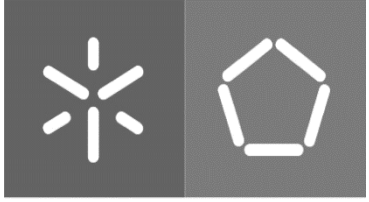
Universidade do Minho
Escola de Engenharia

Cristiano António Azevedo Rodrigues

Heterogeneous Fault Tolerance Architecture based on Arm and RISC-V Processors

outubro de 2019



Universidade do Minho
Escola de Engenharia

Cristiano António Azevedo Rodrigues

Heterogeneous Fault Tolerance Architecture based on Arm and RISC-V Processors

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
Professor Doutor Sandro Emanuel Salgado Pinto

outubro de 2019

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Atribuição-NãoComercial-CompartilhaIgual
CC BY-NC-SA

https://creativecommons.org/licenses/by-nc-sa/4.0/

Agradecimentos

Em primeiro lugar, gostaria de expressar a minha mais profunda gratidão pelo o apoio e contínuo suporte que o meu orientador e professor, doutor Adriano Tavares, me providenciou ao longo deste árduo último ano. Todo o conhecimento passado e pensamento crítico por ele incutido, vieram-se a demonstrar fulcrais ao longo de toda a dissertação. Ainda no âmbito de orientação académica, gostaria também de agradecer toda a ajuda, apoio e críticas construtivas providenciadas pelos professores doutores Sandro Pinto e Tiago Gomes assim como pelo aspirante a doutor, José Martins.

Não poderia deixar de agradecer aos meus colegas de laboratório, Afonso Santos, André Alves e Ricardo Moreira assim como aos meus amigos mais chegados, que foram parte integrante desta minha longa caminhada, tendo eles passando comigo todos os melhores "altos" e dando-me suporte nos meus piores "baixos". Um especial obrigado a todos os membros do "bando": Daniel Barbosa, José Silva, José Pedro, Ivo Marques e Valter Mário.

Por fim, endereço um especial agradecimento a toda a minha família. Família essa que fez e faz com que tudo isto tenha um sentido e propósito. Por toda a crença, apoio e suporte incondicional um obrigado para minha mãe, para o meu pai, para os meus irmãos, André e Sandrina, e para o meu cunhado, Senhor Dr. João Ribeiro. Deixo aqui, para uma futura leitura, um agradecimento ao recém-nascido João Salvador, pela inconsciente, mas poderosa extra motivação trazida para a "espinhosa" reta final.

A todos os que fizeram esta caminhada possível, um forte obrigado do fundo do meu coração.

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

Arquitetura Heterogénea de Tolerância a Falhas Baseada em processadores Arm e RISC-V

Quando sistemas críticos operam em ambientes hostis, estes necessitam de serviços de redundância e de tolerância a falhas para continuarem em funcionamento mesmo na presença de faltas. Embora a técnica de tolerância a falhas seja eficaz para mitigar faltas que ocorrem num único componente, ela perde eficácia, quando múltiplas faltas acontecem simultaneamente em vários componentes. Estes tipos de faltas, despoletam o mesmo erro em todos os componentes afetados, tornando-as indetectáveis. Para solucionar este problema, usualmente, recorre-se a diversidade de desenho para mitigar as Falhas de Modo Comum (FMC), construindo assim um sistema mais robusto e confiável. Várias arquiteturas de tolerância a falhas, baseadas em *Field-Programmable Gate Array (FPGA)*, têm sido descritas na literatura, no entanto, pelas pesquisas efetuadas, nenhuma delas tem como objetivo proteger processadores heterogéneos e aplicar diversidade de desenho ao nível do processador.

Para resolver a supracitada falta de soluções, esta dissertação propõe uma nova arquitetura heterogénea de tolerância a falhas, Lock-V. O Lock-V promove diversidade de desenho, ao nível da arquitetura do processador, assim como técnicas de tolerância a falhas para, respetivamente, mitigar FMC e detetar e recuperar erros despoletados por causas externas, por exemplo, radiação. Para eliminar as FMC, o Lock-V possuí duas unidades de processamento diferentes: um *hard-core* Arm Cortex-A9 e um *soft-core* baseado em RISC-V. Desta forma é aplicada diversidade de desenho, usando heterogeneidade no *Instruction Set Architecture (ISA)*. Por outro lado, para implementar tolerância a falhas, o Lock-V propõe uma solução híbrida de *Dual-Core Lockstep (DCLS)*, onde a deteção de erros é feita em *hardware*, recorrendo a um acelerador na FPGA, e a recuperação dos erros é suportado por *software*, usando técnicas de *rollback*.

Após o Lock-V ser implementado na Zynq-7000 *System-on-Chip (SoC)*, mais de 45000 faltas foram injetadas. Os resultados dessa injeção mostram que quando uma aplicação executa na arquitetura Lock-V, para além de estar protegida contra FMC, devido à diversidade do desenho ao nível dos processadores, também está protegida contra 97% dos erros ocorridos. No entanto, implementar o Lock-V acarreta alguns *tradeoffs*. 79% das *Look-Up Tables (LUT)* e 34% dos *Flip-Flops (FF)* disponíveis na plataforma (Zedboard), são usados. Ao nível do *software*, o Lock-V aumenta em 8% o consumo de memoria e, para o pior cenário testando sem a ocorrência de erros, aumenta em 12% o *overhead* de execução. Tendo em conta que toda a redundância tem o seu custo, o Lock-V provou ser capaz de dotar um sistema com diversidade de desenho e capacidades de tolerância a falhas.

Palavras chave: diversidade de desenho, *lockstep*, redundância, tolerância a falhas.

Abstract

Heterogeneous Fault Tolerance Architecture based on Arm and RISC-V Processors

Safety-critical systems deployed in harsh environments rely on fault tolerance and redundancy techniques to keep them operating even in the presence of faults. Although there are effective techniques to mitigate one side faults, they are not enough to protect the system against simultaneously multi side faults. These kinds of faults trigger the same error in faulty redundant components, which makes resulting errors invisible and undetectable for fault tolerant mechanisms. To overcome this problem, design diversity is applied in fault tolerant system to mitigate the Common-Mode Failure (CMF) and build a more robust and reliable system. Despite several fault tolerance architectures based on FPGA are available in the literature, to the best of our knowledge, none of them aims both hardening of heterogeneous processors and applying design diversity at processor level.

To address this lack of solutions in the current state of the art, this dissertation proposes a novel heterogeneous fault tolerance architecture, Lock-V, which enables design diversity at processors architecture level. It deals with CMF, as well as both error detection and recovery fault tolerance techniques to mitigate errors triggered by external environment interactions, e.g., radiation. To eliminate the CMF, Lock-V explores an implementation based on different processing units: a hard-core Arm Cortex-A9 and a soft-core RISC-V-based processors, to leverage design diversity through ISA heterogeneity. To implement fault tolerance, Lock-V proposes a hybrid DCLS solution where the error detection is done by hardware, resorting to a FPGA accelerator, while error recovery is performed by software using rollback technique.

After the deployment of Lock-V on a Zynq-7000 SoC, over 45000 faults were injected. The results taken from such injection shows that when an application runs on the Lock-V architecture, besides its protection against the CMF due to processors design diversity, it is also protected against 97% of the triggered errors. Nevertheless implement Lock-V came up with some tradeoffs. It used 79% of the LUT and 34% of the FF available on the Zedboard FPGA platform. Regarding the software part, implementing Lock-V leads to an 8% increase in memory footprint and also an increase in the execution overhead around 12%, mainly in the worst case scenario as tested in the absence of errors. Knowing that all the redundancy has its cost, Lock-V proved to be able to grant a system with design diversity and fault tolerance capabilities.

Keywords: design diversity, fault tolerance, lockstep, redundancy.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ABI	Application Binary Interface
ADAS	Advanced Driver-Assistance Systems
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
ASIC	Application Specific Integrated Circuits
AXI	Advanced Extensible Interface
BIST	Built-In-Self-Test
BRAM	Block Random Access Memory
CMF	Common-Mode Failure
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DCLS	Dual-Core Lockstep
DDR	Double Data Rate
DMR	Dual-Modular Redundancy
DWC	Duplication With Comparison
ECC	Error Correcting Code
EDAC	Error Detection And Correction
FF	Flip-Flops
FIC	Fabric Interrupt Controller
FIFO	First In First Out
FMC	Falhas de Modo Comum
FP	Frame Pointer
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GP	Global Pointer
GPIO	General-Purpose Input/Output
I/O	Input/Output
I2C	Inter-Integrated Circuit

IoT Internet of Things
IP Intellectual Property
ISA Instruction Set Architecture
L2C Locked L2 Cache
LIFO Last In First Out
LR Link Register
LUT Look-Up Tables
MBU Multiple Bit Upset
MCU Microcontroller Unit
MMIO Memory-Mapped Input/Output
MMR Multiple-Modular Redundancy
MMUs Memory Management Units
MTBF Mean Time Between Failures
MTTF Mean Time To Failure
MTTR Mean Time To Repair
NASTI Not A Standard Interface
OCM On-Chip Memory
PC Program Counter
PL Programmable Logic
PR Partial Reconfiguration
PS Processing System
RA Return Address
RAM Random Access Memory
RCU Reconfigurable Computing Unit
RISC Reduced Instruction Set Computer
SBU Single Bit Upset
SCU Snoop Control Unit
SDC Silent Data Corruption
SEE Single Event Effect
SEFI Single Event Functional Interrupt
SEL Single Event Latch-up
SET Single Event Transient
SEU Single Event Upset
SHIFT Software-Implemented Hardware Fault Tolerance
SoC System-on-Chip
SP Stack Pointer
SPI Serial Peripheral Interface

TMR Triple Modular Redundancy

TP Thread Pointer

UART Universal Asynchronous Receiver Transmitter

VP Verification Point

WFE Wait For Event

1. Introduction

Processors industry keeps moving fast towards reduced transistor' size, higher clock frequencies, and lower operating core voltages. However, many problems to digital systems have emerged due to such progress, like system failures caused by bit-flipping induced by many possible sources, e.g., radiation. These problems can result in critical issues, not only in aerospace applications, but also in daily basis systems [4, 5, 6, 7, 8, 9]. This boosts research towards developing and deploying fault tolerance systems in order to mitigate several failure situations, while keeping other important requirements such as system robustness, reliability, performance and security.

One way to deploy reliable devices in mixed-critical applications, is to provide them with fault tolerance techniques. Redundancy is one of the most used forms of fault tolerance mechanisms and several solutions can be already found in the literature. While some techniques replicate processing units in a technique called DCLS — implemented either loosely- or tightly-coupled to the processor — [10, 7, 11, 12, 13, 2], others apply a Triple Modular Redundancy (TMR) mechanism, where the processing units are triplicated and a voter module is added to the system [14]. Other techniques can be used in order to achieve fault tolerance systems, such as time redundancy applied to low-cost architectures [15], and virtualization-based systems [16], [17], [18], [19], where several guests can virtually run over the same processing unit as if they were individually running in one unique processor. This way, each guest operating system (OS) can replicate the execution of the same software application, while another guest acts as the voter module. These software-based systems can behave similar to a hardware-based TMR without the need of replicating the hardware resources.

Fault tolerance has proven to be effective in the detection and recovery of errors due to physical faults, e.g., bit-flips by radiation, however, it is vulnerable to either human-made design faults [20] or external disturbances, that affects more then one redundant component at the same time [21]. This type of system's failures called CMF, e.g., power supply disturbances, manufacturer defects, software bugs in tools/compilers, among others, can only be mitigated using design diversity. Design diversity was proposed in [20] in order to protect fault tolerant systems against CMF and is described as " the approach in which the hardware and software elements that are to be used for multiple computations are not copies, but are independently designed to meet a system's requirements ". Nowadays, safety critical systems are aware of the need of design diversity to deal with CMF. To protect the processors against the common-mode failure and build a high-reliability systems, several techniques have been proposed. The design diversity on these

systems is usually applied targeting time diversity — which introduce a execution cycle delay among the processors [22] — but sometimes it can also be implemented with microarchitectural diversity [23] or even ISA diversity [24].

Fault tolerance techniques can be performed both in software and/or hardware, according to the available resources. With the ongoing technological trends, hybrid SoC solutions provide software programmability, available through the hard-core processors, and FPGA technology that can be resorted to deploy soft-core processors or dedicated hardware accelerators in order to enhance the computation of several types of algorithms in terms of speed and energy consumption [25, 26]. These heterogeneous SoC, composed by a hard-core plus a soft-core, make easier the development of a fault tolerant reliable system endowed with design diversity at the ISA level.

1.1 Motivation

Increasingly safety has become a must in the majority of the daily basis commercial systems which consequently, leads to an increase in the demand for fault tolerance systems. This high demand will boost the developing of new architectures and new systems with fault tolerant capability. This opens opportunities to research new techniques and approaches for implementing high-reliable and safety systems, which specially motivated me to jump in this research journey.

Another really interesting point is that, we are in the golden age of processors architectures with the appearance of open-source ISAs like the RISC-V that gives a new level of software and hardware freedom on architectures in an open extensible way, e.g., to develop custom instructions and tightly-coupled co-processors. This, altogether with the high availability of hybrid SoC FPGA-based in the market, brings huge architecture flexibility to explore, develop and delivery fail-functional systems. Despite several architectures and techniques for fault tolerance being available in the literature, to the best of our knowledge, none of them targets heterogeneous architectures that resort hybrid SoC solutions to implement different processor architectures, either deployed in hard- or soft-cores. This is a a kind of ”blue ocean” waiting for being explored.

1.2 Goals

The project developed on this thesis aims to provide an environment for a user to create and run its application in a fault tolerant fashion. After a comprehensive literature review, as exposed in Section 2.3, three main goals were established for this thesis:

1) Develop an architecture on which an application will run in a safety manner based on redundancy techniques to implement fault tolerance;
2) Develop a framework that allows the use of the proposed architecture and recovery the system from errors;

3) Develop a fault injection mechanism that will test the proposed architecture as well as its framework.

In order to achieve the three main goals, some more specific objectives were drawn. They are presented in the following topics.

To reach the first goal

It is necessary to harden the processors on the architecture using a technique called lockstep. This technique should be deployed as a FPGA accelerator and process information in parallel as well as accessing the redundant cores simultaneously. This accelerator needs also to have synchronizations and error detection capabilities, in order to compare at the same execution point the processors' outputs and detect errors (if they exist). The architecture should be also aware of the leveraged design diversity to mitigate CMF.

To reach the second goal

First the framework should be agnostic to the processor's architecture. This means that an application written for the framework, can be compiled either for Arm or RISC-V processors without changing the code. The framework needs to self adapt to the processor architecture. Second, the framework needs to be developed towards a fail-function system, which still keeps delivering its services in the presence of a fault. The system needs to implement techniques for fault resilience and recovering the system to a healthy state when an error is active. To achieve this the the framework have to provide three main functionalities:

- Allow the user (programmers) to choose verification's points in the code where the architecture error detection capabilities will be applied.
- Save a healthy system's state;
- Perform a rollback, recovering the system from an erroneous state to the previous saved system healthy state;

To reach the third goal

The fault injection mechanism should emulate harsh environments where the incidence of radiation cause faults. For doing that, the mechanism must inject faults similar to those occurring in the harsh environments.

1.3 Document Structure

The document of this thesis is structured as follows. Chapter 2 overviews the basic concepts around fault tolerance. It starts to contextualize the fault tolerance and explaining why it is needed. Afterwards it

expose methods that can be used to achieve fault tolerance through redundancy techniques. At the end of the chapter a survey is presented regarding the lockstep implementations in the current state of the art. Chapter 3 presents some requirements that were followed to choose the soft-core as well as the platform constraints imposed by the chosen soft-core. Finally, it presents the chosen platform to deploy the proposed architecture. Chapter 4 proposes the Lock-V, a fault tolerance heterogeneous architecture exposing its design and implementation. Chapter 5 addresses the design and implementation of the framework supporting Lock-V through auxiliary tools for the development of application under the proposed architecture. Chapter 6 presents and discusses the system evaluation as well as the fault injection mechanism that was designed and implemented. The last but not the least, Chapter 7 summarizes the thesis presenting its constraints and limitations as well as the future improvements.

2. Background, Context and State of the Art

As the main subject of this thesis is fault tolerance, this chapter will first contextualize the general definition around fault tolerance and its purpose. After that, it will be explain how fault tolerance can be implemented, followed by some work done in this area. Even though there are many types of redundancy as well as implementations, the focus will be on architectures based in FPGA, using dual-core lockstep for hardened the processors on the architecture, e.g., [3] and [2]. In the section about dependability and redundancy the basic terminologies and concepts are explained for a better understanding of the fault tolerance implementation in the thesis project as well as the need for design diversity in a fault tolerant system. The chapter finishes with a comprehensive analyses of lockstep implementation in current sate of the art as well as a literature gap analysis.

2.1 Dependability

The system dependability is defined as the system ability to avoid service failures that are more frequent and more severe than is acceptable [27]. This concept describes the means used to deal with the threats in order to do not compromise the system reliability. Table 2.1 represents the three basic notions inside the dependability concept. In the first column the dependability attributes are depicted, which are defined as the properties that a dependable system must own. In the next column the dependability threats are represented, which are defined as the reasons for a system to spot performing its function. The threats should be avoided or, in case of being unavoidable, it is needed to deal with them and minimize their consequences. For that reason, it is necessary methods for dealing with the threats. This set of techniques, as shown in the last column, are possible means or approaches for mitigating threats. The approaches can be either preventive or curative. If they avoid the threat and act before they occur, then they are preventive. If they deal with the threats after they occur, they are curative ones. In the following topics 2.1.1, 2.1.2 and 2.1.3 this concept will be explained in more detail.

Table 2.1: Dependability table of concepts.

Dependability		
Attributes	**Threats**	**Means**
Availability	Fault	Fault Prevention
Reliability	Error	Fault Tolerance
Safety	Failure	Fault Removal
Integrity		Fault Forecasting
Maintainability		

2.1.1 Dependability Attributes

The dependability attributes define the properties that a system is expected to have [1]. The five dependability attributes are defined by Avizienis *et al.* [27] as: (1) availability, readiness for correct service; (2) reliability, the system's ability to deliver a correct service in a continuous manner; (3) safety, absence of catastrophic consequences to the system external environment, both for the user(s) and the environment; (4) integrity, absence of improper system changes; and (5) maintainability, system's ability to be repaired and modified. Although dependability only makes sense when all attributes are part of system properties, in most cases the first three attributes, availability, reliability, and safety have the primary focus. In the following topics, each of these three attributes will be explained in more detail.

2.1.1.1 Reliability

Reliability is the probability of a system to perform without any failure for a given interval of time. Thus, the reliability represents a measure of time between the instant zero, when the system is in a fully functional state, and the next expected system failure [1]. This time is known as Mean Time To Failure (MTTF) [28]. In some situations (e.g, harsh environments, remote/inaccessible places) is required a high-reliability system that has to operate in a non-stop fashion. Such systems has to ensure that the delivery service is not degraded, even if some faults occurred. High-reliability systems, like power plants control system, space missions, heart pacemakers, can have faults and still be reliable. Such systems must prevent fault propagation, as they are always safe if no fault is propagated to a system failure. This is possible because high-reliability systems own mechanism to avoid the propagation of the faults. Later in Section 2.1.3 these mechanisms of fault mitigation will be addressed more in detail.

