor took an abnormal amount of time without finishing execution, and the simulation was interrupted. Many conditions may lead to this scenario, such as an incorrect branching due to the injected fault.

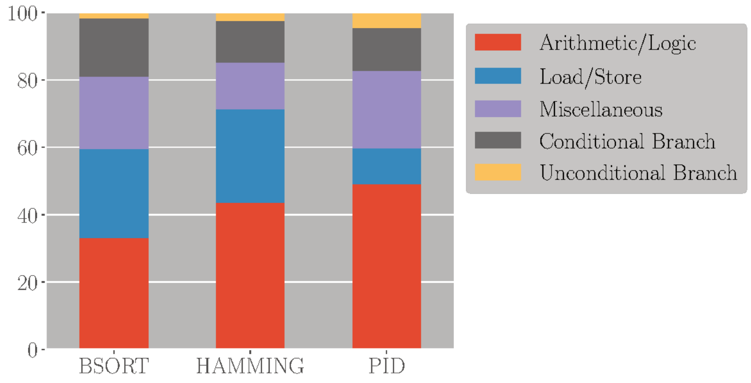
- *Exception* — The processor detected an unexpected event, generating a trap and aborting execution.

Note that only the last three categories (i.e., Wrong Result, Timed Out, and Exception) correspond to effects in which the processor exhibited erroneous behavior. For the sake of clarity, the term harmful effects will therefore be used in the remaining of the text whenever referring to any of these effects.

4.3.3 Workload Description

Three different workloads were used in the fault injection campaign. These include a proportional derivative controller (PID), a bubble sort implementation (BSORT) and a hamming encoder (HAMMING). The number of iterations executed in each workload was adapted so that all three had almost the same execution time, around 35000 clock cycles. About the composition of each workload, Figure 20 presents the dynamic instruction mix (i.e., distribution of executed instructions). In order to obtain these data, a non-synthesizable LEON3 configuration that disassembles instructions to the simulation console was used.

Figure 20 Dynamic Instruction Mix



Source: Author.

Ideally, a more comprehensive set of applications with a higher amount of executed instructions would be desirable. However, the total simulation time required ends up being unaffordable. In addition to simulation being considerably slower than real-time execution, it is essential to perform a high number of experiments with each workload in order to obtain high confidence in the results.

Regarding the compilation of each workload, the LEON3 *Bare-C Cross-Compiler* (COBHAM GAISLER AB, 2016) version 4.4.2 was used.

4.3.4 Implementation Details

The following subsections cover the implementation details of each of the functions performed by the Fault Injection Manager.

4.3.4.1 List Design Flip-Flops

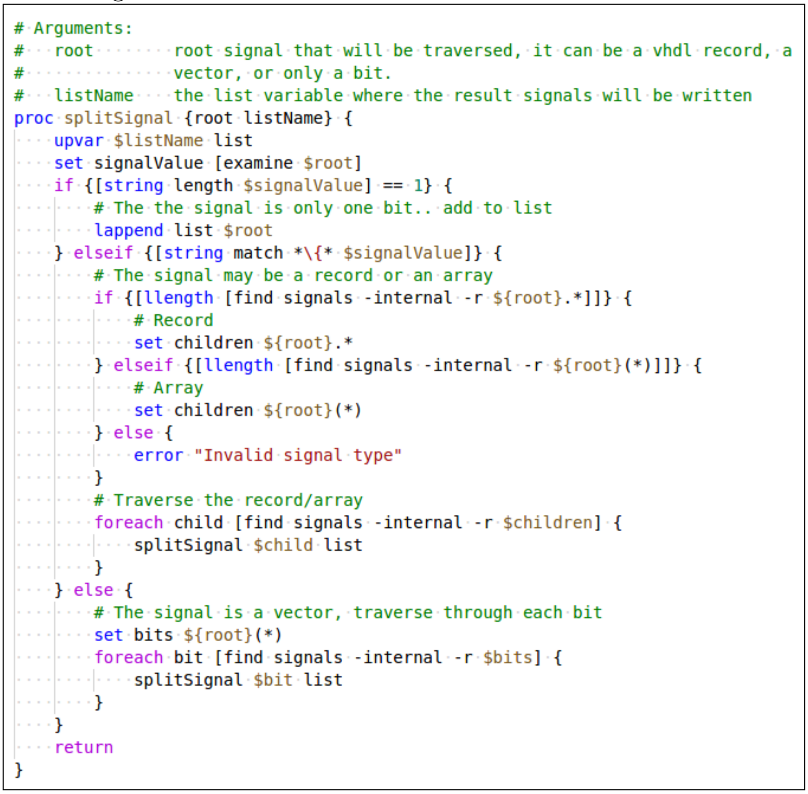
One of the key parts of the fault injection process is the selection of the locations where the faults will be injected. In the fault model chosen for this work, these locations will be the flip-flops inside the PROC3 module. Obtaining a list of all these flip-flops (hundreds) could be done manually, but it would be an exhausting process and susceptible to errors. In this sense, an algorithm has been developed that takes advantage of the centralization of all registers in a few VHDL records, and automatically lists each flip-flop (single bit) within each record. So the only part that must be done manually is to list the records of interest (eight in this work). In order to realize this algorithm, two commands from Modelsim were required:

1. *find* — This command locates objects by type and name. It can be used with the wildcard character (*) to substitute any other character. For example, it is possible to obtain all the signals inside a vector with the call: *find signals vector_name[*]*

2. *examine* — This command examines one or more objects and displays current values. The return value gives useful information regarding the signal type (single bit, vector, array, record). If the return value has only one character, the signal is a single bit. Else, if the return value contains curly brackets { }, it is an array or record.

Figure 21 presents the algorithm implemented in TCL. It receives as argument a VHDL signal of any type and an empty list. Through calls to the *examine* command and string comparison methods, it determines the signal type. If the signal is only one bit, it is appended to the list, and the algorithm returns. Otherwise, if the signal is composed of more than one bit, the algorithm is recursively executed in each of the internal signals, which are obtained through the *find* command. By the end, the list will contain all the one-bit signals that compose the signal that was provided as an argument.

Figure 21 – TCL Procedure that traverses a root signal and lists all the internal signals bitwise



```
# Arguments:
#   root        root signal that will be traversed, it can be a vhdl record, a
#               vector, or only a bit.
#   listName    the list variable where the result signals will be written
proc splitSignal {root listName} {
    upvar $listName list
    set signalValue [examine $root]
    if {[string length $signalValue] == 1} {
        # The the signal is only one bit.. add to list
        lappend list $root
    } elseif {[string match *\{* $signalValue]} {
        # The signal may be a record or an array
        if {[llength [find signals -internal -r ${root}.*]]} {
            # Record
            set children ${root}.*
        } elseif {[llength [find signals -internal -r ${root}(*)]]} {
            # Array
            set children ${root}(*)
        } else {
            error "Invalid signal type"
        }
        # Traverse the record/array
        foreach child [find signals -internal -r $children] {
            splitSignal $child list
        }
    } else {
        # The signal is a vector, traverse through each bit
        set bits ${root}(*)
        foreach bit [find signals -internal -r $bits] {
            splitSignal $bit list
        }
    }
    return
}
```

Source: Author.

