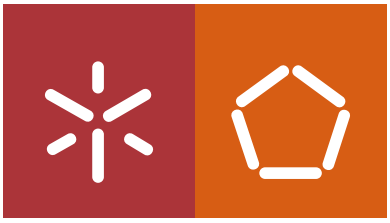
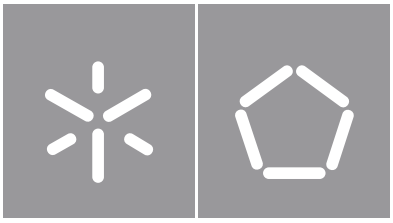


Universidade do Minho
Escola de Engenharia

Ivo da Cruz Marques

A Loosely-Coupled Arm and RISC-V Locksteping Technology

Janeiro de 2020



Universidade do Minho
Escola de Engenharia

Ivo da Cruz Marques

A Loosely-Coupled Arm and RISC-V Locksteping Technology

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Adriano José Conceição Tavares

Janeiro de 2020

Agradecimentos

Primeiramente, gostaria de agradecer ao meu orientador, professor doutor Adriano Tavares, pela dedicação e disponibilidade quando mais foi preciso, assim como todo o conhecimento que partilhou ao longo do meu percurso académico. Também não poderia deixar de agradecer aos professores doutores Sandro Pinto e Tiago Gomes, pelo apoio e orientação ao longo desta dissertação.

Aos meus colegas de laboratório, que me acompanharam nesta jornada e proporcionaram momentos de trabalho assim como de pausas formidáveis. Aos meus amigos mais próximos, que estiveram presentes em vários momentos e foram fulcrais para realizar este percurso, principalmente aos meus amigos Cristiano, Daniel, José Pedro, José Nuno e Valter. E um especial obrigado para a Catarina, por me acompanhar, ajudar e apoiar.

Não podia deixar de agradecer à minha família, aos meus irmãos, aos meus pais e aos meus avós, por sempre me darem apoio, assim como todos os esforços que fazem por mim. Por último, deixo aqui um especial e carinhoso obrigado à minha avó Rosa, por todo o suporte e palavras amigas em momentos de mais aflição. Um grande obrigado a toda a minha família pelo amor, alegria e atenção.

A todas as pessoas que contribuíram para este percurso, um forte obrigado.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Tecnologia *Lockstep* em processadores Arm e RISC-V recorrendo a uma abordagem *loosely-coupled*

Nos últimos anos, o grande crescimento tecnológico tem originado o surgimento de novas necessidades, onde dispositivos eletrónicos e seres humanos passam a ter um maior contacto. Este crescimento, contudo, levanta problemas de fiabilidade e segurança. E apesar de na ciência computacional surgirem melhorias de desempenho e de eficiência energética, devido à redução dos transístores, altas frequências de relógio, e baixas tensões de execução do núcleo de processamento, estas trazem consigo lacunas na fiabilidade dos sistemas, tornando-os mas suscetíveis a faltas. Por exemplo, esta nova geração tecnológica é cada vez mais sensível a radiações que podem despoletar *Single Event Upset (SEU)*.

Esta dissertação visa fornecer uma nova solução para sistemas tolerantes a falhas, denominada de Lock-V, que combina duas técnicas, de forma a responder à lacuna atual. A solução foi implementada sobre a Microsemi SmartFusion2 que inclui um microcontrolador e um *Field-Programmable Gate Array (FPGA)* na mesma plataforma, esta solução consiste numa arquitetura *Dual-Core Lockstep (DCLS)* combinada com diversidade de desenho ao nível do conjunto de instruções que é obtida pelo uso de dois núcleos diferentes, um *hard-core* Arm Cortex-M3 e um *soft-core* com base em RICS-V. O DCLS é apoiado por um acelerador desenvolvido na FPGA e proporciona ao sistema a capacidade de deteção de error, através da comparação *loosely-coupled* das saídas dos núcleos de processamento. Para além disso, esta dissertação fornece uma *framework*, que adiciona ao sistema a capacidade de auto-recuperação.

De forma a validar o sistema, foi desenvolvido um mecanismo de injeção de faltas, que testa a arquitetura Lock-V. Como proteger a memoria está fora do âmbito desta dissertação, e como tal, a injeção de faltas foi aplicada apenas nos registros do processador, que normalmente são os mais vulneráveis a faltas se excluirmos as memórias. Estes testes, provam a eficiência do sistema Lock-V como um sistema tolerante a falhas. Para além disso, esta arquitetura, devido ao seu mecanismo *lockstep* é um sistema tolerante a falhas contra SEU, e também, devido à diversidade de desenho, apresenta proteção contra falhas de modo comum. Resumidamente, o sistema Lock-V tem grande cobertura de faltas tendo em conta as soluções existentes.

Palavras-chave: Diversidade de Desenho, DCLS, Redundância, Tolerância a Falhas.

Abstract

A Loosely-Coupled Arm and RISC-V Locksteping Technology

Due to the technological growth during the last few years, a new market is rising, bringing a huge number of devices that interact with the human being and the environment. However, the dependability of those devices becomes more and more a concern. Furthermore, from what has been seen, in terms of performance and power consumption, these computational systems are constantly being improved due to reduced transistor's size, higher clock frequencies, and lower operating core voltages. However, this leads to a lack in the systems reliability, which turns them more susceptive to faults. For example, systems are becoming more sensitives to radiations that can trigger Single Event Upsets (SEUs) in this new technological generation.

This dissertation aims to provide a new solution for fault tolerance systems, named Lock-V, that combines two fault tolerance techniques, in order to answer the current gap. The solution is deployed under the Microsemi SmartFusion2 that includes a Microcontroller Unit (MCU) and an Field-Programmable Gate Array (FPGA) in the same platform, and the solution consists in a Dual-Core Lockstep (DCLS) combined with design diversity at Instruction Set Architecture (ISA) level. The design diversity is achieved by using two different cores, a hard-core Arm Cortex-M3 and a soft-core RISC-V-based processors. The DCLS is supported by an FPGA-based accelerator and it provides error detection capabilities to the system by comparing, in a loosely-coupled fashion, the outputs from the two cores. Moreover, this dissertation provides a friendly framework, that adds to the system recovery capabilities.

In order to validate the system, a fault injection mechanism was developed, to test the Lock-V architecture. Since protecting the memory is out of the scope of this dissertation, the fault injections are over the register files, which are usually more vulnerable to faults, excluding the memory. These tests, prove the effectiveness of the Lock-V system as a fault tolerance system. Moreover, Lock-V architecture offers fault tolerance against SEU and protection against Common-Mode Failure (CMF) by applying lockstep technique and design diversity, respectively. Summing up, the Lock-V achieved a high fault coverage taking into account the existing solutions.

Keywords: Design Diversity, DCLS, Redundancy, Fault Tolerance.

Contents

List of Figures

List of Tables

Glossary

4-LUT 4-Inputs Look-Up Table

ABI Application Binary Interface

AHB Advanced High-performance Bus

AHB-Lite Advanced High-performance Bus Lite

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

API Application Programming Interface

ASIC Application-Specific Integrated Circuits

AXI Advanced Extensible Interface

BFM Bus Functional Model

CMF Common-Mode Failure

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

DCLS Dual-Core Lockstep

DDR Double Data Rate

DED Double Error Detecting

DFF D-Type Flip-Flop

DMR Dual-Modular Redundancy

DMT Duplex Multiplexed in Time

DWC Duplication With Comparison

ECC Error Correcting Code

EDC Error Detecting Code

FF Flip-Flop

FIC Fabric Interface Controller

FP Frame Pointer

FPGA Field-Programmable Gate Array

FPSoC Field-Programmable System-On-Chip

FSM Finite State Machine

GPIO General-Purpose Input/Output

GTMR Global TMR

HDL Hardware Description Language

I/O Input/Output

IDE Integrated Development Environment

IoT Internet of Things

IP Intellectual Property

ISA Instruction Set Architecture

JTAG Joint Test Action Group

LE Logic Elements

LR Link Register

LSRAM Large Static Random Access Memory

LTMR Local TMR

MBU Multiple Bit Upset

MCU Microcontroller Unit

MMIO Memory-Mapped Input/Output

MMR Multiple-Modular Redundancy

MSS Microcontroller Subsystem

NVM Non-Volatile Memory

PC Program Counter

PCI Peripheral Component Interconnect

PSR Program Status Register

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RTL Register-Transfer Level

SBU Single Bit Upset

SEB Single Event Burnout

SEC Single Error Correction

SEDR Single Event Dieletric Rupture

SEE Single Event Effect

SEFI Single Event Functional Interrupt

SEGR Single Event Gate Rupture

SEL Single Event Latchup

SET Single Event Transient

SEU Single Event Upset

SoC System-on-Chip

SP Stack Pointer

SPI Serial Peripheral Interface

TCLS Triple-Core Lockstep

TMR Triple Modular Redundancy

UART Universal Asynchronous Receiver Transmitter

1. Introduction

Nowadays, we are more and more surrounded by a vast number of technological devices, such as personal computers, smartphones, gadgets, smart devices, and much more. Despite this kind of technology not being developed for critical or safety proposes, it is required the support for some reliability features.

Yet for critical and safety applications, such as a car brake system, as well a medical devices, or even a nuclear reactor system control, the use of reliable technology is a must. It is not desired such critical and safety systems to fail at any time.

Due to human and environment interactions with the technology mentioned above, the research and development of reliable systems with fault tolerance capabilities is a concern, that has been getting particular attention, both from academia and industry. This dissertation explores the integration of a new Instruction Set Architecture (ISA), RISC-V with a fault tolerance technique, the Dual-Core Lockstep (DCLS) over a heterogeneous architecture, adding to the state of the art a new fault tolerance solution in order to achieve reliable technology.

1.1 Motivation

Since the beginning of computing science, the electronic systems face reliability problems. First, because the use of components that presents unreliable characteristics, such as vacuum tubes and relays [1, 2], but later, due to the increased systems' complexity, new types of unreliable systems appear even with the introduction of reliable semiconductors.

Recently, the new generation of components use reduced transistor's size, higher clock frequencies, and lower operating core voltages. If, in one hand, the systems are more efficient in power consumption and in performance, on the other hand, they present new dependability problems. For example, at ground level, these systems are more susceptive to Single Event Upset (SEU) induced by radiations that can cause bit-flips [3]. These issues are widely present in the aerospace environment, but now it is also a concern in daily basis systems, such as [4, 5, 6, 7, 8, 9].

Normally, reliability capability applied to daily basis systems do not include fault tolerance techniques, mainly, due to the rising of complexity and development cost. Moreover, with the complexity of the systems in the current days its harder to achieve dependability with simple methods, demanding for new methodologies that can help to meet the mentioned requirements.

In particular, the combination of design diversity at hardware level with lockstep techniques is a open question in the current state of the art. This alliance between the two concepts can provide to systems a fault tolerance against both SEU and Common-Mode Failure (CMF).

1.2 Main Goal

In order to follow the technologic evolution, and answer the issue that the new systems generation has a lack in dependability, the main goal of this dissertation goes towards developing a fault tolerance system that uses DCLS in a heterogeneous architecture at core level combining different techniques to provide tolerance to a new range of fault, SEU and CMF.

At the same time, it is important to develop the system with focus on low-end devices and to use the new and emerging RISC-V ISA in one of the cores during the implementation.

1.2.1 Objectives

These objectives are divided in three parts and they derived predominantly from the main goal. They are the key to realize the main goal with success.

The first objective is the development of the Lock-V architecture which includes the two cores, a hard-core Arm Cortex-M and a soft-core RISC-V-based, and also an accelerator that supports the heterogeneous architecture. The objective is to provide system synchronization and error detection between cores.

This objective is carried out through several tasks:

- Deploying a soft-core RISC-V-based in a platform that combines a hard-core Microcontroller Unit (MCU) and an Field-Programmable Gate Array (FPGA), in order to provide a dual-core system with heterogeneous architecture at core level;
- Developing *xLockstep* accelerator in the FPGA to provide the system with DCLS capabilities on different core architectures;
- Connecting both cores with the *xLockstep* accelerator in the platform, in order to deploy the Lock-V hardware architecture;
- Developing an Application Programming Interface (API) that explains and details the use of the accelerator while developing a library based on the API, to allow the use of the accelerator in an agnostic way;
- Testing and verifying the synchronization and error capabilities of the Lock-V architecture.

This second objective consists in the development of the Lock-V framework that provides a simple interface to promote in a friendly way the addition of fault tolerance capabilities in a system.

Due to the lack of recovery capabilities on the Lock-V architecture, the framework has the task to implement the system recovery in software.

This objective is performed through the next tasks:

- Writing functions into the framework to apply cores initialization and the synchronization capabilities between them;
- Writing functions to offer the system recovery capabilities, including saving the processor context and restoring the system to the saved processor context, also known as rollback.

The third objective consists in developing a fault injection system to test the system against faults, while verifying and validating the Lock-V as a fault tolerance system.

This is carried out through the following tasks:

- Adapting the main system with a debugging interface, in order to monitoring the system during the verification phase;
- Developing a injection fault mechanism to emulate SEU on the register files in random register through a random time interrupt.

Despite being important, this dissertation does not focus on memory protection mechanisms, thus, the fault injection test system only concerns the register file.

1.3 Dissertation Structure

After this chapter, that includes a brief context, the motivation, and the goals of this dissertation, the chapter 2 overviews the concepts of dependability in section 2.1. Also, it includes different techniques and methodologies to achieve fault tolerance systems, in section 2.2. The chapter 2 ends with the current state of the art for fault tolerance techniques based in lockstep technique in section 2.3. The chapter 3 explains the platform and the soft-core RISC-V-based choice for this dissertation.

Chapter 4 explains the development of the Lock-V architecture by detailing the *xLockstep* accelerator in section 4.1, and how the Lock-V was integrated with the hardware platform in section 4.2. Moreover, in section 4.3 includes the API definition and the library developed to support the *xLockstep* accelerator. The chapter 5, in order to accomplish the second set of tasks mentioned in the objectives, includes in detail the functions provided by the framework for initialization and synchronization purposes in sections 5.2 and 5.3, respectively. It also includes how the system achieves the system recovery using rollback in sections 5.4 and 5.5. The last set of tasks is described in the chapter 6, that also includes resources utilization as well the cost (memory and execution footprint) of the system Lock-V.

Last but not least, the chapter 7 provides a summary and highlights important consideration about this dissertation while mentioning future improvements.

2. Background and State of the Art

This chapter addresses the main concepts of dependability in the section 2.1, which covers attributes, threats, and the means to achieve it. Then, in the section 2.2, it presents fault tolerance techniques as one of the primary ways to achieve dependability. Finally, section 2.3, shows a literature review about fault tolerance techniques.

2.1 Dependability

Nowadays the search for reliance systems is in expansion due to the emergence and increase of safety-critical applications, ranging from daily basis systems such as a braking system in a car, or a financial transaction through an automated teller machine, to medical devices, airplanes, nuclear power plants or even aerospace applications [8, 10, 11, 12].

One of the main properties of the computing systems is dependability [10], and Laprie adapted the dependability definition from the primary definition to computer systems as *"the quality of the delivered service such that reliance can justifiably be placed on this service"* [13]. To understand the previous definition, it is necessary to look for the delivered service meaning, which is the system behavior that the user can observe. A delivered service can be one of the two options: (1) a correct service if the system behavior satisfies the functional specifications previously defined for a concrete system to provide the specified service; or (2) an incorrect service when the system behavior does not match with the expected, i.e., when the delivered service is different from the specified service. The transition from a correct service to an incorrect service is due to failure occurrence, and the reverse transition is named by service restoration [10, 13, 14]. After the dependability definition from Laprie, Avizienis formulated an alternative concept for dependability, taking into account the definition of system failure in order to complement the first definition [10]: *"the ability of a system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s)."*. These definitions lead to the emergence of topics to support the main one since Avizienis et al. wrote the "Fundamental Concepts of Dependability" [10] where, in addition to the dependability definition, they structured the concepts around the main term dependability, and divided into three classes, as shown in Figure 2.1 (adapted from [10, 15]): (1) the threats to dependability, (2) the attributes of dependability, and (3) the means to achieve dependability.

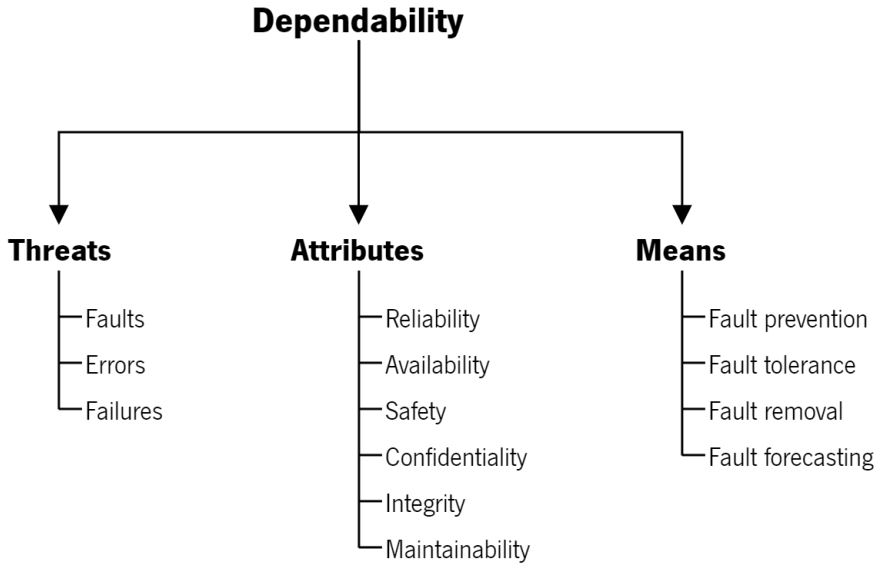


Figure 2.1: Dependability tree.

Faults, errors, and failures are threats to dependability that share a relationship between them. There are six dependability attributes: (1) availability, (2) reliability, (3) safety, (4) confidentiality, (5) integrity, and (6) maintainability. The last class is the means to achieve dependability and they are divided into four main topics: (1) fault prevention, (2) fault tolerance, (3) fault removal, and (4) fault forecasting.

2.1.1 Dependability threats hierarchy

The main terms that characterize the threats to dependability are fault, error, and failure. In this subsection, it is described the terms as well as the connection between them and some important definitions derived from the main terms. In fault tolerance systems, it is necessary to recognize the differences between fault, error, and failure. Moreover, to know the cause and effect of faults, errors, and failures are essential to understand this work.

2.1.1.1 Chain of dependability threats

The dependability threats have a hierarchy that starts in a fault, the fault can be located in a component or a system, for example, due to a defect in a component. The system can wrongly enter into an undesired state when the fault origins an error. Furthermore, this error may lead to a component or system failure. If the failure occurs in a component inside a system, the service failure of the component may be responsible for creating an error in the system, and the error may trigger a failure in the system. Figure 2.2 shows the fault, error, and failure hierarchy. The fault in a component may origin an error, which can lead to failure. Moreover, the failed component may trigger the same cycle in the system [13].

Occasionally these terms are misused, and Parhami bridges this gap. The author presents three different examples to understand the differences. As a reader, the first example transmits the information

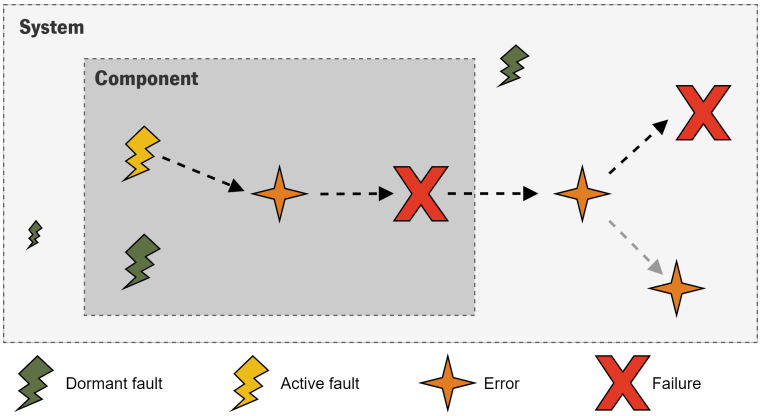


Figure 2.2: Fault, error, and failure hierarchy.

necessary to understand the difference between the terms [16]. In the example, the author explores a car braking system with a defective brake pipe. He reports that a fault occurs when the pipe breaks due to the defective pipe and if this fault results in fluid loss to a dangerously low fluid level in the brake system, the fault leads to an error. However, the error only turns to a failure when the brake system is required, and it does not work correctly.

2.1.1.2 Fault

A **Fault** is a defect in a hardware or software component [17]. A fault may be caused by different problems during the project phases (analysis, design, implementation, or even at the production stage). Beyond these project phases, a fault may emerge due to external factors (human or environment agents). In computation systems, e.g., software processors, Application-Specific Integrated Circuits (ASIC), among others, the transistor size reduction in the last decades leads to smaller working voltage and smaller noise margins, leading to a more significant fault susceptibility in hardware components [4, 18].

A fault can be divided into two different types, taking into consideration its propagation: an active or dormant fault. The active fault generates an error, the dormant fault may generate an error when some interface changes its state.

Faults can be classified, taking into account eight different perspectives, leading to the elementary fault classes [19, 20]: (1) the phase of creation or occurrence of the fault; (2) the system boundaries; (3) The phenomenological cause of the fault; (4) the dimension where the fault occurs; (5) the objective of introducing the fault; (6) the intent of the fault; (7) the capacity of the human that originates the fault; and (8) the persistence of the fault. Figure 2.3 (from [19, 20]) shows the elementary faults based on the eight different classifiers mentioned above.

The phase of creation or occurrence has two different types of faults. The development originated during the system development, or during the system maintenance, or even when procedures are created and the operational fault that occurs during the runtime of the system.

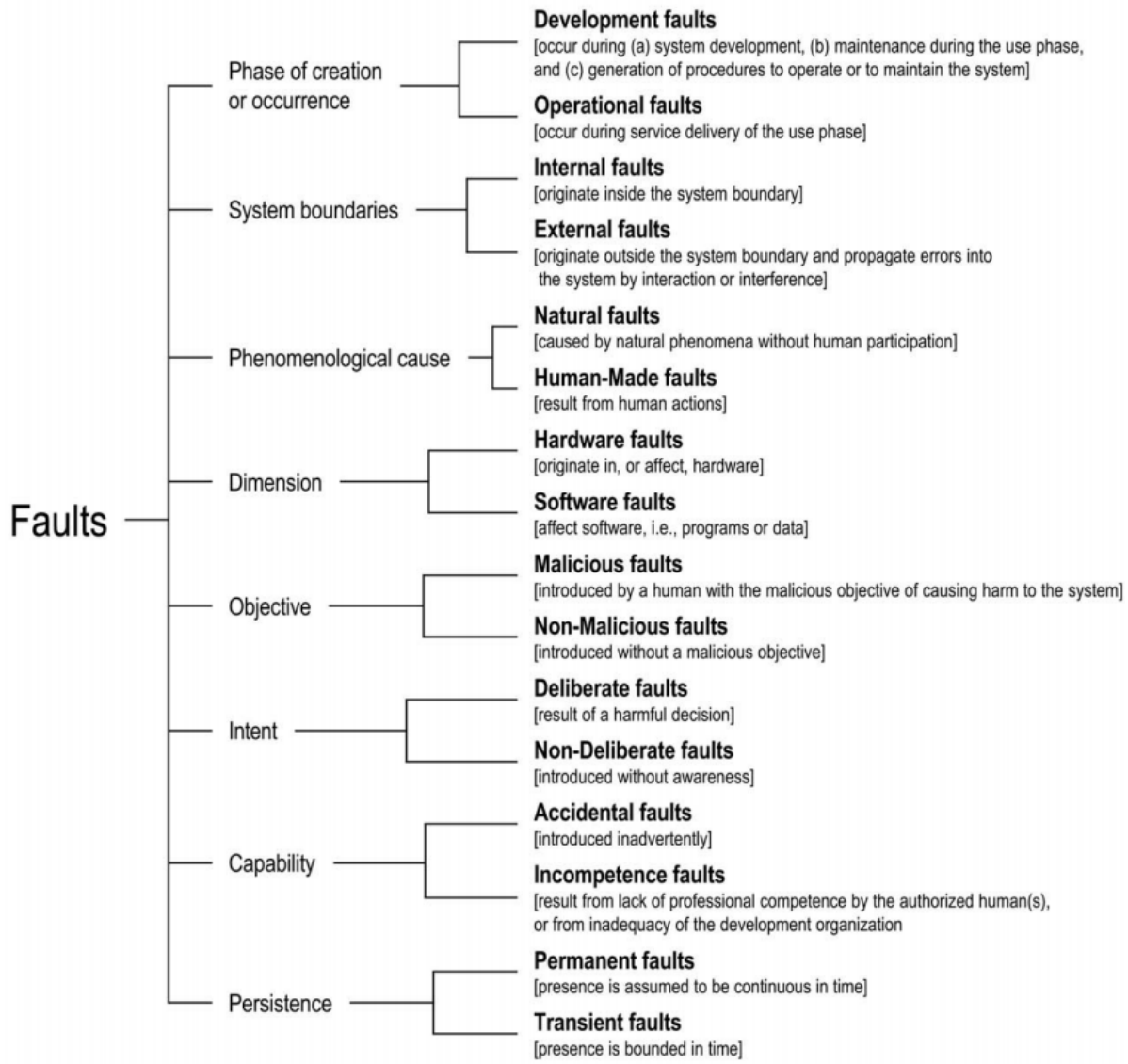


Figure 2.3: Elementary fault classes and its eight different classifiers.

The system boundaries classify the fault taking into account the fault location. A fault can be an internal fault to the system, and this means that the fault occurs within the system. Otherwise, if the fault occurs outside, it is an external fault.