2.1.1.2 Availability

System availability is the probability that the system will function correctly at a given time or period [1]. In order to determine the system availability, it is necessary to know the Mean Time To Repair (MTTR), which is the time since one fault was detected until its full mitigation [28]. Availability is the ratio of the time

a system meets its specification and the elapsed time. Another important concept related to availability is the downtime per year. This represents the time of system's inoperability throughout one year for a given percentage of availability. Based on it systems are classified according to their availability, from unmanaged to ultra available. Systems are sometimes referred in terms of the number of nines that the percentage of availability owns. Table 2.2 depicts the availability of one system for a continuous operation during one year as well as its downtime, class ("class of nines") and system rating.

Table 2.2: Availability percentage corresponding to the different systems types.

System Type	Downtime	Availability	Class	Rating
Unmanaged	35.53 days	90%	1	
Managed	3.65 days	99%	2	Routine
Well Managed	8.77 hours	99.9%	3	Essential
Fault Tolerant	52.6 minutes	99.99%	4	
High Availability	5.26 minutes	99.999%	5	Critical
Very High Availability	31.56 seconds	99.9999%	6	
Ultra Availability	3.16 seconds	99.99999%	7	Safety-Critical

2.1.1.3 Safety

Safety is the absence of catastrophic consequences when one or more faults occur, preventing user damage and environment disasters [27]. This is an important requirement for safety-critical systems. If these systems fail they can either cause harm for the users, e.g., hearts stops beating if a pacemaker fails, or for the surrounding environment, e.g., radiation release if a nuclear power plant had a failure. Safety-critical systems need to have a safety failure mode. This failure mode must analyze, describe and then prevent, in case of a severe failure has occurrence, any damage caused to other equipment, people or to the environment. Despite this, a system can fail and stop its operations and still be safe. A safe system not necessarily implies that the system will continue operating after a fail. If the system when stops operating does not cause dangerous situations, it remains safe. However, if a system stop implies safety issues, it must not stop working at all. For example, the flight control system onboard an F-16 is only a safe system meanwhile the failure does not cause the loss of system functionalities. If the system fails and stops operanting, the F-16 falls down. In terms of safety, a system can has one of the following failure modes:

- **fail-unsafe**: a fail may cause safety hazards that may lead to user or environment harms;
- **fail-safe**: after a failure, the system switches to a safe operating mode with reduced functionality (there are degradation of the delivered service). One example of this failure mode is an elevator. If its cable breaks down, the brakes are applied and the elevator is stopped, avoiding that falling down. The system stop delivery its function, transport people between floors, but remains in a safe state being a safety system;

- **fail-functional or fail-operational**: the system keeps fully functional without any degradation of the delivered service for a certain number of faults. An example of a fail-operational system is the flight control system on a F-16 board, that is quadruple redundant. This means that it owns four fully functional replicas of the system that can start operating at any time, while only one would be required. In this mode of failure, the system may fail and still continue to provide its functionality safely.

2.1.2 Dependability Threats

The dependability threats are usually defined as fault, error, and failure. These threats are what compromise the functionality of a system. This may cause the system to deviate the current service from the previously specified service (its main purpose) preventing it from being delivered. This threats also undermines the system dependability attributes (mentioned in Section 2.1.1.).

2.1.2.1 Fault, Error, Failure

It is important to understand the concepts of fault, error, and failures and how can they trigger a fault tolerance system to recover successfully from failure situations caused by errors. A fault tolerance system must continue to provide the specified service, even at the event of a fault, and should react to errors caused by faults, preventing the error propagation to a state of system failure. In [27] Avizienis *et al.* provides the main concepts and taxonomy for the dependability threats:

- **Fault:** is defined as a logical manifestation caused by one or more physical defects, which change the normal operation of a component in a system;
- **Error:** is caused by one or more faults in a system when it transits into an internal state;
- **Failure:** occurs when some event deviates the delivered service from the specified service. A specified service is defined as a previously agreed description of the system behavior.

Faults can be classified according to many criteria. Regarding the domain, there are hardware or software faults. In terms of their causes, they can be divided into three groups [27]: (1) development faults that include all the faults that are introduced in the system during the design time; (2) physical faults that include all faults that affect hardware; and (3) interaction faults that include all external faults caused by interaction with the environment and/or the user. The software faults are predominantly caused by development faults. A software without development ”bug” works uninterruptible without gives an error and has a software failure. The software can stop working, but just in the case of the hardware (i.e. the memory, processors, etc) also stop working. The hardware fault can be due to a physical or an interaction fault and indirectly tamper with the data or the program in execution. Therefore, the software faults are always permanent, because they always result from development errors [29]. On the other hand, hardware faults are predominantly affected by physical faults and interaction faults (may be affected as well for development faults, but derived from the maturity of the hardware design this rarely happens).

Hardware faults are classified according to their durations into three groups [1]: permanents, transitory and intermittent. The permanent faults are caused by physical defects of the hardware. The transitory faults are triggered by unique events that affect the hardware, known as Single Event Effect (SEE). This can create hard- or soft-errors. In the case of a soft-error occurs, the SEE hit the hardware and just change its state without permanently damaging it. The hardware can be recovered moving to a fully operational state. In another case, when a hard-error occurs, the hardware stays damaged forever. This happens, for example, when there is a bit-flip, and it is never possible to write again to the bit that has been hit. In contrast, when a bit-flip happens due to a soft-error, the bit can be rewritten. The last-mentioned type of faults, the intermittent ones are transitory faults that occur periodically.

2.1.2.2 Causes

The transitory faults can be caused by several reasons. One of them is through the incidence of radiation. This radiation effects called SEEs can be either nondestructive or destructive [30]. SEEs can originate four different types of effects: (1) Single Event Latch-up (SEL); (2) Single Event Transient (SET); (3) Single Event Functional Interrupt (SEFI); and (4) Single Event Upset (SEU). What sets them apart is their cause. Each is triggered when a specific system component is affected by radiation. The SEE can cause permanent damage to the system, destructive SEE or hard SEE, or cause transitory damage that can be recovered, nondestructive or soft SEE. Out of the four above mentioned SEE, one of them can originate a hard SEE and three of them a soft one. The SEL is what causes the hard SEE. This happens when a particle hits a MOSFET or transistor oxide, triggering their gates and hence activating the latch. The three events that causes a soft SEE are the SET, SEFI and SEU. The SET is one of the three events that can originate a soft SEE, and it is characterized by the changing of the expected logical/combinational circuit behavior [31]. When a particle hits. the circuit may charge the P-N junction of the semiconductor and generate current pulses that may affect the MOS transistors and later the circuit behavior. The SEFI generates a system failure, e.g., when the particle hits a bit and generates an improper system interrupt or when the particle hits a bit that triggers a system self-test. In this case, the system goes into an improper self-test which is only solved by a system reboot. The last out of three soft SEE, the SEU, occurs when a particle hits a memory region and changes the value of that location without damaging it. For example, when a particle has flip a bit, a whole byte changes its value. The SEU is known in the literature as a soft error [30] and can has two nuances: (1) it can be a Single Bit Upset (SBU) in case of the radiation just hits a single bit; or (2) it can be a Multiple Bit Upset (MBU) when high energetic radiation hits a memory region and occurs a bit-flip of two or more bits [30].

2.1.3 Dependability Means

Dependability means are methods and techniques for mitigating the threats (Section 2.1.2) and so, delivering a dependable system. For building a dependable system two groups of techniques can be used, the fault avoidance or fault acceptance. They aim for different purposes and deal with faults in a

different manner. The first group of techniques, fault avoidance, tries to avoid the occurrence of a fault, e.g., through preventive maintenance. In contrast, the second group of techniques, the fault acceptance, does not prevent the faults, instead it accepts and isolates them. Thus, the faults are prevented from reach the system boundaries and propagate to errors and, consequently, to failures. The two groups of techniques deal with the threats in different manners. The techniques can be applied in pre-service, during the design-, implement- and test-time, or in-service during the operational lifetime of the system. A system can be made dependable endowing it with four types of techniques: fault preventing, fault removal, fault forecasting, and fault tolerance. Table 2.3 presents those techniques as well as their uses cases according to the phase in the system lifetime.

The Dependability means are described as follows:

- **Fault preventing** is composed by design techniques that can be used during the design-time when the specification, implementation, and fabrication stages are planed;
- **Fault Removal** can be performed either throughout the pre-service or service time. In pre-service, it is applied for testing and debugging the system in order to find faults in the system. This is done using three steps: (1) verification if the system functionalities match its specifications; (2) a diagnosis that comprises the identification of the faults and what cause those faults; and (3) the correction (removal) of the faults. The second possible use case of fault removal technique is during system service. It can be performed through preventive maintenance, where the systems' components are replaced before they fail.
- **Fault Forecasting** aims the detection of present faults and predicting futures ones, to avoid their consequences. Fault forecasting performs the evaluation of the system behavior, resorting to some techniques like failure mode and effects analysis, Markov chains, stochastic Petri nets and fault-trees [27].
- **Fault tolerance** is a technique that ensures the correct function of the system within the presence of faults. This dependability approach will be used in this thesis, so it will be addressed in more detail in the next topic 2.1.3.1.

Table 2.3: Dependability means and their use cases during the lifetime of a system.

	System Lifetime Phases	
Reliability Means	Pre-Service (Development)	In-Service
Fault Preventing	Design and Implementation	✗
Fault Removal	Test and Debug	Preventive Maintenance
Fault Forecasting	✓	✓
Fault Tolerance	✗	✓

✗ Not applied in this phase. ✓ Applied in this phase.

2.1.3.1 Fault tolerance

According to the defined dependability threats in Section 2.1.2, a fault is a malfunction in one component. That malfunction can lead the affected component to have an error that may propagate for a failure. Putting it simply, the failure of a component may lead to a system failure that causes dangerous system's behavior. To avoid the occurrence of a system safety hazard, the chain of fault, error, and failure has to be interrupted. The fault tolerance ensures that the system functionalities do not degrade in the presence of active faults. Hence, fault tolerance techniques need to prevent the propagation of the fault to failures. The fault tolerance mechanism has to detect the errors and recovery the system from them before they reach the component boundaries. As depicted in Figure 2.1, the fault tolerance mechanism needs to perform its functions of detecting and mitigating errors while being an insurmountable fortress to prevent the error propagation. The barrier that prevents the error to be propagated is made up a mechanism in the fault tolerance component. That mechanism transforms, through recovery methods, a system from an error state to a state without detected errors. The fault tolerance mechanisms, as depicted in the Figure 2.1, need to own two functionalities, error detection and error recovery [27]. These functionalities may follow several conceptual implementations to achieve a fault tolerance system that can detect errors and recovery from them.

The first fault tolerance functionality, error detection, identifies the presence of an error in a system. It can be split into two methods: (1) concurrent detection, an online detector that works at the same time at the system delivers its service; and (2) the preemptive detection that is an offline detector, that works when the system suspends its service delivery.

The other fault tolerance functionality, error recovery, is responsible for removing the error and restore the system for one state without errors. This can be reached through two manners, error handling or fault handling. The error handling eliminates the error from the system using one of the following three techniques:

- **Rollback:** brings the system back to a fully functional state. The system is recovered through a system's backup that occurred before the error was triggered. Afterward, the system goes back to

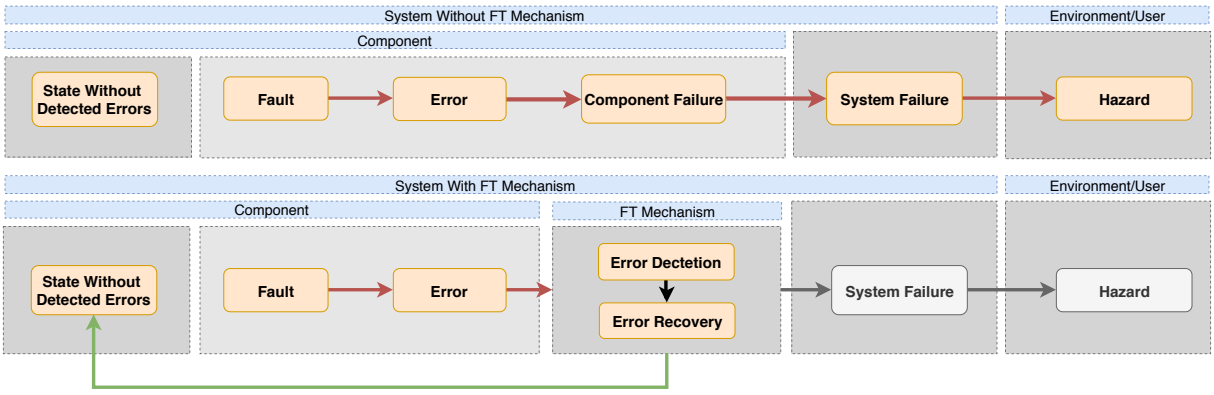


Figure 2.1: Fault tolerant system with error detection and error recovery.

a state that owns no errors, and starts moving forward from there;

- **Rollforward:** changes the system from an erroneous state to a error-free state. This way, the system's errors are mitigated and the system can keep move forward;
- **Compensation:** the system owns several redundancy mechanisms that can be enabled, masking errors when they occur.

The second error recovery technique, the fault handling, averts faults, which triggered errors, from being activated again through four possible manners:

- **Diagnosis** identifies the errors and determines the cause of their occurrence. The errors are cataloged in terms of "where" they had occurred and "how", describing the type of error;
- **Isolation:** the faulty elements, which are responsible for triggering the error are excluded from the system. This method of fault handling makes the fault dormant, excluding the defective component from the system;
- **Reconfiguration:** either switches components from faulty to redundant ones or reassigns tasks among healthy components;
- **Reinitialization:** updates the system state and its configurations. It is done either by resetting the system to an initial state or just updating the system information/configuration without a reset.

The above fault tolerance techniques are implemented in the systems according to their type, the needed degree of availability and the acceptable system downtime. The best-fit technique should be cautiously chosen. The hardware designers need to make tradeoffs when designing the system with these techniques. They have to have into account the level of the system dependability that is delivery versus the cost of the silicon area and engineering efforts that the implementation of the techniques requires.

2.2 Redundancy

Redundancy is the all addition system capabilities that would be unnecessary in a fault-free environment. The main idea is to have extra components in the system, which are designed for doing the same function like the original ones. A redundant system owns two or more copies (replicas) of the same instance when just one is required. Redundancy improves the overall reliability of a system, ensuring that if some part of the system fails, another one assumes the functions of the faulty ones. Thus the system is able to keep its service delivery without having a failure. In the literature there are many concepts for defining and categorizing all redundancy types that can be added to a system. In [1] the author splits the redundancy into three big groups and defines them as: (1) hardware redundancy that will be addressed in topic 2.2.1: (2) software redundancy that will be addressed in topic 2.2.2; and (3) information redundancy that will be addressed in topic 2.2.3.

2.2.1 Hardware Redundancy

Hardware redundancy comprises all the components that are added to the system to perform some functions already provided by the original system. There are two approaches in hardware redundancy: (1) addition of replicated modules; and (2) use of extra circuits for fault detection [1]. In the hardware redundancy one or more redundant units, are added to the system, performing the same functions as the original one. The redundant components can have an active, passive, or hybrid actuation in the system. If they actuate to mask the fault, they are a passive redundant system. These kinds of redundant components in the presence of a fault ensure that, just the correct values pass to the system. This way, it is avoided through fault masking the error propagation and thus, they stay contained within the boundaries of the components. In the opposite, if the redundant components react when a fault occurs, the redundant mechanisms is an active redundant system. In the active redundant mechanism, the fault is detected, and afterward, the system performs actions bring the system back to an operational state without faults. In some cases, both types of actuation are combined, and a hybrid approach is used. Fault masking is used for preventing the propagation of errors (passive approach), and fault detection and recovery is used to detect and replace or reconfigure the erroneous component (active approach).

2.2.2 Software Redundancy

In software redundancy, there are three main techniques, the N-versions programming, the time redundancy, and spatial redundancy. The first consists of writing the same program N times by N different persons and preferably, developed and compiled under N different programming environments, promoting this way diversity in the application development and the odds of the same error occurs in the N-versions and does not be detected is fewer. The second re-execute the code and check its results. The third technique makes replicas of the same code verifying the equality of the redundant code's outputs.

2.2.2.1 Time Redundancy

This particular technique of software redundancy consumes additional time to get a correct and valid result. In the time redundancy, one specific part of the code is re-executed more than once. Execution results are stored, and at the end of all program's executions, the stored results are compared. The outputs are verified if they match, and if not is because an error has occurred. This kind of redundancy in software is suitable to deal with transient faults since they do not occur (or is very unlikely to occur) in the same location twice and cause the same error two times consecutively. So, the re-execution of the same code should not produce the same transient fault, which will be mitigated.

2.2.2.2 Spatial Redundancy

This technique of software redundancy replicates the code to get a compiled redundant machine code that owns different instruction for the same purpose. This software redundancy technique is also known

as instruction redundancy. This named is attributed due to the additional instructions that are added to the binary code whenever a spare code is added to the application code. In the spatial redundancy, one particular part of the code is replicated, e.g., variables or functions. After the replicated code is executed the replicas are compared and checked if they are all equals. If it is not the case then an error has occurred. Like time redundancy, spatial redundancy is suitable to deal with transient faults since it is very unlikely to occur in the same memory region twice. For example, if a variable is replicated when it is affected by an error, it is possible through comparison with the others redundant variables, detect and correct it.

2.2.3 Information Redundancy

Information Redundancy is the encoding of the information the way that the errors are detected and in some cases, corrected. Information redundancy is implemented by adding some additional redundant bits to the original data bits. There are different forms of information redundancies like parity codes, linear codes, cyclic codes, checksum [1], among others. One information redundancy form that is widely used in applications for harsh environments is the Error Correcting Code (ECC). This technique protects the memories from radiation that may cause bit-flips. Usually, the ECC maintains a stored data immune to single-bit errors. It is used to ensure that the read data is equal to the data that had been written [32].

2.2.4 Redundancy Techniques

There are two widely used redundancies techniques, Duplication With Comparison (DWC) and TMR, both having software- or hardware-based implementations. Several DWC- and TMR-based implementations of redundant systems have been proposed like [33], [14], [34], [35] and [36]. Some hybrid applications combine both techniques like the author in [37] propose. Some other techniques can be found in the literature as the Software-Implemented Hardware Fault Tolerance (SHIFT) that implements copies of the same code to introduce redundancy at the instruction level [38].

2.2.4.1 Duplication With Comparison

The DWC replicates the original module to have on the redundant system two instances of the same component, which are operating in parallel. DWC solutions relies on the comparison of the outputs of each module, as depicted in Figure 2.2. Both modules are two instantiations of the same component (e.g., a processor, memory, busses, among others). The replicas outputs, have to match or otherwise, the system is in an incorrect state. That state owns an error that will be signalized by the verification module. In this technique, the error, which was indicated by the verification module, informs that a fault has triggered on the system. The system composed by the two redundant modules are not able to identify if each of them was hit and triggers the error. The only information is if there is a outputs' mismatch or not. Using this technique is impossible to know, which is the erroneous module. In the DWC there is another important

concept, which is the active module and the passive one. This distinction is made because only one module is used for output, the active one. The passive module is used for verification purposes only.