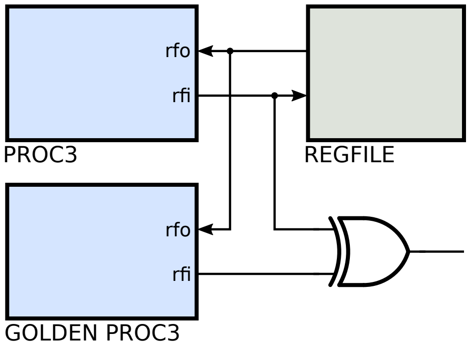
4.3.4.2 Fault Propagation Monitoring

This part of the Fault Injection Manager is responsible for monitoring the manifestation of the injected faults on each of the PROC3 output interfaces (listed in Section 4.2.2). Obtaining this results can help identifying the dominant fault propagation paths inside the processor architecture, which represents useful information for proposing efficient fault detection techniques.

To facilitate this data to be collected, the LEON3 VHDL implementation was modified to include two instances of the PROC3 module, one where the faults are injected which have both its inputs and outputs connected to the rest of the LEON3, and another only for reference with only its inputs connected. Note that by using this configuration, it is guaranteed that the PROC3 module used for reference does not interfere with the operation of the processor.

The comparison between the interfaces of both PROC3 is not made directly in the TCL script, as it would considerably increase simulation time. In this sense, a non-synthesizable comparator was added in the VHDL design. Thus, the TCL script only needs to be executed when the output of this comparator signalizes mismatches between the interfaces. Figure 22 illustrates this arrangement for only the register file interface, but the same approach is extended to all other interfaces. Note that the output of the GOLDEN PROC3 is only used for reference and is not connected to the register file.

Figure 22 – The arrangement used to monitor fault propagation in the PROC3 interfaces



Source: Author.

4.3.4.3 Fault Effect Monitoring

This part of the Fault Injection Manager is responsible for monitoring the effects of the injected faults in the processor operation. In this respect, each of the effects listed in Section 4.3.2 required a different monitoring approach.

The *Exception* condition could be readily observed by monitoring events in the LEON3 trap/exception signals, which are part of the Exception (EX) pipeline stage. This monitoring was entirely implemented in the TCL script, with a minor impact on the simulation time.

For the detection of the *Timed Out* condition, each workload was run once without any fault injected, and the following runs used the observed runtime as a reference. The condition is only triggered if the fault provokes an increase in the execution time of more than 10%. By this means, this approach aims to detect only locking conditions and not small delays.

For the detection of the *Wrong Result* condition, the workload applications were modified to include self-tests. These tests are periodically executed within the workload to verify if the computations performed are correct. To externalize the result of the self-tests, so that the Fault Injection Manager could monitor them, the software triggered events in the LEON3 general purpose I/O pins each time the self-tests identified errors.

If the workload has finished execution and none of the conditions above were identified, the Fault Injection Manager test for a *Latent* fault. This test is done by comparing the contents of all registers inside the PROC3 with the values obtained from a golden run (i.e., without any fault injected). If there is a mismatch in the comparison, the *Latent* condition is triggered.

Finally, if none of the above happened, the result of the injection experiment is that the fault had *no effect* in the processor operation and was overwritten before the end of the simulation.

4.3.4.4 Fault Injection

The primary task executed by the Fault Injection Manager is naturally the fault injection itself. Since the entire fault injection campaign is composed of hundreds of experiments, the injection process is confined within a loop. In this regard, the fault injection loop developed in this work consists of the following steps:

1. Choose a random flip-flop where the fault will be injected. The selection is done from the list generated by the algorithm presented in Section 4.3.4.1. It is important to note, that since the injection target is selected randomly at the bit level, by the end of the fault injection campaign, the number of injected faults by register will be proportional to the register size.

2. Choose a random instant for when the fault will be injected. The time interval considered for fault injection is comprehended between 20% and 90% of the simulation runtime. This prevents the injection of faults in the processor warm-up phase and also provides enough time for the fault to propagate.

3. Simulate until the chosen injection instant.

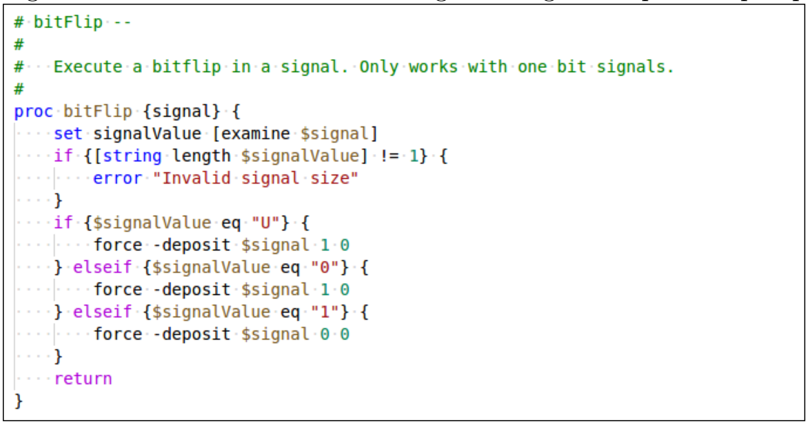
4. Inject the fault by forcing a bitflip in the target flip-flop. Note that as defined in the fault model, the value is not stuck and can be normally overwritten during the remaining of the simulation. The ModelSim command used for this purpose is the *force -deposit*, as shown in Figure 23

5. Simulate until the workload finishes execution, or the predefined timeout is reached.

6. Compare the final registers contents to the ones obtained from a golden run (executed previously), to test for latent faults.

7. Finally, updates all the logs with the collected statistics, for later analysis.

Figure 23 – TCL Procedure used for generating a bitflip in a flip-flop



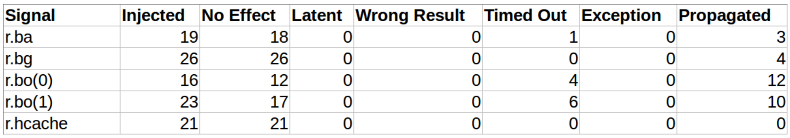
```
# bitFlip --
#
#   Execute a bitflip in a signal. Only works with one bit signals.
#
proc bitFlip {signal} {
    set signalValue [examine $signal]
    if {[string length $signalValue] != 1} {
        error "Invalid signal size"
    }
    if {$signalValue eq "U"} {
        force -deposit $signal 1 0
    } elseif {$signalValue eq "0"} {
        force -deposit $signal 1 0
    } elseif {$signalValue eq "1"} {
        force -deposit $signal 0 0
    }
    return
}
```

Source: Author.