The phenomenological cause classifier defines the fault in natural fault or human-made fault by focusing on the fault's origin with or without human participation. If the fault occurs due to human participation is a human-made, otherwise is a natural fault.

Furthermore, the fault classification depends on its dimension which can be a hardware fault disturbing the hardware, or software fault when the fault disturbs the software (data, programs, etc.). A hardware fault is also known as a physical faults, and a software fault is known as an information fault.

A fault can be a malicious or non-malicious when it considers the objective of causing a fault. Malicious faults occur due to human interaction in order to damage the component or the system. Otherwise, the

non-malicious fault has no intention to damage the component or the system.

The intent of the human is another classifier to faults. If the human deliberately decides to introduce the fault is a deliberate fault, otherwise, when the fault is introduced without perception is a non-deliberate fault. This classifier differs from the mentioned above because the intent classifier defines if the human intended to create a fault or not. A malicious fault is every time a deliberated fault, but a non-malicious one may be deliberated or non-deliberated fault.

The capacity of the human defines two elementary fault classes, the accidental fault, which as its name indicates it is introduced accidentally, and the incompetence fault which occurs due to a lack of professional capacity.

The persistence of the fault classifies the fault in permanent or transient. The permanent fault is expected to remain active during the time unless the fault has been resolved. The transient fault only remains active during a short period of time.

All eight classifiers combined generate all types of faults, but some combinations are not possible. In the Figure 2.4 from [20], Avizienis shows all the possible combinations while classifying them in three main groups: (1) the development faults group, that occurs due to mistakes or other actions during the development phase; (2) the physical faults group, where all the faults interfere in the hardware; and (3) the interaction faults, which contains all the external faults.

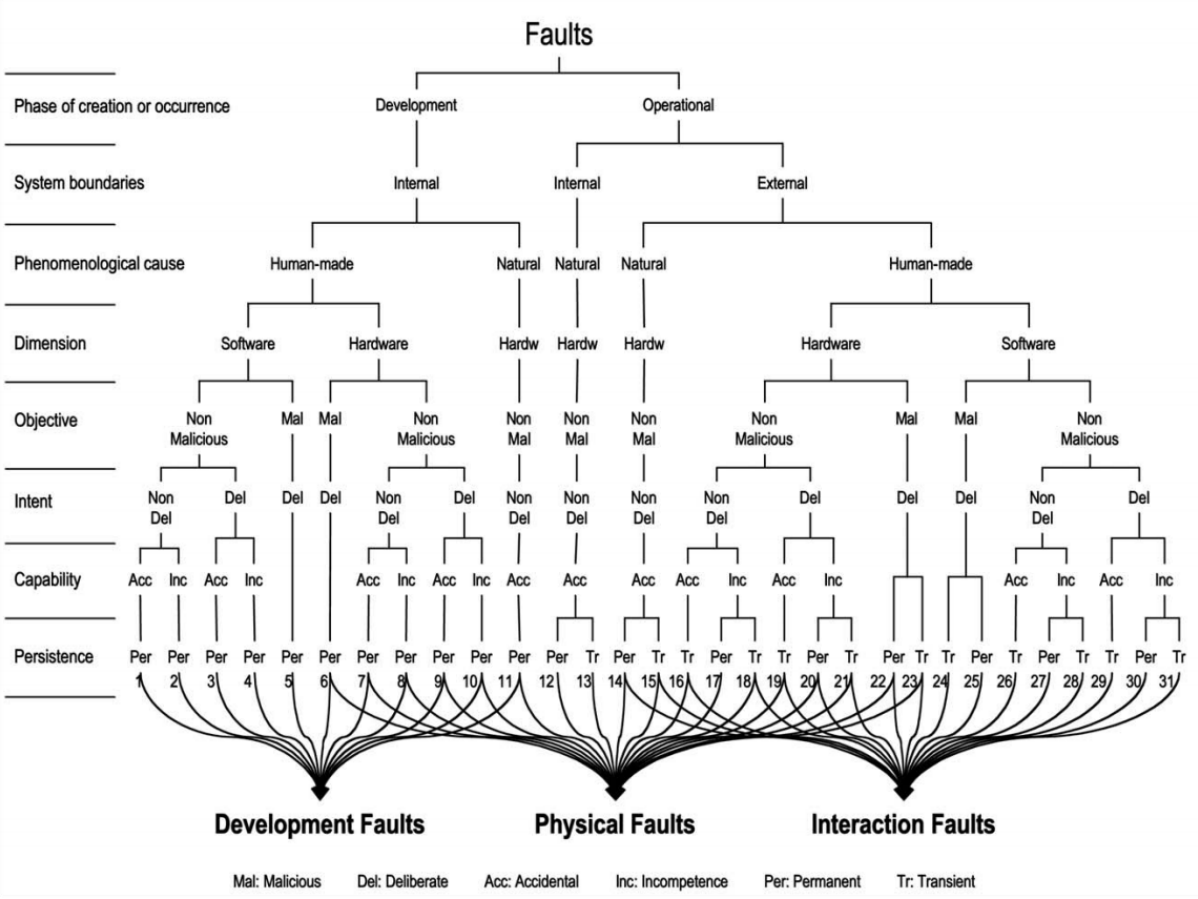


Figure 2.4: The classes of combined faults.

Another important issue, mainly for this dissertation, is the Common-Mode-Faults. It defines faults that occur in more than one redundant module at the same time. This type of fault occurs due to a dependency between the redundant modules, and if the faults origin failures in the redundant modules, then a CMF will be generated.

2.1.1.3 Cause of the faults

Faults can be caused by different situations, and Dubrova divides the origin of faults in four groups: (1) incorrect specification, (2) incorrect implementation, (3) component defect, and (4) external factors [17]. Incorrect specification causes faults due to the use of a wrong algorithm, flowchart, architecture, improper requirement, or specification, between other causes during the specification phase. Incorrect implementation brings to the system faults caused by design and implementation faults, for example, in hardware can be caused by mistakes in component choice, or higher clock frequencies, while in software it can be caused by mistakes that the compiler did not detect or incorrect variable type choice. Imperfections during manufacture and wear in hardware components are the component defects. Although there has been a significant improvement over time in hardware components, the search for lower power systems, smaller devices, and higher performance leads to the use of low voltage levels, thinner encapsulation, and high processing frequencies. These characteristics make systems more sensitive to external factors. These factors are the last cause of faults mentioned by Dubrova and these type of faults, as the name suggests, are caused from external agents outside of the system, such as environmental phenomena, or user perturbance.

In respect to external factors, aircraft systems are subjected to a large amount of radiation [21], and space stations and spaceships systems, even more, probably affecting the system drastically or even cause a catastrophic situation. Also, at ground level, the system is subject to radiation [3]. When single event radiation hits a system with high energy, it may result in failure, such as bit-flips in memory, register, combinational logic, or even damage the component or system [22]. When the single radiation event causes a system failure, it is defined as a Single Event Effect (SEE), and it is divided into various sub-events, as shown in Figure 2.5.

The first level of SEE sub-events is the “soft” SEE (or “soft” error) and the “hard” SEE (or “hard” error). If the SEE does not cause permanent damage in a component or system, then it is a “soft” SEE. Usually, these kinds of events happen at ground level, and the ground commercial applications are the typical systems affected by this type of SEE. In contrast, the “hard” SEE causes permanent damage in a component or system. Usually, these SEE happens at space level and in a critical environment. Space, military, and critical applications must consider protection to “hard” SEE during development [4].

“Soft” SEE includes in its branch the SEU, Single Event Functional Interrupt (SEFI), Single Event Transient (SET):

- A **SEU** is characterized by changing a component state due to ionization, such as a change in a memory cell or a register data. If the SEU affects only one bit, then it is a Single Bit Upset (SBU).

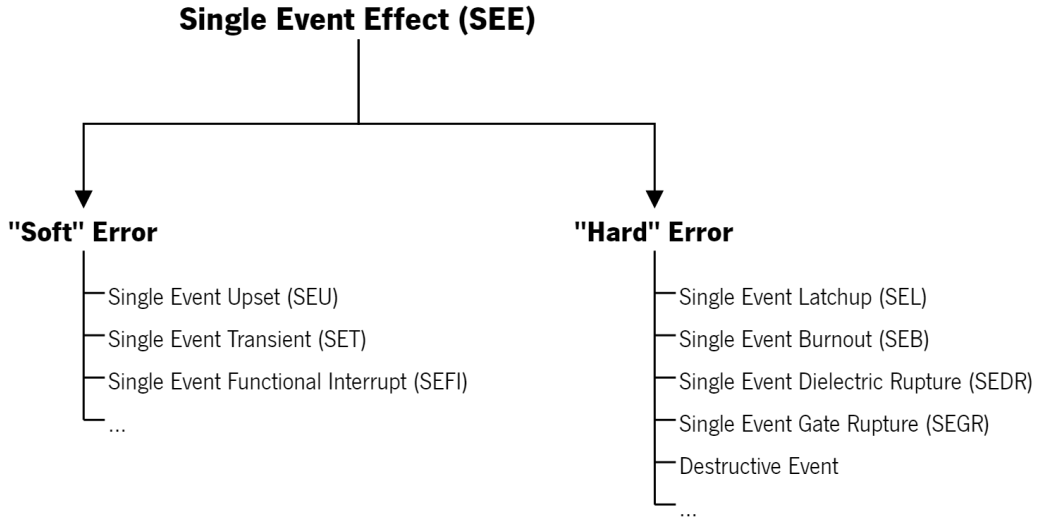


Figure 2.5: Single event effect tree.

However, if the SEU has very high energy to affects more than one bit, it is a Multiple Bit Upset (MBU). Typically, the SBU is more likely to happen, and normally, at ground level, the fault tolerance systems are developed to be protected against SBU.

- A **SET** is characterized by its location, and this type of SEU occurs in combinational logic and generate a transient pulse interference. This event effect in combinational logic is temporary, but if the combinational logic can transfer the sequential logic, it is possible an occurrence of a SEU.

- A **SEFI** is characterized by affecting systems areas that have a global or critical control and leads the system to a functional interrupt. Internal power-on-reset system or a critical system control register are examples of systems that may cause a SEFI, if the radiation event hits them.

There are more “soft” SEE, but the most important for this dissertation context are mentioned above. Also, the “hard” SEE is out of the scope in this dissertation context, however, they are detailed in Yangs’ book in chapter 6.2 [22].

2.1.1.4 Error

An **Error** is originated by a fault, and an error may cause a failure when the error affects the service delivered, and the error changes the service delivered in a system to an incorrect service. Errors can be detected or latent. The detected error is the one when its presence is alerted, for example, with error messages or signals. The latent error is an error that its presence is not detected [17].

An error may cause a failure, but this propagation depends in two factors: (1) The system structure and if redundancy is applied in the system (intentional redundancy for example as fault tolerance techniques or unintentional redundancy); and (2) the system behavior, because an error may never influence the system, for example, if an error is overwriting before its use or never use the faulty component or sub-system [20].

A latent error may never affect the system, and its lifetime in the system is different from system to system due to the system utilization or the origin of the fault. Also, an error may be caused by different types of faults [13].

It is possible to classify an error based on four different classifiers: its domain (content vs. timing errors), its detectability (detected vs. latent), its consistency, and its consequences (minor to catastrophic errors) [20]. This kind of classification is also used for failures in a more frequent way, and the meaning is similar to the classifiers explained above, in subsection 2.1.1.5. When the same fault causes more than an error, for example, electromagnetic radiation that affects different components, they are denominated multiple related errors. If the fault only causes one error, it is denominated as a single error [20].

2.1.1.5 Failure

A **Failure** or service failure occurs when the service delivered by a system does not perform the required services described in its system specification [23]. The cause of failure is always an error that leads the system to deliver an incorrect service. A failure has different modes when it occurs. They are classified by four different classifiers as the errors in subsection 2.1.1.4. Figure 2.6 from [19] shows all the service failure modes and all the four classifiers: (1) the failure domain, (2) the detectability of failures, (3) the consistency of failures, and (4) the consequence of failures on the environment.

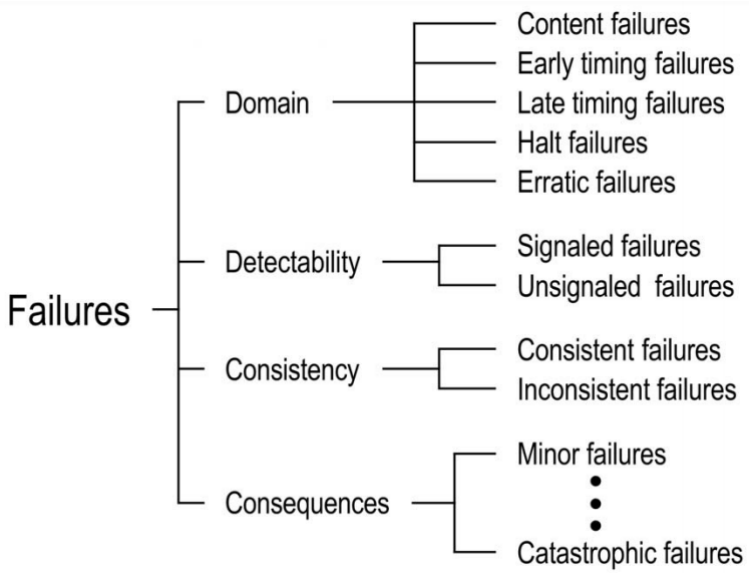


Figure 2.6: Service failure modes.

The failure domain divides into two different ones, the content failures when the system interface delivers incorrect information, and the timing failures when the delivered information from the system interface deviates (early or late timing failures) from the expected. Also, a failure can occur with the two problems in the same failure, information and timing incorrectness. This one can be a halt failure when the service is halted, or an erratic failure when the service is delivered but in an erratic way [19].

The failure detectability considers the alert of a failure to the user to characterize the failure. This classifier divides the failure detectability in signaled or unsignaled failures. The signaled failure uses detected mechanisms to verify if the system delivering the correct service, and in case of incorrect service, the system signals the user with a warning signal. Otherwise, if, in case of failure, the system does not signal the user, it occurs an unsignaled failure [20].

When the system has more than one user, it is possible to consider the consistency of failure to classify the failure. When the failures occur, and it is perceived in the same way by all the users, it is a consistent failure. If the failure reaches the users differently or does not even reach, it is an inconsistent failure [20].

The last classifier is the consequence of the failures in the environment, and this viewpoint to classify failures is defined by levels of failure severities. These levels vary from application to application, and to define the levels of failure severities. It considers the dependability attributes such as availability, safety, confidentiality, among others. Nevertheless, it is possible to define the classifier limits, the minor failures and catastrophic failures [20].

2.1.2 Attributes of dependability

In the first contributions to dependability definitions, Laprie enumerated availability, reliability, safety, and security as the attributes of dependability. However, the author said that the attributes might have different levels of priority according to the application. Laprie also mentioned that availability is always required, but its level of dependency may vary depending on the system and its requirements, while the other attributes may not be necessary at all [15].

After almost a decade, Avizienis et al. updated the attributes and defined them as the six basic attributes of dependability, adding confidentiality, integrity, and maintainability, and removing security from the main attributes [20]. For dependability, security is important but can be mostly fused between the basic attributes: available only for authorized users, confidentiality, and integrity with the assurance that unauthorized information is not changed [19].

Then, the six basic attributes that a dependable system must have are:

- **Reliability** is the ability of a system to continue providing the correct service. Furthermore, reliability can be expressed in function of time, $R(t)$, as the probability that the system will continue providing the correct service in accordance with the specifications for a period of time, t, [17, 24]. When a system needs to run without interruptions or when it is not possible to apply maintenance, high-reliability is necessary [17, 25]. For the fault tolerance system, Lee suggests that reliability is an important attribute [24], and the search for high-reliability in fault tolerance systems is increasing [25, 26]. The main concern during a fault tolerance system design is how to achieve reliability in the presence of faults. To achieve this, the system needs to know the system state. In other words, the system needs to distinguish between expected and unexpected behavior.

- **Availability** is the readiness of the system to provide the correct service. Similarly to reliability, availability also can be expressed in function of time, *A(t)*, as the probability that a system at the instant of time *t* provides the correct service in accordance with the specifications [17]. In order to make the comprehension of availability easier, it can be expressed in *Downtime per year* [27] i.e., the amount of time that a system stays interrupt per year. Table 2.1 (adapted from [17, 27]) shows the availability of system classes and the corresponding downtime per year. For fault-tolerant system is expected an availability of 99.99 percent that corresponds to 52 minutes of service interruption during a year. A system with high-availability can change to a failure state [25], but the time to recovery and the number of failures need to be small enough to meet the conditions presented in Table 2.1.

Table 2.1: Availability of system classes and the corresponding downtime per year.

System Type	Availability Class	Availability (%)	Downtime
Unmanaged	1	90	36.5 days/year
Managed	2	99	3.65 days/year
Well-managed	3	99.9	8.76 h/year
Fault-tolerant	4	99.99	52 min/year
High-availability	5	99.999	5 min/year
Very-high-availability	6	99.9999	31 s/year
Ultra-availability	7	99.99999	3 s/year

- **Safety** relates to the absence of critical situations that can create hazardous effects on the user or/and the environment. In contrast to the two attributes defined before, reliability and availability, that can be quantitative, safety is a qualitative attribute and the amount of safety in a system depends on its purpose. This attribute is required mostly in safety-critical systems where the human life or the environment in case of failure can be affected as a human injury, loss of life, or environment disaster [17]. To develop a safety system it is necessary taking in consideration the safety feature from the beginning. Also software and hardware solutions need to be developed with inextricably intertwined to achieve the attribute safety. Furthermore, it requires special attention in interfaces because they are the first contact to the environment and the users [28].

- **Confidentiality** limits the distribution of unauthorized data or information. To achieve confidentiality, the system applies restrictions in information access and in its disclosures to protect personal privacy and proprietary information [29]. Confidentiality is required in case of systems that allocate sensitive information.

- **Integrity** is the ability to limit access and changes to specific data or information. When integrity is required, the system needs to be protected against unauthorized modification or destruction of information. It also includes the trustiness of information authenticity [29].

- **Maintainability** refers to the possibility of repairs or modifications to the system. The IEEE standard computer glossaries define maintainability as the easiness that a system or a single component can be modified to correct faults, performance improvements, or other attributes. It also can be adapted to a different environment. The purpose of these modifications is to make the system capable of performing its expected functions [23].

The search for a reliable system that includes dependable attributes has been increasing. Reliability and availability are the most required to achieve fault tolerance in industry systems [25]. And in cases where security is a requirement, dependability and security share attributes such as availability, confidentiality, and integrity [19, 20]. In the Internet of Things (IoT) environment, the need for dependable and secure devices is a current demand, mostly due to the number of devices connected to the Internet. For example, IIoTEED is an architecture to achieve attributes like availability, confidentiality, integrity, safety between other industrial IoT edge devices [30]. The authors propose a Trust-Zone-based (by Arm) architecture with two different sides, the security world side and the non-secure side. With this system, the critical processes can be isolated from the non-critical processes, and enhance the industrial IoT devices with a trusted execution environment.

2.1.3 Means to achieve dependability

There are different methods, techniques, and tools to develop dependability systems. The main dependability means are divided into two classes: (1) the dependability procurement, that includes fault prevention, fault tolerance, and (2) the dependability validation, that includes the last two dependability means, fault removal and fault forecasting [13, 14]. The dependability procurement is the means to endow the systems with capabilities to deliver the correct service, whereas the dependability validation aims to reach confidence in the capabilities mention before.

- **Fault-prevention** aims at preventing the injection or occurrence of faults in a system. To achieve that, there is a set of methodologies that will mitigate the fault occurrence, if they are applied during the development phase, more precisely in the specification, design, implementation, production, and test stages. These methodologies are mostly quality control techniques for software and hardware development. In hardware development, it is important to follow design rules, to review the design, to screen component, and to test the components and the system in order to remove specification faults and defective parts from the system, or shielding the weakness parts in the system. In software, the use of methodologies or techniques such as structural programming, modularization, the use of strongly-typed programming languages, and formal verification. Also, the development of procedures to apply maintenance and to use the system are a good way to reduce the fault occurrence [17, 19].

- **Fault tolerance** aims the system to continue delivering the correct service in case of an active fault [31], to achieve this, a fault tolerance needs to include error detection and system recovery

mechanisms. There are different mechanisms for both, detection and recovery, as shown in Figure 2.7. This means that achieving dependability is one of the main characteristics of the project addressed in this dissertation, and it will be more detailed and well-defined in section 2.2.

- **Fault-removal** can be applied during the system development, as well as during system lifetime. As its name suggests, fault removal is the means to remove or to reduce faults in a system. Also, it includes the severities reduction from faults [19]. During systems development, fault removing is achieved by verifications, diagnosis, and correction. And in the system lifetime is achieved with corrective or preventive maintenance.

- **Fault-forecasting** aims to predict the system behavior by estimating faults that can be present in the system, the possibility of fault occurrence in the future, and its consequences [15, 17, 19]. There are two ways to evaluate the system in order to predict the system behavior and the fault occurrence or activation: (1) the qualitative evaluation that try to recognize and classify all the failures modes or events that might happen and leads to a failure situation, and (2) the quantitative evaluation which, in probability way, estimates if the dependability attributes are satisfied according to with the expected.

The system development and its lifetime have an important role in answer the question: *What kind of means to achieve dependability is necessary?* Fault prevention is only applied during the development phase. In contrast, fault tolerance only works during the system lifetime, although it is introduced in the system in the development phase. Fault removal can be applied during the development phase with tests and system debug, but also during the system lifetime with maintenance. Finally, fault forecasting can be used in the two phases: the development phase and the system lifetime phase.

To achieve a dependable system, sometimes the use of only one dependability means, such as fault tolerance is not enough, and depending on the environment and its expected lifetime, it is necessary to adapt and to use more than one dependability mean. However, during this project, only fault tolerance was considered to achieve a dependable system, which will be described below, in section 2.2.

2.2 Fault tolerance

As explained above, in subsection 2.1.3, the purpose of a fault tolerance system is to continue delivering the correct service in case of an active fault, and so preventing that an error caused by an active fault changes the system behavior to an incorrect service. Figure 2.7 (from [19]) depicts the methodologies used in fault tolerance, and normally, the two key bases functionalities that support fault tolerance techniques are **error detection** and **system recovery**[19, 29].

Error detection aims to detect the error and to identify it before the system delivers an incorrect service. There are two main strategies of error detection, the concurrent detection, and preemptive detection:

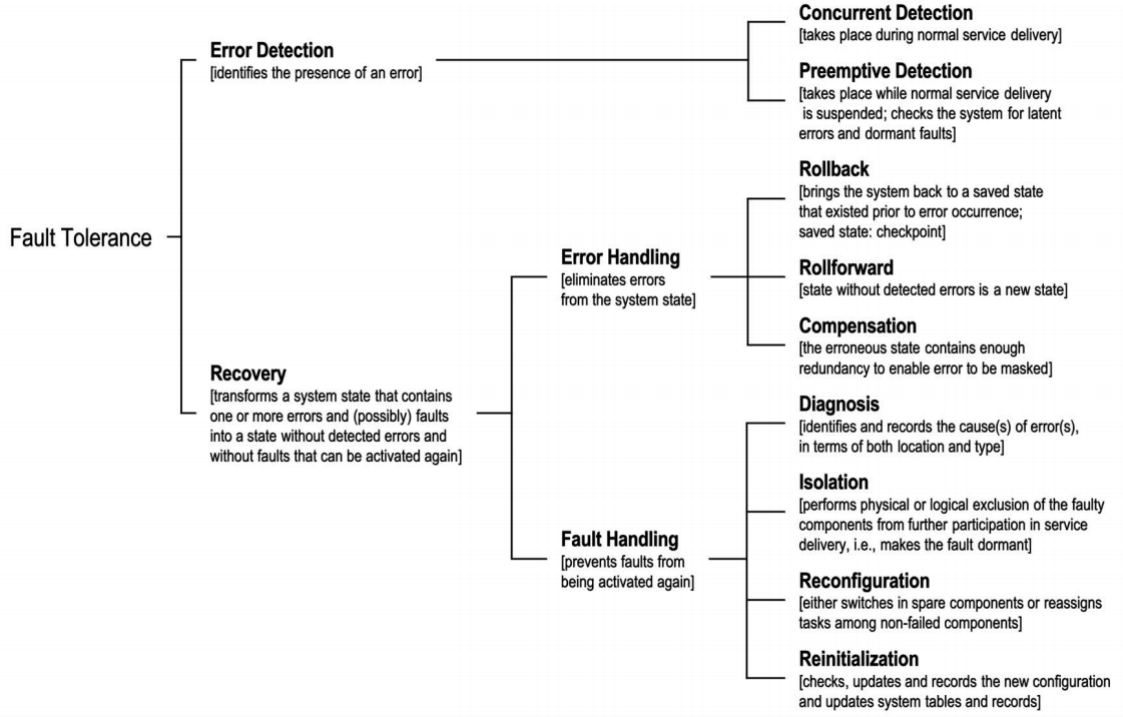


Figure 2.7: Fault tolerance mechanisms.

- **Concurrent detection** - the error detection mechanisms execute in parallel with the normal system delivering;
- **Preemptive detection** - in this case, to executes the error detection mechanisms, the system suspends its system delivering and verifies if there are latent errors and dormant faults.

The second functionality, system recovery (or simply recovery), uses different methodologies to remove detected errors, to remove activated faults that may be activated again, and to restore the system to a state without detected errors. To achieve these results, recovery mechanisms has error handling and fault handling:

- **Error handling** - the objective of this technique is to remove detected errors and to restore the system integrity with one of the three techniques:
    - **Rollback** - this technique restores the system to a previous state without detected errors which is saved at the beginning of the execution or during the execution every time that the system reaches a checkpoint;
    - **Rollforward** - in contrast to rollback, the rollforward restores the system to a new and known state without detected errors;
    - **Compensation** - this technique uses enough redundancy, in order to make the system capable of transmitting correct values to the output or interface, also known as error masking.