2.2.4.2 Triple Modular Redundancy

The TMR technique triplicates the original module and adds one voter module to the system, as depicted in Figure 2.3. The triplicate modules operate in parallel, and their outputs are used by the voter to check if an error occurs or not. This is done by majority voting, which compares if the three redundant components (e.g., processors, busses, memories, among others) are outputting the same object. If not, when one of them has a different output, the voter signalizes an error. This error, in contrast with the given by the DWC verification module, informs what is the erroneous module, due to the majority voting. If just one out of the three outputs differ is because that output is wrong. In this case, the majority voter masks the fault and recognize the correct value to output from the two fault-free modules. In some applications, instead of the fault being masked, the faulty module is recovered and brought back to a fully operational state without active faults. The TMR technique is used when a high-reliability system is required, e.g., in spacecraft, but with the penalty of more silicon area, in hardware-based TMR, or more code footprint and time spent, in software-based TMR.

2.2.5 Redundancy to achieve Fault-Tolerance

When a system needs to own fault tolerant services and be free from failures, mitigation mechanisms based on redundancy are used. These kind of mechanisms resort to hardware-, software- or hybrid-based

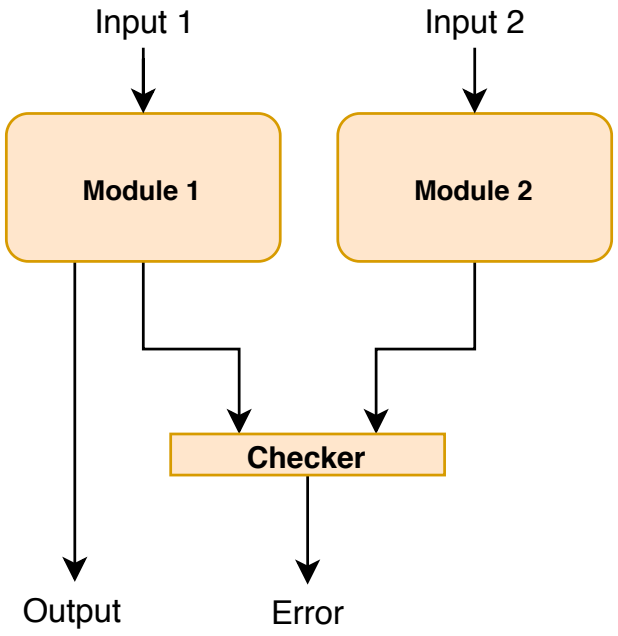


Figure 2.2: Representation of a system with DWC, adapted [1].

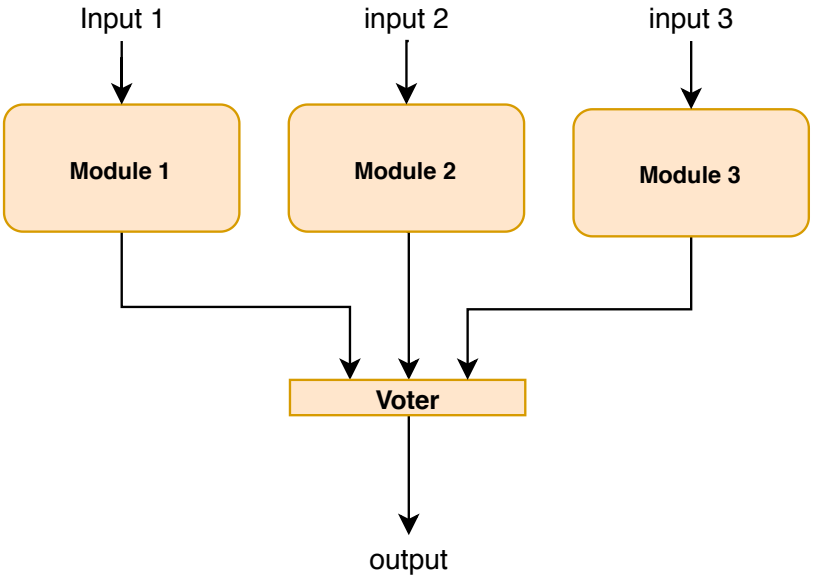


Figure 2.3: Representation of a system with TMR, adapted [1].

redundancy to implement fault tolerance. When a fault occurs in one component, which represents the system's basic unit, the following redundant approaches can be adopted: (1) if the fault is permanent, the component needs to be replaced; and (2) if the fault is transient, the component needs to be recovered. When a redundant system is used for mitigate permanent faults, the faulty component is a substitute for a healthy replica of the component. This substitution is a kind of pseudo-substitution as long as the redundant component does not physically replace the erroneous original component. Instead, the component's functions are reassigned to a redundant component for correcting the faults. When a redundant system is used for mitigate transients faults, it is recovered from an erroneous state to a state without errors. This is possible by applying one of the following redundancy approaches: (1) redundancy that owns two instances of the same component, e.g., DWC implemented in hardware; (2) redundancy that owns more than two instances of the same component, e.g., TMR implemented in hardware; or (3) redundancy that has one instances of the component, but its functionalities are replicated, , e.g., DWC or TMR implemented in software;

In the first redundancy approach, the two components replicas outputs are compared. When these outputs are different, means that an error has occurred. After the error is detected, it needs to be corrected. However it is impossible to know in which component the fault has occurred, since the only information that the fault tolerance system has is that a mismatch in the components' outputs was detected. So, for correct the error, both components need to be recovery. In the second redundancy approach, the fault tolerance mechanism receives information about three or more components. When one of these component's outputs are different, an error in one component is detected. In contrast with the first redundancy type, the fault tolerance system is able to detect the source of the error. So, after the erroneous component is detected, it is recovered. In the third redundancy approach, the component's functionality is replicated instead of the component itself. This means that the component repeats its functions two or more times.

Afterward the results of each execution are compared. This is called time redundancy [1]. Like the other two redundancy approaches, if execution is performed twice, a fault can be detected, if it is performed three or more times the fault location is identified.

In short, according with the type of fault, a redundant system can follow different redundancy approaches to be fault tolerant. For mitigating permanent faults the replacement approach is used. To mitigate the transients ones, is used the recovery approach. This approach can be implemented following one or a combining the above mentioned approaches. Therefore, a system can have only hardware- or software-based redundancy technique, or a hybrid technique when both are used at the same time. It is very usual the use of hardware redundancy with recovery in software, or the use of hardware redundancy to protect the system's hardware and software redundancy to protect the system's software.

2.2.5.1 Design diversity

Although the redundancies techniques can mitigate a wide range of errors, they do not make a system free from them. Some faults can affect the redundant components at the same time [39] and cause the same error in all the redundant systems. This is a false positive of correctness, since the system has an active fault, and hence an active error. In this situation the fault tolerance system is unable to detect it. This is called CMF, which is a fault that simultaneously occurs in two or more redundant modules. CMF are caused by some phenomena that create dependencies among the redundant components and makes them vulnerable to the same faults [40], e.g., single power source, same input/output bus. The shared of resources, design, or architectures makes that the redundant components fail at the same time. This occurs because the same issue are presented in all redundant units, since they have identical operational conditions that triggers a fault in all redundant components. The only fault tolerance approach to deal and mitigate CMF is the design diversity [1]. Design diversity, as the name suggests, explores diversity in the redundant components. The principle is to have components with redundant functions and not just redundant instantiation of the same component. A component A can delivery the same function of component B and have a different implementation. They can rely upon different development algorithms and programming languages under different architectures and designs. They also can be implemented by different teams that following different rules while using different software tools and having a different background (i.e., developed by different professional from different companies and/or universities). The design diversity is an important feature that a mechanism of fault tolerance should be aware of and do something towards it.

2.3 Lockstep

Lockstep is a fault tolerance technique that uses hardware redundancy at the processor level. The lockstep uses a dual- or multi-redundant instances of the main processor. Those redundant processors run at the same time, the same copy of the program and compare their outputs in order to detect a

mismatch among them (signal of an error). The lockstep may use either a Dual-Modular Redundancy (DMR) also known as DCLS, similar to the DWC, or a Multiple-Modular Redundancy (MMR), e.g. TMR. Both techniques have error detection and recovery phases. There are some differences between DMR and MMR as will be explained in the next paragraphs.

Error Detection Phase

At this stage, when using DMR, it is impossible to know on which copy of the processor, the error has occurred. So, the execution of both redundant processors are stopped, and the DMR signalize an error to the system. In the TMR, due to the majority vote, it is possible to know which one was the affected processor by the error. When this occurs, the TMR mechanisms signalized an error sending information to the system identifying the processor where the error occurred.

At this moment, when an error is signalized either by DMR or TMR, the checker does not know if the error is either a soft or a hard one. So, it is necessary to analyze the error and what is its type. If the error is soft, it is possible to recover the system from it. If the error is a hard one, the system cannot be recovered at all. Whenever an error is detected and for it to be classified, it is necessary to run a Built-In-Self-Test (BIST) to analyze the system by searching for a hard error. If the BIST does not find any hard error, it means that the error is a soft one.

Error Recovery Phase

In this phase, depending on the outcome of BIST, some action for recovery or containing the error is taken. In the DMR, if a hard error has occurred, the processors are stopped, a fatal error is signalized, and recovering the system from it is impossible. So, the system stops working and switches to a safe state. When a hard error occurs in a system with MMR redundancy, the erroneous processor is disabled, and the other health processors keep their execution as long as do not occur another hard error in the remaining executing processors. After a hard error, if the MMR was composed of three redundant instances (TMR), the MMR start working like a DMR redundant system. In the case the BIST does not detect any hard error, it means that a soft error has occurred. So, in the system with DMR technique, both processors are recovery to a state without any error, since it is unknown what is the erroneous processor. In the opposite side, in the MMR technique, the recovery is made to the erroneous processor only. The system keeps executing with the remained health processors, and when the erroneous processors are recovered, the MMR change to its fully functional state without had any execution interruption.

Types of Lockstep

Regarding the types of lockstep, it can have a loosely- or tightly-coupled implementation. In the tightly-coupled hardware lockstep, two or more processors are running synchronously, and the outputs are compared at every clock cycle. The comparisons are continuously being made. An error is detected before it propagates to the outside of the system. This type of lockstep is more robust as the granularity is small however it is expensive to implement [41]. In the loosely-coupled hardware lockstep, the outputs are compared periodically. In this type of lockstep, the granularity is higher, so the implementation cost is smaller. However, the error detection is weaker since now the error is only detected after it is propagated. The loosely-coupled lockstep needs less computation once fewer comparisons are performed. Hence, it uses fewer resources compared to the tightly-coupled lockstep, that has to perform comparisons ate every clock cycle. This introduces higher time and space overhead. The type of the lockstep should be chosen according with the type of application, its safety-critical requirements and the hardware systems constraints.

2.3.1 Design Diversity Applied To Lockstep

Lockstep is an appropriate solution for redundancy, but diversity is also needed [42]. Lockstep by its own, cannot detect errors that occur at the same time in the redundant cores since the error does not cause any difference in their outputs. For preventing the system from the CMF, it is necessary to combine the fault tolerance techniques with design diversity.

The lockstep can be implemented at three different design diversity levels:

- **ISA level**, the processors use in lockstep are different once the processors' ISA are different. Moreover, once shuch difference exists, although the same application code may be in execution, the binary code is different. So, never two equal instructions are in execution because the ISAs are not the same. In the worst case scenario, the processors can execute two instructions with similar behaviours, but even in that case, it is almost impossible that CMF happen, since the instructions have similar functions but different implementations. Another advantage of use different ISAs is the delay, which exists by nature, between the executions of the binary code. This happens because different ISAs have different time execution for each instruction when the application code is traduced to binary code and the outcome number of instructions are also different among processors. This also increases the diversity among the different ISAs codes. When the lockstep uses this diversity approach, the processors' CMF are more unlikely to occur and affect the system.
- **Microarchitectural level [23]**, in this design diversity level redundant processors have the same ISA, but different microarchitecture. For example, different teams can design and implement different Arithmetic Logic Unit (ALU) upon the same ISA. The engineering effort to implement this diversity is very high since every microarchitecture component must be implemented differently.
- **Temporal diversity [23]**, in this design diversity level identically instances of the processors, with the same ISA and microarchitecture, are used. The copies of the processors are delayed by some cycles to detect possible CMF. For example, the transient pulse in the energy can trigger

the same error if the processors are simultaneously executing the same instructions. However, it is unlikely that the same error would be triggered if the execution of the processors are delayed for few cycles since it is unlikely that the same instructions are being performed at the same time [22]. This kind of diversity is the most used out of the three kinds because the engineering and implementation cost (silicon area) is the smallest.

2.3.2 Lockstep Implementations

A plethora of possible implementations of the lockstep either software-, hybrid- or hardware-based can be found in the literature. The software-based lockstep can use spatial redundancy (replicating the instructions), temporal redundancy (replicating the execution), or both. For example, in the [43] virtual-lockstep, the authors propose a software lockstep hypervisor-based, that executes the code replicas in an n-modular redundant fashion. Despite the software-based lockstep implementations are effective in delivering a low area overhead and low implementations cost, they fail in to provide a robust, safety-critical and reliable device. They are exposed to the very usual software errors (as known as bugs), which when the system is in operation makes it permanently and unrecoverable. The software lockstep has low scalability, because every time that is done one application, the redundant mechanisms need to be introduced in the code. The software-based lockstep is also more exposed to the software and hardware CMF since hardware diversity does not exist, and software diversity is difficult and expensive to achieve. So, they are considered out of the scope of this thesis. The hardware- and hybrid-based lockstep implementations, found in the available literature and in the current state of the art, are closely related with the hard-core of this thesis and it can be divided into two groups: built-in lockstep implementations and non-native lockstep implementations.

Built-in Lockstep

Due to the industrial demand for safety-critical and reliable systems, Central Processing Unit (CPU) manufacturers were pushed to develop safety processors with built-in lockstep, ready for commercial use. This "Plug-and-Play" processors delivery a reliable solution for a system that needs fault tolerant with a reduced engineering cost. The major players in the processors' industry have launched their solutions to the market. Infineon from the first processor family, AUDO, to the most recent family, AURIX, has been releasing processors for automotive applications compliant to ISO 26262, ISO 25119 and IEC 61508. These processors are based on a 32-bit TriCore microcontroller [44] that owns three processors, two of them running in lockstep. The NXP has launched two processors solutions, the Arm Based Processors and the Power Architecture Processors based in its Microcontroller Unit (MCU)s. Both target ASIL-D and ISO 26262 compliant applications, but one has focus on high-performance computation, for autonomous vehicles applications [45], e.g., active suspension braking and stability control, and the other one low performance needed like chassis applications [46], e.g., gasoline engine management and hybrid and electric vehicle power inverter.

Arm has also been providing a wide range of solutions for a safety-critical system in all three processors groups Cortex-M, Cortex-R, and Cortex-A. In the processors targeting high performance and hard real-time applications, (Cortex-R) Arm offers the Cortex-R4, Cortex-R5, Cortex-R52, Cortex-R7 and Cortex-R8 [47], [48], [49], [50] and [51] with a built-in DCLS that support dual- (Cortex-Cortex-R5, Cortex-R52, Cortex-R7 and Cortex-R8), triple- (Cortex-R52, Cortex-R8) or quad-core (Cortex-R52, Cortex-R8) configuration (up to eight cores by processor, four main processors plus four redundant processors). In processors for high-end application (Cortex-A), Arm offers the Cortex-A65AE and Cortex-A76AE [52]. In low end processors' family (Cortex-M), Arm added the native support for lockstep at the processors Cortex-M23, Cortex-M33 and Cortex-M7 for later use in lockstep mode by the designers as the authors in [22] and [53] have done. New solutions has arisen based on the existing built-in lockstep processors [54], [55]. These works use a DCLS Arm Cortex-R5 processor that has already been verified and certified accelerating the design of the system and thus reducing the costs associated with developing the new architecture.

Some academic solutions have also emerged as proposed in [35]. Han *et al.* create a processing platform that implements in Application Specific Integrated Circuits (ASIC) a built-in dual core dynamic lockstep with time diversity for Advanced Driver-Assistance Systems (ADAS) applications. Despite built-in solutions are well suitable for safety-critical applications, they have a static implementation, what fails in delivery a dynamic and reconfigurable FPGA-based system.

Non-Native Lockstep Implementations

Increasingly, the FPGA have been used in harsh environments and safety-critical applications that require high dependability. For ensuring this, it is necessary that the FPGA owns fault tolerant mechanism for protecting the embedded processors on it. For hardening the processors, an architecture with lockstep is needed.

There are some lockstep implementations [56], [57] and [58] in FPGA-based that use only soft-cores. In [56] and [58] are used two Microblaze soft-cores in lockstep with a partial or full reconfiguration to mitigate any transient or permanent hardware faults. In [58] the authors added a third soft-core processor, a PicoBlaze to identify the faulty core, through a TMR technique. This way, the error location can be determined and correct and afterward mitigated using Partial Reconfiguration (PR) combined with rollback recovery. The lockstep implementation in [56] uses an output comparator to detect an error, and it cannot figure out its location. In [57] the authors implement a DCLS to protect the systems based in soft-core processors against SEU in data and configuration memories. That implementation intended to reduce the context processors saving and its recovery latency.

In [59], [60], [2], [3], [61] and [62] are proposed a loosely-coupled DCLS based in two hard-cores embedded in a SoC FPGA-based. In [60] it is implemented non-invasive lockstep with checkpoint and rollback recovery approach for Commercial Off-The-Shelf (COTS) processors. This implementation is transparent to the processor and software application, once that it does not require modifications either to the processors' architecture or software application. The authors use a checkpoint and rollback to mitigate

SEU. In this implementation the program has been executing and frequently, in some predefined points, it is stopped, and a consistency check is done. If the check passes, the context of the processor is stored in a soft error tolerant memory. If the check fails, a rollback is done, and the system is recovered from the last checkpoint (last processor state without errors).

In [2] is proposed the Transaction Checker Architecture. An architecture based on a DCLS that uses a transition checker (in the Programmable Logic (PL)) combined with some specific processor configuration and interrupt handling to implement a loosely-coupled Lockstep. The Transaction architecture, depicted in Figure 2.4, is composed of two Arm Cortex-A9 processors that execute the same copy of code in parallel. Each processors' code is stored in two different memories. One in a Locked L2 Cache (L2C) and another in an On-Chip Memory (OCM). The processors' memories are isolated from each other since the Memory Management Units (MMUs) and Snoop Control Unit (SCU) limit the range addresses. The processors use shadow registers to communicate with the Transaction checker. It has a set of 20 shadow registers for each processor. However, these two banks of registers are seen as one by the processors, due to a physical to virtual address mapping. Both processors access the same virtual address, but different physical address (different banks of shadow registers). The Transaction Architecture provides a processor event bus that is responsible for the lockstep services. This bus owns a set of signals that informs the transaction checker when it has to read the shadow registers to perform the comparisons of processors' outputs. When the processors perform a transaction, after the shadow registers have been configured, the Wait For Event (WFE) signal is used to inform the checker that the data is ready for being read. When the WFE is low, the processors are executing, when is high, the processor is waiting for an event. The processors stay waiting for an event until the transaction checker toggled the event input signal. When this occurred, the processors resume their executions. The transaction checker controls the reading and writing in the Input/Output (I/O) and the interruptions. When the processors want to access to the I/O they send that information to the transaction checker and, if the data is equal, the checker writes or read the I/O. The interrupts are also controlled by the checker. It is connected to a Fabric Interrupt Controller (FIC) that it is connected to both processors, so when an interrupt is triggered both processors receive that interrupt signal equally and at the same time. The Transaction Architecture is composed by hardware-based loosely-coupled lockstep that in the presence of an error, reports it and cause an alarm and afterward, to prevent the error propagation, the system is halted or reset.