4.3.4.5 Log Generation

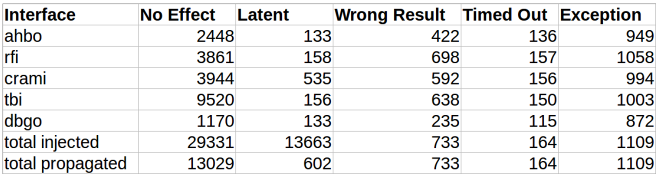
The last operation performed by the Fault Injection Manager is the log generation. This activity consists of compiling all the information gathered throughout the fault injection experiments, and exporting it to structured tables in CSV files. Given the variety of the information collected, three distinct tables are generated. The first table, exemplified in Figure 24(a), contains the compilation of the fault effects observed for each of the signals investigated during the experiments. The second table, exemplified in Figure 24(b), contains the number of observed faults and their respective effects in each of the analyzed interfaces. The third and last table, exemplified in Figure 24(c), contains for each interface the calculated latencies between the injection instant and the moment the faults manifested.

Figure 24 – Example logs generated by the Fault Injection Manager



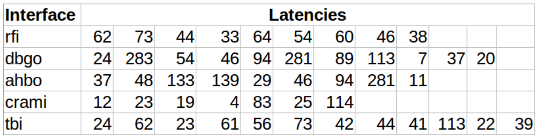
Signal	Injected	No Effect	Latent	Wrong Result	Timed Out	Exception	Propagated
r.ba	19	18	0	0	1	0	3
r.bg	26	26	0	0	0	0	4
r.bo(0)	16	12	0	0	4	0	12
r.bo(1)	23	17	0	0	6	0	10
r.hcache	21	21	0	0	0	0	0

(a)



Interface	No Effect	Latent	Wrong Result	Timed Out	Exception
ahbo	2448	133	422	136	949
rfi	3861	158	698	157	1058
crami	3944	535	592	156	994
tbi	9520	156	638	150	1003
dbgo	1170	133	235	115	872
total injected	29331	13663	733	164	1109
total propagated	13029	602	733	164	1109

(b)



Interface	Latencies											
rfi	62	73	44	33	64	54	60	46	38			
dbgo	24	283	54	46	94	281	89	113	7	37	20	
ahbo	37	48	133	139	29	46	94	281	11			
crami	12	23	19	4	83	25	114					
tbi	24	62	23	61	56	73	42	44	41	113	22	39

(c)

Source: Author.

4.4 PARTIAL REDUNDANCY TECHNIQUE

Among the forms of redundancy presented in Section 2.4, the strategy chosen for implementation in this work is a variation of the TMR technique. The variation happens in the sense that the technique will not be applied in whole modules, but instead in only the components identified as most vulnerable in the fault injection campaign. Therefore, a more appropriate denomination that will be used in the remaining of this work is partial TMR (PTMR).

Following the choice made for the fault model in Section 4.3.1, the PTMR technique developed in this work is not intended for protecting the design against SETs, but rather to protect the sequential components from SEUs. This option is in accordance with the objectives of this work, in the perspective that sequential elements are more vulnerable than the combinational elements in FPGAs, and therefore should be the focus on a partial protection strategy. With this in mind, the TMR structure adopted is the one from Figure 5 in Section 2.4.1, where only the sequential elements are replicated, with all three replicas receiving the same input from the combinational logic.

In order to implement the PTMR technique, the necessary modifications were made directly to LEON3 VHDL model, more specifically in the VHDL entities containing flip-flops identified as vulnerable. Note that implementing directly in the VHDL model requires some attention during the synthesis step, since the tools may perform optimizations that remove redundant logic. The necessary actions will be described later in the text.

Furthermore, given the adoption of a structured VHDL design methodology by the LEON3 authors, the modifications to implement the PTMR technique could be realized in a systematical way in each VHDL entity. The necessary steps are summarized below with the effects of each modification represented in Figure 25.

1. The implementation begins with the definition of a new VHDL record type containing all the registers that will be protected. The use of a record type facilitates the recognition of the tripled registers throughout the code and also respect the design methodology. Next, two VHDL signals are declared, using the recently defined record as the type. The combination of these two new signals, with the registers already present in the design, set the triplication of the sequential logic. Figure 25(a) presents the resulting design. Note that the replicated registers are still not

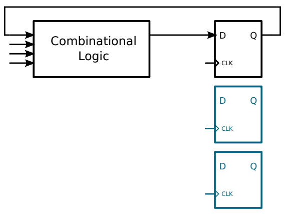
connected to the rest of the design.

2. The next step consists in defining the majority voter logic and connecting to its inputs the replicated registers. As there will be as many majority voters as there are protected registers, a reasonable approach is to define the majority voter logic in a VHDL function and call it for each register. Better yet is to have a procedure that loops over each bit within the record calling the majority voter function. Observe that this procedure must be executed at the beginning of the combinational process (from the two processes per entity approach) so that the voter output is accessible for the remaining of the combinational logic. Figure 25(b) presents the design after this step. Note that still only the majority voter inputs are connected.

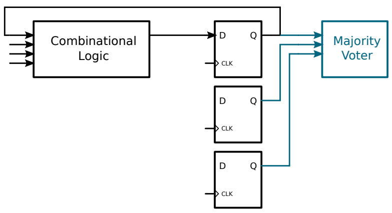
3. The next step is to disconnect the protected registers from the remaining of the combinational logic (except the voter inputs) and connect the respective majority voters outputs instead, as can be seen in Figure 25(c). Unfortunately, this step cannot be performed with a straightforward Find & Replace, on the contrary, it is the most demanding part and requires the manual inspection of each signal assignment resulting in hundreds of modifications in the VHDL source code.

4. Finally, the last remaining modification is to ensure that all replicas of each protected register receive the same input at the same instant, as can be seen in Figure 25(d). In the structured VHDL design methodology, this modification is performed within the sequential process and consists of replicating the signal assignments involving the registers of interest.