- **Fault handling** - it aims to remove faults that can be activated again and may propagate to failure. There are four basic techniques that can be mixed to achieve the fault handling goal:
  - **Diagnosis** - localization, and categorization of faults that can cause errors;
  - **Isolation** - turn activated faults in dormant faults with a physical or logical exclusion, avoiding the participation of component with faults in system behavior;
  - **Reconfiguration** - change the system in order to use spare components or allocate the tasks from the faulty component by others;
  - **Reinitialization** - this can be achieved by a system reset, leading the system to integrity and known state or by a system update in the configuration.

There are different ways to develop a fault tolerance system. During the development phase, it is necessary to specify the dependability attributes that are important to achieve. These choices lead to different error detections and recovery methodologies.

Typically, fault tolerance systems use redundancy, which allows applying different techniques to error detection, and in almost techniques, redundancy is necessary to recover the system as detailed in subsection 2.2.1.

There are different strategies to develop fault tolerance systems [17, 19, 29]. Essentially, fault tolerance is achieved by using error detection mechanisms with a system recovery technique. Some examples of strategies to develop fault tolerance are:

- **Checkpoint and restart** - this strategy uses checkpoint over the execution to call or to apply error detection mechanisms and in case of error, the system restarts.
- **Checkpoint and recovery** - the checkpoint follows the same methodology mentioned above, but instead of a system restart, it restores the state to integrity and knowing state. The recovery can be a rollback or rollforward. Figure 2.8 shows this type of strategy with rollback to develop a fault tolerance system.
- **Masking and recovery** - in contrast to the other mentioned strategies, this one uses compensation to mask errors instead of checkpoints to detect errors. Error masking always has the same time overhead, and it is shorter than the error detection. But to apply compensation, it is necessary triple or multiple redundancies and a voting system.

In general, all the strategies for fault tolerance systems are described above except for adaptations from them. The strategy to develop the fault tolerance system depends on the system requirements. The use of checkpoints creates a time window between them, and in case of error, the duration from the error until system recovery varies with the distance between checkpoints. Also, if the checkpoint is placed in a bad location, in case of error, it can propagate to a system failure. Besides that, the use of system restart in case of error may not be desired, depending on system requirements.

This dissertation work uses the fault tolerance strategy depicted in Figure 2.8. This strategy uses checkpoints to apply the error detection methodology and rollback to system recovery.

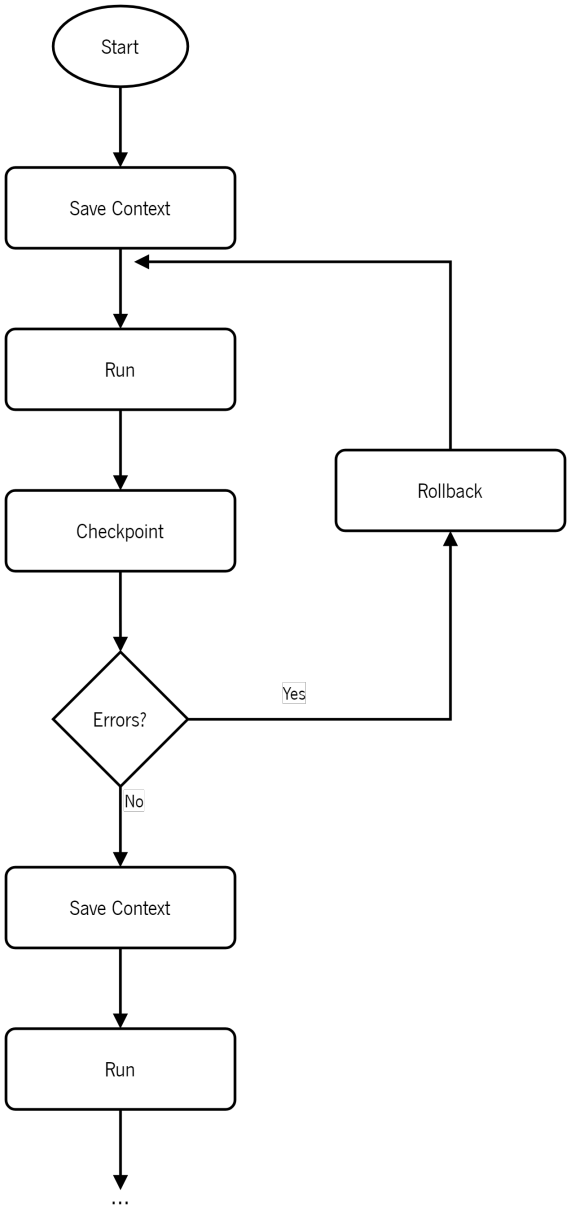


Figure 2.8: Fault tolerance strategy, error detection with system rollback recovery.

After the system start-up, saving of the system context is needed in order to have an integrity state in case of a rollback is executed. The system continues its normal execution until reaching a checkpoint. After the checkpoint, the system uses its error detection mechanisms to verify its own state. In case of error, a rollback is executed. Otherwise, the context of the system is saved and the system continues its normal execution.

2.2.1 Redundancy

The main way to achieve fault tolerance is the use of redundancy [25]. There are different ways of redundancy, and it is possible to use more than one type of redundancy to achieve better results in fault tolerance. Moreover, redundancy techniques are used since the very beginning of the computer science to leverage fault tolerance capabilities, based on the repetition of digital logic or component [2], to the repetition of program execution [1, 18, 32, 33], or even the repetition of an entire system [34, 35, 36]. One function of redundancy in fault tolerance systems is error detection. The redundancy allows the system to compare results from multiple copies of components, modules, or software [8]. An error leads to a failure state if the results from the copies do not match.

There are four different key types of redundancy to achieve fault tolerance, and they can be interconnected in the same system in order to get a better error detection ratios. The four types of redundancy are:

- **Hardware redundancy** - the main purpose of using hardware redundancy is to have two or more identical or similar copies of the same hardware component, where the copies can acquired the same values in the input interface in order to compare the output values between copies and enabling error detection in case of different values from the components [12].

  Also, according to Pierce's studies [2], the use of redundancy for fault tolerance systems is highly efficient in case of single fault presence.

  Unfortunately, the use of hardware redundancy may overload the system in different ways, such as weight, size, power consumption, cost, between others [17]. During the development phase, there are choices as the quantity of redundancy as well as its location in the system that can reduce or increase the weight, the cost, among others. Redundancy techniques may be easily adopted in several systems, however, they are not preferred in aero-space applications due to size and weight costs.

  Hardware redundancy splits into three types: passive, active, and hybrid hardware redundancy.

  - **Passive redundancy** - it uses redundancy in order to masking faults instead of detecting it without a system or a user interaction. Passive redundancy achieves fault tolerance without interrupting the system. It is a desired feature for the system where suspending of system delivering or reparation during runtime is undesirable, as a brake system in a car, or an aircraft system control, or even a heart pacemaker [17].

    Some examples of passive hardware redundancy are the modular redundancy with three or more copies of the hardware, knowing as TMR and n-modular redundancy, respectively. These ones includes a voter system to compare the output and to decide the correct value with majority consideration for the output values [25].

- **Active redundancy** - primarily, this fault tolerance technique uses hardware redundancy to detect the faults or errors in the system and to restore the system to a normal state without detected errors. To perform this kind of fault tolerance, the system stays out of the normal state for a period of time to verify the existence of errors, and, in case of an error, the system restores itself to an integrity state. It is necessary to take this into consideration during system development. The three main active redundancy techniques are Duplication With Comparison (DWC), standby, and pair-and-a-spare.

  DWC, as depicted by Figure 2.9 from [17], is the most basic form of an active redundancy that consists in comparing outputs from two identical modules using a comparator, for example in this dissertation the modules are the processors, but they could be memories, hardware modules, among others. If the output values from the modules do not match, the comparator signalized that an error is detected. The DWC issue is its detection capability since it only

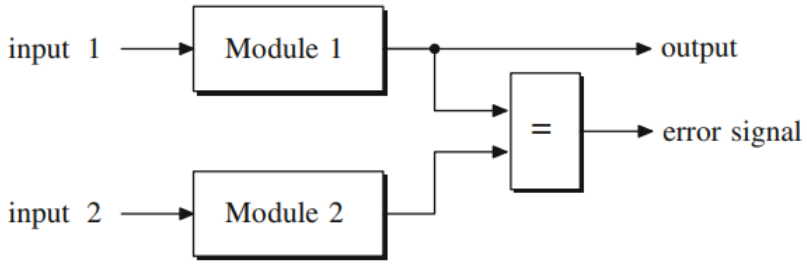


Figure 2.9: Duplication with comparison.

  detects one module error, but does not know what is the faulty module. If the two modules fail in the same way and they generate equal outputs, the comparator will see the occurrence as a correct one.

  The second basic active redundancy hardware is standby redundancy which uses *n* identical modules, each module includes an error detection unit, and all modules are connected to a switch monitor that controls and selects the module to be active. In case of an error in the active module, the switch changes the active module to another in order to generate the correct output. There are various applications with this type of active hardware redundancy [37]. An advantage of this method is the easiness of maintenance since it allows the system to be updated or repaired in a module during the execution of another.

  The last basic active redundancy hardware is the pair-and-a-spare which uses and combines the two techniques mentioned above, DWC and standby redundancy. The scheme is similar to standby redundancy, but, it uses two active modules instead of one active module. This way, it detects the faulty module and changes it to the other one.

  - **Hybrid redundancy** - this type of redundancy incorporates the other two types, the passive and active redundancy. It is used predominantly in safety-critical application because it leverages the advantage of both passive and active redundancies. Self-purging redundancy and N-modular redundancy with spares are the main techniques used in hybrid redundancy [17].

- **Information redundancy** - it is the group of techniques to achieve fault tolerance using code techniques [17]. There are many information redundancy techniques, but one of the most famous and older way of information redundancy in computer science that is still in use today [38], is the parity-code [39]. It allows to detect single-bit error and an odd number of multiple-bit errors by using an extra bit. Although parity-code can only detect and cannot correct errors in bits, there are adaptations from the first implementation that can detect and localize errors. This allows the system to correct the single- or multiple-bit errors [39, 40]. There are more ways of information redundancy such as linear codes, cyclic codes, unordered codes, and arithmetic codes [17].

- **Software redundancy** - this type of redundancy is used at different levels, such as functions, processes, or even system level. It consists of two classes, the single- version and multi-version [17]. The single-version uses techniques for error detection and recovery in a software component or system. The multi-version uses design diversity, explained below in subsection 2.2.2, and it applies different versions of software to do the same task, in order to reduce the probability of having common faults. Typically, the two classes adapt the hardware redundancy techniques to software redundancy. There are various solutions in state of the art: (1) the use of multithreading to get fault tolerance in systems against transient hardware faults [18] by running similar program copies concurrently in independent threads to obtain a better coverage in error detection; (2) another solution uses aspect-oriented programming to performs fault tolerance systems [31], which allows to apply fault tolerance at thread level to real-time embedded systems using a framework in order to reduce the effort during the system development; and (3) there are other solutions that use redundancy at the process or program level such as [32, 33].

- **Time redundancy** - it uses time as a resource to allows systems to be fault tolerance when other resources such as size, weight, and others are limited. This type of redundancy repeats the same task and saves the task results to compare and detect errors. Time redundancy is accurate and efficient when the system is affected by transient faults. When integrated with encoding technique, fault tolerance capabilities for permanent faults are achieved. Due to permanent and transient fault detection, the system becomes able to distinguish what kind of fault occurs in terms of fault persistence [17]. Some fault tolerance systems run the same program code multiple times and compare the results [32]. Alternatively, Kottke uses a lockstep technique with two cores which uses a delay of 1.5 clock cycles between the cores [35]. This way, the author achieves time redundancy in another way.

There are multiple solutions that use a combination of the redundancy types, for example, the combination of time and information redundancy [38], to achieve a fault tolerance system against transient faults with less power consumption and a better recovery time vs. an exclusive information redundancy system. Another solution uses the combination of software and information redundancy [32] to run similar program copies under the same processor. This kind of system is a good option for fault tolerance against transient faults, also against some types of permanent faults. Another example, a fault tolerant RISC-V softcore [41], it uses Multiple-Modular Redundancy (MMR) with Hamming codes [39] as an Error Correcting Code (ECC) mechanism to integrate hardware redundancy with information redundancy. And the last example, the combination of hardware and time redundancy [35], uses hardware redundancy with DCLS to run two copies of the program, achieving time redundancy with a delay of 1.5 clock cycles.

2.2.2 Design diversity

A fault tolerance system that uses redundancy-only to achieve dependability attributes has a lack of fault tolerance against faults affecting redundant modules at the same time. These type of faults cause failures knowing as CMF [42, 43] which lead systems to failure because all the modules that are affected may fail, making harder the error detection or the system recovery. One of the best ways to protect fault tolerance systems of CMF is the usage of design diversity [43, 44].

Design diversity is achieved by using different designs in the redundant modules applied to multiple levels of hardware, software, or information [45]. For example, in hardware, a system can acquire design diversity by using different components to do the same function. On the other side, the use of different programs, algorithms, programming languages, or even data structures to save the same data are examples of design diversity in software and in information techniques. When a fault tolerance system uses redundant software modules following the multi-version technique approach, typically, design diversity is applied between redundant software modules [17].

However, a trade-off between the quantity of design diversity is required during the development phase, and as the higher diversity a system acquires, the less stable it will be, but maybe the system is more resilient [46].

2.2.3 Lockstep technique

The main idea of a lockstep system consists of executing the same program on different processors, comparing constantly their generated outputs. If the outputs match, the execution follows normally, otherwise, at least one error occurred in the system. The principle is similar to DWC, TMR, or n-modular redundancy, depending on the number of replicated components, however, it adds a recovery mechanism, and sometimes a checkpoint system.

Ozer characterizes lockstep in three types, shown in Figure 2.10 from [8]: (1) system level, (2) subsystem level, and (3) Central Processing Unit (CPU) level.

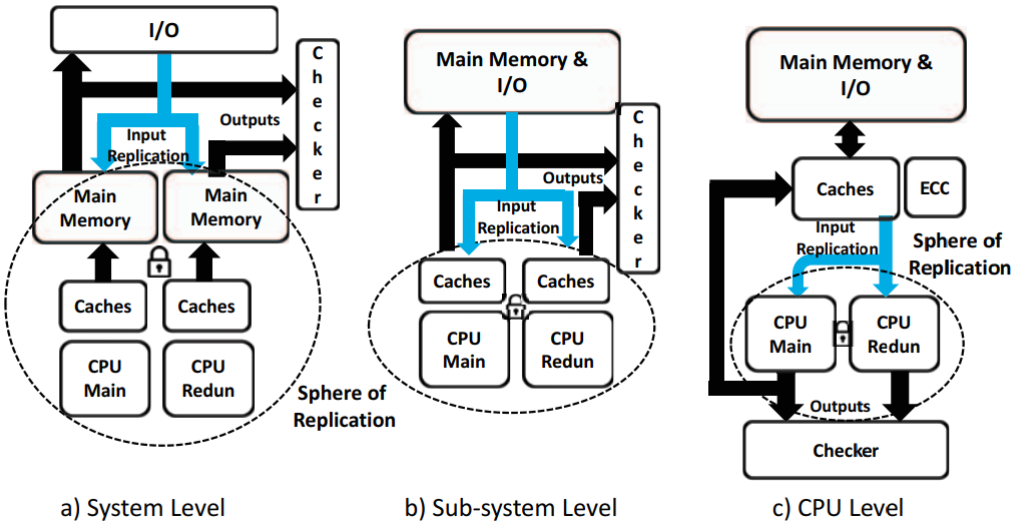


Figure 2.10: Three types of lockstep in computer systems.

Lockstep at system level replicates core, cache and memory. Also, the input is replicated in the Input/Output (I/O) interface, and the outputs are compared through a checker entity. The lockstep at sub-system level replicates the core and cache, and the input and output have the same management as the system level, while at the sub-system level the main memory is shared between cores. Finally, the CPU level only replicates the core, and the output from the cores is compared in the checker.

Lockstep mechanisms can be implemented in a tightly- or loosely-coupled approach. In a tightly-coupled approach, the comparison is performed in each system clock. In a loosely-coupled approach the comparison is periodically or it is through a checkpoint feature [47]. When comparing both approaches, in terms of the error detecting time, a loosely-coupled approach performs worse, which makes the error to be propagated during more time. However, it offers less overhead in execution time.

When the system uses Dual-Modular Redundancy (DMR), the system is a DCLS. If the number of replications in the system is more than two, it is a MMR, with the specific case of three copies known as Triple-Core Lockstep (TCLS).

2.2.3.1 Error detection

In lockstep systems, the error detection is performed by comparing the values output from each processor at a certain time. However, the way these values are compared depends not only on the number or processing cores but also on the approach that is adopted, i.e., loosely- our tightly-coupled. DCLS can detect that an error occurs, but it cannot know the core with the error [48]. Then, in case of error detection, both cores stopped and signalized that an error exists. In MMR implementation, it can detect and disable the core that is at an error state and signalized it. This allows the system to continue the execution and apply the system recovery in parallel.

2.2.3.2 System recovery

When the system receives the error signal, its subsystem recovery also depends in its number of redundancy. In the DCLS case, the system recovery applies a rollback or a rollforward to a state of integrity in both sides. In contrast to the MMR implementation, the system recovery is only applied in the core in error state.

2.2.4 Fault tolerance techniques

This subsection provides, in the end, an overview of the typical fault tolerance techniques and compares them in a qualitative approach in different characteristics, such as redundancy, execution time overhead, fault coverage, and fault correction.

Before the overview analyses, it is important to clarify the technique classification. Azambuja divides the fault tolerance techniques into three groups [49]:

- **Software-based techniques** - this group of techniques is characterized by the use of software, time and information redundancy to develop error detection or system recovery for system program execution, and typically they are applied over the program code.

  The advantages of software-based techniques are their flexibility for implementation or modification. And some techniques have error correction, and do not need hardware adaptations.

  The use of software-based techniques means more processor execution time and overhead in memory that leads to the main issue of the performance degradation. Another problem in software-based techniques to achieve fault tolerance is the inability of the error detection system to detect faults when a fault in hardware reaches the program execution, but the system sees the fault as a false positive error.

- **Hardware-based techniques** - these techniques add hardware redundancy or time redundancy to fault tolerance systems. The time redundancy used in these techniques is the one applied directly in the hardware (see the clock delay example inside the time redundancy topic in subsection 2.2.1).

  An advantage face to the other techniques are their high fault coverage, their fast detection, and implementations do not use software adaptation. Hardware redundancy is a large part of hardware-based techniques. DWC and TMR are the mains techniques in this group.

- **Hybrid techniques** - the last group of techniques integrate both techniques described above in the same system, i.e., the hardware-based technique with the software-based technique. This combination allows systems to be more fault tolerant (higher fault coverage from the hardware-based techniques) with low cost and smaller solutions (from software-based techniques).

  The biggest advantage of hybrid techniques is the possibility to get the best from both hardware- and software-based techniques. This advantage also brings bad features (normally, the same bad

features from hardware- and software-based techniques) to the hybrid techniques. However, these negative features are smaller than when the system is only hardware- or software-based.

Some examples of hybrid techniques are derived from hardware-based techniques and originates hybrid techniques such the DCLS and the TCLS.

2.2.4.1 Fault tolerance techniques overview

Table 2.2 includes a panoply of the main fault tolerance techniques. It details in a qualitative way the set of techniques accordingly to different characteristics, such as type and area overhead from redundancy, execution time overhead, and fault coverage.

The first three techniques DWC, TMR, and n-modular redundancy, use only hardware redundancy and normally it provides a high fault coverage. The increase of redundancy also increases cost, size, and weight. In contrast, the time overhead during execution is low compared with other techniques. The advantage of TMR and n-modular redundancy techniques compared with DWC is the error masking feature, but to achieve it, these techniques need more than two modules in the system. There are some variations of TMR, for example, the Local TMR (LTMR), where only flip-flops in FPGA are triplicated, or the Global TMR (GTMR) that triplicates all the system modules, even the clocks [50]. Other implementations use TMR in the most sensitives components, for example, triplication of only the memory [51], or in the configuration engine, responsible to reconfigure the soft-core in an FPGA [52].

Single- and multi-version techniques have a high execution time overhead, and their fault coverage is not as high as the hardware techniques. These types only use extra memory for fault tolerance algorithms code or extra diversity code. Time redundancy techniques do not have area overhead, but they have a high level of execution time overhead, and this overhead is proportional to the number of time redundancy in the system. Parity code and its adaptations are a good option to use in memories. The extra area used is minimum compared to hardware techniques. However, those techniques that use information redundancy have low fault coverage. If the systems need fault tolerance, and there is no possibility of adding hardware redundancy, the use of software techniques is a good option. For example, Duplex Multiplexed in Time (DMT) [53] achieves fault tolerance by combining software and time redundancy. This solution does not use extra hardware and in contrast, it incurs a high execution time overhead.

The derived technique DCLS from DWC has a similar size to DWC, a bit higher than two, because, in addiction to the dual-core architecture, it also needs a comparator system. This system at hardware level uses the same mechanisms of a DWC, and thus takes the same advantage, higher fault coverage. On the other hand, the software system recovery add some overhead in execution time in case of error. There are various solution that uses DCLS [26, 35, 36, 47, 52, 54, 55, 56], mainly because of its high fault coverage and reduced size, weight, and cost compared to TMR or N-Modular redundancy techniques. Moreover, it is always preferable to use DCLS instead of DWC due to its software system recovery.

When the system includes a TMR at core level, e.g., a TCLS [57], the execution time overhead is low compared to the DCLS, and if an error occurs, the system can recover one of the cores while the other

two cores mask the error. If the size, weight, and cost is not a restriction to achieve fault tolerance, TMR is a good solution. It also has a high fault coverage and low overhead during time execution.

Finally, to cover CMF, the last technique combines design diversity with DCLS with the advantage of very high fault coverage with a similar area overhead DCLS. When the design diversity is applied at core level, all types of redundancy will be used. This leads to a bigger effort during the development phase because it is necessary to synchronize both cores. The information and time redundancy is achieved by using two different cores with different ISA, that uses different data structures, registers, instructions, among others. Despite its development effort, this solution is the best solution to fault coverage. In case of low execution time overhead demand, this solution can be incremented to a TCLS with design diversity, at the cost of the development effort.

Table 2.2: Fault tolerance techniques overview.

Technique	**Technique Group**	**Redundancy**		**Execution Time Overhead**	**Fault Coverage**
		Type (*)	Area Overhead		
TMR	HW-based	H	∼3x	Low	High
N-Modular Redundancy	HW-based	H	>3x	Low	High
DWC	HW-based	H	∼2x	Medium	High
Single-version	SW-based	S	-	High	Low
Multiple-version	SW-based	S	-	High	Medium
Time Redundancy	SW-based	T	-	High	Medium
Parity Code	SW-based	I	-	-	Low
Parity Code Adaptations	SW-based	I	-	-	Low
DMT	SW-based	S,T	-	High	Medium
DCLS	Hybrid	H,S	∼2x	Medium	High
TCLS	Hybrid	H,S	∼3x	Low	High
DCLS + Design Diversity	Hybrid	H,I,S,T	∼2x	Medium	Very High

(*)Type of redundancy: H - Hardware redundancy; S - Software redundancy; T - Time redundancy; I - Information redundancy

Finally, some consideration and notes about fault tolerance techniques are:

- Baumann refers that lockstep mechanisms are a good solution for fault tolerance systems against SEE [4];
- If the cost and performance are the most important criteria, then TMR is the best choice for fault tolerance hardware technique [2];
- If the availability is not the main concern, then DWC or DCLS with design diversity is a good option for fault tolerance system. It is smaller than TMRs solutions and it has a very high capability for fault detection.

2.3 Related work

This work is part of a project that includes two different systems. However, they share the same fault tolerance methodologies and techniques [58, 59]. The project aims the development of a fault tolerance system using redundancy and design diversity at the core level. The redundancy is defined by a DCLS system with two different core architecture in order to achieve benefits from redundancy and design diversity. The most significant difference between systems is the platform and the core architecture. The system developed in this dissertation has a hard-core Arm Cortex-M3 and a soft-core Mi-V by Microsemi (RISC-V-based), both a 32-bits Reduced Instruction Set Computer (RISC) processor. In contrast, the other system includes a 32-bits hard-core Arm-A9 and a 64-bits soft-core lowRISC (also RISC-V-based).

So far, to the best of the author's knowledge, there are no similar implementation beyond those mentioned above [58, 59]. However, there are important works in different fields that supports this project for fault tolerance in a heterogeneous architecture (subsection 2.3.2) or lockstep systems (subsection 2.3.3). Also, there is a project that can complement this work in order to make it more consistent, such as the fault tolerance memories, mentioned in subsection 2.3.1, where they can be attached in parallel to this project to turn the memory system a fault tolerance one because it is a weak point in this work.

2.3.1 Fault tolerance on memory

Verhage proposed a scheme with two different levels of fault tolerance for cache structure for a RISC-V soft-core [60], one of them provides a light protection that uses distinct error detection and correction techniques depending on the number of bits. This solution, in order to achieve the low protection level, uses the Hsiao code technique [61] for Single Error Correction (SEC) and Double Error Detecting (DED) when the number of bits is larger than a byte, or it uses a parity bit and a write-through policy for exactly a byte. Only this level of protection was implemented and verified. The light-weight protection reduces performance when a fault occurs. However, the system is fault-tolerant, and the results are coherent with the simulation and the FPGA behaviour. The other protection level, the high level one, just has the design description, and by the lack of data results, it is not possible to implement directly without a development effort and because it was not examined for this project.

Although this subject is out of the dissertation scope, it is important to mention that a system to be a complete fault tolerance system, all the modules need to be verified in order to block the error propagation in case of an error appears. The example above shows an example to protect the memory at different levels. Solutions like that one can be integrated into this dissertation project, making the system more robust and dependable, either at the core level or either at the memory level.