In [3], [61] and [62] Oliveira *et al.* proposed a loosely DCLS depicted in Figure 2.5, that own one Double Data Rate (DDR) memory, two Block Random Access Memory (BRAM) memories, one for each processor and a checker module. The DDR memory, which is used by the two Arm Cortex-A9 processors, stores the program that will be executed and is used as an alternative safe memory for storing the checkpoint, once that the checkpoint stored memory used by default is BRAM-based. Although is protected by a Error Detection And Correction (EDAC), it is vulnerable to radiation. The BRAM memories are connected to each own processors by an AXI interconnect and are used either for all the application data or for saving the processors' context at the checkpoint time (writing the context) or rollback time (read the

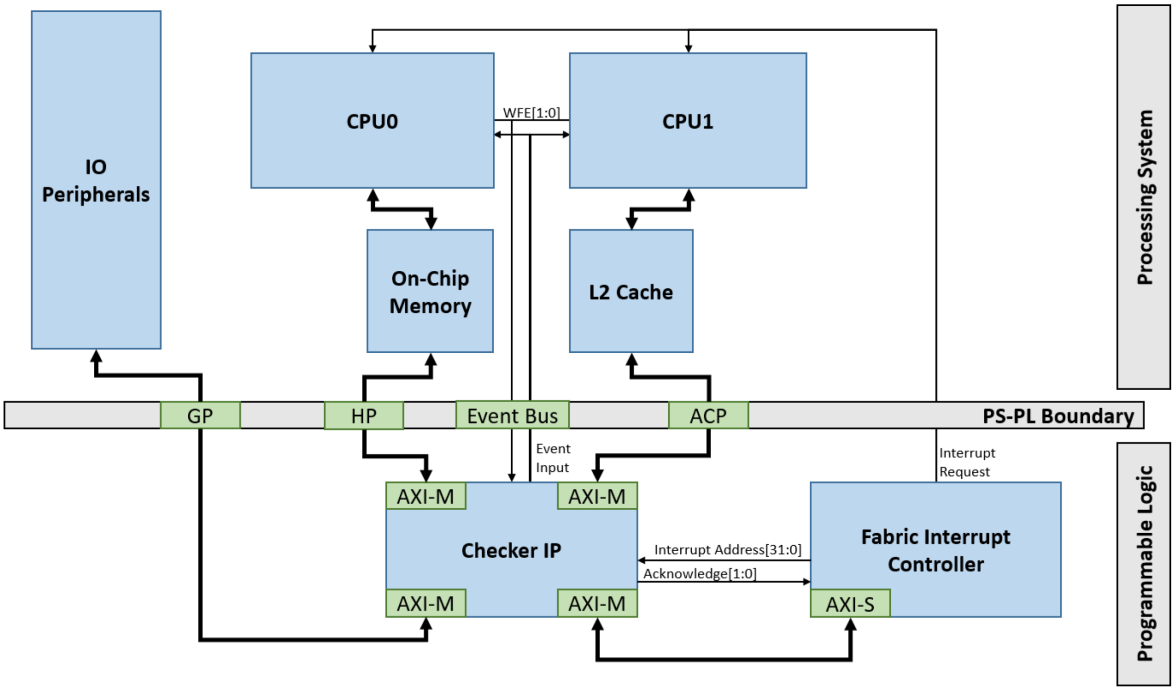


Figure 2.4: Transaction Architecture block diagram [2].

context). The checker module is responsible for verifying the processors' consistency and for performing the lockstep services. The application is executed simultaneously in both processors. In order to perform the consistency checking, the application is divided into blocks, and at the end of each block, a Verification Point (VP) is added. When a VP is performed, the processors' status are stored in their BRAM memories, and the execution is locked. Afterward, the checker triggers an interrupt to read the processors' registers and to compare the outputs of each unit of processing. If they are equal, the processor is considered in a safe state, and the execution is resumed, by a new CPU interruption, and the context of the processors' are saved. Otherwise, if a mismatch has occurred, it is generated an interrupt to perform the rollback. After that, the checker gives a signal to unlock the processors. The rollback can be done by two different manners. The first one, accessing the BRAM memory for saving processors' context (checkpoint) and later recovering to a safe processor´s state, read from BRAM (rollback). This first approach can be done by another way if more reliability in the rollback is needed. Since the context processors data saved can also be affected by an error, accessing the BRAM and DDR memories for saving and recovering the processor context is more reliable than just using the BRAM. In this second checkpoint/rollback setup, the context is stored in both BRAM and DDR memories. The default recovery memory still is the BRAM, although if after the rollback from BRAM the error in the block code persists, another rollback is performed, but this time from the external DDR memory. When this situation occurs, the previously checkpoint data, which is corrupted, is overwritten by the DDR data. The checker can have one of the following two approaches to detect errors. First, in each VP comparing all the outputs position of the BRAM0 and BRAM1 memories. Using this approach requires no added code to the application. Second, generate a signature from the outputs before the VP that choose just some outputs to be compared for the checker, i.e., the signature ones.

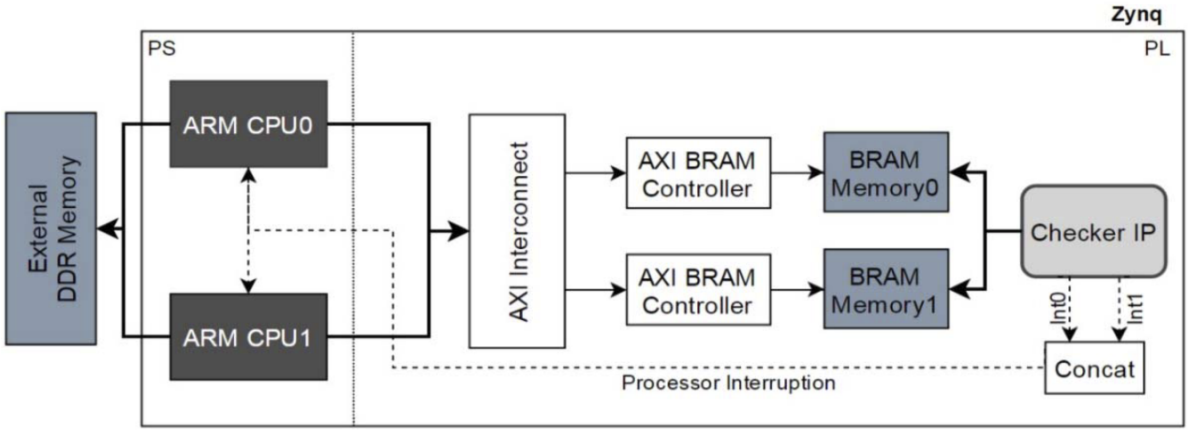


Figure 2.5: Proposed loosely DCLS architecture implemented in the Zynq-7000 APSoC [3].

This has less overhead, once that fewer comparisons are performed, but could mask an error because the outputs were not fully verified.

In [59] is proposed a DCLS based in [3], [61] and [62] implementations, with small architectural variations, it is used two individual DDR memories, rather than one shared DDR, while targeting a different processor, Arm Cortex-A53. It seems to be a porting of Oliveira et al. lockstep implementations for a new board and a new processor. The main contribution of this paper relies on heavy reliability analyses that prove, through Markov model and Matlab testing, that the DCLS architecture proposed in [59], [3], [61] and [62] is suitable for microsatellites applications, once that it meets the operational requirements of them. Sun *et al.* determine that the proposed DCLS has a Mean Time Between Failures (MTBF) at approximately 9.55 years, what it is bigger than the average life span of microsatellites, 5-8 years.

2.3.3 Discussion

Table 2.4 summarizes the gap analysis targeting commercial or academical lockstep implementations compared with the envisioned solution in this thesis. The lockstep implementations have been compared based on the type of lockstep, the processors' lockstep support, its diversity (time, microarchitectural and ISA) and relation at its architecture components, if it has hard-cores, sofcores or FPGA.

Although several FPGA- and ASIC-based lockstep mechanisms have been proposed in the literature for protecting the processors, they fail into providing a robust solution in terms of design diversity. After some literature research, it is possible to see two types of scenarios. On one hand, the FPGA-based lockstep solutions, just ensure redundancy capabilities to the system, but they do not leverage any design diversity. On the other hand, the ASIC-based lockstep solutions protect the system against CMF, but only using time diversity. This happens because implementing design diversity entails more engineering cost and silicon area overhead. For that reason, the processors' designers choose to use only time diversity since it is the cheapest solution in terms of implementation and developed costs. However, time diversity in some cases

is not enough because the outputs are just delayed few clock cycles, typically 2 cycles and some CMF (e.g., power source and clock tree issues) may last too longer than that clock offset.

Table 2.4: Gap analysis among Lockstep implementations.

	Arch. Components			**Diversity**			**Lockstep**	
	Hard-core	Soft-core	FPGA	Time	Micro.	ISA	Type	Support
Hanafi *et al.* [56]	✓ (2)	✗	✓	✗	✗	✗	L	NN
Sun *et al.* [59]	✗	✓ (2)	✓	✗	✗	✗	T	NN
Pham *et al.* [58]	✗	✓ (3)	✓	✗	✗	✗	T	NN
Cornejo *et al.* [57]	✗	✓ (2)	✓	✗	✗	✗	T	NN
Han *et al.* [35]	✓ (2)	✗	✗	✓	✗	✗	T	B
Abate *et al.* [60]	✓ (2)	✗	✓	✗	✗	✗	L	NN
Kral *et al.* [2]	✓ (2)	✗	✓	✗	✗	✗	L	NN
Oliveira *et al.* [3]	✓ (2)	✗	✓	✗	✗	✗	L	NN
Iturbe *et al.* [54]	✓ (6)	✗	✗	✓	✗	✗	T	B
Yiu [42]	✓ (2)	✗	✗	✓	✗	✗	T	N

(L) Loosely-Coupled **(T)** Tighly-Coupled **(B)** Built-in **(N)** Native **(NN)** Non-Native

3. Platform

The choice of a platform that supports the architectures developed by the systems' designers, take an important role in a project. This chapter will explain the criteria used for choosing the platform that will support this thesis. A key demand for the thesis architecture is the deployment of RISC-V processor. In the first section, the demand for choosing a RISC-V processor is discussed, as well as it will be exposed a candidate analyses for the RISC-V processor. In the second and last section, it will be addressed the choice of the platform according with the constraints imposed by the chosen RISC-V processor.

3.1 Processors

Due to the demand for design diversity, addressed in Subsection 2.2.5.1, it was used two different processors, to ensure diversity at time, microarchitecture and ISA level. The selected architectures for the processors were the Arm and RISC-V. The first was chosen because it represents one of the most used processors' architecture for embedded systems applications. According to the Arm media fact sheet (Sept. 1, 2016) [63] at the time, more than 86 billion Arm-based chips had been shipped, 42% of them for the embedded space. Arm processors are becoming widespread in embedded space due to their well-balanced capabilities since they combine performance with low cost while offering wide processors' family portfolio. The RISC-V processor was chosen because it is a novel open-source ISA [64] based on a Reduced Instruction Set Computer (RISC) architecture. It was designed with a focus on embedded systems, Internet of Things (IoT), and other modern devices. RISC-V allows a new level of software and hardware freedom on architectures in an open extensible way. This ISA allows the implementation of RISC-V ISA-based cores and adapts them to, for example, fault tolerance techniques like DCLS or TMR. It is possible to create new instruction due to architecture freedom and focus them for a specific purpose. Because the RISC-V is a very recent ISA (the RISC-V Foundation was founded just 4 years ago), at the time of the starting of this thesis (autumn 2018) for the best of my knowledge, none FPGA own hard-core Arm alongside a soft-core RISC-V. So, it is necessary to use different core implementations, using one hard-core Arm (already deployed at Processing System (PS)-side) plus one soft-core RISC-V (to be deployed in PL-side).

Some of the freely available RISC-V soft-core implementations, requires host environment features, both for the booting process and for the processor to run and execute the application. Such implementations are called tethered processors [65], as they require a host processor to start up and to interact

with the environment. For this reason, be untethered becomes an important requirement for the soft-core. The soft-core must provide too, an open implementation that makes it flexible, adaptable and scalable for allowing the deployment of the lockstep, and for futures implementations of new features. To be easily scalable for new features, the soft-core must also be developed through a high abstraction level using one hardware construction language (like Chisel). This kind of soft-cores implementation allows faster hardware development since it abstracts the designer from the specification of the basics hardware units which the most of the time cause design and implementations errors. The soft-core must also be a general-purpose processor (RV64GC) to be similar to Arm's general-purpose hard-core and thus, it should be programmed for a wide variety of applications. Due to the time shortage, in this case is the time to finish the thesis, once processor' design is not the scope of this thesis (they just are a component of the architecture), the soft-core must support a heterogeneous platform with an Arm hard-core plus a FPGA part. This is an important requirement because doing a soft-core porting for a new platform that is not supported, may last some time. For summarizing, the soft-core must fulfill the following requirements: (1) be untethered; (2) be flexible, adaptable and scalable; (3) be implemented in a high abstraction language; (4) be a general-purpose processor; and (5) has support for a heterogeneous board. These requirements were used to do a comprehensive analysis of the soft-core candidates, compiled in table 3.1. There are several processors that are good candidates for be used as soft-core such as the Ibex (Zero-riscy) [66], Pulpino [67], picoRV32 [68], Ariane [69], Orca [70], Mi-V_RV32 cores [71], BOOM [72], Freedom [73], Rocket [74], lowRISC [75], among others. However, not all of them fulfill the requirements specified above.

The Ibex, Pulpino, picoRV32, Orca, and Mi-V_RV32 are not general-purpose processors, and they are not programmed in a high abstract language hence the scalability is medium or low (Mi-V_RV32 is not open-source because is a Microsemi proprietary implementation of RISC-V ISA). For these reasons, these

Table 3.1: Soft-Core Candidates Analysis.

	Untethered	Scalability	Language	Board Support*	ISA
Ibex	Yes	Medium	SystemVerilog	No support	RV32IMC
Pulpino	Yes	Medium	SystemVerilog	ZedBoard	RV32IMF
picoRV32	Yes	Medium	Verilog	No support	RV32IMC
Ariane	No	Medium	SystemVerilog	No support	RV64GC
Orca	Yes	Medium	VHDL	ZedBoard	RV32IM
Mi-V_RV32	Yes	Low	Verilog	No support	RV32IMAF
BOOM	No	High	Chisel	ZedBoard, ZC706	RV64G
Freedom	Yes	High	Chisel	No support	RV64GC
Rocket	No	High	Chisel	Zybo, ZedBoard, ZC706	RV64GC
lowRISC	Yes	High	Chisel	ZedBoard**	RV64GC

* Just was considered boards that support the implementation of a heterogeneous architecture composed by one hard-core plus a soft-core (hybrid SoC). If the cores just support boards with FPGA, that soft-core is considered that has no support.
** For the lowRISC version 0.3

processors were excluded from the soft-core candidate list. Although the Ariane, BOOM, and Rocket be general-purpose processors, they fail into providing an untethered implementation, since they rely on a host environment to start-up, run programs and interact with the surrounding. So, they are excluded too. Only two candidates remain on the list, Freedom, and lowRISC. They are very closed candidates. However, Freedom does not provide support for a heterogeneous board. Therefore, the only soft-core on the list that meets the requirements is lowRISC.

The lowRISC processor fits in all essential characteristics that soft-core need to own: (1) it is an untethered soft-core processor, which is a crucial aspect for the implementation of the lockstep mechanism since each processor (Arm and RISC-V) have to execute their binary machine code independently; (2) it has high scalability once it is a customizable core, enabling the refactoring of the lowRISC processor to the project requirements, such as adding a master/slave Not A Standard Interface (NASTI) bus, which is similar to AXI, for connecting the core to loosely-coupled accelerators or for adding tightly-coupled accelerators that can work as co-processors; (3) it is coded in a high abstraction language, chisel; (4) it has support for a heterogeneous board, ZedBoard; and (5) it is a general-purpose processor, RV64GC ISA. In the following topic, it will be addressed and explained in more detail.

3.1.1 The lowRISC

The lowRISC is a 64-bit SoC platform based on the Rocket Chip RISC-V ISA implementation. The lowRISC aims bringing the benefits of the open-source development to hardware world and being "The Linux of the hardware world". In doing so, it allows the design of more custom hardware to endow the next generation of computing of personalizing, secure and safety hardware.

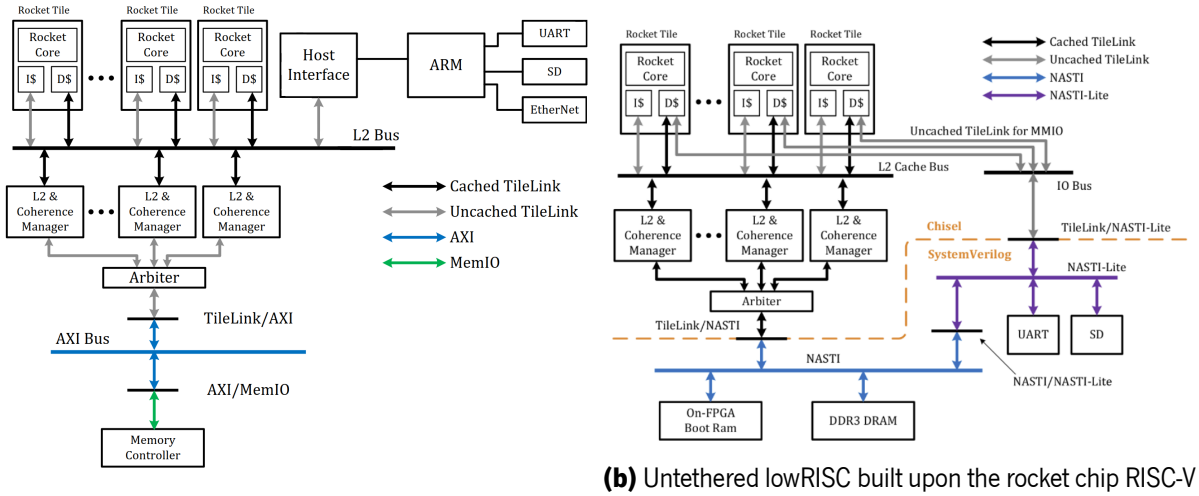
The lowRISC has been releasing several version of the lowRISC. So far, there are six releases:

- **version 0.1**: tagged memory (04-2015). The lowRISC first code release that is built upon the Rocket RISC-V implementation offers support for tagged memory;
- **version 0.2**: untethered lowRISC (12-2015). This lowRISC version has a standalone implementation achieved through untethering the Rocket Chip. The support previously developed in version 0.1 for tagged memory was not added;
- **version 0.3**: trace debugger lowRISC (07-2016). In this code release, lowRISC gained its first debug infrastructure. This is an extended untethered implementation of lowRISC (version 0.2) to which debug support has added;
- **version 0.4**: minion tag cache lowRISC (06-2017). This release provides a more complete prototype of tagged memory implementation. A "minion" core was also added for delivery an SD-card interface. The support for keyboard and VGA compatible text display was added too;
- **version 0.5**: ethernet multiuser lowRISC (12-2017). This release provides a complete Ethernet reference design. The SD-card interface is now a Rocket peripheral for having more performance. The keyboard and VGA compatible text display were set as the default Rocket console;

- **version 0.6**: technical refresh lowRISC (06-2018). In the last release, the Rocket core updated to the March 2018 Rocket version and was added JTAG debugging conforming to the RISC-V specification and this version now has also support to Debian Linux.