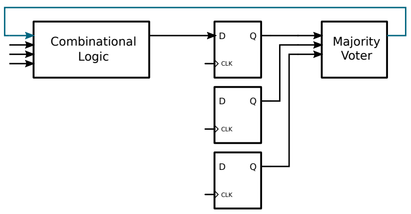
Figure 25 – Steps for implementing the PTMR technique



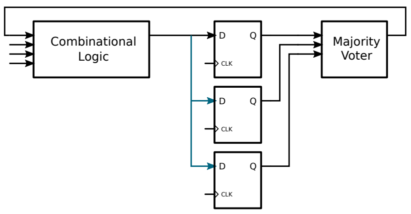
(a)



(b)



(c)



(d)

Source: Author.

5 RESULTS AND DISCUSSION

The presentation and discussion of the results is carried out in two parts: it begins with an extensive analysis of the fault injection campaign in the LEON3's unprotected architecture (i.e., non-fault tolerant), and continues with the evaluation of the performance of the proposed partial redundancy technique. In total, 45000 fault injection experiments were performed in the non-fault tolerant LEON3, and another 18000 experiments were performed in the version of the processor with the proposed technique implemented. Note that most of the analyzes that will be presented refer to the aggregate result of the three workloads, except when expressly mentioned.

5.1 FAULT INJECTION IN THE UNPROTECTED LEON3

5.1.1 Overall Performance

In Figure 26 a plot containing the overall results of the fault injection experiments targeting the LEON3 processor core is presented. It is interesting to see that even without any mechanism for SEU mitigation, most of the injected faults were overwritten without compromising the program execution. This can be explained by the fact that a large part of the processor components are not used in every instruction, and the injected faults in those components are masked when new instructions are fetched into the pipeline. Another relevant result is that the processor trap/exception mechanism was able to detect only slightly more than half of the faults that led to harmful effects, with the other half going completely unnoticed, indicating the need to include other detection mechanisms in the case of critical applications.

Figure 27 presents a plot with the results broken down by the workload running on the processor during the experiments. The observed effects are overall consistent throughout all three programs, with a discrepancy in the BSORT, which presented a more significant occurrence of wrong results. This result was surprising, given that the BSORT had the least amount of arithmetic/logic instructions. The expectation was that this type of instruction would be more prone to this effect since the injected faults could lead to wrong calculations. However, it has been realized that other forms of data manipulation (e.g., memory operations) are equally likely to generate this effect in the presence of faults, so that is difficult to draw any conclusion regarding vulnerability based solely on the distribution of instructions.

Figure 26   Overall PROC3 Performance

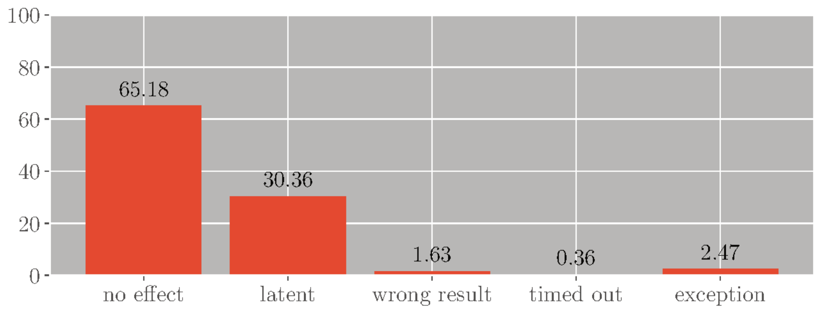
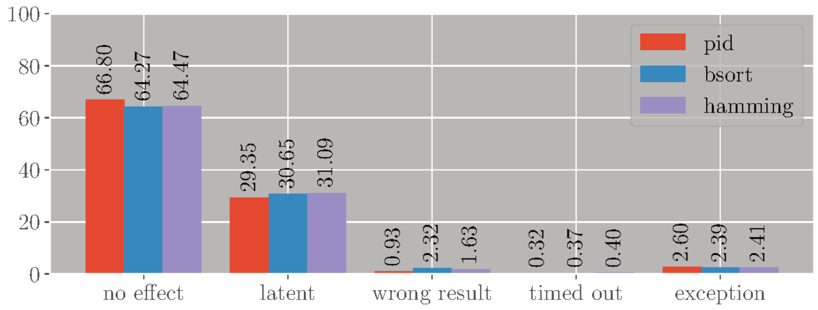
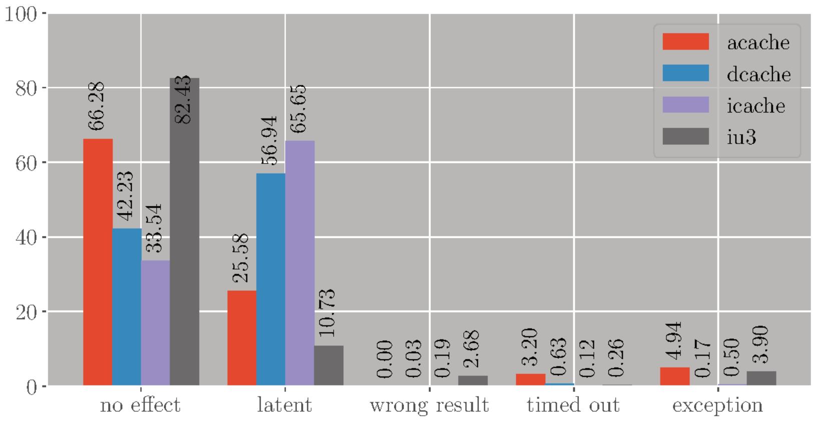


Figure 27   Overall Performance by Workload



With respect to the individual performance of each module that composes the PROC3, the results can be seen in Figure 28. It is possible to observe that the cache controllers present a much higher ratio of latent faults than the integer pipeline. This is due to some of the registers being used on only some specific cache configurations (e.g., when using the LRU replacement policy), which causes the injected faults in those registers to never be overwritten. Note that during the synthesis step, these registers tend to be removed from the design. Furthermore, the higher exception value observed in the *acache* module is expected, since this module has a small number of registers, and one of them corresponds to a critical register used for error warning.