2.3.2 Heterogeneous architecture

There are solutions that uses heterogeneous architecture in order to achieve multiple proposes, such as energy efficiency [62], performance [59, 63], between others, and also error detection [64].

The architecture proposed by Ainsworth is composed of a main high-performance out-of-order core that executes in parallel with small multiple checker cores [64]. The mechanisms to detect errors consist in verify application fragments independently in each small checker core while the main core executes the entire application. The heterogeneity is applied with partial replication of the main core with the multiple checker cores, and neither in the DCLS, nor at ISA level. However, the architecture achieves low overhead in area, performance, and reduced energy consumption. Despite the extra effort during the development phase, this type of solution supports this dissertation in the sense that heterogeneity can bring better performance to systems.

2.3.3 Lockstep systems

Many fault tolerance solutions exist in the literature, either at academic researchs or at industrial applications. There are Commercial Off-The-Shelf (COTS) solutions that use built-in lockstep to aims fault tolerance systems [65, 66], also there are hard-cores that makes it possible to operate in lockstep mode [67, 68, 69, 70, 71, 72]. Some of this cores are used in the lockstep solutions that uses hard-cores to develop fault tolerance systems [6, 26, 34, 36, 47, 54, 57, 73, 74]. In contrast, other solutions uses FPGA to build lockstep systems [35, 52, 75], offering freedom to choose and adapt if necessary the cores, and to develop the extra hardware necessary to apply the lockstep between cores.

The solutions provided by [26, 34, 75] use a DCLS following a tightly-coupled approach. The biggest difference between the three solutions is the delay clock cycles between cores. In [26] there is a delay of 2 clock cycles between the cores Arm Cortex-M7. Also, in [35], a DCLS tightly-coupled implemented in an FPGA with two SPEAR soft-cores [76], includes a delay of 1,5 clock cycles between cores. This time redundancy brings more performance in error detection, mainly against to common-mode faults compared to solutions without time redundancy.

In lockstep implementations [6, 47, 54, 73], the authors use DCLS with hard-core, but all the implementations include an FPGA, mainly because they use a loosely-coupled DCLS approach. These implementations use the FPGA to implement custom modules to support the synchronization and comparisons between cores.

The architecture in [52], shown in Figure 2.11 (form [52]), includes a DCLS with two soft-core in the FPGA side (both processors are MicroBlaze). The architecture also includes a comparator in order to detect errors and a multiplexer to connect the output from one of the soft-core in the system output. Lastly, this system uses TCLS with three PicoBlaze soft-core in the configuration engine The reason to use TCLS is claimed by the importance of the configuration engine to run without interrupt and errors. Both

lockstep systems are tightly-coupled and the system is implemented in an FPGA mainly because of the core's nature.

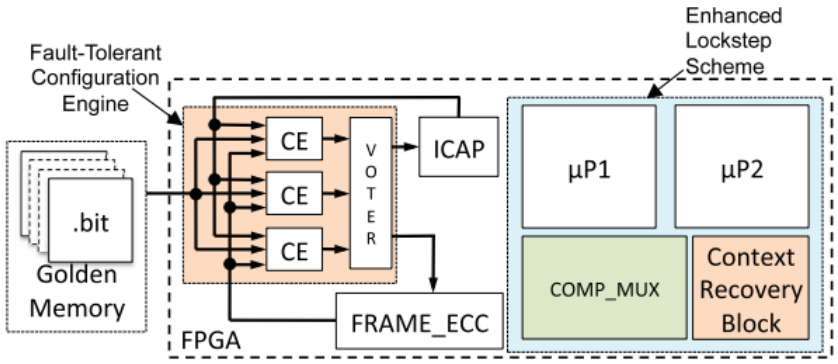


Figure 2.11: Fault tolerance architecture for FPGA with soft-core DCLS and TMR in configuration engine.

In [57], a TCLS tightly-coupled between three Arm Cortex-R5 is used due to the dual-core safety-critical nature of the Arm Cortex-R5, each one includes hardware mechanisms to deal with errors, and it includes inside of each unit a DCLS. In contrast to DCLS solutions, this solution has a recovery system that executes without software interaction.

Figure 2.12 (from [77]) shows a fault tolerance architecture that triplicates all logic, adds multiplexers and corruption detectors before each register, and adds voters after each register, to achieve the fault tolerance system [77]. This work, due to TMR and the voting system allows detection and masking of the error until the system eliminates it. Also if a SEU affects the combinational parts and the corruption detectors do not match between them, an error is alerted and the system starts an FPGA reconfiguration to repairs the system.

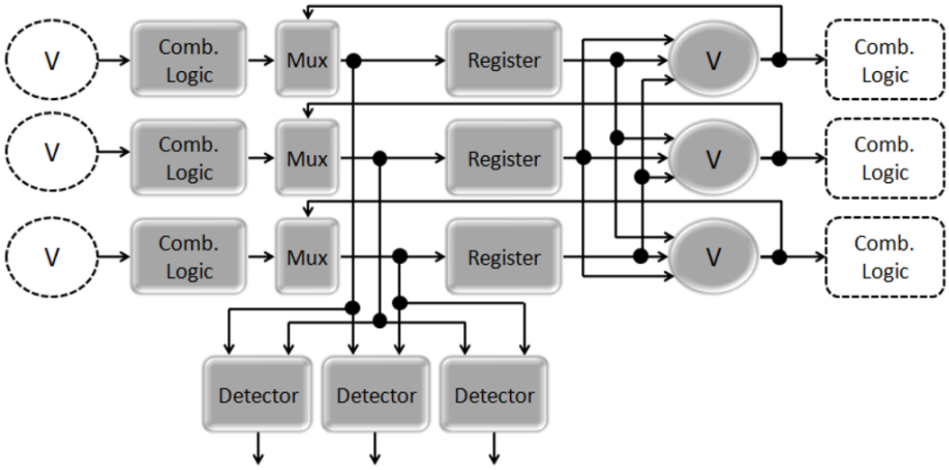


Figure 2.12: Fault tolerance soft-core architecture for FPGA with triplication of all units.

In [64], it is the only in the panoply of solution analysed here, that uses design diversity in lockstep, but this is only a proposed solution without an implementation. The proposed solution aims a high performance technique for fault tolerance by using a new way of parallelism in the state of the art.

Table 2.3 summarizes the main lockstep implementations in the literature, showing architectural characteristics such as hard- or soft-core and if it resorts FPGA fabric. It also shows the type and the redundancy dimension of the lockstep. Lastly it informs the use of design diversity or not in the system.

Table 2.3: Lockstep related work overview.

	Architecture		**Lockstep**		**Design**
	Core	FPGA	Type (*)	Redundancy	**Diversity**
Klecka et al. [34]	hard-core	No	T	DCLS	No
Abate et al. [6]	hard-core	Yes	L	DCLS	No
Yiu [26]	hard-core	No	T	DCLS	No (**)
Kral et al. [47]	hard-core	Yes	L	DCLS	No
Oliveira et al. [54]	hard-core	Yes	L	DCLS	No
Sun et al. [73]	hard-core	Yes	L	DCLS	No
Kottke and Steininger [35]	soft-core	Yes	T	DCLS	No (**)
Cornejo et al. [75]	soft-core	Yes	T	DCLS	No
Pham et al. [52]	soft-core	Yes	T	DCLS and TCLS	No
Iturbe et al. [57]	hard-core	No	T	TCLS	No
Chaudhry and Tremblay [74]	hard-core	No	T	MMR	No
Ainsworth and Jones [64]	-	-	L	MMR	Yes (***)

(*) T - tightly-coupled; L - loosely-coupled.

(**) Lockstep implementation includes time redundancy with clock delay.

(***) Design diversity at size level.

2.4 Summary

This chapter introduces the necessary background to understand dependability systems, focusing mainly on fault tolerance technique, one of the main means to achieve dependability and ends with the related work, listing some important fault tolerance solutions that use lockstep (Table 2.3).

The information along this chapter allows to retain some important notes: (1) The use of redundancy in a fault tolerance system is not enough for safety-critical applications [50], and combining different methodology is a solution to get more performance in fault tolerance systems; (2) The lack in the state of the art of fault tolerance system using hardware redundancy and design diversity opens an opportunity to explore it. Although the development effort to build these systems, it allows to achieve a fault tolerance with a very high fault coverage, mainly against common mode faults; (3) Taking into account the solutions above and the techniques in Table 2.2, the development of a DCLS with design diversity at core level is a good option to explore in fault tolerance environment. (4) With the lockstep related work analysed here, see Table 2.3, there is no implementation with design diversity, and the only solutions that are developed

against CMF are the ones with time redundancy. However, the delays in the two implementation [26, 35] may not be enough against CMF due to its short delay.

3. Platform and tools

This chapter identifies the platform and the hardcore MCU (section 3.1), and tools used throughout the development of this work (section 3.2) which shows a list of soft-cores RISC-V-based, and it explains the selected soft-core for this project.

3.1 Reconfigurable technology

The main goal of this dissertation is to implement a DCLS under a heterogeneous architecture at core level, which implies the usage of two different cores in the same platform. Moreover, FPGA technology allows design diversity and it makes possible the deployment of customized hardware in order to achieve the heterogeneous DCLS at core level. And when compared with ASIC implementations, FPGA, due to its features such as reprogrammability enables fast prototyping at reduced costs [43].

FPGA technology enables the development of hardware that can be customized for a specific propose [78]. In the context of this dissertation, FPGA can be used to deploy soft-cores, as well as modules to support the DCLS system. There are several platforms that combines FPGA with MCU, and typically, the platform allows the connection between the MCU and the remaining part of the Fabric FPGA. This characteristic provides added value in term of development effort as it is possible to use the hard-core MCU in the DCLS system. This way, the FPGA is responsible to include only one soft-core and the extra modules to support the DCLS.

During the specifications of the project, the chosen core architectures are Arm and RISC-V. The other work carried out in parallel with this one uses the ZedBoard with two Arm Cortex-A9 [59], to evaluate the system in an alternative platform. The chosen platform for this dissertation is the Microsemi SmartFusion2 with one low-power hard-core Arm Cortex-M3, that fits with the desirable low-end characteristics.

3.1.1 Microsemi SmartFusion2

There are various project that uses Field-Programmable System-On-Chip (FPSoC) which combines MCU and FPGA in the same platform. The Microsemi SmartFusion2 is an example of a FPSoC platform used to implement hardware-assisted solutions [51, 63, 79, 80, 81] due to the fusion between the MCU Arm Cortex-M3 and the FPGA fabric.

Figure 3.1 (from [82]) shows the Microsemi SmartFusion2 SoC FPGA, a platform which includes a flash-based FPGA fabric and a hard-core Arm Cortex-M3 that can connect soft-peripherals (deployed in the FPGA), through the Advanced Microcontroller Bus Architecture (AMBA) interfaces, Advanced High-performance Bus Lite (AHB-Lite) or Advanced Peripheral Bus (APB). This platform includes some reliable features, that can be used to make the fault tolerance system stronger in terms of dependability attributes: (1) SEU-immune (Zero FIT FPGA configuration cells); (2) SEC/DED in the Cortex-M3 eSRAMs, the Double Data Rate (DDR) memory controllers, and others peripherals; (3) buffers with SEU resistant latches in the DDR bridges, in the instructions cache, between others; and (4) Non-Volatile Memory (NVM) integrity check. Also, it includes various securities features that matches with the dependability attributes [82].

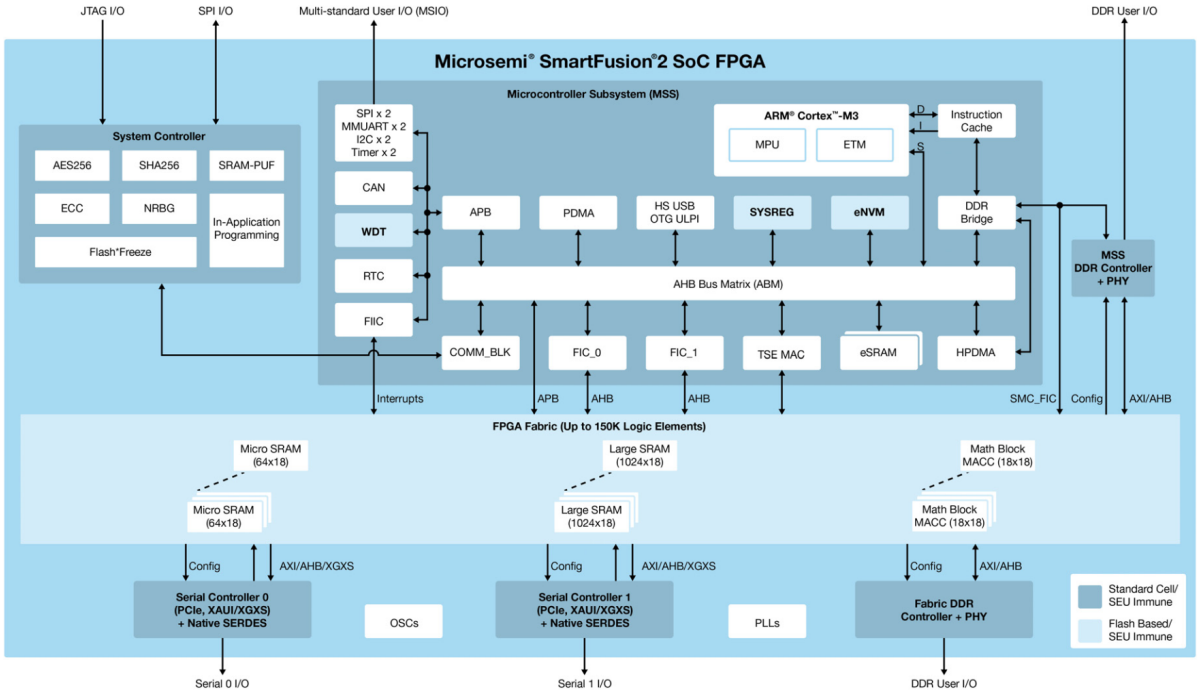


Figure 3.1: Microsemi SmartFusion2 SoC FPGA Block Diagram.

More than just leveraging a hard-core with low-end characteristics in the Microsemi Smartfusion2 platform, the Microsemi environment provides the Mi-V RISC-V ecosystem [83]. This ecosystem offers RISC-V-based (soft-cores) Intellectual Propertys (IPs), and design resources to integrate them in their FPGAs. This environment meets all the requirements to develop the proposed fault tolerance system, and in addiction, it is a valuable tool for this dissertation because it helps to reduce the engineering effort during the implementation process.

The development of this project performed under the SmartFusion2 Security Evaluation Kit, that includes 90k Logic Elements (LE) and it includes all security and reliable features from SmartFusion2. The board includes a 64 Mb SPI Flash memory, a 512 Mb low-power DDR, a Peripheral Component Interconnect (PCI) Express interface, a Joint Test Action Group (JTAG)/Serial Peripheral Interface (SPI) programming interface, and a set of headers for SPI, General-Purpose Input/Output (GPIO), between other

features [84].

Microsemi provides Libero SoC Design Suite for design and deployment under Smartfusion2 platform. It includes a toolset to design, synthesize and simulate for SmartFusion2, it also integrate a firmware flows for SoftConsole, an Integrated Development Environment (IDE) that provides development and debug support for Arm Cortex-M3, and RISC-V soft-cores from Mi-V ecosystem. The Microsemi Libero SoC Design Suite allows the use of Hardware Description Language (HDL) to design and test FPGA peripherals/accelerators. It also provides a set of tools to simulate using Bus Functional Model (BFM), to emulate any type of AMBA interface, leading to an effort reduction during the verification step.

3.2 RISC-V-based soft-core solutions

For implementing the DCLS system proposed by this dissertation, it was used a soft-core RISC-V processor, mainly due to its free and open-source nature, as well as its impact in computer systems environment.

RISC-V is a free and open-source ISA based in a RISC architecture, designed with a modular concept with different extensions to be adapted to all type of applications and focusing mainly in modern devices as cloud, mobile, embedded, IoT systems, among other devices. The ISA has been designed mainly for 32-bit and 64-bit architectures, although the RISC-V manual specifies a flat 128-bit address space [85]. More details about its extensions and characteristics can be consulted in [86].

RISC-V processors are in constant expansion since 2010 in both academic and industrial context [87, 88, 89, 90]. One of the reasons why the RISC-V ISA has been impacting is its open-source feature that allows to modify the core for a specific purpose, which originated the development of several soft-core implementations such as: Rocket, BOOM, lowRisc, PULPino, PicoRV32, and Mi-V RV32.

- Rocket core uses the Rocket Chip Generator [90] that generates a synthesizable Register-Transfer Level (RTL) general-purpose 5-stage in-order cores. It is a tethered processor as it demands for a host environment for boot processing, i.e., a companion core that is usually present in the platform, used to help in the execution of the Rocket core;

- BOOM core, also generated by Rocket Chip Generator, leverages an out-of-order architecture. Similar to the Rocket core, BOOM core is a tethered processor [91];

- lowRisc core is an untethered soft-core based on Rocket core;

- PULPino is an untethered soft-core and an in-order one that allows configurations between 2- or 4-stages, but requires modifications to integrate an AMBA interface;

- PicoRV32 is an untethered soft-core which provides three variations and one of them includes an AMBA Advanced Extensible Interface (AXI)4-Lite interface;

- Mi-V RV32 soft-core includes three different variations that includes different AMBA interfaces. This soft-core is also untethered and it is supported by Microsemi SmartFusion2 without any modification.

Table 3.1 lists the soft-cores mentioned above, showing their differences at the processor level, available bus interfaces, ISA, and platform support. The soft-core selection depends especially in its tethered/untethered nature and the support of an AMBA interface. Untethered soft-core is a preferable solutions, because it does not demand for a extra host environment to boot the soft-core. And due to the chosen platform, that includes a hard-core with a AMBA interface, if the soft-core also includes a AMBA interface it reduces the development effort.

Table 3.1: List of RISC-V soft-core.

Core	Untethered	ISA	Board Support (*)	AMBA Interface
Rocket [90]	No	RV32G / RV64G	No	Yes
BOOM [91]	No	RV64G	No	Yes
lowRisc [92]	Yes	RV64GC	No(**)	Yes
PULPino [93]	Yes	RV32IMF	No	No
PicoRV32 [94]	Yes	RV32IMC	No	Yes
Mi-V RV32 [83]	Yes	RV32IMA(F)	Yes	Yes

(*) Implementation in Microsemi SmartFusion2 without any modification.

(**) Only supported for Microsemi polarfire.

Mi-V RV32 is the chosen soft-core due to its reduced development effort by supporting an AMBA interface and the Microsemi SmartFusion2 support platform, it is also an untethered core, which means that it does not need any host environment for the boot processing in contrast with Rocket and BOOM core. The next subsection 3.2.1 explores the 3 different variations of Microsemi Mi-V soft-core.

3.2.1 Microsemi Mi-V

All the three soft-core variations that Microsemi Mi-V environment provides are untethered and all the soft-cores include an AMBA interface, AXI or Advanced High-performance Bus (AHB). The three variations are:

- **Mi-V RV32IMAF L1 AHB** - This version, is a soft-core based in the SiFive Coreplex E31 soft-core (RISC-V standard RV32IMAF) and it includes two AHB interfaces, one for memory and other for I/O. In the SmartFusion2, this soft-core can achieve a maximum frequency of 58.38 MHz [95].
- **MIV RV32IMA L1 AHB** - This soft-core is a Rocket core based with RISC-V standard RV32IMA. As the version above, it includes two AHB interfaces and one for memory and other for I/O. In the SmartFusion2, this soft-core can achieve a maximum frequency of 85.0 MHz [96].

- **MIV RV32IMA L1 AXI** - This soft-core is also a Rocket core based with RISC-V standard RV32IMA, but this version includes two AXI interface. In the SmartFusion2, this soft-core can achieve a maximum frequency of 90.0 MHz [97].

All the cores requires the use of a memory controllers, provided by Microsemi IP cores, to reload the instruction and data caches. The I/O interface by AXI or AHB requires another core to convert the AXI or AHB signals to APB in order to use the peripherals provided by Microsemi IP cores and the hardware accelerator developed. And in the soft-core that uses AXI it is necessary a converter with two stage (if the implementation uses only Microsemi IP cores), an AXI to AHB conversion and an AHB to APB conversions.

The MIV RV32IMA L1 AHB is the chosen soft-core version, because it uses only one conversion, AHB to APB, which is the AMBA interface to implement the hardware accelerator in Lock-V. In a future work in can be used the Mi-V RV32IMAF L1 AHB without adaptations, but this soft-core has slower clock frequency and uses more LE than the Rocket-based. Moreover, the extensions of this RISC-V core are used for low-end implementation: (1) the "I" - base integer instruction set; (2) the "M" - standard extension for integer multiplication and division; and (3) the "A" - standard extension for atomic instructions [86].

Figure 3.2 (from [96]) shows the soft-core MIV RV32IMA L1 AHB block diagram, a soft-core which includes a JTAG interface for debug, an external interrupt interface, two AHB interface for memory and I/O. It also includes a ECC feature through an SEC-DED interface for instruction and data caches.

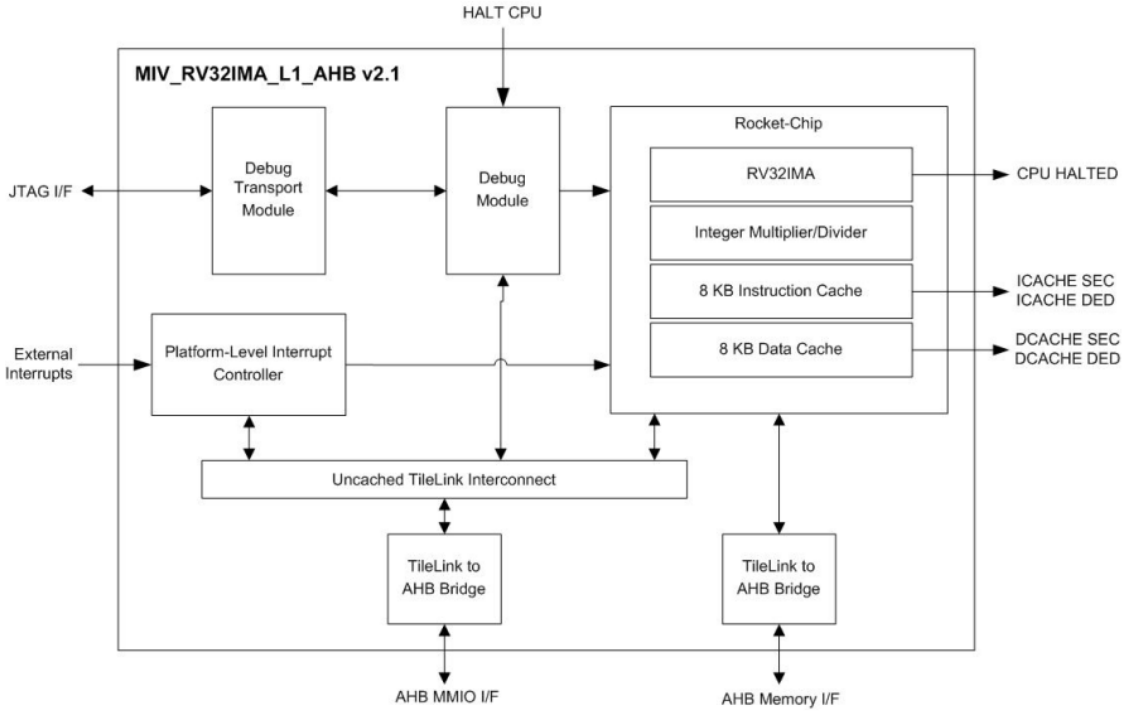


Figure 3.2: MIV RV32IMA L1 AHB soft-core block diagram.

3.3 Summary

The chosen platform for this project is the Microsemi SmartFusion2 that includes an hard-core Arm Cortex-M3 with a directly interface AMBA to the FPGA fabric. This platform includes dependability features that can be used to improve the fault tolerance system. Moreover, the hard-core embedded in the SmartFusion2 presents low-power characteristic that fits with low-end focus of this dissertation.

The soft-core Mi-V RV32IMA with AHB interface is the chosen soft-core for this project due to the AHB interface and the reduced LE utilization compared to the other AHB soft-core implementations from Microsemi. Furthermore, its untethered characteristic and the board support without requiring any modification are taken into account during the core selection.

4. Lock-V Architecture

The main goal of this work is the development of a fault tolerance mechanism and the core components of a DCLS system with design diversity at core level leveraged by one hard-core Arm Cortex-M3 and a RISC-V-based the soft-core Mi-V RV32IMA. Figure 4.1 depicts the proposed system architecture, Lock-V, which is responsible to supports all the components of a DCLS system, core synchronization, output comparation, error detection, and rollback features.