The chosen release is the lowRISC version 0.3 due to two reasons. First, it fits the thesis soft-core requirement since it is an untethered standalone processor built upon the Rocket Chip implementation of the RISC-V ISA. It eliminates the need for a companion core, which is replaced with FPGA peripherals. Second, it is the only version that has support for a heterogeneous board (ZedBoard), which is crucial for deploying a DCLS with different processors, that have both to run independent binary code.

A high-level view of the previous version of the tethered Rocket chip is shown below Figure 3.1a and in Figure 3.1b it is shown the untethered version of Rocket chip that lowRISC was generated. The Rocket chip needs a companion core to initialize the state of memory, execute programs, and interact with the external world. The Rocket chip is connected through a host interface to an Arm processor (the companion core), which is connected to its L2 bus. In this processor, it is run a Linux that uses a RISC-V frontend server to interact with the Rocket core, e.g., execute binaries code or start up a Linux in the Rocket Core. The booting of the rocket chip is not standalone, i.e., it is dependent on the Arm processor. In the lowRISC there is no such processor, so it is easier to boot from a boot loader code. The lowRISC removed the Arm companion core and remapped the peripherals for FPGA. Furthermore, the Universal Asynchronous Receiver Transmitter (UART) and SD peripherals were connected to the lowRISC through the FPGA and the boot is done either through an on-FPGA Random Access Memory (RAM) (the method used by the lowRISC in the ZedBoard) or through a DDR3 RAM. For connecting the Rocket tiles to the peripherals, lowRISC implemented two NASTI/NASTI-Lite interfaces, that is a limited subset implementation of the AXI/AXI-Lite bus functions. The NASTI interface is used by the L2 cache to read and write in memory, and the NASTI-Lite interface is used by the I/O bus to access the peripheral. This way the Rocket chip was made



(a) Tethered Rocket chip, [76].

(b) Untethered lowRISC built upon the rocket chip RISC-V ISA implementation, [77].

Figure 3.1: Tethered and untethered implementations based on the Rocket chip.

untethered and can be used as a standalone processor. Now it is possible adding more custom Memory-Mapped Input/Output (MMIO) peripherals through more NASTI-Lite interfaces, which enables the use of this upgrade version of rocket chip in a wide range of applications. It is also possible a direct bootload while working with the lowRISC in a standalone manner.

3.2 ZedBoard

Regarding the hardware platform, according to imposed requirements for the soft-core, the chosen one, lowRISC, restricts the possible range of FPGA solution to a ZedBoard (as we saw in Section 3.1). The ZedBoard Figure 3.2 is a low-cost, flexible, and scalable development board that features an XC7Z020 Zynq device. This integrates a dual-core Arm Cortex A9 in its PS side and PL in a single device. The Arm Cortex A9 is the heart of the PS which also include caches, OCM, external memory interfaces, DMA controller and several I/O peripherals and interfaces, e.g., General-Purpose Input/Output (GPIO) with four 32-bit banks, high-speed UART, master and slave Inter-Integrated Circuit (I2C) interfaces, full-duplex Serial Peripheral Interface (SPI) ports, among others. Many different resources compose the PL as 85K programmable logic cells, 53.2K LUTs, 106.4K FF, 400 DSP slices and 140 Block RAM of 36 KB, making it very suitable for a wide range of FPGA-based applications. The dual-core Arm Cortex A9 in the PS always boot first,

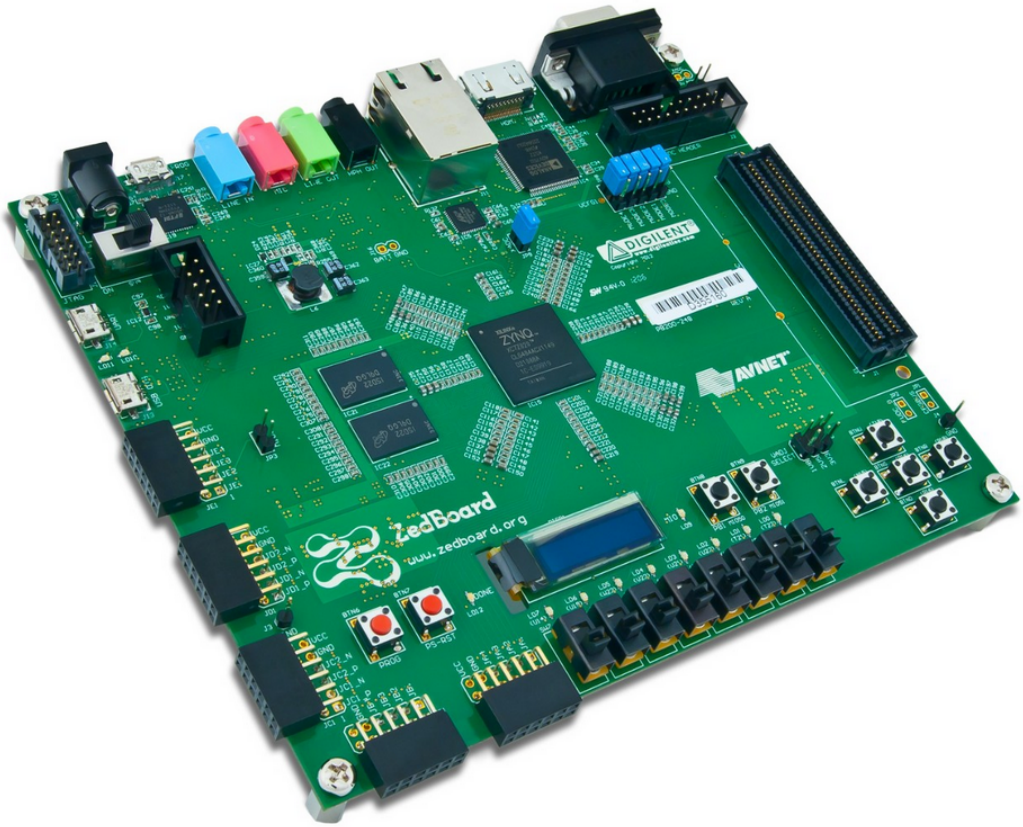


Figure 3.2: ZedBoard development board.

allowing a software centric approach for PL system boot and PL configuration. The PL can be configured in the boot process or in the runtime of the system. Using this feature, the PL can be reconfigured either for full or partial configuration. PR allows configuration of a portion of the PL. This enables optional design changes such as updating coefficients or correct an error when the FPGA is hit by radiation. PR capability is analogous to the dynamic loading of software modules, offering several useful applications.

4. Proposed Architecture (Lock-V)

This chapter presents the Lock-V, a heterogeneous architecture that explores a DCLS fault tolerance technique in two different processing units: a hard-core Arm Cortex-A9 and a soft-core RISC-V-based processor, lowRISC. At the beginning of the chapter is discusses the best lockstep technique to be applied to the Lock-V architecture. Afterward, Lock-V architecture design and implementation are explained. The next section describes the adaption done to the lowRISC soft-core in order to fit it in the lock-V requirements. Furthermore, the *xLockstep* is introduced, and its functionalities are detailed. Next, it is shown how the accelerator was implemented to be suitable to connect two different processors. Near the end of the chapter, it is described how the *xLockstep* was deployed in the proposed fault tolerance architecture. Lastly, in the final section, it is presented the *xLockstep* Application Programming Interface (API), a bunch of functions that allow the external world access and use the lockstep capabilities of the Lock-V.

4.1 Adding Lockstep Capabilities

The ideal redundant system has 100% availability, ensuring this way that the system never fails. Despite that some system can achieve availability closely of 100%, they require a lot of resources, either physical (silicon area) or design ones (engineering costs). So, it is necessary to make some tradeoffs in order to chose the redundancy level that best fits the system requirements, having into account the imposed constraints. Once the main goal of this thesis is developing a redundant system that leverages design diversity, having a high-availability system, for now, it is not a concern.

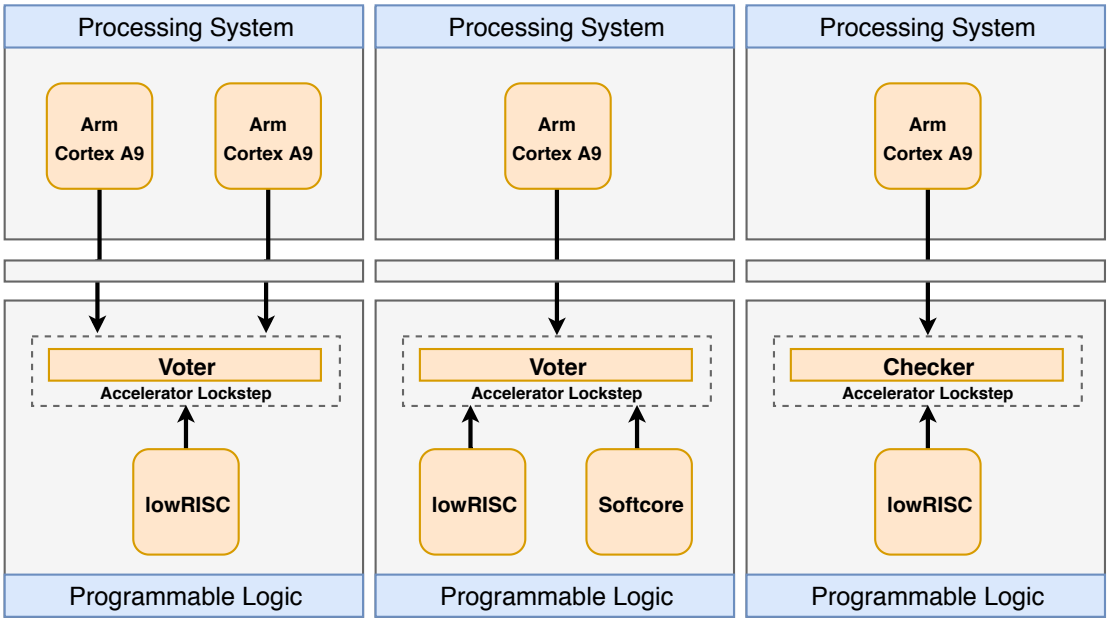
The first initial step was to choose which lockstep technique and redundancy level would be used. As we saw in sections 2.3, there are two main techniques that a lockstep system can implement, a DCLS or a TMR lockstep. Thus, to develop such systems, some aspects must be considered, such as area overhead, implementation and design costs and system effectiveness to meet the project requirements (in this case, design diversity). Figure 4.1 depicts the three main design scenarios for deploying a fault tolerance heterogeneous architecture in the heterogeneous board selected in Section 3.2, with the soft-core that was chosen in Section 3.1, lowRISC, and onboard Arm Cortex A9 processor. In all alternatives of heterogeneous design there are the chosen RISC-V soft-core (lowRISC), the dual-core processor that integrates the Zedboard, a lockstep accelerator and in some design options it is considered using an extra soft-core. According to the type of lockstep applied to the architecture, the configurations of the design

may vary between the following combinations: (1) a TMR, depicted in Figure 4.1a, with one soft-core, two hard-core Arm Cortex A9 plus one lockstep accelerator; (2) a TMR, depicted in Figure 4.1b, with two soft-cores, one hard-core plus one lockstep accelerator; or (3) a DCLS, depicted in Figure 4.1c, with one soft-core, one hard-core plus one lockstep accelerator.

Figure 4.1a illustrates a possible design option leveraging a TMR-based lockstep solution. This solution is composed by one soft-core lowRISC, in the PL, alongside with one accelerator lockstep, which implements a voting mechanism for detecting which is the erroneous processor when an error occurs. The other two processors are the hard-core Arm Cortex A9 integrated in the chosen board. Once that two out of the three processors are identical, i.e., they have the same ISA, the system will have less design diversity. If a CMF hits the two Arm processor, the voter does not detect it. However, a mismatch will emerge because the output of the two Arm processor would be different from the lowRISC processor. The voter will try to recover the lowRISC processor because the majority voting and the system will start working in an erroneous state that may cause a safety hazard. So, this design solution should be avoided.

The design solution illustrated in Figure 4.1b, is similar to the previously mentioned design, but instead of using the two Arm Cortex A9 it is added an extra soft-core. This solution is composed of two soft-cores lowRISC plus another like for example, Microblaze plus one hard-core Arm Cortex A9. This way, the design diversity requirements are accomplished since all three processors have different ISAs. Yet, this would take too long to develop and implement the system. Additionally, it would require a significant amount of programmable logic, that may be a limiting factor for futures improvements to the project.

Figure 4.1c illustrates a different solution from the previous ones which instead of implementing a TMR-based lockstep mechanism, it uses a DCLS mechanism. This solution is composed in the PS side by



(a) TMR, 2 hard-cores. **(b)** TMR, 2 soft-cores. **(c)** DCLS.

Figure 4.1: Design options for the lockstep architecture.

a hard-core Arm A9 processor, and in the PL side by the lowRISC soft-core beside a lockstep accelerator with checker capabilities. In spite of a DCLS solution has less fault coverage and is slower in error recovery than the other two TMR-based design alternatives, it promotes design diversity in the whole system. Such capability is not offered by the design in Figure 4.1a, while it requires less area overhead and development time than the TMR in Figure 4.1b. The DCLS design is the chosen one, once it has well-balanced tradeoffs between the system design diversity and the resources spent either regarding development time or needed area. Of the 3 designs, this one is the best suited for use in Lock-V architecture.

4.2 Architecture Overview

Lock-V is a heterogeneous architecture that explores a DCLS fault tolerance technique in two different processing units: a hard-core Arm Cortex-A9 and a soft-core RISC-V-based processor. The system, depicted by Figure 4.2, can be split into two main components: the software block and the hardware block.

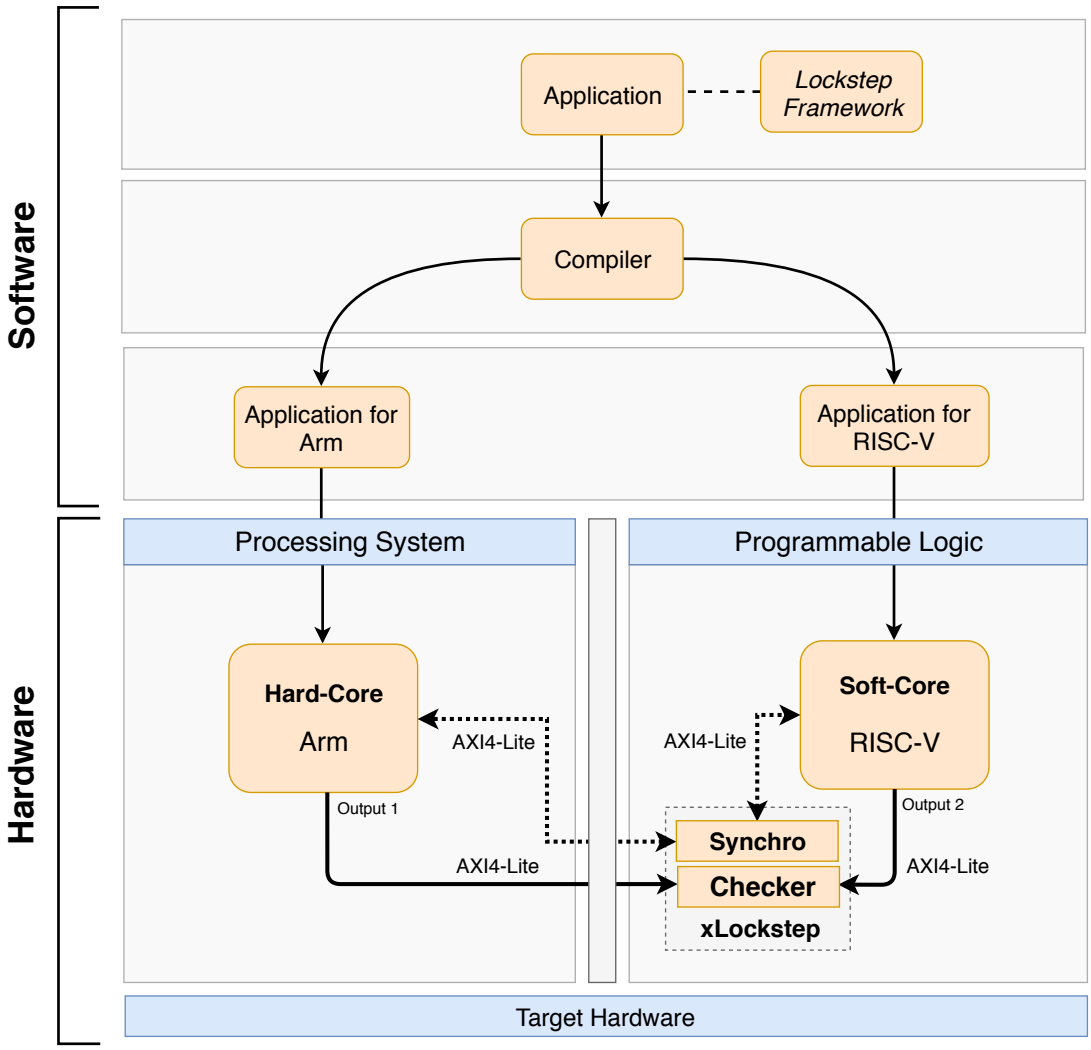


Figure 4.2: Proposed DCLS heterogeneous architecture.

Regarding the software, the Lockstep framework is responsible to generate the final machine binary code for a given application. The inputted software block, compiled for the two target architectures (Arm and RISC-V), was accordingly generated and patched from the same source code application. The framework also provides a set of functionalities in order to allow users to insert and configure execution checkpoints in the source code. The checkpoints are predefined verification points, introduced prior the compilation time, in order to endow the system with lockstep functionalities. Such checkpoints are essential for the auxiliary mechanisms of the DCLS architecture, the **Synchro** and **Checker** blocks. Their main tasks are, respectively, the synchronization of both cores and the verification of the processors' output in order to detect data integrity problems during code execution.

Regarding the hardware part, the Lock-V is divided into two main areas, the PS and the PL. The PS is mainly composed by a hard-core Arm Cortex-A9 processing unit and the associated software application. By its turn, the PL hosts a soft-core RISC-V processor (where the same software application also runs), and the hardware accelerators, which are responsible for deploying the lockstep functionalities performed by the **Synchro** and **Checker** sub-modules. The PS and PL execute concurrently and are both connected through a standard Advanced Microcontroller Bus Architecture (AMBA) protocol, the AXI, in order to exchange information among all hardware modules. The main hardware components of the Lock-V architecture are detailed as follows:

- **Arm Cortex-A9 processor:** a 32-bit processor that follows the ARMv7-A architecture and available in the PS as a hard-core processor. It runs the application machine binary code, in parallel with the soft-core processor.
- **RISC-V processor:** a soft-core processor deployed in the FPGA fabric of the PL and it also runs the application code. This 64-bit processor is based on the lowRISC, an untethered implementation of the RISC-V ISA based on the Rocket Chip.
- **Lockstep accelerator (*xLockstep*):** a hardware accelerator deployed in the PL following a loosely-coupled approach, which was developed under the specification of the Chisel hardware construction language [78]. Such approach provides several advantages when compared with the tightly-coupled design, such as hardware customization, flexibility and portability for using the *xLockstep* in other SoC and processor architectures. The *xLockstep* is responsible for the auxiliary lockstep mechanisms and its main tasks are: (1) the synchronization of the code execution on both cores; (2) the comparison and verification of the outputs from each processor; (3) the control on the code execution when the compared outputs are validated and coherent; and (4) the ability to suspend the processors' execution when an error is found, until the error is processed and marked as solved.