Figure 28 Overall Performance by Module

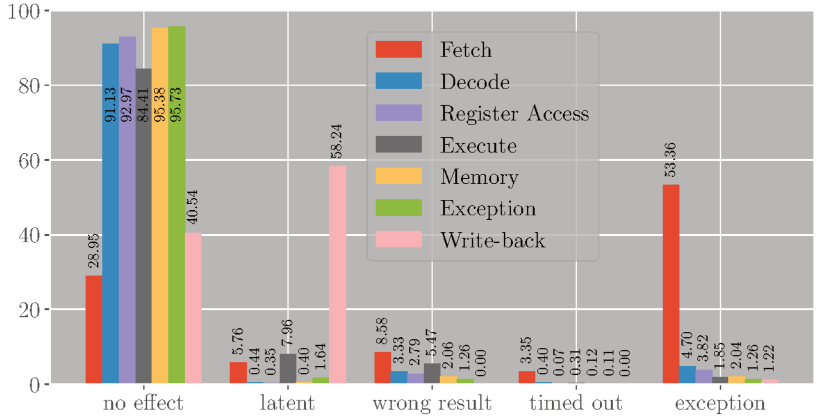


5.1.2 Integer Pipeline Performance

Being the integer pipeline the largest module in the LEON3 processor core, it is of great interest to analyze it closely. For this purpose, the IU3 registers were split by pipeline stage, and the SEU vulnerability obtained for each stage was investigated separately. The same analysis was done in (TOULOUPIS et al., 2007) for the LEON2 processor, which has a slightly different pipeline structure with only five stages, and the conclusions were very similar. The investigation results are presented in Figure 29.

According to the results, the *Fetch* stage presents a much higher rate of harmful effects than the other pipeline stages. This behavior is expected, since the *Fetch* stage contains only two registers, with one of them being the *Program Counter* (PC). Faults injected into the PC may cause the wrong instruction being fetched, or even an attempt to read an invalid memory location. Another remark is that the *Write-Back* stage contains a high rate of latent faults. This is due to this stage being composed mostly of special registers that were barely accessed during the execution of the workloads.

Figure 29 IU3 Performance Split by Pipeline Stage

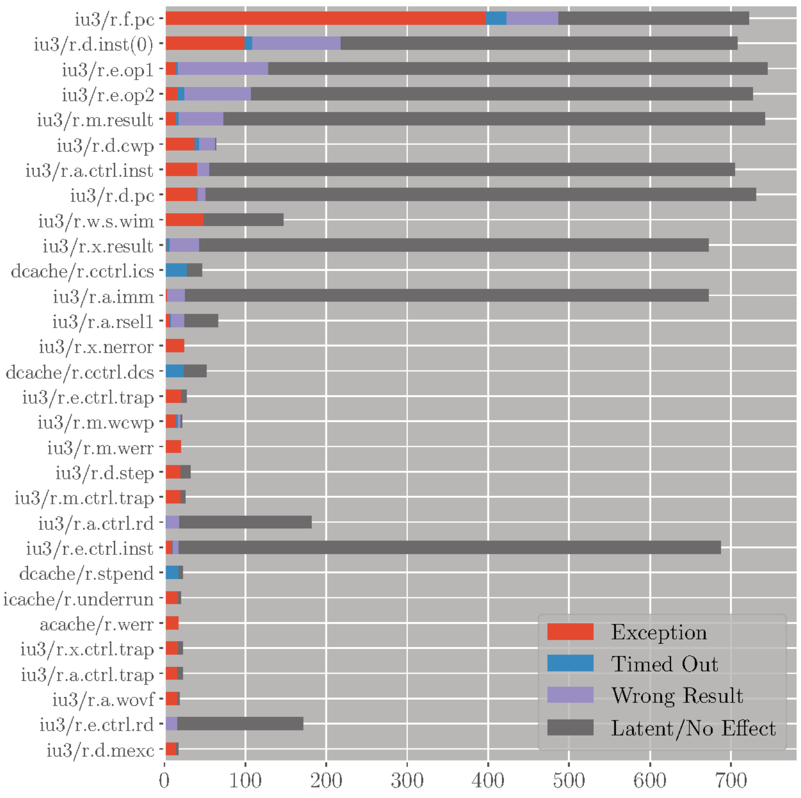


5.1.3 Individual Register Performance

Figure 30 contains the individual performance of the thirty registers which presented the highest number of harmful effects and therefore are the primary candidates to be protected through the proposed partial redundancy technique. The registers are sorted with the most vulnerable placed at the top. Note that as mentioned before, the number of injected faults by register is proportional to its size. Also, since the analysis is based on absolute quantities, even though some one-bit registers presented a 100% exception rate, they are considered less vulnerable than larger registers with smaller exception rates, but higher absolute values. Ideally would be interesting to perform a bit-level analysis instead of a register-level, as even inside the same register, some bits may be more or less sensitive to faults. However, the effort required to implement triplication at the bit-level is considerably higher, requiring modifications to many more lines in the VHDL description. Having an automated tool to perform TMR insertion would allow following this approach.

As can be seen in the results, the program counter (*r.f.pc*) presented the worst performance by a significant margin, followed by other important registers such as the fetched instruction (*r.d.inst*), both the ALU operands (*r.e.op1* and *r.e.op2*) and the ALU operation result (*r.m.result*). Note that the non-uniform distribution of harmful effects between the registers, jointly with the abrupt drop on these effects moving down the plot, is a strong indication that it is possible to achieve significant improvements in fault tolerance by protecting only some registers. Moreover, it is interesting to observe how the fault effects relate to the register functionality. Faults injected in registers used by the ALU frequently lead to wrong results, while the most common effect of faults injected in error and trap registers is the generation of exceptions.

Figure 30 Individual Register Performance



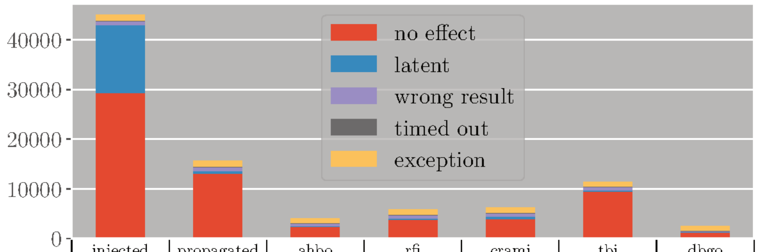
5.1.4 Fault Propagation

Figure 31 presents the results obtained regarding fault propagation to the PROC3 boundaries. The *injected* bar contains the fault effect distribution of all injected faults, the *propagated* bar only contains the ones that propagated to one or more interfaces, and the rest of the bars correspond to the propagation on each interface individually. As shown in the results, only one-third of the injected faults have led to a disturbance in the interfaces. Most *no effect* and *latent* faults remained inside the PROC3. Also, as expected, all the faults that generated harmful effects propagated to at least one interface.

Between the interfaces, the TBI had the highest propagation rate, mostly due to the high number of *no effect* faults. This behavior is consistent since the trace buffer track statistics of all the executed instructions (e.g., address, opcode and result). As to faults that led to harmful effects, the register file interface is where they have manifested the most. It is important to note, however, that the results indicate that harmful effects cannot be detected by looking to only a single interface, for complete coverage, a fault detection strategy would need to look more than one location.