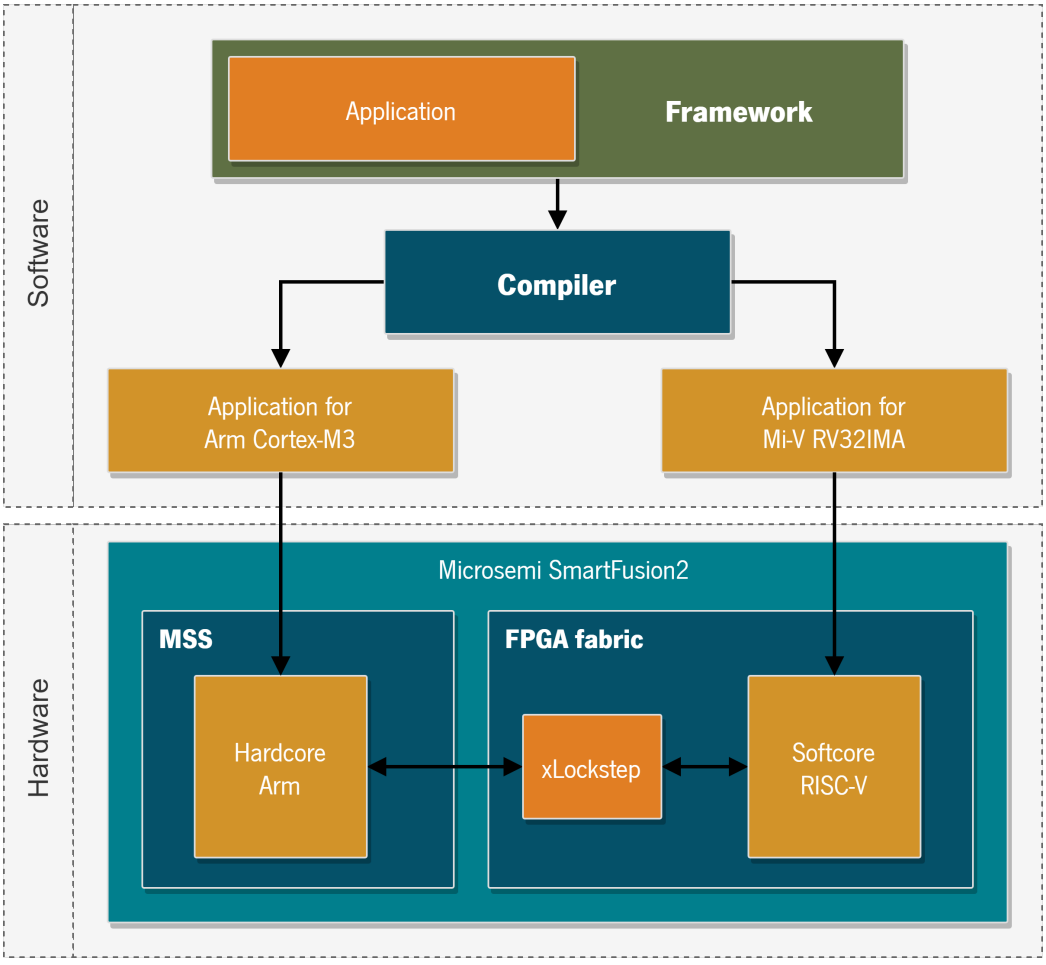


Figure 4.1: Proposed DCLS architecture, Lock-V.

Due to design diversity requirements, the Lock-V resorts two different processor architectures. Given

so, this approach does not allow the error detection at instruction level. This can be overcome by implementing checkpoints along the application execution flow, which can output specific data/register values for the comparation. Moreover, this type of architecture, the DCLS, is an active redundancy which requires a execution interruption whenever a checkpoint is reached, in order to apply its error detection mechanism.

The Lock-V is divided in two main modules: (1) the software module, and (2) the hardware module. The hardware module includes the two different processors, the *xLockstep* accelerator that provides to the system the DCLS capabilities, and all the interfaces to connect the cores with the *xLockstep* accelerator, through memory mapped interface provided by APB3. The *xLockstep* architecture as well its submodules are detailed in the section 4.1. The accelerator, the processors and their connections in the platform Microsemi SmartFusion2 establish the architecture Lock-V, described in section 4.2. The software module is responsible to compile and to produce the binary machine code for each core architecture from the main application. This work also provides a framework in order to simplify the adaptation process of a normal application to a DCLS application, as detailed in the next chapter 5. To provide these characteristics, the module is composed by an API that details how the software uses the memory mapped feature to use the accelerator. The section 4.3 presents the API that supports the accelerator.

4.1 *xLockstep* architecture

The *xLockstep* accelerator was developed under a modular design methodology, and written with verilog. The Figure 4.2 shows all the modules and interfaces combined to generate the *xLockstep* accelerator.

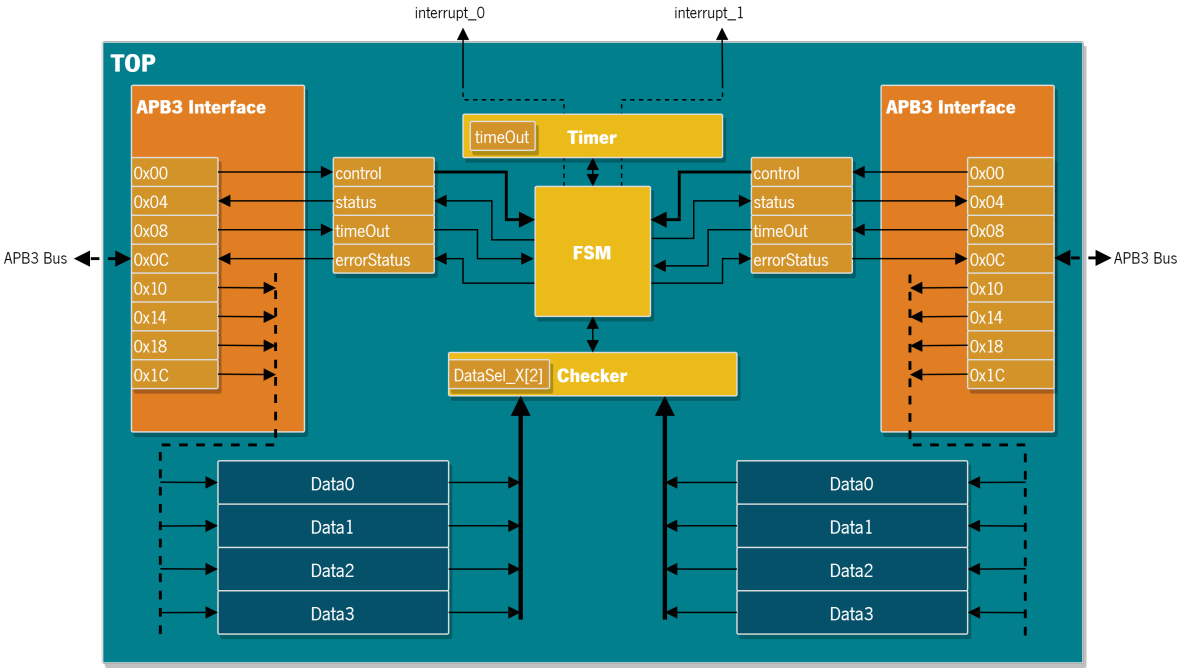


Figure 4.2: *xLockstep* architecture overview.

The accelerator has four interfaces, two APB3, and two interrupt signals, one for each core. Inside the top module is included all the modules to provide the DCLS capabilities, two APB3 interfaces, a timer, a checker, a FSM, and a set of data storage.

The accelerator includes a set of control, status and data registers to allow the system to be controlled and used from outside using registers handled by the cores through the APB3 interface. All the registers are 32-bits and they includes two read-only status registers, two write-only control registers, and four write-only registers for each core:

- **Control Register** - Write-only register (Figure 4.3) that is responsible to control the system, and is composed by 6 bits:
  - *bit_Enable*, it enables the system when it is set, and it resets the system when it is clear;
  - *bit_SynchroX*, it starts the synchronization when it is set and the necessary conditions are met, and it finishes the synchronization and comparison when the bit is clear;
  - *bit_DataSel_X* where *X* can be 0 or 1, indicates the number of data to compare, which can varies from 1 to 4 according to Table 4.1;

Table 4.1: *bit_DataSel_X* selection according with number of data to compare.

Number of data to compare	bit_dataSel_1	bit_dataSel_0
1	0	0
2	0	1
3	1	0
4	1	1

  - *bit_EnInterrupt*, when this bit is set the system uses the interrupt interface showed in Figure 4.2. When *xLockstep* finishes the comparison the interrupt signal is activated until the *bit_SynchroX* is cleared;
  - *bit_ErrorSolve* is used to inform the accelerator in case of error that the error was fixed.

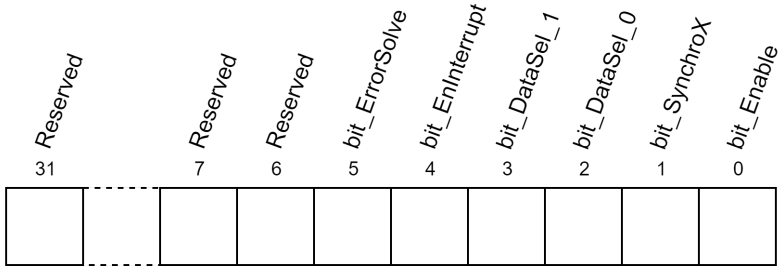


Figure 4.3: Control Register.

- **Status Register** - This register, Figure 4.4, is read-only, and it provides information about the accelerator state. The register has the *bit_busy* to inform if the *xLockstep* accelerator is busy or

not, the system is busy in the *Synchro* and *Checker* states. Also, the register has three others bits, *bit_stateFSM_X* (*X* can be 0, 1, or 2) to indicate the actual FSM state. Every time that the FSM state updates, the states of these bits are also updated;

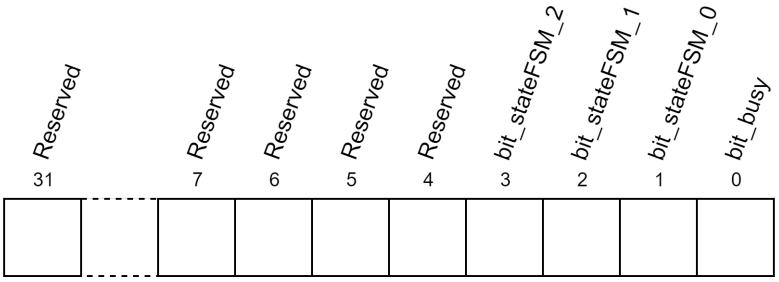


Figure 4.4: Status Register.

- **TimeOut Register** - This write-only register used to load the timer module with the time out value (see subsection 4.1.2 to understand the register utility);

- **Error Status Register** - This read-only register, Figure 4.5, provides information about the errors status. If the data comparison gets a mismatch between data, the *xLockstep* uses the bits *bit_dataError_X* to inform what *dataX* is corrupted. If one of the cores does not reach the checkpoint within the allocated time as defined in the *TimeOut* register, then *xLockstep* will set the *bit_timeOut_CoreX*, where *X* corresponds to the core that did not reach the checkpoint;

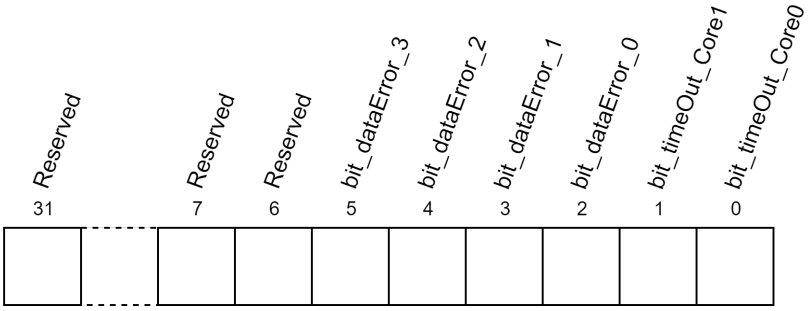


Figure 4.5: Error Status Register.

- **Data*X* Register** - Write-only registers, where *X* go from 0 to 3. These registers are used to compare the specified data when a checkpoint is achieved. See subsection 4.1.3 to know more about these registers.

The *xLockstep* accelerator has four different modules: (1) the FSM module that coordinates all the others submodules; (2) the *Timer* module is responsible to synchronize the cores; (3) the *Checker* module is in charge of data comparison between the cores; and (4) the APB3 interface modules which provides the communication between the cores and the *xLockstep* accelerator.

4.1.1 Finite State Machine Module

The FSM module is responsible to manage the other modules. To achieve that, it uses the FSM showed in Figure 4.6. The FSM has six states: *Start*, *Idle*, *Synchro*, *Checker*, *Resume*, and *Error*.

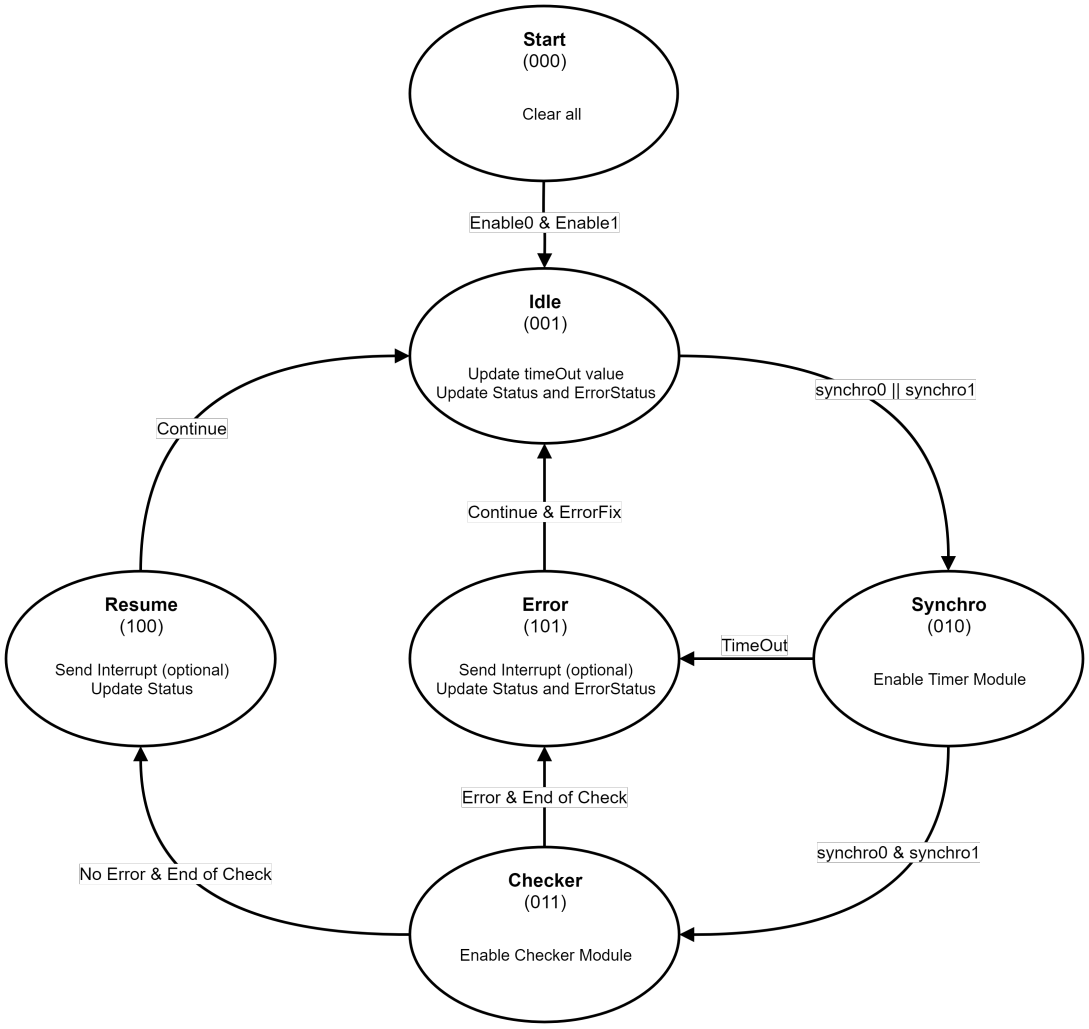


Figure 4.6: FSM to manage the *xLockstep* architecture.

On system-reset this module goes to the *Start* state. The *Start* state only changes to the *Idle* state, and to change to *Idle* it requires that the two cores set the *bit_Enable*. If one of the bits *bit_Enable* is clear, the *xLockstep* change to *Start* state.

When the *xLockstep* accelerator is in the *Idle* state, it updates the *TimeOut* value in the *Timer* module. Both status registers are updated, and it is continuously verifying the *Control* register. If any change occurs in the *Control* register, the *xLockstep* updates its configuration (for example, enable interrupt). If one of the two *bit_SynchroX* is activated, the state change to *Synchro* state, this happens when one core reaches a checkpoint and sets its *bit_SynchroX*.

In *Synchro* state, the *xLockstep* enables the *Timer* module to start a counter, and one of the next two situations can occurs: (1) the other core reaches the checkpoint and set the *bit_SynchroX* into the time

specified in the *TimeOut* value, and the state change to the *Checker* state, if no time out occurs; or (2) the counter in *Timer* module reaches the *TimeOut* value, which means that the other core did not reach the checkpoint into the specified time, and a *TimeOut* flag is set in the *Timer* module, leading to a state change to *Error* state.

During the *Checker* state, the system will enable the *Checker* module, that compares the output from both processors. If the outputs are not the same, the *Checker* module will set the error flag. At the end of the comparison the *Checker* will set the *endOfCheck* flag. After the comparison, if there were errors in the processor's output, the FSM state changes to *Error* state, otherwise it will change to *Resume* state.

If an error happens in the data comparison, or if one of the cores do not reach the checkpoint within the *TimeOut* value, the *xLockstep* changes to the *Error* state until both cores notify the *xLockstep* that the error was fix through the bit *bit_ErrorSolve* (*ErrorFix* flag is equal to the logic and between the two bits *bit_ErrorSolve*, one for each core) and that the synchronization is over through the bit *bit_SynchroX* (*Continue* flag is equal to the logic not-and between the two bits *bit_SynchroX*, one for each core). If *Continues* and *ErrorFix* flags are activated, the system changes to *Idle* state. During the *Error* state, the *xLockstep* updates the *Status* and *Error Status* registers and if the interrupt is enable, the system sets the interrupt flag.

In the *Resume* state, the system updates *Status* register and sets the interrupt flag in the case of the interrupt being enabled. The system stays in this state until both cores clear the two bits *bit_SynchroX* in order to set the flag *Continue*, if the flag *Continue* is enabled, the system changes to *Idle* state.

4.1.2 *Timer* Module

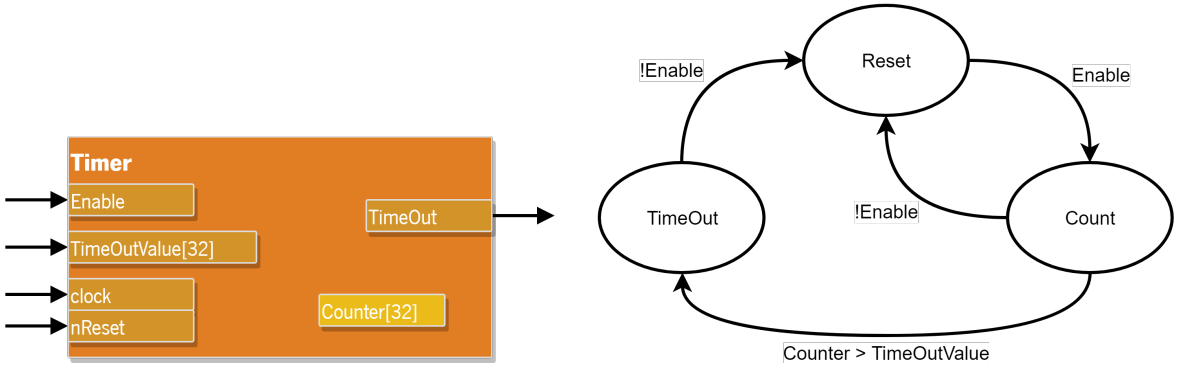
This module is responsible for synchronizing both cores, and in doing so, it includes a counter system. This mechanism is activated when one of the processors reaches a checkpoint. Then the *Timer* module is enable, and it starts the counter. If the counter reaches the *TimeOutValue* before the second processor reaching its checkpoint, the module outputs a timeout error, otherwise no errors were found, the timer is stopped, and the system follows its normal execution.

Figure 4.7 shows the architecture overview of the module in the subfigure 4.7a. It includes two inputs apart from the clock and *nReset* inputs. The *Enable* input starts the counter when is set, and the *TimeOutValue*, a input with 32-bit wide, is loaded by the first core that reaches the checkpoint, and it is provided by the main system from the *TimeOut* register. It includes a output *TimeOut*, that signalizes the occurrence of timeout situations. Also, it has a counter register to support the counter system. The subfigure 4.7b shows the FSM that describes the module behaviour, and it is composed by three states:

- *Reset* state - in this state the counter value is zero. If the enable bit is set, the module changes to the *Count* state. Also during this state the flag *TimeOut* is set to low;
- *Count* state - this state increments the counter at every clock cycle. The module stays in this state until the counter reach the *TimeOutValue* (and it changes the state to *TimeOut*) or when the enable

input is disabled (in this case, it changes to *Reset* state). During this state, the flag *TimeOut* does not change its value;

- *TimeOut* state - when this state is reached, the flag *TimeOut* is set to high, and only changes to *Reset* state when the enable input is disable.



(a) Architecture Overview **(b)** FSM

Figure 4.7: Architecture overview and FSM of the *Timer* module.

4.1.3 *Checker* Module

The *Checker* module compares the data from both processors, but it only compares four 32-bit registers at each execution. However, it is in this module that the system implements the error detection capabilities. If any mismatch between the data occurs, the module signalizes it.

Figure 4.8 shows the architecture overview and the FSM that controls the module. The architecture overview, in subfigure 4.8a, reveals the module inputs, apart from clock and *nReset*: (1) the *Enable*, that starts the comparison; (2) the *DataSelX* is responsible to inform the number of data registers to compare according with Table 4.1, and it is provided by the bits *bit_DataSel_X* from the *Control* register; and (3) the *Data_Core0* and the *Data_Core1* that provide data for comparison. Also this module includes three different outputs: (1) the *EndOfCheck*, which is the flag that informs the main system that the comparison is concluded; (2) the *ErrorDataX*, which informs the system in case of mismatch between data, it has four bits, one for each comparison, which is sent to the *ErrorStatus* registers; and (3) the *SelectData*, this output is responsible to select data from both processors.

The FSM in subfigure 4.8b describes the module behaviour, which starts in the *Idle* state. During this state, the outputs *EndOfCheck* and *SelectDataX* are set to zero when the the input *Enable* is activated the state changes to the *ErrorClean* state, clearing the output *ErrorDataX*. The module stays only one clock cycle in the *ErrorClean* state, and immediately changes to the *Compare* state. During this state, the first data to be compared are already available in the *Data_Core0* and *Data_Core1* inputs. The *Compare* state compares the input values, and changes the bit in *ErrorDataX* according to the compared data in case of an error. The next state is the *End* state if the module finish all the data comparison (defined by *EndOfData*

flag, this flag is set to high when the *DataSelX* is equal to *SelectData*) and it sets the *EndOfCheck* to high and it stays in this state until the system disable the *Checker* module, otherwise, the next state is the *ImportData* that increments the *SelectData* and imports the data from the *DataX* registers.

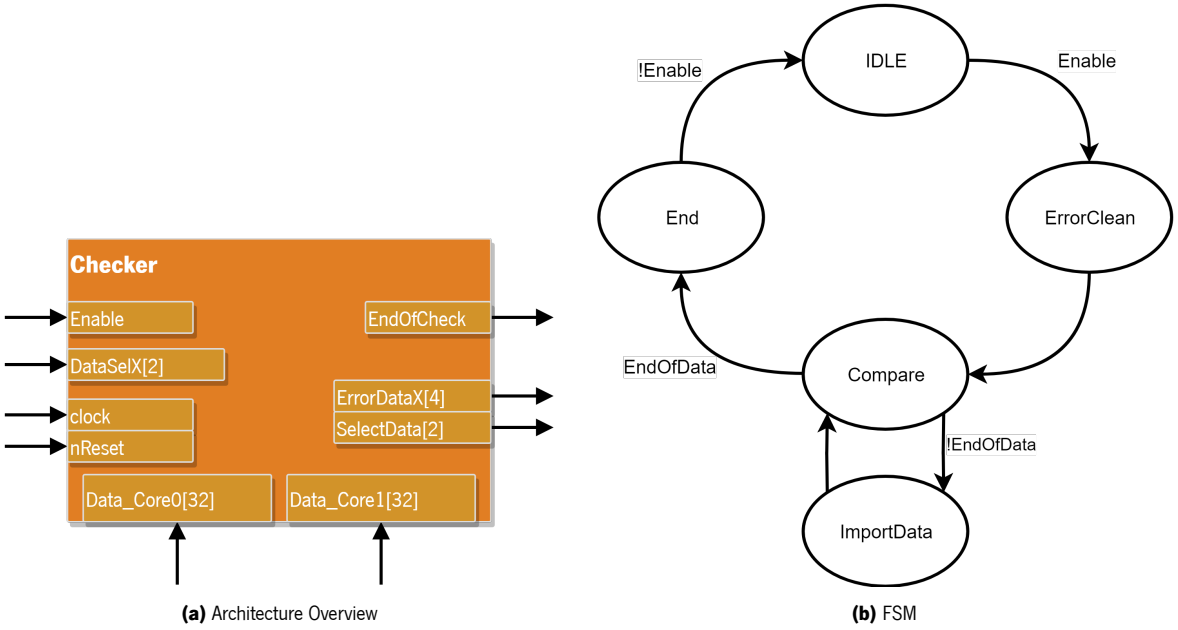


Figure 4.8: Architecture overview and FSM of the *Checker* module.

4.1.4 APB3 Interface Module

In the system there are two slave-APB3 interface modules, one for each core. Each module includes eight 32-bit registers to be aligned with the APB3 bus wide. This interface allows the communication between both cores and the *xLockstep* accelerator.

This module was developed based on the APB3 specification [98], that allows to create a memory mapped interface (Figure 4.9).

0x00	**Control Register**	Write-only
0x04	**Status Register**	Read-only
0x08	**TimeOut Register**	Write-only
0x0C	**Error Status Register**	Read-only
0x10	**Data0 Register**	Write-only
0x14	**Data1 Register**	Write-only
0x18	**Data2 Register**	Write-only
0x1C	**Data3 Register**	Write-only

Figure 4.9: *xLockstep* memory-mapped address.