4.3 The lowRISC Adaptations

The lowRISC was adapted to fit in the specification of Lock-V architecture. The lowRISC has a NASTI and NASTI-Lite crossbar that is implemented in SystemVerilog. This crossbar was implemented by the promoters of lowRISC, to access MMIO peripherals, in order to make the lowRISC untethered. Although there is a NASTI-Lite crossbar, it has no extra interface instantiated that can be used for connecting the lockstep accelerator to the processor. For that reason, it was necessary to change the source code of the lowRISC to add a NASTI-Lite slave interface.

4.3.1 Adding A New Peripheral

With the purpose of adding a MMIO peripheral, two files were changed, one chisel file, which is part of the source code lowRISC processor, and one SystemVerilog file, which is part of the source code of the lowRISC NASTI-Lite crossbar. The chisel file was changed to offer a new entry into the address map and the second file was changed to add the new NASTI-Lite interface to the NASTI-Lite crossbar. Listing 4.1 contains the added code to the chisel file under the configurations of the lowRISC processor.

Listing 4.1: New entry added into lowRISC address map, changing the chisel file $TOP/src/main/scala/Configs.scala.

```
1
2 // address for the lockstep accelerator
3     entries += AddrMapEntry("xLockstep", MemSize(1<<16, 1<<30, MemAttr(AddrMapProt.RW)))
```

The previous code, assigns an address for the lockstep accelerator peripheral with 16 bits of size and 30 bits of alignment. The address was configured to be readable and writable through the MemAttr parameter AddrMapProt.RW. After the address for the new MMIO peripheral has been set in lowRISC, it was necessary to add a NASTI-Lite interface to the NASTI-Lite crossbar, as shown in Listing 4.2. The NASTI-Lite crossbar supports up to 8 interfaces, however, in the original lowRISC, only three interfaces were used, with five dummy interfaces free to futher usage.

Listing 4.2: New NASTI-Lite interface added to the NASTI-Lite crossbar, changing the SystemVerilog file $TOP/src/main/verilog/chip_top.sv.

```
1
2 //connecting  the lockstep accelerator
3 nasti_channel
4  #(
5  .ADDR_WIDTH  ( `ROCKET_PADDR_WIDTH       ),
6  .DATA_WIDTH  ( `LOWRISC_IO_DAT_WIDTH     ))
7 io_xLockstep_lite();
8
9 // IO crossbar
10 localparam NUM_DEVICE = 4;
```

```
11
12 // dummy channels
13 nasti_channel ios_dmm4(), ios_dmm5(), ios_dmm6(), ios_dmm7();
14
15 nasti_channel_slicer #(NUM_DEVICE)
16 io_slicer (
17           .master   ( io_cbo_lite   ),
18           .slave_0  ( io_host_lite  ),
19           .slave_1  ( io_uart_lite  ),
20           .slave_2  ( io_spi_lite   ),
21           .slave_3  ( io_xLockstep_lite  ),
22           .slave_4  ( ios_dmm4      ),
23           .slave_5  ( ios_dmm5      ),
24           .slave_6  ( ios_dmm6      ),
25           .slave_7  ( ios_dmm7      )
26           );
27
28 defparam io_crossbar.BASE3 = `DEV_MAP__io_ext_xLockstep__BASE;
29 defparam io_crossbar.MASK3 = `DEV_MAP__io_ext_xLockstep__MASK;
```

So, one of the five free interfaces was used to create a NASTI-Lite interface for connecting the lockstep accelerator. That was done in five steps: (1) a new device was instantiated through *nasti_channel* (*io_xLockstep_lite*) and the clock and reset signal were connected to it; (2) the number of devices connected to the NASTI-Lite crossbar was updated to four. They were three; (3) the used channel is no longer a dummy one, therefore it was removed from the list of dummy channels; (4) the previously created *nasti_channel* was added to the *io_slicer* in the channel four of the slave NASTI-Lite interfaces; and (5) the peripheral base address and mask were assigned to the labels that are used to, accordingly to the already used range of address, automatically generating an address. As a result of these steps, lowRISC was connected to a new peripheral, the lockstep accelerator, which can be accessed as a MMIO peripheral through its base address.

4.4 *xLockstep*

The *xLockstep* accelerator, depicted in Figure 4.3, is a memory-mapped AXI-compliant peripheral deployed in the PL, which implements auxiliary lockstep mechanisms. It has two slave AXI-Lite interfaces, one for each processor and an exclusive bank of registers dedicated for each processor, being their access restricted by hardware. Therefore, each processor can only access their register bank. This logic, composed of two slave AXI-Lite interfaces and the hardware to restrict the access to the register banks, is implemented in the top module of the *xLockstep* **xLockstep_AXI**. This top module is responsible for connecting the *xLockstep* accelerator to the processors and converting it into a MMIO peripheral. Between the **xLockstep_AXI** and **xLockstep** modules lies the **TopXLockstep** that convert the information in registers that comes from the processors into signals and data for being processed by the *xLockstep*. The accelerator has more three sub-modules, which are the building blocks of the *xLockstep*, i.e., two instances

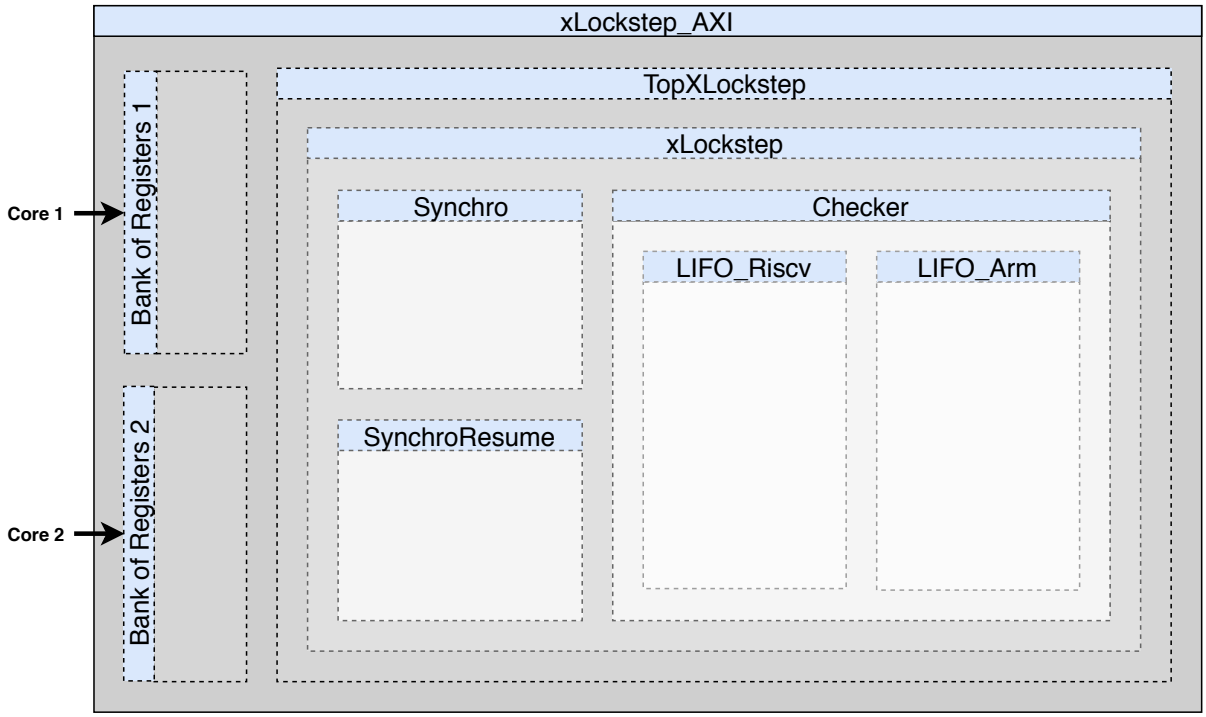


Figure 4.3: Design of the *xLockstep* accelerator with its modules and sub-modules.

of the *Synchro* and one of the *Checker* modules. The *Synchro*, *SynchroResume* and *Checker* modules are responsible for ensuring that both processors are synchronized, resuming the processors' execution after the end of *xLockstep* execution and comparing the outputs of both processors, respectively. The *Checker* owns another two sub-modules, two instantiations of the Last In First Out (LIFO) module, the **LIFO_Arm** and **LIFO_Riscv**, that are responsible, respectively, for storing Arm and lowRISC processors' data.

Figure 4.4 depicts the control logic of the Finite State Machine (FSM) of the *xLockstep* accelerator, which is composed by five states: *Idle*, *Synchro*, *Checker*, *Resume*, and *Error*. The FSM stays in the *Idle* state until the first checkpoint (from Arm or RISC-V processor) is reached. When this event occurs, the FSM changes the state to *Synchro* and waits for the second checkpoint of the second processor to be reached, until a programmer-defined timeout occurs. If that time is exceeded, an error by timeout in synchronizations is signalized and the FSM changes to the *Error* state. If the timeout is not exceeded, the FSM changes to *Checker* state. In the *Checker* state, a vector of processors' outputs are compared and if they are different, the FSM changes its state to *Error*. Otherwise, if the outputs are the same, then the FSM moves to the *Resume* state in order to resume the processors execution. When reaching the *Error* state, the *xLockstep* stays in that state until both processors signalize that the error will be corrected. After this, the system will be ready to resume its execution from a recovered health state.

4.4.1 Synchro

Due to the difference in clock domains and architectures between the soft-core and hard-core processors, the program execution between them is asynchronous, demanding for the synchronization of both

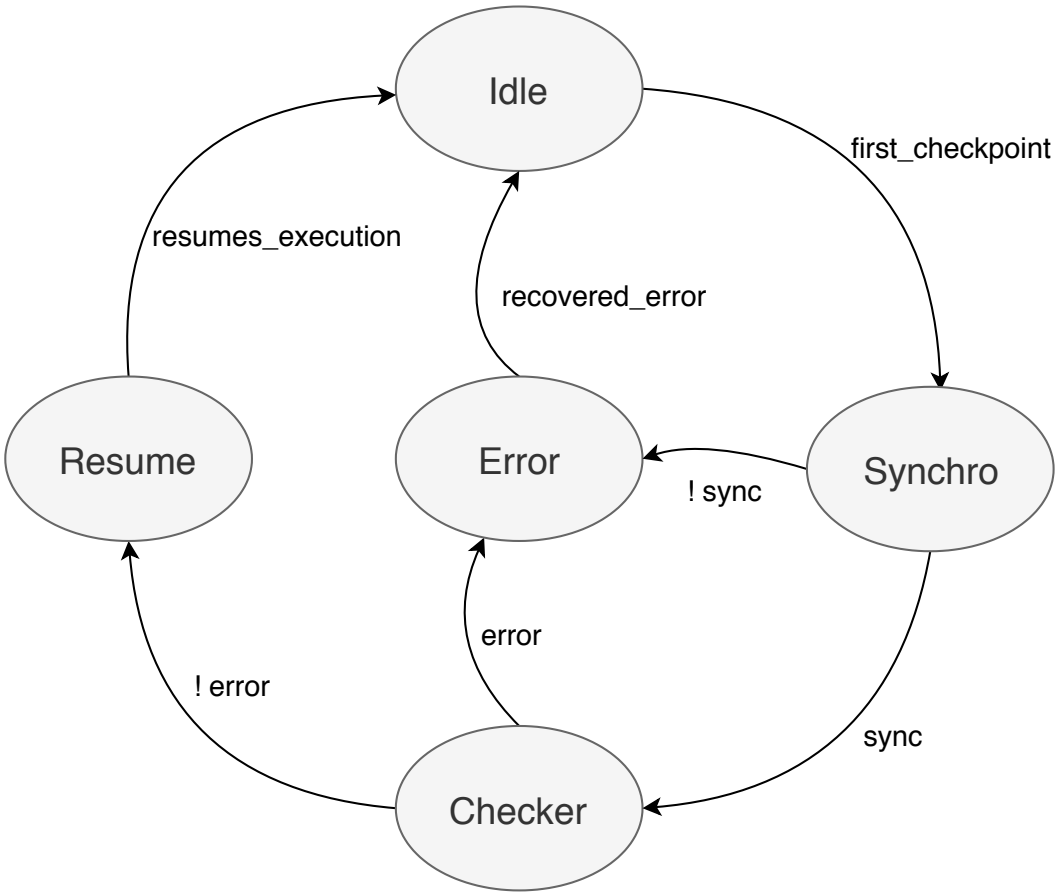


Figure 4.4: Main FSM of the *xLockstep*.

processors. For this purpose, it was created the *Synchro* module, which is used in two different scenarios. First, to synchronize the processors when a checkpoint is achieved, and second, to simultaneously return the code execution after the verification mechanisms of the lockstep have actuated. In both system operation scenarios, the *xLockstep* has to wait for both processors to indicate that they are ready to synchronize. This is achieved when: (1) the program reaches the checkpoint, and (2) both processors are ready to resume the execution. Therefore, to achieve those functionalities, the *Synchro* module implements a FSM with three states: *Idle*, *Ready*, and *Sync*. In the *Ready* state, the *Synchro* module expects both processors to enable the ***b_ready_to_sync*** bit, and afterwards, the *Synchro* gives feedback to both cores and enables the ***b_ready*** bit. Then, the state of the FSM changes to the state *Sync*. At this moment, the *Synchro* module is waiting for the synchronization's acknowledgment from both processors, which consists in disabling the ***b_ready_to_sync*** bit. As a result, the processors' synchronization ends and both cores are synchronized.

4.4.2 LIFO

For the *Checker* to perform the comparisons of the processors' data, they have to be stored. Therefore, a storage and data flow control method, which manages the writing and reading of data vectors, has been implemented. The most often used methods are the First In First Out (FIFO), that is like a queue, and the LIFO, that is like a stack. The first needs two pointers, one for reading data and another for writing data, while the second only needs one, that points to the top of the stack. The LIFO was the chosen method to store and control data flow because it fits in the *Checker* data requirements while using only one pointer which makes its implementation simpler. For the *Checker*, the data comparison order does not matter. The only concern is that each equivalent element of each LIFO must be compared at the same time. However, it does not matter, if the first comparisons are made with the last or the first stored data, once that all the data must be compared. When a LIFO is used, if for any reason the numbers of stored data are different, e.g., if one core sent five units of data and another core just sent four, an error is given in the first comparisons because the fifth element is different from the fourth one. This avoids all data from being compared. If a FIFO approach is used, the error would be detected only in the comparison of the last FIFO elements.

Listing 4.3 contains the interface of the module LIFO, which was implemented in chisel. The LIFO has a data width of 32 bits and owns, beyond the data I/O ports, 4 input signals and 2 output signals. The input signals, **iPop**, **iPush**, **iFlush** and **iEn** are responsible for controls the module operational mode. When the signal associated with pop is high[1], the last element of the LIFO is removed from it and putted in the output data port. In contrast, when it is needed stored data, the the signal associated with the push is set to high and the value in input data port is stored in the top of the stack. When all the stored data must be removed, the signal associated to the push is set to high, and the LIFO will be cleared. Enabling or disabling the LIFO through the enable signal, just must be done after changing the other input signals. This signal is needed because once the LIFO is a pure combinational circuit, if the module is always-on, then the push, pop or flush will always occur and the data is not stored. For correct working, the enable signal should not be high more than 1 clock cycle. If this is not respected, the LIFO performs more then one push, pop or flush (one for every clock cycle that the enable signal is high). The output signals, **oEmpty** and **oFull**, carry information about the module, as the names suggest, when the LIFO has all the storage capacity used, the full signal stays high, and when the LIFO has no data, the empty signal stays high.

Listing 4.3: LIFO Module interface, which was implemented in chisel.

```
1
2 class LIFO extends Module {
3     val io = IO(new Bundle {
4         //Inputs
5         val iData = Input(UInt(32.W))
6         val iEn = Input(Bool())
7         val iPop = Input(Bool())
```

[1] Every time that is used the term **high**, that means the bit has the logical value of 1.

```
8         val iPush = Input(Bool())
9         val iFlush = Input(Bool())
10        //Outputs
11        val oData = Output(UInt(32.W))
12        val oEmpty = Output(Bool())
13        val oFull = Output(Bool())
14
15    })
16  ...
17 }
```

4.4.3 Checker

For implementing the lockstep mechanism, the processors' output have to be compared. For that purpose, each core sends its output vector to the *Checker* module in order to perform their verification. The received outputs are stored in two different memory regions (one for each processor) by the *Checker* using a LIFO approach. Because both parts are involved in the data transfer process (processors and *Checker*), both of them need to know the state of each other. For that, the *Checker* uses a control bit, **b_Tx**, to coordinate the data transfer, working in the following ways. Firstly, when the *Checker* is available to receive and store an output, it puts its **b_Tx** bit to 0, signalizing the processor that it is available to perform the transaction. Next, it waits for the processor to signalize its availability to initialize a data transfer. After the data transfer, the *Checker* module clears the **b_Tx** bit and it becomes ready for another transaction. Secondly after the data is received from both processors, at a given checkpoint, the *Checker* performs the comparison of the entire LIFO contents, checking for data integrity errors. There are two possible error cases that can be detected and signalized by the *Checker* to both processors. The first case occurs when an element from LIFO 1 is different from the respective element from LIFO 2. The second case results when the number of written outputs in both LIFO memories are different. The *Checker* LIFOs work as a circular buffer with limited size. Therefore, if one processors' output vector size can not be accommodated by its respective LIFO, the *Checker* signalizes to the processor a busy state. This way, the *Checker* module is unaware of the data size and content, being the main concern only its storage and comparison. While the data is being processed, the processor waits for the *Checker* confirmation to allow new data to be transferred (for the next checkpoint or for repeating the previous one).

4.4.4 *xLockstep* AXI-aware Interface

To enable the connection of the *xLockstep* to the processors, two slave AXI-Lite interfaces were used, each of them with a set of eight 32bits registers. In order to keep the integrity of the system and prevent processors from accessing the registers that do not belong to them, the access to the bank of registers has been restricted. To achieve this, *xLockstep* registers have been replicated, and now each processor has its unique bank of registers. Each slave AXI-Lite interface was mapped in a different region of memory.