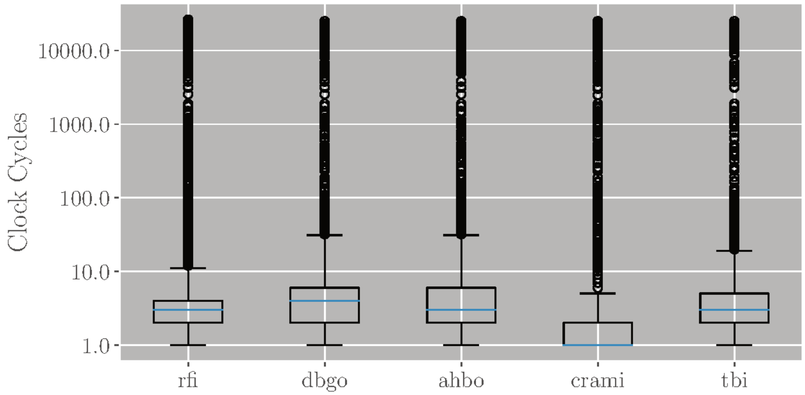
With respect to the fault manifestation times, Figure 32 contains a box plot with the obtained values in each PROC3 interface. It follows that for all interfaces, most propagated faults take less than ten clock cycles to manifest; however, they can remain latent for tens of thousands of cycles before they manifest. Notably, in the cache memory array interface, more than half of the propagated faults manifested in the clock cycle following the injection.

Figure 31 Fault Propagation in PROC3 interfaces



	injected	propagated	ahbo	rfi	crami	tbi	dbgo
no effect	29331	13029	2448	3861	3944	9520	1170
latent	13663	602	133	158	535	156	133
wrong result	733	733	422	698	592	638	235
timed out	164	164	136	157	156	150	115
exception	1109	1109	949	1058	994	1003	872

Figure 32 Boxplot of the Fault Manifestation Time in PROC3 interfaces. The bottom and top of the box corresponds to the first and third quartiles, the line inside the box is the median, the whiskers are at 1.5 IQR, values outside of that range are represented by dots.



5.2 FAULT INJECTION IN THE PROTECTED LEON3

5.2.1 Overall Performance

As shown in Figure 30, the concentration of harmful effects in a small number of registers presents a clear indication of the possibility of obtaining a significant improvement in the overall reliability by using a selective protection strategy. The same conclusion can be obtained by looking at Figure 33, where is derived the expected SBU tolerance when the most sensitive registers are protected (e.g., through spatial redundancy). In order to validate these results, the thirty registers identified as most vulnerable (from Figure 30) were protected using the partial TMR technique previously introduced, and the fault injection experiments were redone. Figure 34 contains the updated results. As expected, the number of *latent* and *no effect* faults improved to around 99.25%, which is a 3.71% increase compared to the original version. The improved fault tolerance means having only one incorrect computation every 133 faults, whereas in the original architecture there was one every 22.

Figure 33 Expected SBU Tolerance

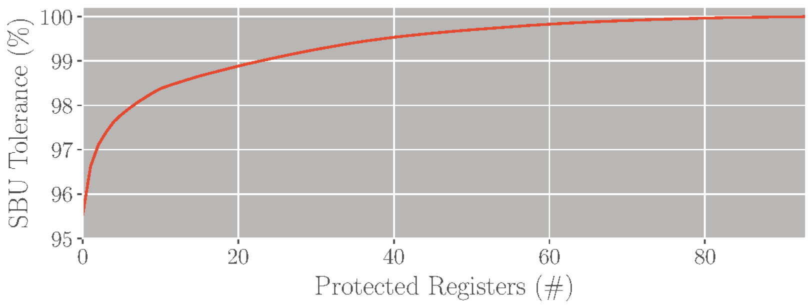
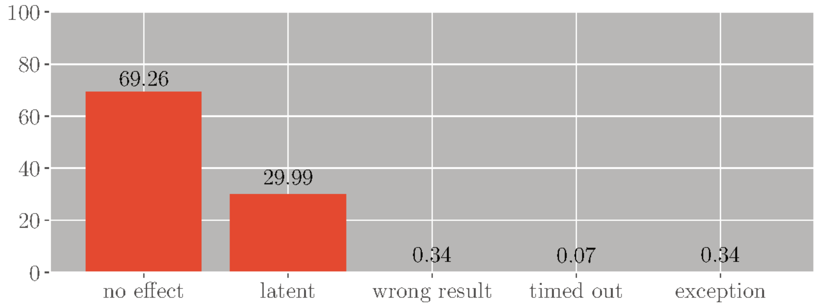


Figure 34 Protected LEON3 Overall Performance



5.2.2 Synthesis Area/Performance Overhead

In order to evaluate the performance and area penalty of the proposed technique, three distinct variants of the LEON3 were synthesized for the Xilinx Spartan 6 FPGA (XC6SLX45). The first variant corresponds to the LEON3 without any custom fault tolerance strategy applied. The second variant is the LEON3 with the partial redundancy strategy employed (LEON3 PTMR). Finally, the third variant consists of the LEON3 with complete triplication of the PROC3, however without tripling any other module (LEON3_TMR).

The tool used in the synthesis was the Xilinx PlanAhead 14.7. As previously mentioned, with default flags the tool optimize out all the replicated registers. In order to keep the redundancy, the flag *equivalent_register_removal* was set to False.

Tables 2 to 4 present the synthesis results. Note that both the TMR and PTMR variants showed a small drop in the maximum achievable frequency. This decrease occurred mainly due to the inclusion of the majority voters in the critical path. It is important to observe that the use of majority voters is different between the TMR and PTMR approaches. Particularly, the PTMR technique required a more significant number of voters (one voter for each protected bit), with the voters placed internally the PROC3. In contrast, the TMR technique had the voters outside the PROC3, with one voter for each output.

Concerning the resources utilization, the advantage of the PTMR technique is clear, showing only a modest increase over the unprotected LEON3. Note that the figures in resources utilization comprise the design as a whole, including parts that were not protected (memory hierarchy and peripherals), for that reason the TMR overhead is less than 200% in most resources.

Table 2 – Design Maximum Frequency

	LEON3	LEON3 PTMR		LEON3 TMR	
	(Mhz)	(Mhz)	Overhead	(Mhz)	Overhead
Frequency	51.031	50.546	-0.960%	50.279	-1.496%

Source: Author.

Table 3 – FPGA Resources Utilization

	LEON3	LEON3 PTMR		LEON3 TMR	
	(#)	(#)	Overhead	(#)	Overhead
Slice Registers	2864	3720	29.888%	5932	107.123%
Slice LUTs	5664	6249	10.328%	12932	128.319%
DSPs	4	4	0.000%	12	200.000%

Source: Author.