The addresses are displayed in a offset system, because the absolute address varies with the bus connection.

4.2 Lock-V hardware

The DCLS mechanism is implemented and deployed under the Microsemi SmartFusion2, depicted in Figure 4.10, which corresponds to the SmartDesign from Microsemi Libero tool.

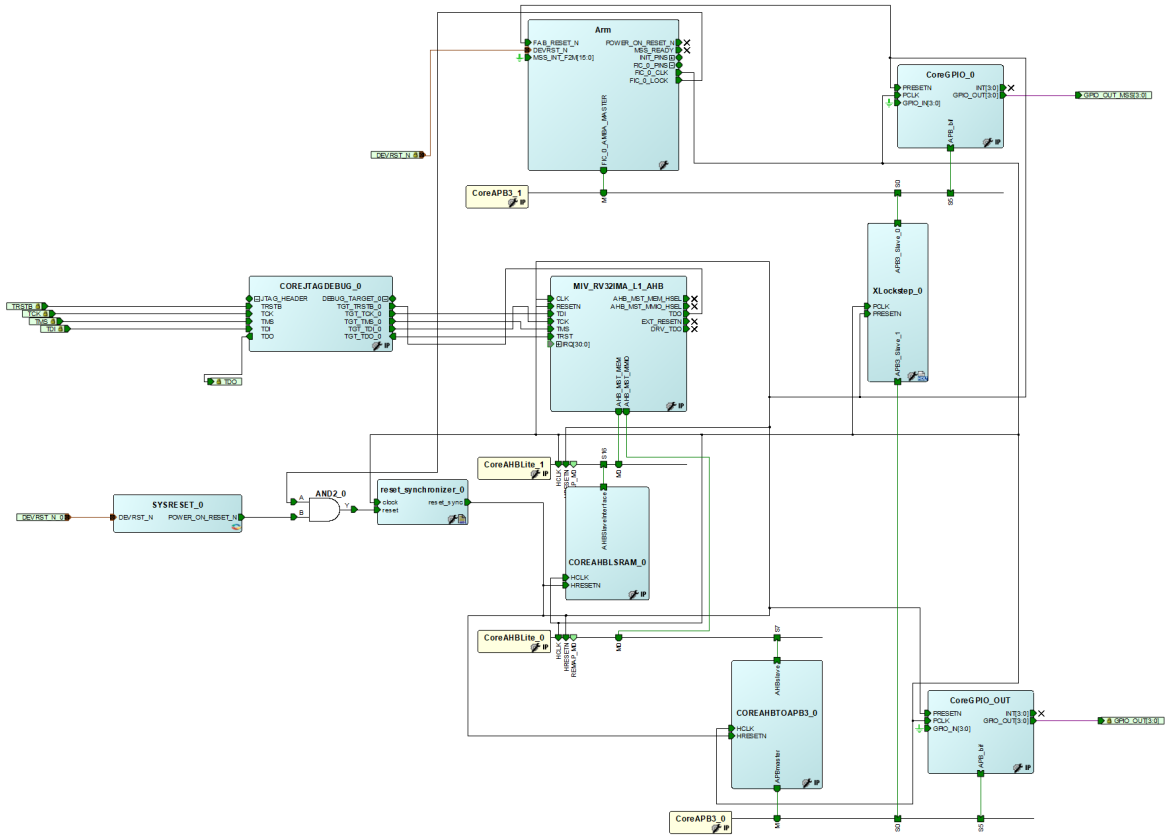


Figure 4.10: *xLockstep* design on Microsemi SmartFusion2.

The first step of the implementation was to add the Arm Cortex-M3 to the project, using the System Builder that allows to configure the Microcontroller Subsystem (MSS). In the MSS was activated the use of Direct Connection Mode, meaning that the Fabric Interface Controller (FIC) is exported out of the System Builder to generate the clock for the other peripherals. Due to time constraints grasped during the time analyses, the main clock exported from the MSS and shared with the other core and the peripherals was reduced to 25MHz. Also, it was necessary to add FIC_AMBA_MASTER to the System Builder in order to enabled the APB3 interface.

The second step was adding the Mi-V core to the SmartDesign which demands for extra cores to run, such as a LSRAM controller (COREAHBLSRAM in the SmartDesign) to get access to an embedded Large Static Random Access Memory (LSRAM) in the SmartFusion2. For this implementation, it is necessary

to add extra hardware modules in order to synchronize both processors with the same system reset, as detailed in [96]. Also it is necessary to add an APB3 to AHB bridge in order to connect the *xLockstep* accelerator and the core Mi-V. In order to add debug capabilities to the MI-V processor, a JTAG debug interface was added (COREJTAGDEBUG).

The third step was to add the *xLockstep* accelerator, and connect the APB3 interfaces with the respectively APB3 bus. Also in each APB3 bus is included a CoreGPIO that allows to output signals from both processors.

4.3 Lock-V API

In order to use the DCLS accelerator, an API was developed to allow the use of the *xLockstep* accelerator with different processors due to its agnostic characteristic. The *xLockstep* accelerator is a memory mapped peripheral that uses APB3 protocol to create the interface between both processors and the accelerator. The API simplifies the use of the memory mapped interface, while offering an abstraction to the user with the accelerator registers.

There is a library, that implements the API, *x_lockstep*, where are implemented the set of functions below, that allows to initialize the accelerator, start the synchronization between cores, compare data to achieve error detection, among other setting and runtime functions:

- *XLOCKSTEP_init()*;
- *XLOCKSTEP_synch()*;
- *XLOCKSTEP_checker()*;
- *XLOCKSTEP_resume()*;
- *XLOCKSTEP_errorFix()*;
- *XLOCKSTEP_config()*;
- *XLOCKSTEP_getStatus()*;
- *XLOCKSTEP_getErrorStatus()*.

These functions are divided by different modules described in sub sections bellow: the initialization (subsection 4.3.1), the synchronization (subsection 4.3.2), the comparison (subsection 4.3.3), the runtime duties (subsection 4.3.4), the configuration (subsection 4.3.5), and the status information (subsection 4.3.6). These set of modules described altogether the API to use the accelerator *xLockstep*.

4.3.1 Initialization

The initialization is responsible to enable the *xLockstep* accelerator by setting to high the *bit_Enable* in the *Control* register, see Figure 4.3.

The developed library includes the function *XLOCKSTEP_init()* to start the initialization. This function initialises one side of the *xLockstep* accelerator by set the *bit_Enable* bit. It is necessary to initialise both sides (one side for each processor) of the accelerator before starting the synchronization and comparison tasks. When both sides are enabled, the *xLockstep* will change the state from *Start* to *Idle*.

4.3.2 Synchronization

The synchronization is the process to synchronize both cores at the same checkpoint without comparing data. The synchronization only starts if the *xLockstep* are in *Idle* state, and both processors are synchronized if both have reached the checkpoint within the *TimeOutValue*. This process is only possible if both cores already initiate the *xLockstep* accelerator. If the initialization is achieved with success, the *xLockstep* changes to *Idle* state. The first core that reaches the checkpoint in the application will start the synchronization by setting the *bit_SynchroX* in the *Control* register, and the *xLockstep* will start immediately the *Timer* module by changing to *Synchro* state. If the other processor reaches the checkpoint within the time specified in *TimerOut* value, it will finish the synchronization by setting the bit *bit_SynchroX* and the state will pass to *Checker* without comparing data and changes to *Resume* state. But if the last core does not reach the checkpoint, the system will change to *Error* state with a timeout error, meaning that the processors are not synchronized.

The library *x_lockstep* provides the function *XLOCKSTEP_synch()*, which starts or finishes the synchronization by setting the bit *bit_SynchroX*. It is possible to configure the *timeOut* value for synchronization, but it must be done before the function *XLOCKSTEP_synch()* call, see subsection 4.3.5 to know more about *xLockstep* configuration.

4.3.3 Comparison

The comparison tasks is responsible to compare data between the cores in order to provide error detection capabilities to the *xLockstep*. This process requires that both cores are synchronized and in comparison to the synchronization, the former differs when the cores reaches the checkpoint, i.e, before start the synchronization, they need to load the data to be compared. Because of that, when the cores are synchronized, the *xLockstep* compares the data if any difference between data occurs, the comparison module changes to *Error* state with a data error. Then the state changes to *Resume* state when the system finish the comparison.

One constraint of this DCLS mechanism is the number of data to compare per execution. The system can only compare up to four data blocks with 32 bits. The number of data to compare is defined in the

two bits *bit_DataSel_X* before the core starts the synchronization. Table 4.1 shows the bits configuration according with the number of data. Beyond these two bits, the system uses the same bits described in synchronization to start and finish the synchronization process.

Despite the fact that ArmV7 architecture supports both little- and big-endian, in this work, it uses only little-endian as the RISC-V processor. Then during comparisons it is not necessary to convert the data, but if the data comparison occurs between two cores with different endianness, it is necessary to convert the output of one of the two cores when it sends the data.

In the library, there is the function *XLOCKSTEP_checker()* to apply the comparison which requires as argument the data and their size for performing the comparison.

4.3.4 Runtime duties

During the *xLockstep* execution, the application has some responsibilities to keep the integrity of the system. When the accelerator finishes the synchronization or the comparison processes, it changes to the *Resume* state if no error occurred. Afterwards, it is the accelerator's responsibility to finish all the process, by clearing the bit *bit_SynchroX* in each processor, allowing the *xLockstep* to change to *Idle* state. If an error occurs during the accelerator execution, it changes to *Error* state, and in order to enable the system to start a new synchronization or comparison, the application needs to inform the accelerator that the error was solved by setting the bit *bit_ErrorSolve*, as well clearing the bit *bit_SynchroX*. Both bits are loaded into the accelerator through the *Control* register. This will set the FSM to *Idle* state when the system changes to *Idle*, the application needs to clear the bit *bit_ErrorSolve*.

The library includes two functions to apply the runtime responsibilities, the function *XLOCKSTEP_resume()* and the *XLOCKSTEP_errorFix()*. The first function is used when no error happens and the *xLockstep* state is in *Resume*. The second function is used when a error occurs it informs the accelerator that the error was solved and it is possible to continue the normal execution. When the state changes to *Idle*, the application needs to call the function *XLOCKSTEP_resume()* in order to clear the *bit_ErrorSolve* bit.

4.3.5 Configuration

The accelerator has two different fields for configuration. It allows to update the *TimeOut* value through the *TimeOut* register and to enable or disable the interrupt interface through the bit *bit_EnInterrupt* in the *Control* register. This configuration only comes into effect at the *Idle* state, therefore, it is important to send the configuration before starting the synchronization. By default, the accelerator starts with the *TimeOut* value 0xFFFFFFFF, and with the interrupt interface disabled.

The library provides one function that allows to configure both fields, the function *XLOCKSTEP_config()*. It receives two arguments, the timeout value, a 32-bits unsigned variable, and a "boolean" variable based on a enumeration to enable or disable the interrupt interface.

4.3.6 Status information

The accelerator includes two read-only registers: the *Status* register that provides the accelerator state; and the *Error Status* register to get any error information. It is required to check the *Status* register before starting any synchronization or comparison. The interrupt interface, when enabled, allows the application to be interrupted only when the synchronization or comparison processes finishes. But to know if an error has occurred, it is necessary to check the *Status* register.

There are two functions in the library that returns the registers mentioned above. The function *XLOCKSTEP_getStatus()* that returns the *Status* register, and the function *XLOCKSTEP_getErrorStatus()* that returns the *Error Status* register.

4.4 Summary

The DCLS architecture deployed on the SmartFusion2 presents several advances over traditional DCLS mechanisms as it allows the development on an heterogeneous architecture with two different core processors in a agnostic fashion. To use other cores, they just need to be able to connect to the APB3 interface and to map the accelerator in memory. However, depending on the core, it may be necessary to modify the library to be coherent with the core architecture (for example, if the cores have different endianness in data structures).

The *xLockstep* accelerator was deployed in FPGA in order to provide the DCLS mechanisms such as error detection, however the recovery capabilities were not implemented in hardware, and such functionalities are supported in software by the framework descried in the next chapter.

5. Lock-V Framework

This chapter presents the framework to support the DCLS system developed and explained in chapter 4. Also, this framework provides to the system the recovery capability to achieve a complete fault tolerance system.

Section 5.1 presents the framework, describing its behaviour and the supported functionalities. In section 5.2 and section 5.3, it explains how initialize and synchronize the system. Section 5.4 and section 5.5 detail how the framework provides the recovery capabilities.

5.1 Framework Overview

The main goal of this dissertation is the development of a fault tolerance system, a DCLS with a heterogeneous architecture at core level. As mentioned in subsection 2.2.3, the lockstep technique uses the output from both processors for error detection, and due to the design diversity, it is only possible to apply the comparison between both processors in a loosely-coupled approach. This means that the system requires the use of checkpoints throughout the application code in order to synchronize and to compare the outputs from both processors. These outputs need to be previously selected because the design diversity does not allow instruction comparison.

Normally, the system recovery in DCLS is applied by using rollback or rollforward to an integrity state. In this work, the rollback was the chosen technique, and to use it, it is necessary to save states of integrity during the program execution. The Figure 5.1 shows the execution flow of an DCLS as a fault tolerance system with a rollback technique in a loosely-coupled comparison.

In this technique, both processors execute the same application, but only one processor outputs the results, in this case the core master. First, the system starts the synchronization between both processors. If the synchronization is successful, the system saves the processor contexts and runs the application code. When a checkpoint is reached on both sides, the system compares the output from both processors, and if any error is detected, the new processor contexts is saved and it continues the application code until a new checkpoint is reached. Figure 5.1 shows an example of an active fault that affects the application and corrupts the data in one of the processors. When the next checkpoint is reached, the comparison detects a data error and notifies both processors about the detected error, which triggers the rollback system in

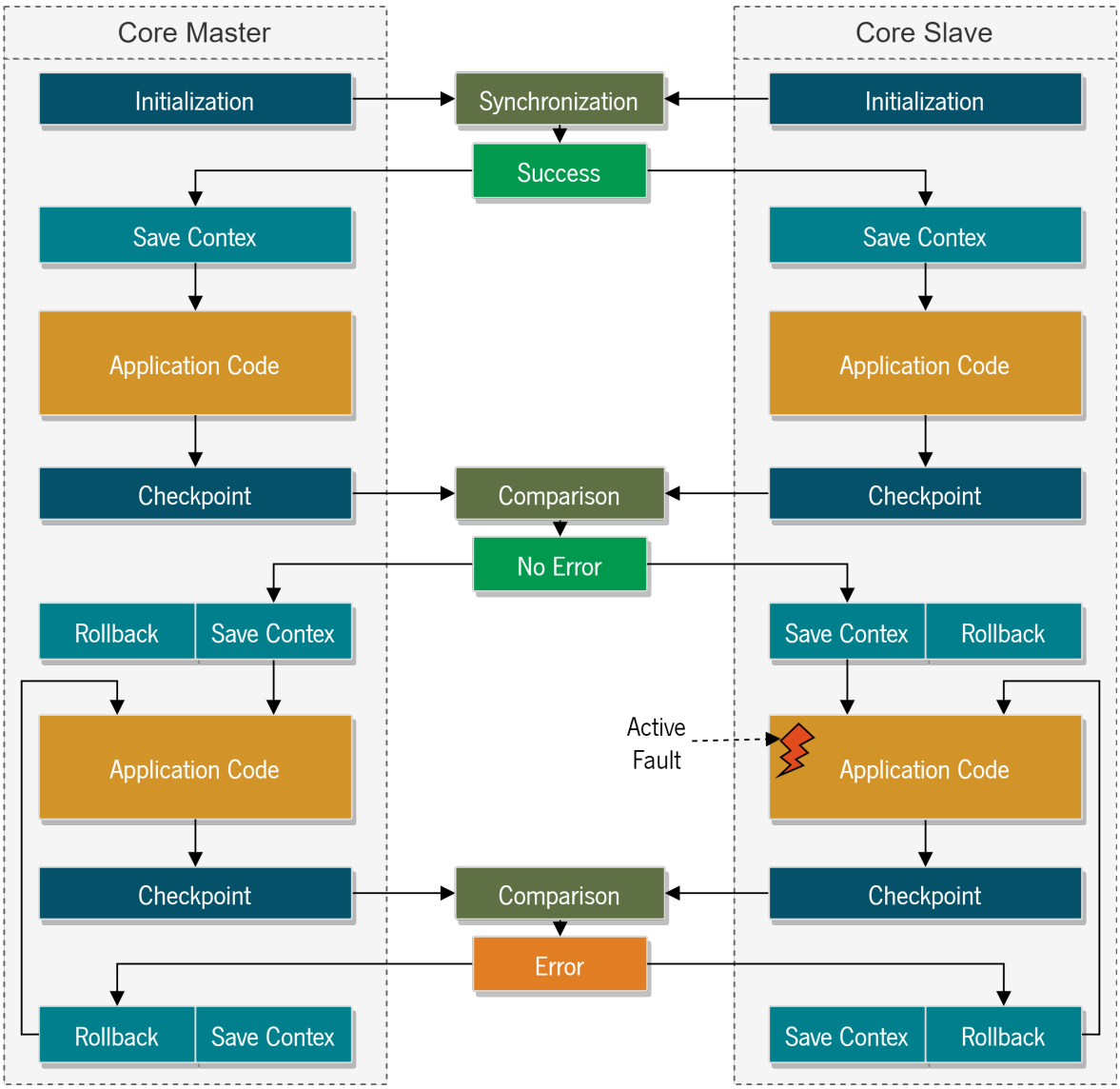


Figure 5.1: DCLS execution flow with rollback.

both processors, taking the execution to the previous checkpoint state where a valid context is saved and possible to be recovered.

In order to provide the error detection and the system recovery with a friendly interaction, this framework offers to the user a set of functions to use the Lock-V architecture, as described in chapter 4. The framework, using the library *x_lockstep*, provides five functions embedded in the library *xLockV* to achieve the fault tolerance in the system:

- *initLockV()* - this function initializes the *xLockstep* accelerator and synchronizes the cores;
- *checkpoint()* - this function synchronizes both processors, but in contrast with the *initLockV()*, it compares the selected data from the processors. Moreover, it returns a negative value in case of error, or zero if the comparison executed without errors;

- *saveContext()* - this function saves the processor context. This function is different from processor to processor due to the different used ISA. To save the processor context, the function uses assembly inline to copy the registers and the main stack. However, this function presents some constraints in its utilization, as explained by the save context process at the section 5.4;
- *rollback()* - this function, as well the function saveContext above, uses assembly inline, but the function *rollback()* resorts the saved data in the last call of the function *saveContext()* to restore the system in the last saved integrity state. Due to the use of assembly inline, the function also presents some constraints, that are detailed in section 5.5;
- *errorFix()* - This function is used to inform the *xLockstep* accelerator that the system fixed the error and it is reliable to continue the execution.

5.2 Initialization

The initialization purpose is to start the accelerator from both processors, and at the same time to synchronize the processors. This process consists of calling the function *initLockV()* at the application start. The first processor to execute this function will send the start signal to the *xLockstep* accelerator through this function *XLOCKSTEP_init()* and will wait until the other processor also executes the same process. Afterwards, the *xLockstep* changes the state from *Start* to *Idle*.

5.3 Checkpoint

The checkpoint functionality aims to synchronize both processors due to the loosely-coupled implementation, and it is also used to compare the data output between both processors, providing the error detection capability to the DCLS system. Figure 5.1 shows how the checkpoint system is used throughout system execution.

The system only compares four words for each comparing round (each word has 32-bits), as shown by the FSM behaviour in chapter 4. If it is necessary to compare a bigger amount of data outputs, the system will require a new synchronization after comparing each four words. The function *checkpoint()* abstracts the user from dealing with the required synchronization for more than four words during comparison.

This function returns the value 0, if no error occurs. However, if an error occurs during the process, the function returns a negative integer. It can return -1 or -2, for a data error or timeout error, respectively.

5.4 Save Context

This functionality is used for context-switches during the application execution. After a checkpoint being successfully reached, the application context is stored. Therefore, when the next checkpoint is

reached and there are errors detected in the comparing outputs, the application can return to a previous (valid) checkpoint and restore the previous processor's state. Figure 5.1 shows when the save context needs to be done. It is not mandatory to save the processor context after all the checkpoint, however, it is advisable to do it. However, it is crucial to save the processor context after a checkpoint without errors. During the development phase of an application with the Lock-V, the designer needs to know, that in case of a detected error, the system will restore to the last call of the save context.

This framework provides the function *saveContext()* which does not use any arguments, and does not return any value. It just saves to a protected memory area, the register file and the main stack. Due to its implementation with assembly inline, it needs to be called every time in the function *main()*, as will be explained later in this section.

5.4.1 Save Context Implementation

Due to the load-store type architecture of Arm Cortex-M3 and RISC-V (first the program loads the registers with data from memory, executes the operations with the registers, and after, if necessary, it stores the results to memory). Furthermore, since to the register file and memories are the most sensitives to SEU in computer systems [99], it becomes crucial to save the register context when a rollback is needed. Since memory fault tolerance is out of the scope of this dissertation, it is not considered as a fault sensitive component. There are various solutions that use ECC or parity protection that can be implemented to make the memory system more fault tolerance.

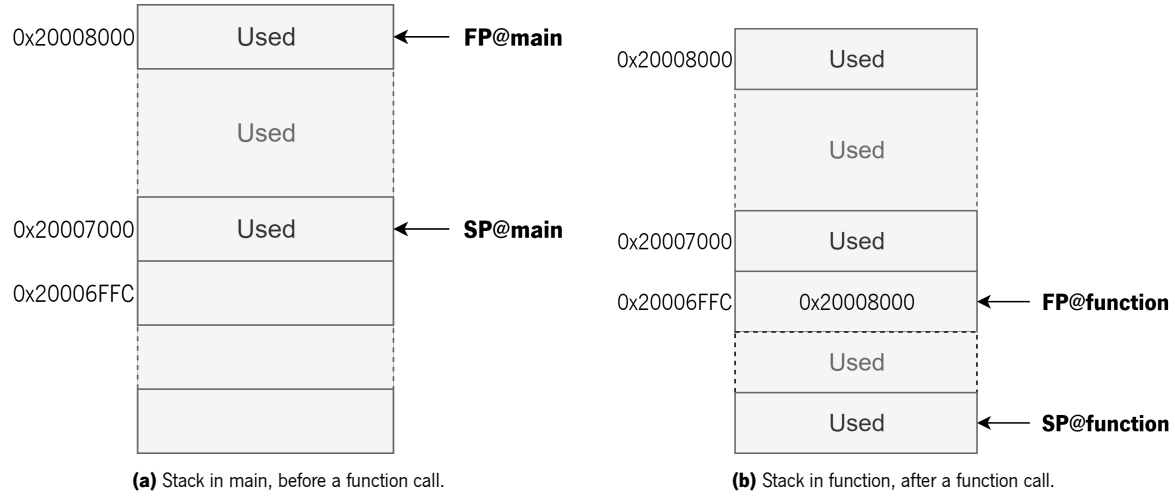
The function *saveContext()* when called, first saves the file registers, the main Frame Pointer (FP) and the main Stack Pointer (SP). After saving these registers, it saves the main stack, using the previous FP and SP. Finally, It saves the Program Counter (PC).

To save the register file and the main stack, it is necessary to know the register file and how the stack system works, before and after the call of function. The Arm Cortex-M3 includes 17 registers [100]:

- R0 to R12 - This registers are for general-purpose;
- R13 or SP;
- R14 or Link Register (LR);
- R15 or PC;
- and the Program Status Register (PSR).

The Arm Cortex-M3 uses the R7 as a FP. If the system is in the *main()* function, and calls another one with no arguments, the FP and the SP change, in order to provide a new stack. Figure 5.2 shows the stack, the SP, and the FP before and after a function call. Before the function call, the *main()* FP points to the address 0x20008000, and the *main()* SP points to the address 0x20007000, as shown in Figure 5.2a.

After the function call, the FP points to the address, 0x20006FFC which stores the address of the FP in the *main()*, 0x20008000, and the address of the four bytes above is the old address of the *main()* SP.



(a) Stack in main, before a function call. **(b)** Stack in function, after a function call.

Figure 5.2: Stack in Arm Cortex-M3 before and after a function call without arguments nested functions. The address are merely illustrative.

These considerations are important, because the *saveContext()* saves all the register file, and also the stack. And the reasons to be called every time this function in the *main()* are: (1) to avoid saving long stack by constraining it only to the *main()* stack; (2) the complexity and the execution time of the called function only increases if the system needs to save the *main()* stack and nested functions stacks; and (3) if one function calls other functions, a different strategy should be drafted to save the *main()* FP.

The RISC-V core includes 32 registers [86]:

- X0 - hard-wire zero;
- X1 - return address;
- X2 - SP;
- X3 to X27 - multi purposed;
- X28 to X31 - temporary registers.

It is important to take some considerations about the architectures and their Application Binary Interface (ABI) during the development of this function and the rollback function described in subsection 5.5:

- The stack alignment proposed in RISC-V with XLEN equal to 32 is 8 bytes. If the XLEN is equal to 64, the alignment will be 16 bytes. In the core RV32IMA, it uses XLEN equal to 32 [101]. However, the compiler supported by Microsemi does not follow the proposed alignment, and it uses a 4 bytes alignment saving the stack and the FP during a function call;

- As this framework saves the system stack, to restore it, in case of error, it is important to consider the stack type, which are both descending. But the Arm Cortex-M3 has a full descending stack, in contrast to Mi-V RV32IM that has an empty descending stack.
- Although, the core Mi-V RV32IMA has a different ISA, the way the system save the FP and the SP is similar, but it requires some adaptations to develop the *saveContext()* function.

The *saveContext()* does not receive any argument and does not return any value. It needs to operate in different ways from processor to processor due to the process of saving the processor context. In order to solve that, the library *xLockV* includes a define statement to target the architecture to compile the application.