Although the *xLockstep* is the same for both cores, they see it as a different MMIO peripheral, since each AXI-Lite interface is mapped in one unique memory region. Figure 4.5 depicts the bank register associated with the Arm and RISC-V processors, on base addresses **0x83C0_0000** and **0x8000_0000**, respectively. The set of registers are made of eight registers, although four are unused. The minimum number of registers needed is four (4 address bits), but in order to have a margin for future improvements, it was used 5 bits for the register's address (8 registers of 32 bits), justifying the four unused registers. The other registers are:

- **x_DATA_REG** [offset 00] - is used to send data of 32 bits from the processors to the *xLockstep*. It is through this register that the processors send, for being compared, their outputs' vector;
- **x_CONTROL_REG** [offset 04] - is used to send control signals to the *xLockstep*, in order to: (1) enable the use of the lockstep accelerator; (2) synchronize the processors; (3) control the data flow between processor and accelerator; (4) resume the processor execution after the auxiliary lockstep mechanisms execution; and (5) recover the processor from an error.
- **x_TIMEOUT_REG** [offset 08] - is used to configure the maximum time between processors checkpoints, i.e., if processor A reaches the checkpoint, the processor B has the maximum time defined by this register to achieve its checkpoint, otherwise, if the timeout is exceeded, an error is signalized. Just exists one value for the timeout, but to be consistent in design, both register banks have a replica of the timeout register, allowing each individual processor to set the timeout. In case of both processors tries to configure the register and the values are not equal, the timeout used is the one with the smallest value.
- **x_STATUS_REG** [offset 1C] - is from this register that the processors receive all the feedback given by the *xLockstep* for example, if the accelerator is busy or if an error had occurred either in

83C0001C	ARM_STATUS_REG	8000001C	RISCV_STATUS_REG
83C00018	UNUSED	80000018	UNUSED
83C00014	UNUSED	80000014	UNUSED
83C00010	UNUSED	80000010	UNUSED
83C0000C	UNUSED	8000000C	UNUSED
83C00008	ARM_TIMEOUT_REG	80000008	RISCV_TIMEOUT_REG
83C00004	ARM_CONTROL_REG	80000004	RISCV_CONTROL_REG
83C00000	ARM_DATA_REG	80000000	RISCV_DATA_REG

Figure 4.5: The *xLockstep* peripheral memory address space.

the process of synchronization or when the *Checker* verifies the processors' outputs.

After *xLockstep* registers were replicated, access to them was restricted by hardware to ensure that only the respective processor in each registers bank has access to them. For doing this, the two AXI-Lite interfaces generated by Vivado had to be changed. Two signals **wValidWriteAddr** and **wValidReadAddr**, illustrated in Listing 4.4, were added for restricted the access.

Listing 4.4: Signals for restrict the access to the Arm registers bank. Two equal signals were used in the RISCV AXI-Lite interface.

```
1    assign wValidWriteAddr = (S_AXI_AWADDR[31:5] == ARM_BASEADDR[31:5]) ? 1'b1 : 1'b0;
2    assign wValidReadAddr = (S_AXI_ARADDR[31:5] == ARM_BASEADDR[31:5]) ? 1'b1 : 1'b0;
```

These signals are needed because, by default, the AXI-Lite interfaces just check the lower bits of the read or write address (the register offset), in this case, the lower 5 bits, while the upper bits are ignored. Regardless of the base address, the AXI-Lite interface always takes the offset of the register from the address lower bits, reading or writing on it. The new signals are in a high state when the base address of the AXI-Lite interface matches with the base address that is used by the processor to access the *xLockstep*. Each AXI-Lite interface has a parameter, **x_BASEADDR**, that allows the user to change the base address of each AXI-Lite interface. So despite on this thesis the used base addresses are **0x8000_0000** and **0x83C0_0000**, for futures designs, any base address can be used. Either **wValidWriteAddr** and **wValidReadAddr**, are used throughout the AXI-Lite interface source code for enable or disable the reads and writes in the registers, depending on whether the base address is correct or not. If one processor tries to access a registers bank that does not belong to it, a **DECERR** error is generated, which is a decode error to indicate that there is no slave at the transaction address.

4.5 *xLockstep* deployment in Lock-V

For adding the *xLockstep* peripheral to the Lock-V each processor must have a free AXI interface to be connected to the peripheral. As described in Section 4.3, the original lowRISC was adapted to incorporate an extra NASTI-lite interface, which is compliant with the AXI-lite interface. For the Arm to be connected to the accelerator, it was configured to use the free AXI high performance slave interface. After the configuration of these two interfaces in both cores, they become connected to *xLockstep* and the Lock-V architecture is now completed. It features a lowRISC and an Arm Cortex A9 processors in lockstep, which is guaranteed by the *xLockstep* accelerator. The Figure 4.6 depicts a part of Lock-V design without the lowRISC, because this processor does not own an Intellectual Property (IP) interface and so it is used in the design as a verilog file. This is the setup used by the lowRISC, one core in verilog plus a logic that interact with the Arm processor, but does not interact with the lowRISC. The lowRISC implemented its design this way, because it was not projected to work in lockstep, so the core does not need to interact with the Arm side once their executions are independently from each other. The Arm side Lock-V design has the

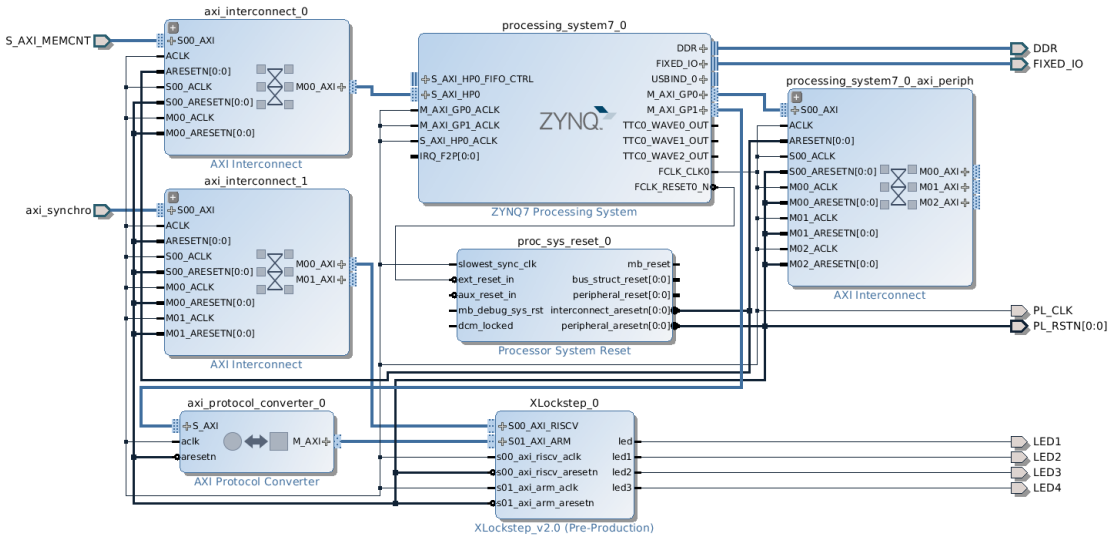


Figure 4.6: Lock-V design (Arm side).

connection to the PS (Arm processor) and the lockstep accelerator as well as an input bus that is connected to the lowRISC interface, which is reserved for the *xLockstep*. It is through this bus that both sides of the Lock-V designs are connected. This connection is made in one wrapper that has instantiated the lowRISC and the Arm side design. The NASTI-Lite interface is passed, through the **axi_lowrisc** interface, from the lowRISC to the Arm side, where is used as an AXI-Lite interface.

4.6 *xLockstep* API

The *xLockstep* API is a tool that will help programmers to easily configure and interact with the *xLockstep* accelerator in order to deploy its application in the Lock-V architecture and running it in a lockstep fashion. This API is processor-agnostic, and it can be used either in the Arm processor or the RISC-V processor. The only change needed in the use of the API between the cores is the macro that defines which processor is using the API. Two macros, **__ARM** and **__RISCV** are used to specify the processor targeted by the written code. This is required to change the base address used by the API, because each processor sees the *xLockstep* as a different peripheral, once it is mapped in different memory region for each processor. So, each core must define its base address that is associated with its *xLockstep* AXI interface in order to access its unique bank of registers. The *xLockstep* API is composed of the following four functions, which are used to interact with the *xLockstep*:

- `initXLockstep()` is responsible to setup and initialize the *xLockstep*, as well as all the memory address space registers for each processor (Figure 4.5). This function also sets the timeout value for the next checkpoint.

- **`sync()`** is used for processors' synchronization. The *sync* function starts to verify two things: (1) if the *xLockstep* is busy to perform the cores' synchronization or not; and (2) if there is an error. After these verifications, if any error had occurred and the *xLockstep* is not busy, the *sync* set the checkpoint bit to high, informing this way the *xLockstep* that the code reached one checkpoint. At this moment, the cores' synchronization starts, and the *sync* waits for the end of it. When the cores are synchronized, the bit checkpoint is clear, and the synchronization process ends. In case of failed synchronization, an error is returned.
- **`checker()`** function is responsible for handling the *Checker* functionalities, returning an error if the processors' output, reported by the *Checker* module, are different. The *checker* function owns two input parameters, a data vector to be compared, and the size of that data vector. First, the *checker* informs the *xLockstep* that it wants to start new writing of data, setting the **bitStart** to high. After this, the *checker* waits for the *xLockstep* acknowledgment in order to start the writing. When this confirmation is given, the *checker* sends all the data vector to the *xLockstep*. At the end of the writing, the **bitStart** is clear, and the *checker* is ready to resume the code execution. At this moment the *checker* waits for the *xLockstep* confirmation that everything went well for the execution to be returned. If the data vector that each processor sent to *xLockstep* is different, an error is returned by the *checker* API function.
- **`errorFixed()`** is used to inform the *xLockstep*, before the error recovery processing, that the system will be changed for a state without errors. When an error occurs, the programmer should define the desired behaviour, according to the application needs and call this function before the error recovery for signalizing the *xLockstep* accelerator that the error will be processed.

The *xLockstep* hardware accelerator is a memory mapped peripheral, consequently, for the API functions to be implemented, they must access to the *xLockstep* registers. That *xLockstep* registers access was implemented through three internal functions that feature all the needed registers operations: read a register, write in a register and write in one bit of a register. For read a register, the API offers the internal function **read()** shown in Listing 4.5. It has one input parameter that is the address of the register to be read. The function points to the address passed into the parameter and returns its value.

Listing 4.5: Internal API function for read a MMIO peripheral register.

```
1 static unsigned int read( unsigned int addr){
2     unsigned int *ptr = addr;
3     return *ptr;
4 }
```

To write a 32 bit data in one *xLockstep* register, the internal API function, **write()** shown in Listing 4.6 is used. It requires two input parameters that is the address to be written and its respectively value. This function does the opposite of the previous one. It points to the address passed into the first parameter but instead read, it writes a value on it.

Listing 4.6: Internal API function for write in a MMIO peripheral register.

```
1 static void writeData(unsigned int addr, unsigned int value){
2     unsigned int *ptr = addr;
3     *ptr = value;
4     return;
5 }
```

To write an isolated bit of a register, e.g., when it is needed to manipulate a control register, the internal API function **writeBit()** shown in Listing 4.7 is used. It has three input parameters, the mask for the bit to be written, the address of the register and the value to be written in the bit. The function points to the address, and for writing "1" in the bit, it is performed a bitwise OR with the mask and for writing "0" in the bit, it is performed a bitwise AND with the complemented mask.

Listing 4.7: Internal API function for write in a bit of a MMIO peripheral register.

```
1 static void writeBit(unsigned int addr, unsigned int mask, int value){
2     unsigned int *ptr = addr;
3     if(value)
4         *ptr |= mask; // write 1
5     else
6         *ptr &= ~mask; // write 0
7     return;
8 }
```

These three functions are responsible for the interaction with the *xLockstep*, at a more low level. Whenever one of the four functions provided by the API is used to read or write to the accelerator, the previous internal functions are used to interact with the *xLockstep*, and pass information or read information from it. This information flow is performed using four 32 bit data registers depicted in Figure 4.5. Two registers the **x_Control_reg** and the **x_status_reg** have their information encoded in their bit-field. The first aforementioned register has five bits, depicted in Figure 4.7, to control the accelerator actions and second one has ten bits, depicted in Figure 4.8 to provide information about it. Whenever the *xLockstep* API functions want to trigger an action in the accelerator, they have to write in the bit-fields of **REG_CONTROL**. The **bCheckpoint** bit-field is used to indicate that a checkpoint was reached. The **bStart** bit-field is used to indicate that the processor wants to send a data vector. The third bit is used to control the flow of data. Whenever it sends a 32 bit data the **bTx** bit-field is set to high. The **bReady2Resume** bit-field is used to inform that the processors has sent all the data and is ready for resuming its execution, if no error had happened. The **bErrorFixed** bit-field allows the processors to inform the *xLockstep* that the error was recovered and this way the accelerator can keep going with its lockstep functions.

Whenever the *xLockstep* API functions want to get the feedback of its action and the status of the accelerator, they have to read the register **REG_STATUS** and decode its bit-fields. The **bSynchroBusy** bit-field is set when the *xLockstep* is performing another action, like checking the outputs or resuming the cores' execution and is busy doing the synchronization. The **bSynchroBusy** bit-field is cleared when

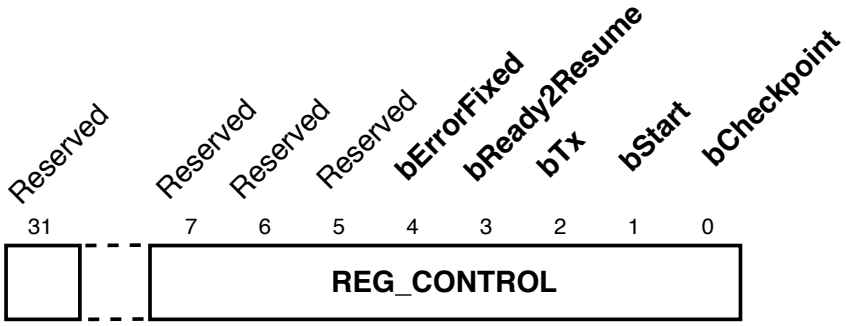


Figure 4.7: Control register field.

the synchronization ends and both cores are in sync. The **bArmTx** and **bRiscvTx** bit-fields are set when the *xLockstep* receive and store a 32 bit data from the processor Arm or from the lowRISC. The **bCheckerBusy** bit-field is set when the *Checker* is comparing the stored data of the Arm_LIFO and RISCV_LIFO or when the LIFOs are full. The **bReady2Write** bit-field is set when the *Checker* is ready to receive data from the Arm or RISC-V processor. The **bArmBusy2Write** and **bRiscvBusy2Write** bit-fields are set when the Arm or RISC-V processor are not writing data, means that the other processor is writing data or the *Checker* is compared outputs. The first processor to write wins the writing access, so the other need to wait for the end of the data writing. The **bResumeExecution** bit-field is set when, after the checker compares the processors' data and detects an error, both processors are synchronized for returning at the same time the code execution. The last used bit-field, **bError**, as the name indicates is set when an error is detected either by the *Checker*, reporting divergent outputs, or by the *Synchro*, reporting inability to synchronize the cores. All the registers as well as their bit-fields are defined in the *regxLockstep.h* file for being used by the *xLockstep* API.

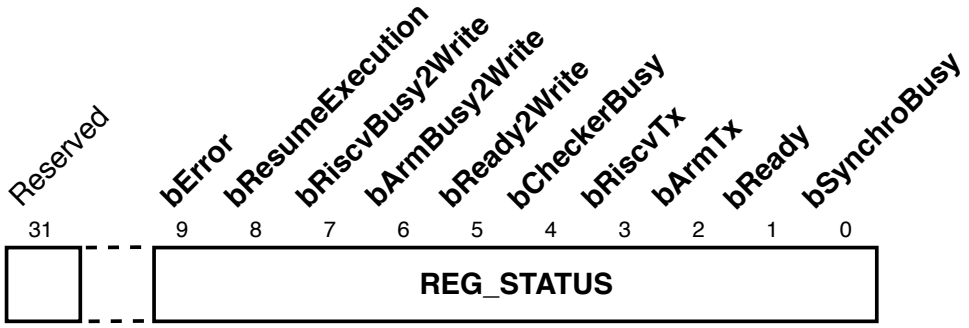


Figure 4.8: Status register field.

5. Lock-V Framework

This chapter addresses the framework for the proposed fault tolerance architecture, Lock-V. This thesis has as main goal the development of a heterogeneous fault tolerance lockstep-based architecture. This is split into two parts, the implementation of hardware for error detection, addressed in the previous Chapter 4, and the implementation of a framework that leverages software-based system recovery, after error detection. This chapter presents the lock-V framework, a toolchain that supports the development of an application, running in lockstep, in the Lock-V architecture. At the beginning of the chapter, it is explained how the framework works and how it should be used. In the next section, it is explained how the framework uses the *xLockstep* API in order to detect an error. Afterward, it is present how the error recovery is performed through saving the processor context and rollback technique. In the last section, it is detailed the constraints to use the Lock-V framework.

5.1 Framework Overview

As it was explained in 2.1.3.1 a fault tolerance system must provide two main services. It detects an error and recover from it by changing the system from an erroneous state to a error-free state. The chosen technique for the Lock-V, DCLS, typically uses as error recovery the rollback technique. This technique is responsible to correct an error avoiding its propagation to a system failure. With this in mind, it was developed a framework that model, through three C functions, the behavior of a system with error detection and error recovery. In order to detect an error, the framework uses the *xLockstep*. For recovery from an error, the framework uses software techniques. Both error detection and recovery are addressed respectively in Section 5.2 and 5.3. Figure 5.1 depict the execution flow of an application running in DCLS-fashion in the Lock-V architecture using the Lock-V framework. The framework offers three main tools, providing services with the same names of tools:

- **checkpoint**, which is responsible for verifying the integrity of the code execution;
- **saveContext**, which is responsible for saving the processor's context. This tool should be used when the previous one verifies that the system is in an integrity state;
- **rollback**, which is responsible for restoring the processor's context to the last known integrity state. It should be used when the checkpoint tool confirms an error occurrence in the system.

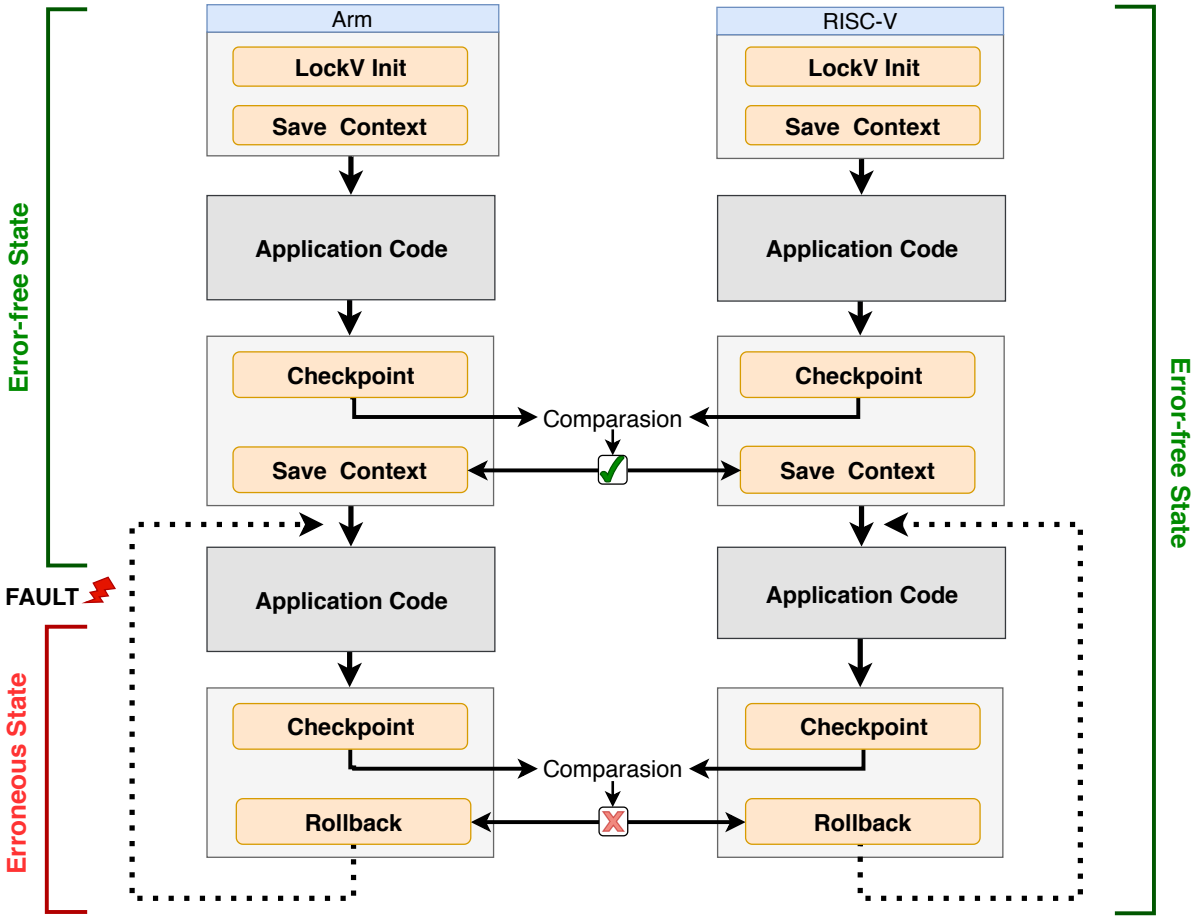


Figure 5.1: Flow execution of an application running in Lock-V, coded using the Lock-V framework.