Table 4 – FPGA Power Consumption

	LEON3	LEON3 PTMR		LEON3 TMR	
	(W)	(W)	Overhead	(W)	Overhead
Supply Power	0.903	0.918	1.661%	0.986	9.192%

Source: Author.

6 CONCLUSIONS

This work presented a partial redundancy strategy for protecting the processing logic of soft-core processors from SEUs. The suggested approach consists of only using TMR in the processor's registers identified as most vulnerable. The number of registers protected depends heavily on the fault tolerance required by the application and other constraints such as area and power. In the task of identifying the most sensitive registers, it is proposed to conduct an extensive fault injection campaign. In this sense, the most sensitive registers are those that in the presence of faults are more likely to cause the processor to exhibit erroneous behavior.

Given the characteristics of this study, it was decided to use a simulation-based fault injection technique. This choice was in agreement with previous works, which also opted for this same method. This technique proved to be satisfactory, considering that it allowed the execution of most of the desired tests. The main limitation found, which had already been reported in other studies, is the simulation time, which prevented the use of more representative workloads such as MiBench (GUTHAUS et al., 2001). In the experiments performed, using an Intel Core i7–4700MQ @ 3.2Ghz with four cores, the total simulation time was about 15 days. Four instances of the simulator were run simultaneously, one in each core of the processor, and this strategy helped avoiding an even longer simulation time.

The experimental results from the campaign in the unprotected LEON3 revealed that most of the faults that led the processor to erroneous behavior were limited to a small group of registers, mainly the program counter and the ALU operands. Moreover, was found that only a third of the injected faults propagated to the CPU core interfaces. These results presented substantial evidence that a reliability improvement was possible by protecting only the most vulnerable parts of the processor.

Later, with the partial redundancy technique implemented in the LEON3, a new fault injection campaign showed that by protecting only 30 registers out of a total of 362, it was already possible to have a reduction over six times in the number of faults that led to harmful effects. At the same time, the costs in terms of FPGA resource utilization were much lower than a full triplication (between 3 and 12 times). It is necessary to note, however, that the proposed technique is not ideal for every application, and it does not in any way replaces the classical

TMR technique when higher fault tolerance levels are required.

Finally, the main contribution of this research was the implementation and evaluation of the partial redundancy technique, taking into account that in the considered references this approach had only been suggested. The importance of this contribution was confirmed by the community with a paper accepted for oral presentation in the 19th Latin-American Test Symposium.

6.1 FUTURE WORKS

The work described in this document represented a first step towards the goal of developing a complete, low overhead, fault tolerance technique for soft core processors. As such, this first iteration was not expected to tackle all the problems at once and, therefore, there are still many studies to be performed and improvements to be made. In this sense, the following future works are suggested:

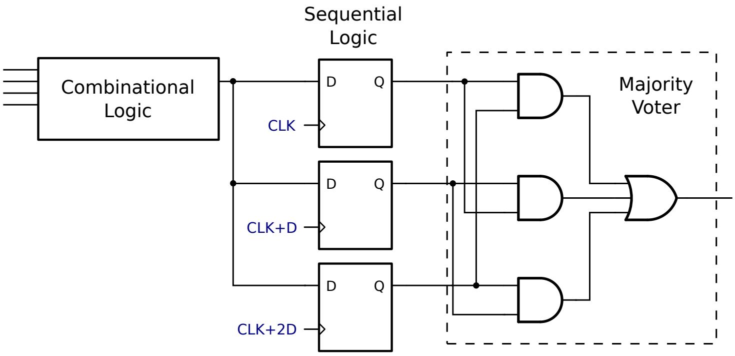
- Explore alternative strategies for identifying the most vulnerable components in the processor. These strategies could be used in conjunction with the fault injection campaign, and could even be used to limit the number of experiments required. One such alternative is to record the circuit toggle activity while running a workload. This information could be used to identify unused or rarely accessed elements, which have a high probability of not impacting the processor operation even in the presence of faults.

- Develop an automated design flow for employing the partial redundancy in the processor. With automation it would be possible to employ the redundancy at the flip-flop level rather than the registers, leading to an even more efficient technique. For implementation by hand, this possibility was discarded due to the tremendous effort required.

- Evaluate the proposed technique in combination with strategies to protect the memory hierarchy. With the complete system fault-tolerant, hardware-based fault injection campaigns could be performed to measure the overall reliability.

- Improve the fault tolerance by including SET mitigation, thus preventing radiation-induced transients in the combinational logic from being stored in the registers protected by the PTMR strategy. However, instead of using the Full TMR approach presented in Section 2.4.1, which would result in significant costs in area utilization, diverting from the proposal of this work (i.e., develop a low overhead technique), the concept would be to merge both hardware and time redundancy, as shown in Figure 35. In this approach, the flip-flops from the TMR strategy are latched at different time instants, thus ensuring that the radiation-induced transients are latched by only one of the flip-flops, being subsequentially masked by the majority voter. Note that this implementation requires three separate clock signals, with the clocks separated by a delay (d). The length of the delay will determine the maximum SET duration that the circuit can still mask, posing a trade-off between performance and fault tolerance.

Figure 35 – Triple Modular Redundancy with SET detection



Source: Adapted from (KASTENSMIDT; CARRO, et al., 2006).

REFERENCES

ALTERA CORPORATION. *White Paper: Introduction to Single-Event Upsets*. 2013.

ARLAT, J. et al. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, IEEE, vol. 52, no. 9, pp. 1115–1133, 2003.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, IEEE, vol. 1, no. 1, pp. 11–33, 2004.

BAUMANN, R. C. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, IEEE, vol. 5, no. 3, pp. 305–316, 2005.

________. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, IEEE, vol. 22, no. 3, pp. 258–266, 2005.

________. Soft errors in commercial integrated circuits. *International Journal of High Speed Electronics and Systems*, World Scientific, vol. 14, no. 02, pp. 299–309, 2004.

BENEDETTO, J. et al. Digital single event transient trends with technology node scaling. *IEEE Transactions on Nuclear Science*, IEEE, vol. 53, no. 6, pp. 3462–3465, 2006.

CARMICHAEL, C. Triple module redundancy design techniques for Virtex FPGAs. *Xilinx Application Note XAPP197*, vol. 1, 2001.

CARMICHAEL, C.; TSENG, C. W. Correcting single-event upsets in Virtex-4 FPGA configuration memory. *Xilinx Corporation*, 2009.

CHEN, C.-L.; HSIAO, M. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, IBM, vol. 28, no. 2, pp. 124–134, 1984.

COBHAM GAISLER AB. BCC-Bare-C Cross-Compiler User's Manual, 2016.

________. *LEON3 Processor*. Dec. 2017. Address: `<http://www.gaisler.com/index.php/products/processors/leon3>`.

DUBROVA, E. *Fault-tolerant design*. Springer, 2013.