5.5 Rollback

The rollback aims to restore the system to the previous valid state, saved by the saveContex() function. This gives to the Lock-V system a strong fault tolerance capability. This framework includes a *rollback()* function without arguments to apply the rollback over the saved state. As a constraint, this function requires that the *saveContext()* function has been called at least once before a rollback call.

The subsection 5.4 details some specifications to take into considerations during the development of the *saveContext()* and the *rollback()* functions. This function is only inline assembly, and it cannot have any nested function to keep the rollback as simple (and faster) as possible.

The rollback process consists in restoring all the stack in the *main()*, after, restoring the register file with the saved register file and finally, updating the PC. This way, at the end of this last update, the system changes to the state exactly after the end of the *saveContext()*.

5.5.1 ErrorFix Function

The *xLockstep* accelerator needs to receive the information that the error was fixed. However, due to the characteristic mentioned before, the *rollback()* function cannot have any nested function call. Then, the function *errorFix()* needs to be called after or before the *rollback()* execution in order to change the state of the *xLockstep* accelerator from Error to *Idle* state.

5.6 Summary

The distance between the checkpoints in the application will define the time to error detection in the system. If the system aims for critical or safety purposes, the distance should be shorter as possible. However, shortening such distance, the application may loose its real-time capabilities. This trade-off must be carefully taken into good consideration.

As well, the number of data during the comparison needs to be taken into account during the development phase. More comparing data between checkpoints causes the system to spend more time in the error detection task.

Although the Lock-V hardware has an agnostic interface to any processor that allows the connection with APB3, this framework needs to be adapted to the processor architecture because to achieve the system recovery, it is necessary to use the functions *saveContext()* and *rollback()* written through inline assembly. Therefore, saving register file as well as the stack, is ISA-dependent.

6. Evaluation and Results

This chapter presents the analysis performed under the Lock-V in order to test, evaluate, and validate the fault tolerance system.

The first section presents the resources needed by the Lock-V architecture in the FPGA. Also, it presents the memory and execution footprint caused by the use of the Lock-V in different applications.

The second section, as an answer to the third objective of this dissertation, explores the fault injection system applied to a case study that uses the Lock-V architecture. The section finishes with the results achieved by tests carried out on the case study.

All the tests are executed in the Lock-V architecture and the results that needed the application compilation, were obtained without optimizations. The proposed DCLS system was deployed on a Microsemi SmartFUsion2 with an Arm Cortex-M3 and a RISC-V-based processors are independently connected to the *xLockstep* accelerator. The lock-V system runs at a frequency of 25MHz, which is applied to all modules: Arm-CortexM3, RISC-V-based soft-core, and *xLockstep* accelerator. Moreover, during the test, all the results obtained, are using the Lock-V framework.

6.1 Lock-V Implementation Analysis

This section reports the cost by using the Lock-V coupled to a normal application compared to not using the Lock-V mechanism. First, in subsection 6.1.1, it is presented the FPGA resource usage in the Lock-V hardware. Subsection 6.1.2 covers the amount of memory that an application uses with and without the Lock-V. Finally, subsection 6.1.3, also compares the execution footprint by comparing different applications with and without Lock-V.

6.1.1 Fabric FPGA Resources Utilization

Table 6.1, shows the resource usage in the *xLockstep* accelerator, and how much resources are needed by each submodule. Moreover, it includes the hardware resources required by the RISC-V soft-core and its dependencies.

The report that returns the resources needed distinguishes the resources between the fabric and the interface elements, in both, the report includes the 4-Inputs Look-Up Table (4-LUT) and the D-Type Flip-Flop

(DFF) usage. Libero uses DFF as the sequential elements representation and the 4-LUT as a combination element. The difference from the DFF and 4-LUT in the interface to the fabric is that the elements in the interface use the Random Access Memorys (RAMs) and the math blocks, respectively. Table 6.1 only shows the sum of the 4-LUT and the sum of the DFF from both, interface and fabric. This simplification is due to the system which only uses interface elements in the RISC-V processor and its memory.

Table 6.1: Resource utilization of the *xLockstep* and the entire system described in section 4.2.

Module		4-LUT	DFF
xLockstep	- APB3 Interface Slave	158	224
	- APB3 Interface Master	158	224
	- Checker	129	12
	- Timer	81	35
	Total	**588**	**544**
Mi-V RV32IMA		10511	5555
LSRAM Controller		2026	1219
Reset Synchronizer		0	4
Bus Connections		369	194
Total		**13494**	**7516**
	Percentage (100%)	15.66%	8.72%

Lock-V requires a total of 13494 4-LUTs and 7516 DFFs, which represent around 15.66% and 8.72% of the available 4-LUT and DFF elements respectively for the SmartFusion2 M2S090TS System-on-Chip (SoC).

As it is possible to see in the Table 6.1, the *xLockstep* only uses 588 4-LUTs and 544 DFFs, that corresponds to 0.68% and 0.63% of the available 4-LUT and DFF elements in the chosen platform, respectively. Also, this resource utilization corresponds to 4.36% 4-LUTs and 7.24% DFFs compared to the Lock-V implementation. The *xLockstep* accelerator includes two APB3 interfaces that uses 158 4-LUTs and 224 DFFs each one. The module *Checker* requires 129 4-LUTs and 12 DFFs, and the module *Timer* 81 4-LUTs and 35 DFFs. To combine all the modules inside the accelerator, the *xLockstep* needs 62 4-LUTs and 49 DFFs.

The RISC-V soft-core uses a total of 10511 4-LUTs and 5555 DFFs, which corresponds respectively to 77.89% and 73.91% of the system Lock-V, that it is a large part of the system. However, the soft-core only represents 12.20% of 4-LUTs and 6.45% of DFFs in the entire fabric FPGA. To complete the system, it was required to use the AHB and APB3 buses, the LSRAM Controller and the Reset Synchronizer, they use 2395 4-LUTs and 1417 DFFs, these modules use in the Lock-V system a total of 17.75% 4-LUT and 18.85% DFF.

These results show that the *xLockstep* accelerator uses only a small fraction of the Lock-V system, less than 10% of the resource usage. And the soft-core uses more than 70% of the entire system resources.

6.1.2 Memory Footprint

In order to evaluate the memory footprint of the Lock-V framework, and to know its impact in the system, two cases were considered with the same application:

- Without the Lock-V framework;
- With the Lock-V framework.

The application with the Lock-V framework is a simple code with a loop, it includes a *initLockV()* call at the application beginning and two *saveContext()* calls, one after the initialization, and the other inside the loop. Also, it includes a *rollback()* and a *checkpoint()* call inside the loop.

The results are obtained from the two processors, Tables 6.2 and 6.3, in the Arm side by invoking the compiler with GNU ARM Cross Print Size, and in the RISC-V side by invoking the compiler with GNU RISC-V Cross Print Size. Both results are in bytes and they include the size of: (1) the code segment (.text); (2) the global or static variables segment (.data); and the uninitialized data segment (.bss).

Table 6.2: Arm memory footprint (values in bytes).

Arm side	.text	.data	.bss	total
Application without Lock-V	4656	16	59072	63744
Application with Lock-V	6384	80	60864	67328
Lock-V Overhead	1728	64	1792	3584

Table 6.3: RISC-V memory footprint (values in bytes).

RISC-V side	.text	.data	.bss	total
Application without Lock-V	3824	128	8196	12148
Application with Lock-V	6624	256	8864	15744
Lock-V Overhead	2800	128	668	3596

The different memory overhead in the global or static variables segment between the two compilations, it is due to the fact that the Arm side has an alignment of 4 bytes in the stack, and the RISC-V side has an alignment of 8 bytes.

Although the compilation results have very different values from each other. To include the framework in the application there is only an increase of 3600 bytes approximately. More specifically, an increase of 3584 bytes through the compiler for the Arm core, and an increase of 3596 bytes for the RISC-V core.

6.1.3 Execution footprint

In order to evaluate the execution performance of the Lock-V, it was performed a set of micro benchmarks. The obtained results represent the system overhead, in terms of clock cycles, caused by inserting

the Lock-V framework. However, the results are only provided by the Arm Cortex-M3, because the instruction *RDCYCLEH* [1], which returns the clock cycles in the core RISC-V-based is out of date (the register to read the clock cycles changes from the Microsemi implementation to the compiler implementation). To measure the latency between executions it was used the available *SysTick* registers in the Arm Cortex-M3.

6.1.3.1 System recovery execution footprint

The first measurements, Table 6.4, presents the latency by calling individually the *saveContext()* and the *rollback()* functions from the framework. The results for each function were obtained with the *main()* stack size of 4x4 bytes.

Table 6.4: *saveContext()* and *rollback()* execution footprint (the *main()* stack size was 4x4 bytes).

Arm side	Latency (clock cycles)	Time (at 25 MHz)
saveContext() function	335	13.4 μs
rollback() function	248	9.92 μs

The *saveContext()* function executes in 335 clock cycles, which is equivalent to 13.4 μs at 25 MHz frequency clock. And the *rollback()* function executes in 248 clock cycles that is 9.9μs at 25 MHz frequency clock. The extra clock cycles in the *saveContext()* is due to the register reposition before it returns the function call, such as the FP and the SP, while the *rollback()* function copies only all the previous saved registers.

6.1.3.2 Error detection execution footprint

Tables 6.5 and 6.6 display the microbenchmarks performed over the checkpoint functionalities, under different conditions. Although the results were extracted from the hardcore side, due to the DCLS synchronization dependency during the checkpoint phase, the time execution between both processors remains the same. The measurements to the checkpoint were performed by sending different numbers of data to compare between the processors, from 0 to 100 blocks of 32 bits. These experiments were performed under three different conditions: (1) without errors; (2) with an error in the first element to compare; and (3) in the last element to compare.

Table 6.5 shows the results from the measurements for the case without errors and with error in the first comparing element. If the checkpoint does not sends data to compare, and only uses the functionality to synchronize both processors in the same point, the execution footprint is 1119 clock cycles in the hardcore Arm, this latency corresponds to 44.75 μs at 25 MHz. In this case, there is no example with error because there is no data to compare. When the checkpoint compares only one data with 32 bits, in the case without error the checkpoint spends 1459 clock cycles. If the data is wrong, the checkpoint spends less 815 clock cycles, that corresponds to an overhead reduction. This difference is due to the checkpoint return from

[1] *RDCYCLEH* is a pseudo-instruction specifically for the ISAs RV32, for the ISAs RV64 it is used the *RDCYCLE*

the function, as soon as the error is detected, without synchronization. If the checkpoints compare ten elements, the latency is 2154 clock cycles without errors, and in case of error in the first element the latency is 2134 clock cycles, which corresponds to a decrease in latency of 20 clock cycles. In this situation, the overhead reduction is much smaller compared to the first case with one element to compare, due to the synchronization after sending the first four data (section 5.3 explains this limitation). When the system sends more 10 elements in the checkpoint to compare, it increases, on average, by 1767 clock cycles, whether or not there is any error in the first element. The first element in the checkpoint is always the last element to compare in the *xLockstep* accelerator, and due to that, when the system compares more than four elements, and when errors are detected, the overhead increases, on average, by 22.60 clock cycles.

Table 6.5: Checkpoint execution footprint without errors and with error in first element to compare.

Number of Data (32 bits each)	**Without Errors**		**With Error (in 1st element)**		**Overhead** (*)	
	Clock cycles	Time at 25 MHz (μs)	Clock cycles	Time at 25 MHz (μs)	Clock cycles	Time at 25 MHz (μs)
0	1119	44.75	-	-	-	-
1	1459	58.35	644	25.76	↓ 815	32.59
10	2154	86.16	2134	85.36	↓ 20	0.80
20	3675	147.00	3661	146.44	↓ 14	0.56
30	5757	230.28	5735	229.40	↓ 22	0.88
40	7272	290.88	7256	290.24	↓ 16	0.64
50	9360	374.40	9328	373.12	↓ 32	1.28
60	10869	434.76	10843	433.72	↓ 26	1.04
70	12949	517.96	12925	517.00	↓ 24	0.96
80	14474	578.96	14464	578.56	↓ 10	0.40
90	16556	662.24	16512	660.48	↓ 44	1.76
100	18059	722.36	18041	721.64	↓ 18	0.72
Avg. Increment (10 in 10 elements)	1767.22	70.69	1767.44	70.70	-	-
Avg. Overhead (10 to 100 elements)	-	-	-	-	↓ 22.60	0.90

(*) The overhead is the difference between the case without error and the case with error.
Overhead symbols: ↑ - Increase of overhead; ↓ - Decrease of overhead.

Table 6.6 shows the results for the cases without errors and with error in the last element to compare through the checkpoint. The obtained results when there are no errors in the processor's output, are the same as presented on Table 6.5. Additionally, when the checkpoint compares only one element, the results are equal to the mention before, since the last element is at the same time the first element during the comparison process. Moreover, in case of error in the last element, when the data to compare is bigger or equal to four elements, the system spends only on average 771.90 clock cycles, that corresponds to 30.88 μs. If the checkpoint uses four or more elements to compare, in case of error in the last element, the clock cycles spent is approximately the same, mainly, due to the the *xLockstep* accelerator characteristic

that always compares four elements each time. The overhead from no error to error in the last element reduces approximately in 1767 clock cycles when ten more elements are added to the checkpoint array.

Table 6.6: Checkpoint execution footprint without errors and with error in last element to compare.

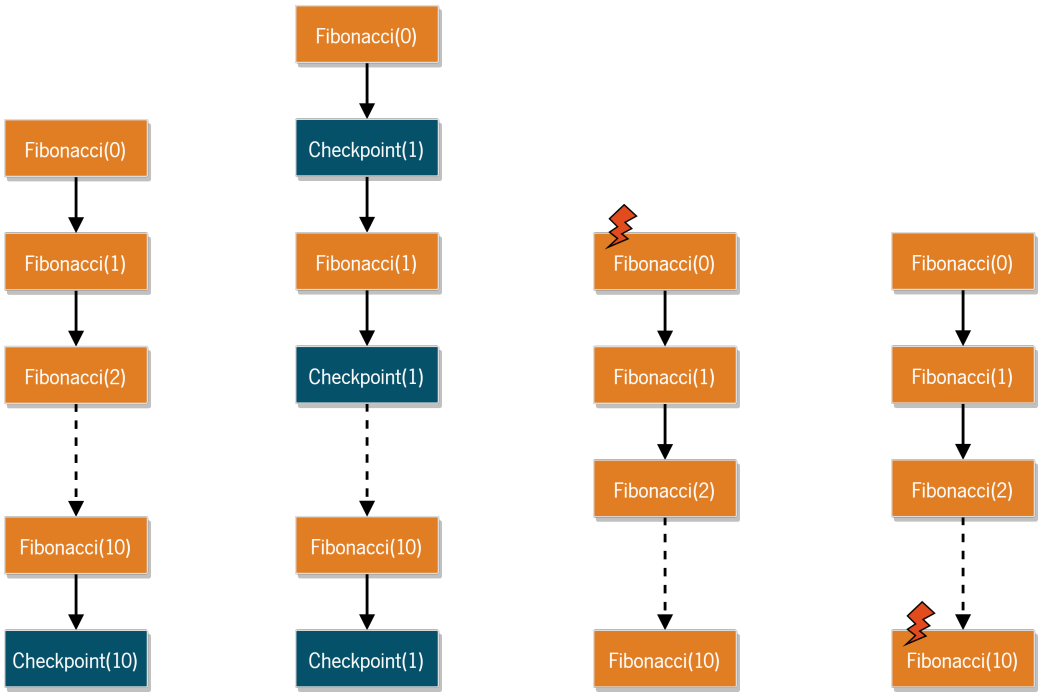
Number of Data (32 bits) each)	**Without Errors**		**With Error (in last element)**			
					Overhead (*)	
	Clock cycles	Time at 25MHz (μs)	Clock cycles	Time at 25MHz (μs)	Clock cycles	Time at 25MHz (μs)
0	1119	44.75	-	-	-	-
1	1459	58.35	644	25.76	↓ 815	32.59
10	2154	86.16	771	30.84	↓ 1383	55.32
20	3675	147.00	773	30.92	↓ 2902	116.08
30	5757	230.28	771	30.84	↓ 4986	199.44
40	7272	290.88	771	30.84	↓ 6501	260.04
50	9360	374.40	773	30.92	↓ 8587	343.48
60	10869	434.76	776	31.04	↓ 10093	403.72
70	12949	517.96	769	30.76	↓ 12180	487.2
80	14474	578.96	771	30.84	↓ 13703	548.12
90	16556	662.24	771	30.84	↓ 15785	631.4
100	18059	722.36	773	30.92	↓ 17286	691.44
Avg. Increment (10 in 10 elements)	1767.22	70.69	0.22	0.01	1767.00	70.68
Avg. Overhead (10 to 100 elements)	-	-	771.90	30.88	-	-

(*) The overhead is the difference between the case without error and the case with error.
Overhead symbols: ↑ - Increase of overhead; ↓ - Decrease of overhead.

These results show that, in case of error, the worst case for the error detection capability occurs when the faulty data is allocated in the first element of the array to compare. And the best scenario occurs when the faulty data is allocated in the last element of the array to compare.

Table 6.7 represents the execution footprint with and without errors using a function to calculate the *Fibonacci* sequence. The *Fibonacci* function scalability allows to understand the impact of using the Lock-V with one or more checkpoints during the function execution. These measurements show the difference between using more or less checkpoints, and to analyse the difference, the tests were made in different conditions, and for each condition, the tests were performed by calculating the first 10, 15, or 20 *Fibonacci* sequence values. The conditions include the function execution without errors, with error in the first element, as shown in the figure 6.1c, or in the last element, as shown in the figure 6.1d, from the *Fibonacci* sequence. Moreover, in each condition, the Lock-V was tested with one or *N* checkpoints, where *N* corresponds to the number of elements that the *Fibonacci* function calculates. For example, when the *Fibonacci* function calculates the first ten elements, if the system includes only one checkpoint (figure 6.1a), this one will be reached after the function calculates the ten elements, and the checkpoint sends ten elements to compare. But if the system includes *N* checkpoints, in this case ten checkpoints (figure

6.1b), the checkpoint will be reached after each calculated element, and the checkpoint only sends one element to compare.



(a) Fibonacci function with one checkpoint. **(b)** Fibonacci function with *N* checkpoints. **(c)** Fibonacci function with with error in the first element. **(d)** Fibonacci function with with error in the last element.

Figure 6.1: Fibonacci sequence calculation with one (a) or *N* (b) checkpoints. And *Fibonacci* sequence with error in the first (c) or in the last (d) element.

In order to compare and to analyse the overhead by using the Lock-V mechanism, it was also measured the *Fibonacci* function without the Lock-V integration. To calculate the first ten elements of *Fibonacci* sequence, the system without Lock-V spends 26785 clock cycles. For the first fifteen elements, the system uses 299132 clock cycles, and to calculate the first twenty elements the system spends 3323420 clock cycles.

When the system includes the Lock-V mechanism, in the case that no error happens and there is only one checkpoint as it is demonstrated in figure 6.1a, the system uses 27253 clock cycles, 299789 clock cycles, and 3324077 clock cycles to calculate the first 10, 15, and 20 *Fibonacci* elements respectively. That corresponds to an increase in the overhead of 1.75% for 10 elements, 0.22% for 15 elements, and 0.02% for 20 elements. In the case when the system uses *N* checkpoints, figure 6.1b, the system increases the overhead from 1.75% to 29.65% for 10 elements, also it increases from 0.22% to 3.84% for 15 elements, and it increases from 0.02% to 0.44% for 20 elements in the *Fibonacci* sequence calculation.

If the sequence has an error in the first element of the *Fibonacci* sequence, for the case when the system only includes an checkpoint, the overhead increase is 1.68%, 0.21%, and 0.02% for 10, 15, and 20 elements, respectively. The overhead is smaller compared to the case without error, because when the system detects the error it discards the synchronization and instantly returns to apply the recovery if possible. When the system includes *N* checkpoints and the error is in the first element, the first checkpoint

will detect the error. In this case, the Lock-V detects the error faster than the previous case. Although this is a specific case of the system, the results show an decrease of overhead compared to the case without Lock-V of 94.58%, 99.51%, and 99.96% for 10, 15, and 20 elements calculation. It is also necessary to take into account that the system when detects the error, it does not finish the *Fibonacci* calculation successfully, and it will require to start the execution from the last saved state if the system has the recovery enabled.

In the case with one checkpoint and the error is in the last element from *Fibonacci* sequence, the overhead is smaller than the case with an error in the first element, the reason for this small reduction is the error detection during the comparison, the last element in the checkpoint is the first element to be compared. The results show an increase of overhead of 1.59%, 0.21%, and 0.02% for 10, 15, and 20 elements. When the system includes *N* checkpoints and the error is also in the last element, the results are quite similar to the case with *N* checkpoints without errors. There is an overhead increase of 30.10%, 3.87%, and 0.45% for 10, 15, and 20 elements calculation respectively.

Table 6.7: Lock-V execution footprint with and without errors, using *Fibonacci* function.

	Fibonacci(10)		**Fibonacci(15)**		**Fibonacci(20)**	
	Clock Cycles	Overhead (%)	Clock Cycles	Overhead (%)	Clock Cycles	Overhead (%)
			Without Lock-V			
	26785	-	299132	-	3323420	-
			With Lock-V			
No Errors						
1 checkpoint	27253	↑ 1.75%	299789	↑ 0.22%	3324077	↑ 0.02%
N checkpoints (*)	34727	↑ 29.65%	310628	↑ 3.84%	3338172	↑ 0.44%
Error in 1st element						
1 checkpoint	27236	↑ 1.68%	299766	↑ 0.21%	3324062	↑ 0.02%
N checkpoints (*)	1451	↓ 94.58%	1451	↓ 99.51%	1451	↓ 99.96%
Error in last element						
1 checkpoint	27212	↑ 1.59%	299772	↑ 0.21%	3324060	↑ 0.02%
N checkpoints (*)	34846	↑ 30.10%	310708	↑ 3.87%	3338431	↑ 0.45%

(*) The number **N** of checkpoints is equal to the argument of the function Fibonacci(**N**)
Overhead symbols: ↑ - Increase of overhead; ↓ - Decrease of overhead.

In case of error, the best scenario occurs with *N* checkpoints and the error is in the first element. The worst case occurs when the system has *N* checkpoints and the error is in the last element. It is possible to see, due to the *Fibonacci* scalability, that the impact of the Lock-V reduces with the increase of the application execution time. Also, the impact of using 1 or *N* checkpoints become more similar to each other with the increase of the application execution time.

6.2 Fault Injection Case Study

For this case study, a new design was developed to validate the Lock-V system as a fault tolerance one. Figure 6.2 shows the new design that includes a peripheral with UART features to monitor the system.

Also, this new design includes the fault injection mechanism.

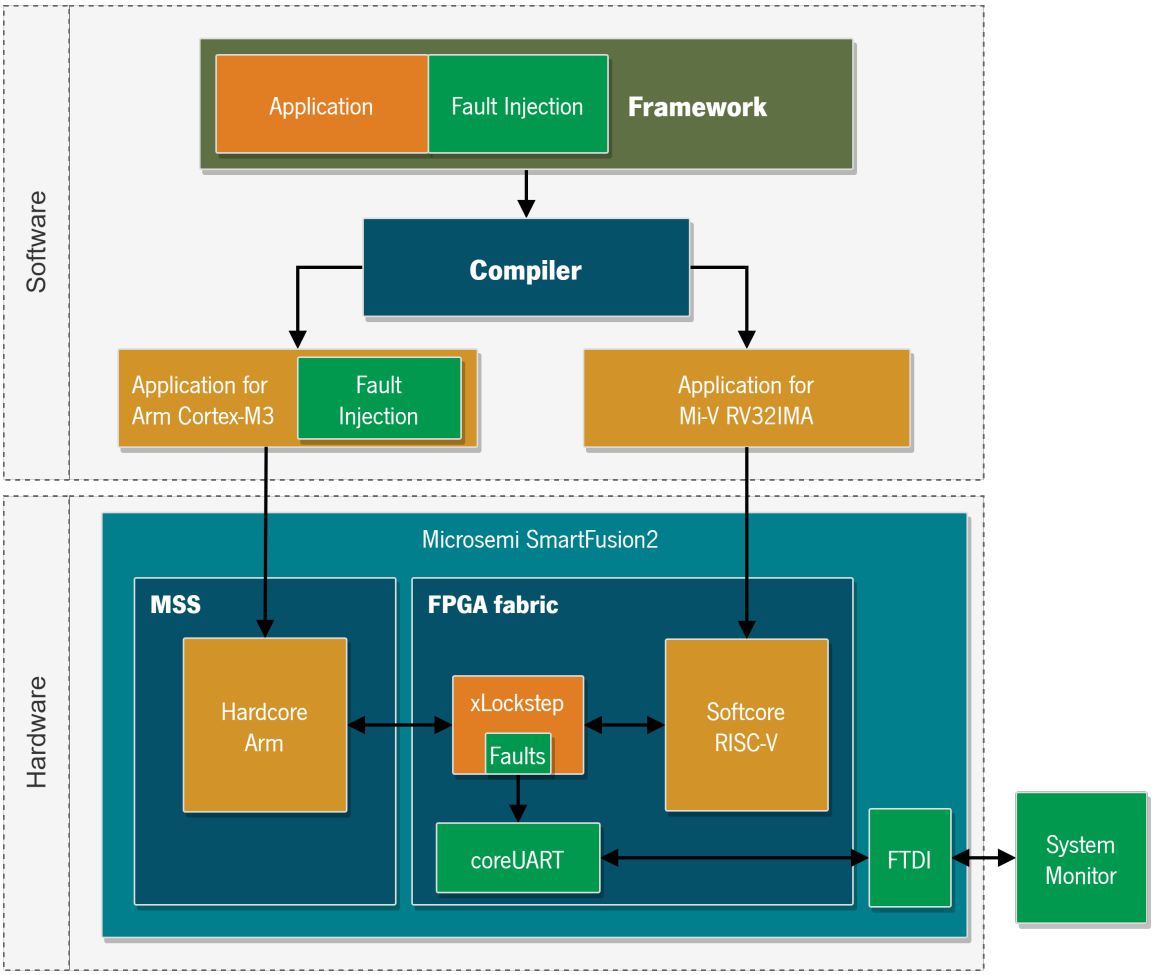


Figure 6.2: Lock-V architecture with fault injection mechanism and monitor system.

The fault injection mechanism is only implemented in the Arm Cortex-M3 through a random timer interruption. At every interruption, the mechanism provokes a random bit-flip in one of the Arm bank registers, also randomly, in order to provoke a fault in the processor. Moreover, to the monitor system knows the number of faults that had already been provoked, it was added to the *xLockstep* a new register in the APB3 interface to count the number of faults, this register is updated at each interruption.

The *xLockstep* suffered some adaptations which allow monitoring the system. It was added to the the fabric FPGA a *coreUART* peripheral that receives data from the *xLockstep* accelerator and sends it through the onboard serial port adapter.

Every time an error occurs and the *xLockstep* detects it, the *coreUART* sends to the system monitor the number of faults and the error type. If the error occurs due to a timeout, the system will need a reset. But, if the error is a data mismatch, both cores will execute the rollback function, which sends the execution flow to a previous valid state. If the core RISC-V-based executes the application correctly, and if an error occurs in the Cortex-M3 is expected that the accelerator recognizes it and reports to the system monitor.

To know if the system continues the execution correctly, after both cores send the *errorFix()*, the coreUART informs the system monitor that the execution continues correctly.

6.2.1 Results

During this test, it was injected a total of 98957 faults in the bank register of the hardcore Arm. These faults originated 93 errors, where 85 are data errors and the 8 are *timeOut* errors. The data error occurs when the data from the two cores is different. The *timeOut* error occurs when one of the cores does not reach the checkpoint.

Table 6.8 shows the fault injection results with and without the Lock-V DCLS The system with and without Lock-V had 8 *timeOut* errors, and due to the Lock-V inability to recover from *timeOut* errors, its error correction is 0% for this type of error. In the case of data errors, the Lock-V detects 85 errors, without Lock-V the system is not able to correct the errors, but with Lock-V the system was able to correct 83 (out of 85), which give a success rate of 97.65% error correction for data errors.

Table 6.8: Fault injection results with and without Lock-V.

	Faults	**Errors**		**Error Correction (%)**
		Without Lock-V	With Lock-V	
TimeOut error	-	8	8	0.00%
Data error	-	85	2	97.65%
Total	**98957**	**93**	**10**	**89.25%**

This results proved the high fault coverage in the error detection mechanism. However, it only presents a high rate of error correction against data errors. To increase the recovery and error capabilities, it is necessary to implement an extra type of rollback for the cases when the error is a *timeOut* one.

6.3 Summary

Due to the small resource utilization of the *xLockstep* accelerator, shown in subsection 6.1.1, it is possible to implement the Lock-V in a largest number of FPSoC platform.

In case of data error, the error detection capability achieves the best performance when the checkpoint sends the faulty data in the last element of the array to compare, and the worst case occurs when the faulty data goes in the first element of the checkpoint array.

The results obtained in the case study show that the Lock-V mechanism has a good fault tolerance capability against data errors, however, when the system suffers a *timeOut* error, the Lock-V cannot recover from it.

7. Conclusion

The research and development for technology with power consumption efficiency while keeping good performance ratios does not stop. This can be achieved with reduced transistor's size, higher clock frequencies, and lower operating core voltages. Consequently, the systems present new dependability lacks. To answer this issue, this dissertation presents a solution for fault tolerance with focus on low-end devices.

The major contribution of this work is the new fault tolerance system that uses a lockstep technique with design diversity. Whereas the typical lockstep approach provides a strong fault tolerance technique, they present a weakness against CMF. Then, this dissertation concept, adds to lockstep technique an effective technique against CMF, the design diversity at ISA level. So far, to the best of the author's knowledge, there are no similar implementations beyond the mention before.

The solution uses DCLS in a heterogeneous architecture that includes two different processor architectures, a hard-core Arm Cortex-M3 and a soft-core RISC-V, the MI-V RV32IMA. Also it includes extra hardware developed in FPGA, the *xLockstep* accelerator, which supports the lockstep methodology and applies the comparison between both processors in a loosely-coupled fashion. The Lock-V uses APB3 interfaces to connect both processors with the *xLockstep* accelerator. All the hardware combined provides synchronization and error detection capabilities to the system.

Moreover, this solution provides a framework that combines the hardware and software to achieve error detection and system recovery capabilities in the Lock-V system, that are the keys capabilities to achieve fault tolerance. Also, this framework provides a simple and easy use of the Lock-V architecture.

This solution is agnostic enough to provide a fault tolerance approach in different FPSoC platforms, however, due to architectural incompatibilities, minor changes always need to be performed for other processor architectures.

To increase the effectiveness of the error detection, first the user should select the critical parts in the application, for example, before generating an output. And second, use inside the critical parts the checkpoint function with more intensity, for example, compares the outputs.

The recovery capability in the Lock-V provides a good performance against data errors, with 97.65% of error correction. The system can detect and recover from error states, except when one of the processors does not reach the checkpoint.

Depending on the distance between checkpoints and the number of data to compare, the time to execute the application varies. The more data the application sends to compare, the more time the

system uses in fault tolerance duties. Also, the less is the distance between checkpoints, the higher is the percentage of execution time in fault tolerance tasks compared to the entire execution time. Using the Lock-V involves a trade-off during the conversion from the normal application to the fault tolerance one. The checkpoints distance and the amount of data between comparisons need to be defined according to the application purpose. The efficiency of the Lock-V system was verified and validated by using fault injection over the register file. Also, it was analysed the extra latency by adding the Lock-V to a system. This latency varies according to the use of the checkpoint during the application.

Thus, it is concluded, that the proposed Lock-V system, composed by the DCLS architecture with design diversity, and the Lock-V framework, is a fault tolerance system that offers a high rate of error detection and protection against SEU and CMF, in other words, the systems presents a high fault coverage.

7.1 Future Work

This dissertation presented the research and development work in regards to the combination of two techniques, the lockstep technique, a DCLS methodology, and the design diversity at core level. Although this dissertation presenting, there are some future implementations that can be pointed out:

- Adapt the system to other bus widths. The system only works with APB3 with a 32-bits bus width. To perform another bus width it is necessary to change how the registers work in the APB interface, also the checker and the timer modules uses 32 bits registers to compare the timer counter and the data respectively.

- Increase the support for more processor architectures and versions. The xLockstep accelerator presents an agnostic feature, however, the framework will need modifications to use the save context and rollback features in other core architectures.

- Change the system to multi-core (more than two cores). This adaptation is more complex, the checker module will need a full reconfiguration to compare all data. It is possible to upgrade the system and to add a voting module, this allows the fault tolerance system to choose the best way to follow in case of error.

  Taking into consideration the agnosticism of the system developed for lockstep, the adaptation of the system for a TMR is a good step for higher performance and a better availability in the system. Also, according with Pierce, TMR is the best choice for fault tolerance systems if the cost and performance are the most important criteria [2].

- The core RISC-V-based used in this dissertation, the MIV_RV32IMA_L1_AHB, provides ECC functionalities that include SEC and DED techniques. It is possible to use this feature in the instruction or data cache, and in case of error inside, an event signalizes the user that an error has been corrected or detected. If this functionality is added to the system, it will be more effective in fault

tolerance capabilities. However, it requires adaptation, to in case of error correction, do not apply the rollback, and in case of error detection, apply specific processes to correct the error.

- The case study shows that the Lock-V does not present a good fault tolerance capability against *timeOut* errors. The next step to improve the Lock-V is to implement a rollback solution for this type of error.

- Implement adaptative checkpoints, which can be enabled/disabled in run-time, according to the error rate that is being achieved during the execution. This way, when the system is performing in a scenario that is more prone to faults, checkpoints that were initially disabled can be activated to improve the error detection.

7.2 Publications

During the development of this work, this dissertation has contribute for two publications:

- **I. Marques**, C. Rodrigues, S. Pinto, T. Gomes, and A. Tavares, "Arquitetura Heterogénea para Sistemas Tolerantes a Falhas Baseada em Arm e RISC-V," *in XV Jornadas sobre Sistemas Reconfiguráveis – REC'2019*, (Guimarães, Portugal), Feb. 2019

- C. Rodrigues, **I. Marques**, S. Pinto, T. Gomes, and A. Tavares, "Towards a Heterogeneous Fault-Tolerance Architecture based on Arm and RISC-V Processors," *in IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, vol. 1, pp. 3112–3117, Oct. 2019.

References

[1] A. Aviziens, "Fault-Tolerant Systems," *IEEE Transactions on Computers*, vol. C-25, pp. 1304–1312, Dec 1976.

[2] W. H. Pierce, "Failure -Tolerant Computer Design," p. 247, 1965.

[3] E. Normand, "Single event upset at ground level," *IEEE Transactions on Nuclear Science*, vol. 43, pp. 2742–2750, Dec 1996.

[4] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 305–316, Sep. 2005.

[5] I. Hwang, S. Kim, Y. Kim, and C. E. Seah, "A Survey of Fault Detection, Isolation, and Reconfiguration Methods," *IEEE Transactions on Control Systems Technology*, vol. 18, pp. 636–653, May 2010.

[6] F. Abate, L. Sterpone, C. A. Lisboa, L. Carro, and M. Violante, "New Techniques for Improving the Performance of the Lockstep Architecture for SEEs Mitigation in FPGA Embedded Processors," *IEEE Transactions on Nuclear Science*, vol. 56, pp. 1992–2000, Aug 2009.

[7] A. B. de Oliveira, G. S. Rodrigues, and F. L. Kastensmidt, "Analyzing Lockstep Dual-Core ARM Cortex-A9 Soft Error Mitigation in FreeRTOS Applications," in *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design: Chip on the Sands*, SBCCI '17, (New York, NY, USA), p. 84–89, Association for Computing Machinery, 2017.

[8] E. Ozer, B. Venu, X. Iturbe, S. Das, S. Lyberis, J. Biggs, P. Harrod, and J. Penton, "Error Correlation Prediction in Lockstep Processors for Safety-Critical Systems," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 737–748, Oct 2018.

[9] J. Han, Y. Kwon, Y. C. P. Cho, and H. Yoo, "A 1GHz fault tolerant processor with dynamic lockstep and self-recovering cache for ADAS SoC complying with ISO26262 in automotive electronics," in *2017 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pp. 313–316, Nov 2017.

[10] A. Avizienis, J.-C. Laprie, B. Randell, *et al.*, "Fundamental concepts of dependability," 2001.

[11] V. P. Nelson, "Fault-tolerant computing: fundamental concepts," *Computer*, vol. 23, pp. 19–25, July 1990.

[12] Z. Gao, C. Cecati, and S. X. Ding, "A Survey of Fault Diagnosis and Fault-Tolerant Techniques—Part I: Fault Diagnosis With Model-Based and Signal-Based Approaches," *IEEE Transactions on Industrial*

Electronics, vol. 62, pp. 3757–3767, June 2015.

[13] J.-C. Laprie, "Dependable computing and fault-tolerance," *Digest of Papers FTCS-15*, pp. 2–11, 1985.

[14] A. Avizienis and J. . Laprie, "Dependable computing: From concepts to design diversity," *Proceedings of the IEEE*, vol. 74, pp. 629–638, May 1986.

[15] J.-C. Laprie, *Dependability: Basic concepts and terminology*, vol. 5. Springer, 1992.

[16] B. Parhami, "Defect, Fault, Error,..., or Failure?," *IEEE Transactions on Reliability*, vol. 46, pp. 450–451, Dec 1997.

[17] E. Dubrova, *Fault-Tolerant Design*. New York, NY: Springer, 2013. OCLC: 843460688.

[18] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pp. 25–36, June 2000.

[19] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan 2004.

[20] A. Avižienis, J.-C. Laprie, and B. Randell, "Dependability and its threats: a taxonomy," in *Building the Information Society*, vol. 156, pp. 91–120, Boston, MA: Springer US, 2004.

[21] E. Normand, "Single-event effects in avionics," *IEEE Transactions on Nuclear Science*, vol. 43, pp. 461–474, April 1996.

[22] M. Yang, G. Hua, Y. Feng, and J. Gong, *Fault-Tolerance Techniques for Spacecraft Control Computers*. John Wiley & Sons, Jan. 2017.

[23] A. Geraci, F. Katki, L. McMonegal, B. Meyer, J. Lane, P. Wilson, J. Radatz, M. Yee, H. Porteous, and F. Springsteel, *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, 1991.

[24] P. A. Lee and T. Anderson, *Fault tolerance*. Springer, 1990.

[25] A. K. Somani and N. H. Vaidya, "Understanding Fault Tolerance and Reliability," *Computer*, vol. 30, pp. 45–50, apr 1997.

[26] J. Yiu, "Design of SoC for High Reliability Systems with Embedded Processors," in *Embedded World Conference*, 2015.

[27] J. Gray and D. P. Siewiorek, "High-availability computer systems," *Computer*, vol. 24, pp. 39–48, Sep. 1991.

[28] J. Bowen and V. Stavridou, "Safety-critical systems, formal methods and standards," *Software Engineering Journal*, vol. 8, no. 4, pp. 189–209, 1993.

[29] M. Al-Kuwaiti, N. Kyriakopoulos, and S. Hussein, "Network Dependability, Fault-tolerance, Reliability,

Security, Survivability: A Framework for Comparative Analysis," in *2006 International Conference on Computer Engineering and Systems*, pp. 282–287, Nov 2006.

[30] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares, "IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices," *IEEE Internet Computing*, vol. 21, pp. 40–47, Jan 2017.

[31] F. Afonso, C. Silva, N. Brito, S. Montenegro, and A. Tavares, "Aspect-Oriented Fault Tolerance for Real-Time Embedded Systems," in *Proceedings of the 2008 AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS '08, (New York, NY, USA), Association for Computing Machinery, 2008.

[32] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," in *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pp. 84–91, June 1999.

[33] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, (Edinburgh, UK), pp. 297–306, IEEE, June 2007.

[34] J. S. Klecka, W. F. Bruckert, and R. L. Jardine, "Error self-checking and recovery using lock-step processor pair architecture," May 21 2002. US Patent 6,393,582.

[35] T. Kottke and A. Steininger, "A Reconfigurable Generic Dual-Core Architecture," in *International Conference on Dependable Systems and Networks (DSN'06)*, pp. 45–54, June 2006.

[36] . B. de Oliveira, L. A. Tambara, and F. L. Kastensmidt, "Applying lockstep in dual-core ARM Cortex-A9 to mitigate radiation-induced soft errors," in *2017 IEEE 8th Latin American Symposium on Circuits Systems (LASCAS)*, pp. 1–4, Feb 2017.

[37] R. D. Yearout, P. Reddy, and D. L. Grosh, "Standby redundancy in reliability - a review," *IEEE Transactions on Reliability*, vol. 35, pp. 285–292, Aug 1986.

[38] A. Ejlali, B. M. Al-Hashimi, M. T. Schmitz, P. Rosinger, and S. G. Miremadi, "Combined time and information redundancy for seu-tolerance in energy-efficient real-time systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 323–335, April 2006.

[39] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, pp. 147–160, April 1950.

[40] G. Edward, "Error-detecting and correcting system," Aug. 30 1960. US Patent 2,951,229.

[41] W. F. Heida, "Towards a Fault Tolerant RISC-V Softcore," Master's thesis, Delft University of Technology, Aug. 2016.

[42] B. Littlewood, "The impact of diversity upon common mode failures," *Reliability Engineering & System Safety*, vol. 51, no. 1, pp. 101–113, 1996.

[43] S. Mitra, N. R. Saxena, and E. J. McCluskey, "Common-mode failures in redundant VLSI systems: a survey," *IEEE Transactions on Reliability*, vol. 49, pp. 285–295, Sep. 2000.

[44] I. M. Jacobs, "The Common Mode Failure Study Discipline," *IEEE Transactions on Nuclear Science*, vol. 17, pp. 594–598, Feb 1970.

[45] F. C. Afonso, *Operating system fault tolerance support for real-time embedded applications*. PhD thesis, Universidade do Minho, 2009.

[46] J.-C. Laprie, "From dependability to resilience," in *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, pp. G8–G9, 2008.

[47] R. D. Kral, J. S. M. Chong, and A. L. Schreiber, "Implementation of a Loosely-Coupled Lockstep Approach in the Xilinx Zynq-7000 All Programmable SoC? for High Consequence Applications.," 2017.

[48] N. Vaishnavadevi, S. Ramkumar, and R. Ganesan, "Efficacious redundancy technique for enriched lockstep architecture," *International Journal of Advanced Information Science and Technology (JAIST)*, vol. 11, no. 11, 2013.

[49] J. R. Azambuja, F. Kastensmidt, and J. Becker, *Hybrid Fault Tolerance Techniques to Detect Transient Faults in Embedded Processors*. Springer, 2014.

[50] M. Berg and C. Michael, "FPG Mitigation Strategies for Critical Applications, support of NASA/GSFC," Sept. 2018.

[51] E. Kyriakakis, "Implementation of a Distributed Fault-Tolerant NoC-based Architecture for the Single-Event Upset Detector," Master's thesis, KTH, School of Information and Communication Technology (ICT), 2017.

[52] H. Pham, S. Pillement, and S. J. Piestrak, "Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor," *IEEE Transactions on Computers*, vol. 62, pp. 1179–1192, June 2013.

[53] M. Pignol, "DMT and DT2: two fault-tolerant architectures developed by CNES for COTS-based spacecraft supercomputers," in *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, pp. 10 pp.–, July 2006.

[54] . B. de Oliveira, G. S. Rodrigues, F. L. Kastensmidt, N. Added, E. L. A. Macchione, V. A. P. Aguiar, N. H. Medina, and M. A. G. Silveira, "Lockstep Dual-Core ARM A9: Implementation and Resilience Analysis Under Heavy Ion-Induced Soft Errors," *IEEE Transactions on Nuclear Science*, vol. 65, pp. 1783–1790, Aug 2018.

[55] K. D. Safford, C. L. Lyles, and E. R. Delano, "Core-level processor lockstepping," Oct. 30 2007. US Patent 7,290,169.

[56] K. D. Safford, "Method and apparatus for recovery from loss of lock step," May 6 2008. US Patent 7,370,232.

[57] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pp. 246–249, June 2016.

[58] I. Marques, C. Rodrigues, S. Pinto, T. Gomes, and A. Tavares, "Arquitetura Heterogénea para Sistemas Tolerantes a Falhas Baseada em Arm e RISC-V," in *XV Jornadas sobre Sistemas Reconfiguráveis – REC'2019*, (Guimarães, Portugal), Feb. 2019.

[59] C. Rodrigues, I. Marques, S. Pinto, T. Gomes, and A. Tavares, "Towards a Heterogeneous Fault-Tolerance Architecture based on Arm and RISC-V Processors," in *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, vol. 1, pp. 3112–3117, Oct. 2019.

[60] A. A. Verhage, "A fault tolerant memory architecture for a RISC-V softcore," Master's thesis, Delft University of Technology, 2016.

[61] M. Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, pp. 395–401, July 1970.

[62] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pp. 81–92, Dec 2003.

[63] T. Gomes, F. Salgado, A. Tavares, and J. Cabral, "CUTE Mote, A Customizable and Trustable End-Device for the Internet of Things," *IEEE Sensors Journal*, vol. 17, pp. 6816–6824, Oct 2017.

[64] S. Ainsworth and T. M. Jones, "Parallel Error Detection Using Heterogeneous Cores," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 338–349, June 2018.

[65] Infineon, "Highly Integrated and Performance Optimized 32-Bit Microcontrollers for Automotive and Industrial Applications," *Infineon Technologies AG*, Mar. 2016.

[66] Infineon, "AURIX™ 32-bit microcontrollers for automotive and industrial applications," *Infineon Technologies AG*, Aug. 2019.

[67] Arm, "Arm® Cortex®-R5 and Cortex®-R5F," *Arm Limited, Tech. Rep.*, Feb. 2011.

[68] Arm, "Arm® Cortex®-R8 MPCore Processor Technical Reference Manual," *Arm Limited, Tech. Rep.*, Feb. 2019.

[69] Arm, "Arm® Cortex®-R7 MPCore Technical Reference Manual," *Arm Limited, Tech. Rep.*, Nov. 2014.

[70] Arm, "Arm® Cortex®-A76AE Core Technical Reference Manual," *Arm Limited, Tech. Rep.*, Dec. 2018.

[71] Arm, "Arm® Cortex®-R52 Processor Technical Reference Manual," *Arm Limited, Tech. Rep.*, Feb. 2018.

[72] Arm, "Arm® Cortex®-M33 Dual Core Lockstep," *Arm Limited, Tech. Rep.*, June 2017.

[73] Y. Sun, P.-f. Wu, J. Li, and Z.-f. Ma, "Research on Dual-Core Lock Step Mechanism and Its Application for Commercial High Performance APSoC," *Advances in Astronautics Science and Technology*, pp. 1–5, 2019.

[74] S. Chaudhry and M. Tremblay, "Providing fault-tolerance by comparing addresses and data from redundant processors running in lock-step," Mar. 1 2005. US Patent 6,862,693.

[75] J. Gomez-Cornejo, A. Zuloaga, U. Kretzschmar, U. Bidarte, and J. Jimenez, "Fast context reloading lockstep approach for SEUs mitigation in a FPGA soft core processor," in *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2261–2266, Nov 2013.

[76] M. Delvai, "SPEAR handbook," *Vienna University of Technology, Embeded Computing Systems Group, Vienna, Austria, Technical Report*, vol. 70, p. 2002, 2002.

[77] P. Garcia, T. Gomes, F. Salgado, J. Cabral, P. Cardoso, M. Ekpanyapong, and A. Tavares, "A Fault Tolerant Design Methodology for a FPGA-based Softcore Processor," *IFAC Proceedings Volumes*, vol. 45, no. 4, pp. 145–150, 2012. 1st IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control.

[78] T. Miyazaki, "Reconfigurable systems: a survey," in *Proceedings of 1998 Asia and South Pacific Design Automation Conference*, pp. 447–452, Feb 1998.

[79] Â. Ribeiro, C. Rodrigues, I. Marques, J. Monteiro, J. Cabral, and T. Gomes, "Deploying a Real-Time Operating System on a Reconfigurable Internet of Things End-device," in *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, vol. 1, pp. 2946–2951, Oct. 2019.

[80] T. Gomes, F. Salgado, S. Pinto, J. Cabral, and A. Tavares, "A 6LoWPAN Accelerator for Internet of Things Endpoint Devices," *IEEE Internet of Things Journal*, vol. 5, pp. 371–377, Feb. 2018.

[81] F. Salgado, T. Gomes, J. Cabral, J. Monteiro, and A. Tavares, "DBTOR: A Dynamic Binary Translation Architecture for Modern Embedded Systems," in *2019 IEEE International Conference on Industrial Technology (ICIT)*, pp. 1755–1760, Feb. 2019.

[82] C. Microsemi, "SmartFusion2 SoC FPGA Overview," Aug. 2018.

[83] C. Microsemi, "Mi-V RISC-V Ecosystem." https://www.microsemi.com/product-directory/fpga-soc/5210-mi-v-embedded-ecosystem, jan-2020.

[84] C. Microsemi, "M2S090TS - Microsemi SmartFusion2 Security Evaluation Kit." https://www.microsemi.com/existing-parts/parts/143988, 2019.

[85] D. Kanter, "RISC-V offers simple, modular ISA," *The Linley Group MICROPROCESSOR report*, 2016.

[86] A. Waterman, K. Asanovic, and C. Division, "The RISC-V Instruction Set Manual," p. 145, May 2017.

[87] S. Pinto and J. Martins, "The Industry-First Secure IoT Stack for RISC-V: A Research Project," in *RISC-V Workshop*, (Zurich), June 2019.

[88] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-T: A RISC-V Processor with Light Weight Security Extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, HASP '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[89] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, P. Chiu, R. Avizienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, and K. Asanovic, "An Agile Approach to Building RISC-V Microprocessors," *IEEE Micro*, vol. 36, pp. 8–20, Mar 2016.

[90] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[91] C. Celio, D. A. Patterson, and K. Asanovic, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015.

[92] lowRISC, "lowRisc Project." https://github.com/lowRISC/lowrisc-chip, Jan. 2020.

[93] P. platform, "Pulpino: An Open-Source Microcontroller System Based on Risc-v." https://github.com/pulp-platform/pulpino, Jan. 2020.

[94] C. Wolf, "PicoRV32 - A Size-Optimized RISC-V CPU." https://github.com/cliffordwolf/picorv32, Jan. 2020.

[95] C. Microsemi, "MiV_RV32IMAF_L1_AHB V2.0." Nov. 2017.

[96] C. Microsemi, "MIV_RV32IMA_L1_AHB v2.1." July 2018.

[97] C. Microsemi, "MIV_RV32IMA_L1_AXI V2.0." Mar. 2018.

[98] Arm, "AMBA 3 APB Protocol Specification," *Arm Limited, Tech. Rep.*, Aug. 2004.

[99] F. M. Lins, L. A. Tambara, F. L. Kastensmidt, and P. Rech, "Register File Criticality and Compiler Optimization Effects on Embedded Microprocessor Reliability," *IEEE Transactions on Nuclear Science*, vol. 64, pp. 2179–2187, Aug 2017.

[100] Arm, "Cortex-M3 Technical Reference Manual," *Arm Limited, Tech. Rep.*, Dec. 2005.

[101] K. Asanovic, “Proposal for a RISC-V Embedded ABI (EABI).” https://github.com/riscv/riscv-eabi-spec/blob/master/EABI.adoc, Oct. 2019.