FENG, S. et al. Shoestring: probabilistic soft error reliability on the cheap. In: ACM, 1. *ACM SIGARCH Computer Architecture News*. 2010. vol. 38, pp. 385–396.

GADLAGE, M. J. et al. Single event transient pulse widths in digital microcircuits. *IEEE transactions on nuclear science*, IEEE, vol. 51, no. 6, pp. 3285–3290, 2004.

GAISLER, J. A structured VHDL design method. *Fault-tolerant microprocessors for space applications*, pp. 41–50, 2011.

GAISLER, J. et al. GRLIB IP core user's manual. *Gaisler research*, 2017.

GOLOUBEVA, O. et al. *Software-implemented hardware fault tolerance*. Springer Science & Business Media, 2006.

GOMES, I. A. et al. Exploring the use of approximate TMR to mask transient faults in logic with low area overhead. *Microelectronics Reliability*, Elsevier, vol. 55, no. 9-10, pp. 2072–2076, 2015.

GORDON, M. et al. Measurement of the flux and energy spectrum of cosmic-ray induced neutrons on the ground. *IEEE Transactions on Nuclear Science*, IEEE, vol. 51, no. 6, pp. 3427–3434, 2004.

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. In: IEEE. *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. 2001. pp. 3–14.

HAMMING, R. W. Error detecting and error correcting codes. *Bell Labs Technical Journal*, Wiley Online Library, vol. 29, no. 2, pp. 147–160, 1950.

HUSSEIN, J.; SWIFT, G. *Mitigating single-event upsets*. 2015.

ITURBE, X.; VENU, B.; OZER, E. Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU. In: IEEE. *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2016 IEEE International Symposium on*. 2016. pp. 91–96.

JOHNSON, B. W.; AYLOR, J. H.; HANA, H. H. Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder. *IEEE journal of solid-state circuits*, IEEE, vol. 23, no. 1, pp. 208–215, 1988.

KASTENSMIDT, F. L.; CARRO, L.; REIS, R. *Fault-tolerance techniques for SRAM-based FPGAs*. Springer, 2006. vol. 32.

KASTENSMIDT, F. L.; REIS, R. Fault tolerance in programmable circuits. *Radiation Effects on Embedded Systems*, Springer, pp. 161–182, 2007.

KEYS, A. et al. Radiation hardened electronics for space environments (RHESE) project overview. In: GEORGIA INSTITUTE OF TECHNOLOGY.

KIM, S.; SOMANI, A. K. On-line integrity monitoring of microprocessor control logic. *Microelectronics journal*, Elsevier, vol. 32, no. 12, pp. 999–1007, 2001.

KLETZING, C. et al. The electric and magnetic field instrument suite and integrated science (EMFISIS) on RBSP. *Space Science Reviews*, Springer, vol. 179, no. 1-4, pp. 127–181, 2013.

LABEL, K. A.; GATES, M. Single-event-effect mitigation from a system perspective. *IEEE Transactions on Nuclear Science*, IEEE, vol. 43, no. 2, pp. 654–660, 1996.

LESEA, A. et al. The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs. *IEEE Transactions on Device and Materials Reliability*, IEEE, vol. 5, no. 3, pp. 317–328, 2005.

LYONS, R. E.; VANDERKULK, W. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, IBM, vol. 6, no. 2, pp. 200–209, 1962.

MARKIEWICZ, W. J. et al. Venus monitoring camera for Venus Express. *European Geosciences Union. 1st General Assembly Nice*, 2004.

MARTINS, V. M. G. et al. A dynamic partial reconfiguration design flow for permanent faults mitigation in FPGAs. *Microelectronics Reliability*, Elsevier, vol. 83, pp. 50–63, 2018.

MENTOR GRAPHICS. *ModelSim*. Address: `<https://www.mentor.com/products/fv/modelsim/>`.

NEUMANN, J. van. Probabilistic logics and synthesis of reliable organisms from unreliable components, Automata Studies. *Annals of Mathematical Studies*, vol. 34, pp. 43–98, 1956.

POIVEY, C.; GRANDJEAN, M.; GUERRE, F. Radiation characterization of microsemi ProASIC3 flash FPGA family. In: IEEE. *Radiation Effects Data Workshop (REDW), 2011 IEEE*. 2011. pp. 1–5.

PRADHAN, D. K. *Fault-tolerant computer system design*. Prentice-Hall, 1996.

PRATT, B. et al. Improving FPGA design robustness with partial TMR. In: IEEE. *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*. 2006. pp. 226–232.

REBAUDENGO, M.; REORDA, M. S.; VIOLANTE, M. Accurate analysis of single event upsets in a pipelined microprocessor. *Journal of Electronic Testing*, Springer, vol. 19, no. 5, pp. 577–584, 2003.

SAJID, M. et al. Single Event Upset rate determination for 65 nm SRAM bit-cell in LEO radiation environments. *Microelectronics Reliability*, Elsevier, vol. 78, pp. 11–16, 2017.

SAMUDRALA, P. K.; RAMOS, J.; KATKOORI, S. Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs. *IEEE transactions on Nuclear Science*, IEEE, vol. 51, no. 5, pp. 2957–2969, 2004.

SANCHEZ-CLEMENTE, A. J.; ENTRENA, L.; GARCIA-VALDERAS, M. Partial TMR in FPGAs using approximate logic circuits. *IEEE Transactions on Nuclear Science*, IEEE, vol. 63, no. 4, pp. 2233–2240, 2016.

TONG, J. G.; ANDERSON, I. D.; KHALID, M. A. Soft-core processors for embedded systems. In: IEEE. *Microelectronics, 2006. ICM'06. International Conference on.* 2006. pp. 170–173.

TOULOUPIS, E. et al. Study of the effects of SEU-induced faults on a pipeline protected microprocessor. *IEEE Transactions on Computers*, IEEE, vol. 56, no. 12, pp. 1585–1596, 2007.

TRAVESSINI, R. et al. Processor Core Profiling for SEU Effect Analysis. 2018.

VILLA, P. R. et al. Processor Checkpoint Recovery for Transient Faults in Critical Applications. 2018.

WANG, J. Radiation effects in FPGAs. CERN, 2003.

WANG, J. et al. SRAM based re-programmable FPGA for space applications. *IEEE Transactions on Nuclear Science*, IEEE, vol. 46, no. 6, pp. 1728–1735, 1999.

WU, K.; KARRI, R. Fault secure datapath synthesis using hybrid time and hardware redundancy. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, vol. 23, no. 10, pp. 1476–1485, 2004.

ZIADE, H.; AYOUBI, R. A.; VELAZCO, R., et al. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.