First of all, the framework needs to be initialized using the **initLockV()** function, and soon after that, the function of the **saveContext** tool must be used. The application code can only be written after doing this initial configuration. To ensure the system integrity under rollback capability, it is important to make the initialization code the entry-point of the system execution. Putting it simply, the application code must be executed only after or patched after it at design time. The checkpoint tool must be always used when programmers want to verify the code integrity. Checkpoints should be patched at critical code points. For example, when is sent a command message to another component or when the application interacts with the external world (surrounding environment). Although the framework's tools can be used separately, they are designed to work together. In doing so, the application code must be accordingly patched under the framework control to instantiate calls to appropriate tools' services. For instance, when the checkpoint detects an error, the rollback function should be invoked, and when no error is detected, the **saveContext** service should be invoked. This ensures that in each verification block, the integrity is checked. If the system has no errors, its state is saved. Otherwise, when there is an error, a system recovery, is performed. Afterward, the system change to a error-free state, which was previously saved. Lock-V framework enables fault tolerance capability to applications built upon a mechanism in hardware for error detection (DCLS-based) and another mechanism in software for error recovery (rollback-based).

5.2 Error Detection Capabilities

As said above, to verify the integrity of the system, the Lock-V framework provides a tool for error detection. This tool is called checkpoint and resort to the *xLockstep* accelerator, presented in the previous Chapter 4, to perform the error detection. This is done through the use of the *xLockstep* API that allows synchronizing both processors, comparison of their outputs, and simultaneous error recovering in both processors. The processors' outputs are compared, and if any data integrity faults are detected, the system stops working to enter into an error recovery state.

5.2.1 Checkpoint

The checkpoint execution flow is depicted in Figure 5.2, which details how the checkpoint is performed. The checkpoint tool is built upon the *xLockstep* API. It starts to synchronize both cores and checking for a timeout error. If this error occurs, it means that the code was stuck at some execution point. For preventing the system from crashing or entering in an infinite loop, an error is signalized by the checkpoint. On the other hand, if the synchronization succeeded, the checkpoint execution will proceed. At this point, the

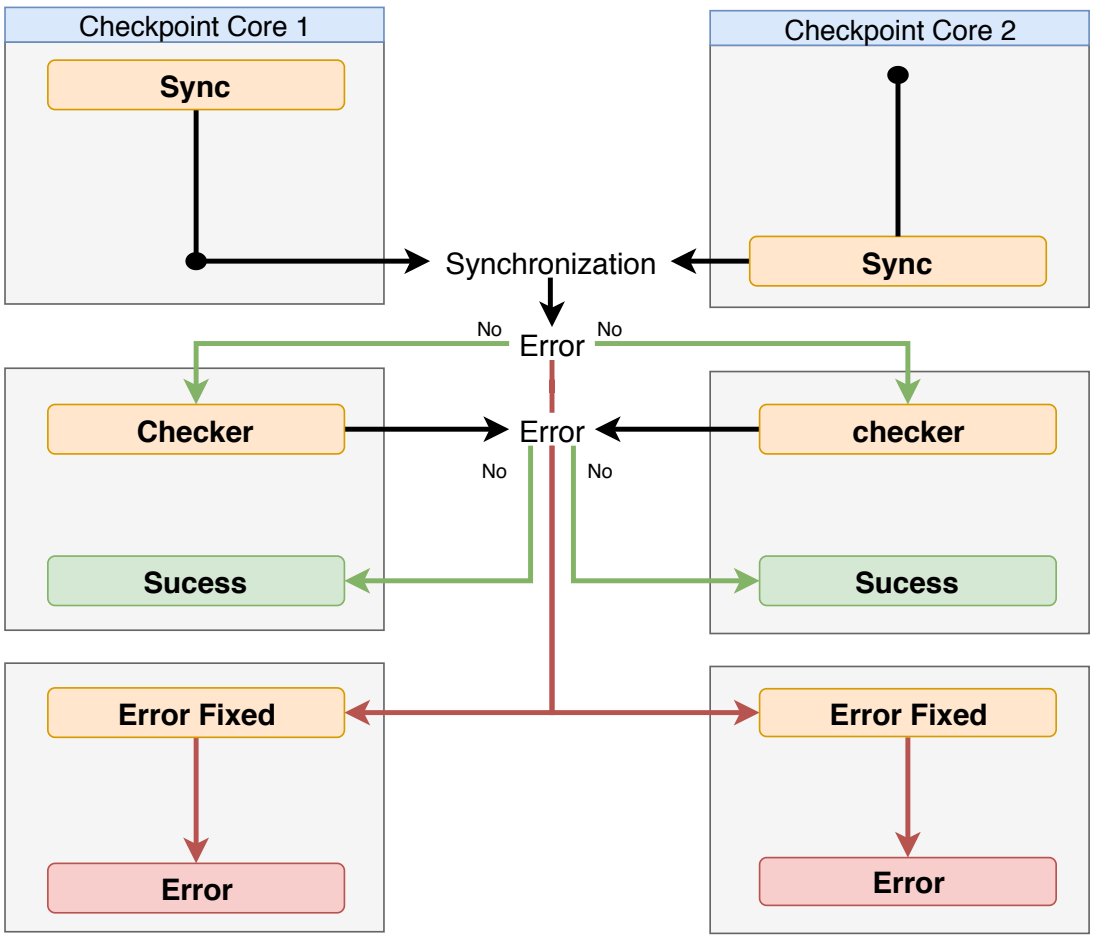


Figure 5.2: Flow execution of the checkpoint tool.

xLockstep's checker module will start sending the programmer's chosen data to the verification module. If no error in the data is detected, the checkpoint will return with a *success* signal. Otherwise, an error due to data' mismatch is signalized. When an error is signalized, either by the synchronization or the verification phase, the **errorFixed()** function will be invoked. When both processors reach this stage, the checkpoint returns an error.

5.3 Error Recovery Capabilities

On concluding the error detection phase with an error, the system must be recovered by bringing it to an error-free state to operate normally. To achieve that goal, the system needs to have some kind of error recovery capabilities. The Lock-V framework has implemented that through rollback technique. This brought some challenges as:

1. How should the system be backed up? It must save a system state that has no active errors;
2. How and what data should be stored? If the amount of data is very high, the fault tolerance system overhead can increase drastically;
3. What is the minimum amount of data needed for ensuring the system can be restored?

For a fully and integral rollback, all the application code and data should be saved. However, this requires a lot of redundant memory as well as increases the silicon overhead of the redundant system. Furthermore, the time wasted to perform a fully context save and rollback is high. Furthermore, this thesis assumes that memory is safe and protected through possibly one of the well-known and effective techniques, such as, ECC or TMR-based memories, among others [79, 30]. For this reason, protect memories errors is out of the scope of the Lock-V rollback. Having excluded memories from the possible causes of lock-V errors, the most likely source of errors are the processors' register files. Those errors may occur due to SEU that origin bit-flips in the registers of the processor. The register file have been identified as one of the most critical part of the processor [80, 81, 82, 83, 84]. So to protect the Lock-V registers' files, was implemented a rollback mechanism. Doing only the protection of the register file rather than all memory, presents two main benefits: (1) one of the most critical part of the processors, register file, is protected. When a fault is triggered in the register file and causes an error in the system, the register file is restored to a fault-free backup; and (2) save and restore the register file requires few saved data amounts. and hence the save context and rollback will be faster and lightweight.

Although the memory can be protected from external faults, e.g., with ECC memories, they can be affected by the propagation of the faults that appear in the register file [84]. This can be worst in the Lock-V architecture because both processors are load/store architectures. All the instructions are between registers or register and memory. This means that in all the instructions that involve memory, the registers are used. So, the memory can be affected by faults in registers. To avoid the error propagation from the registers to the rest of the memory, the use of memory was restricted. Just the memory directly used by the registers can be used. This means that the programmers only can use local variables. This type of

variables are stored either in registers or in the stack. So, beyond the protection of the register file, the rollback also needs to protect the stack. Performing stack and register file backup is the least amount of saved memory that ensures the operation of a rollback. This prevents those register file faults from propagating to the memory and a lightweight rollback implementations can be achieved.

5.3.1 Save Processors' Context

As mentioned above, the saving of processors' context is done by backing up the processors' register file and stack. Although the logic behind saving the processors' context are the same for both cores, the registers file are different. While the Arm processor has a register file of sixteen registers, RISC-V processor has one with thirty-two. The Arm own twelve general-purpose registers, r0-r12, plus three special registers, the r13 is the Stack Pointer (SP), the r14 is the Link Register (LR), and the r15 is the Program Counter (PC). The general purpose r11 register is usually used as a Frame Pointer (FP). The RISC-V processor has thirty-two registers in the integer register Application Binary Interface (ABI) convection. The x0 register is hardwired zero, so it does not need to be saved. The x1-x4 are special registers: x1 is the Return Address (RA), x2 is the SP, x3 is the Global Pointer (GP) and x4 is the Thread Pointer (TP). The other registers are all general-purpose registers. The presence of a FP is optional, however, if it exist then it must be mapped in x8 register. In the RISC-V register file, there is no PC. The PC is an extra register that can not be directly accessed. So, for the rollback to be done successfully, r0 to r15 registers in Arm processor and x1 to x31 plus the PC registers in RISC-V processor need to be saved. In addition, the stack of each processor must be saved. The Arm processor stack is implemented as full descending stack aligned at 4 bytes. This means that in each push for stack, the SP decrements 4 bytes. In a full descending stack first the SP is decremented and after that the data is stored (push). The RISC-V processor stack is also implemented in a full descending but instead of being aligned to 4 bytes is aligned at 16 bytes. This stack alignment is used because the RISC-V ISA supports word-width up to 128 bits. Although the lowRISC is a 64-bits processor, the alignment of 16 bytes has to be maintained in order to make context saving and rollback compliant with the RISC-V ABI. This means that in each load to the stack, SP is decremented 16 bytes and after that, the data is stored. Doing a copy of the RISC-V stack, for the same amount of stack data, is four times more costly. This means that if the stack has for example 10 data units, it is needed 40 bytes for saving the Arm stack and 160 bytes for the RISC-V stack. So, for saving the registers' files and stacks two hardware IPs were created. The IP for saving the Arm context is by default 464 bytes of storage (16×4 bytes for saving the 32-bit register file plus 100×4 bytes for the stack). The IP for saving the RISC-V context is by default 1728 bytes of storage (32×8 bytes for saving the 64-bit register file plus 100×16 bytes for the stack).

Figure 5.3 depicts the procedures for the context saving. First, when the **saveContext** service is invoked, the main FP and the LR are saved on the function stack. This is done in any function call and is introduced by the compiler when it translates the C code for assembly. Afterward, a register file copy is carried out and the LR and the FP is copy from the function stack to the saved register file. After that,

this saved register file is stored in the *ContextSave_Arm* IP. Now, a full copy of the register file is already stored. The next step is to make a copy of the stack. For this to be done, the main FP and SP were used. In order to copy all the stack data, a pointer is assigned with the value of the base of the stack (hold by the FP). After that, another pointer is assigned with the top of the stack (hold by the SP). Afterward, a third pointer is assigned with the base address of the saved stack. When these pointers are set, the stack is then copied byte to byte to the saved stack until the base pointer (r1) be equal to the top pointer (r0). Now there is a copy of the register file and the **main()** stack. These allow a future rollback for this safe point of the program execution.

5.3.2 Rollback Processors' Context

In order to perform the processors' rollback, the stack and register file need to be restored. These are done by copying the stored stack and register file that were previously saved by the **saveContext** service or function. First of all, the stack needs to be restored, and just after that, the register file. Contrarily

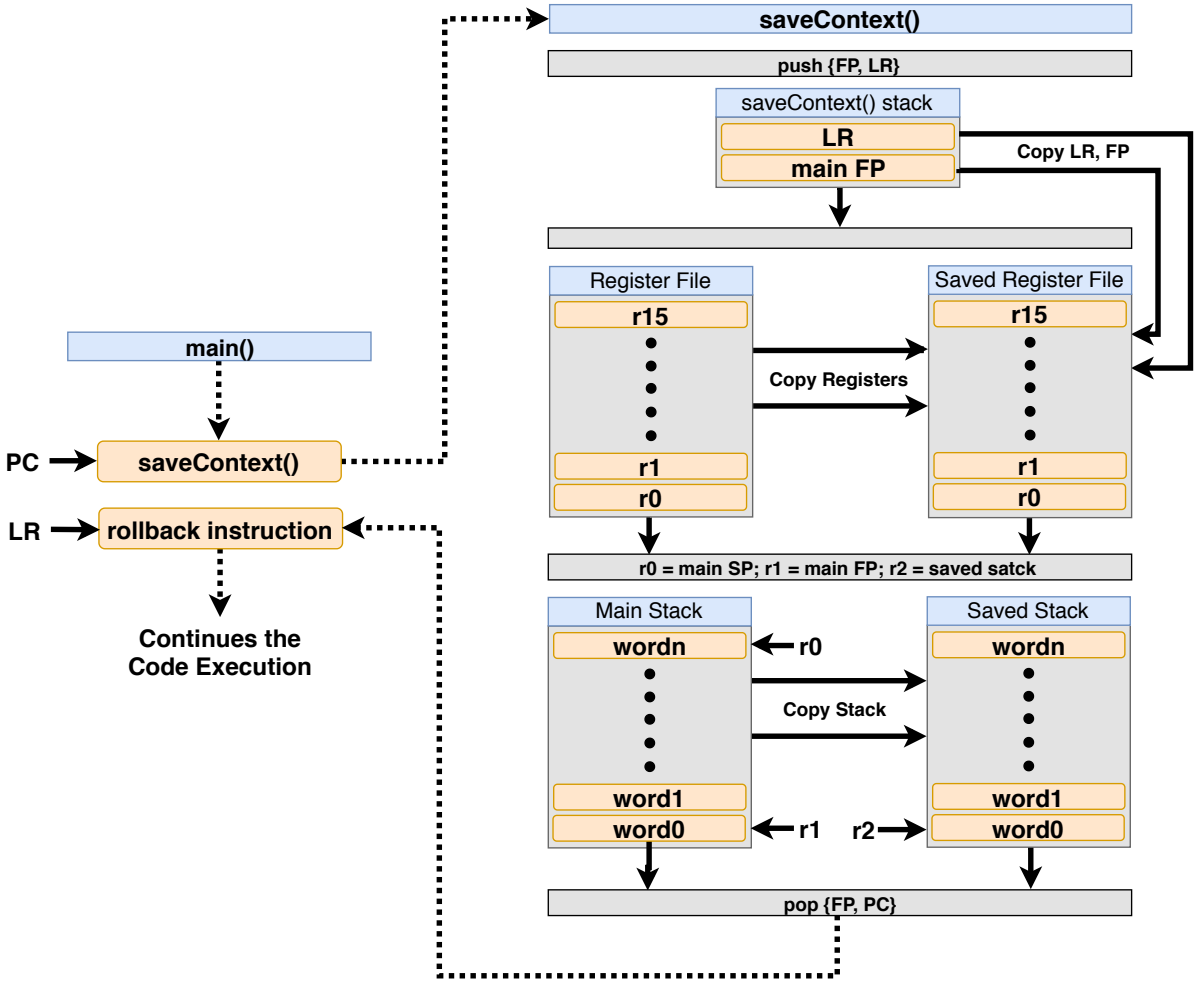


Figure 5.3: Example of the Arm save processor's context. Although the number of registers and stack alignment are different, the logic in the RISC-V save processor's context is the same.

to the **saveContext** service or function, to perform the rollback service or function correctly, the stack must be restored first. This should be done in this order (restore stack and after restore the registers), because to restore the stack it is needed to use some register for copying the data from the storage unit to the memory allocated to the processor stack. If the register file is restored first, some registers would be corrupted with improper data. Regarding the type of error, two different types of rollback can be performed. One that is performed when the *xLockstep* detects an error an another when the processor detects an incorrect operation like, execution of undefined instructions, load or store data at illegal address, among others. The first rollback is performed when a bit-flip occurs in the register file and affects the processor outputs. This means that when the bit-flip occurs, the execution or data flow of a program is changed, and its outputs are different from the expected. When this occurs, a mismatch between the processors' outputs is detected by the *xLockstep* and a rollback is performed. This rollback occurs due to the lockstep mechanism. The second rollback, which we named rollbackAbort, is performed when a bit-flip change some special registers, like SP or PC. These bit-flips may cause the processor to read a wrong data, tries to access nonexistent memory, or execute wrong instructions. When these faults occur, the processor goes to an exception (in Arm) or a trap (in RISC-V). When an error is detected by the *xLockstep*, a rollback needs to be performed in both cores because it is impossible to know each of them is the source of the error. However, when an exception occurs, the rollback just needs to be performed in the erroneous core, once is known which core triggered the exception.

Figure 5.4 depicts the steps to rollback the processors' context to a safe state. At the beginning, after the **rollback** service or function be invoked, the FP and the LR are load to the stack. The **rollback** function for the stack is created to follow the calling convection, but these values will not be used. Nevertheless, this stack will be harnessed to restore the FP and the PC. That will be done by overwriting the FP and LR with the saved FP and LR. After the **rollback** private stack be created, the main stack is restored using a similar process to the way that it was saved. A pointer is assigned with the value of the base of the stack (saved FP), other is assign with the top of the stack (saved SP), and another is assign with the base address of the saved stack. After that, the stack is restored copying all the bytes of the saved stack until the base stack pointer (r1) is equal to the top stack pointer (r0). Next, the register file is restored from the saved register file. Afterward, it is performed a pop instruction that restores the FP and put the PC pointing to the saved LR, i.e., the rollback instruction. The rollback instruction is the first instruction to be executed after the rollback is performed and is the first instruction that immediately follows the **saveContext()** function. After the PC is pointing to the rollback instruction, the code keeps with its normal execution. At this point, the system is recovered from the error, and it is in a safe state without errors.

5.4 Framework Constraints

Due to the complexity of implementing a rollback mechanism, some tradeoffs had to be done. These tradeoffs led the lock-V framework to have some constraints to its use. While the rollback tool can be

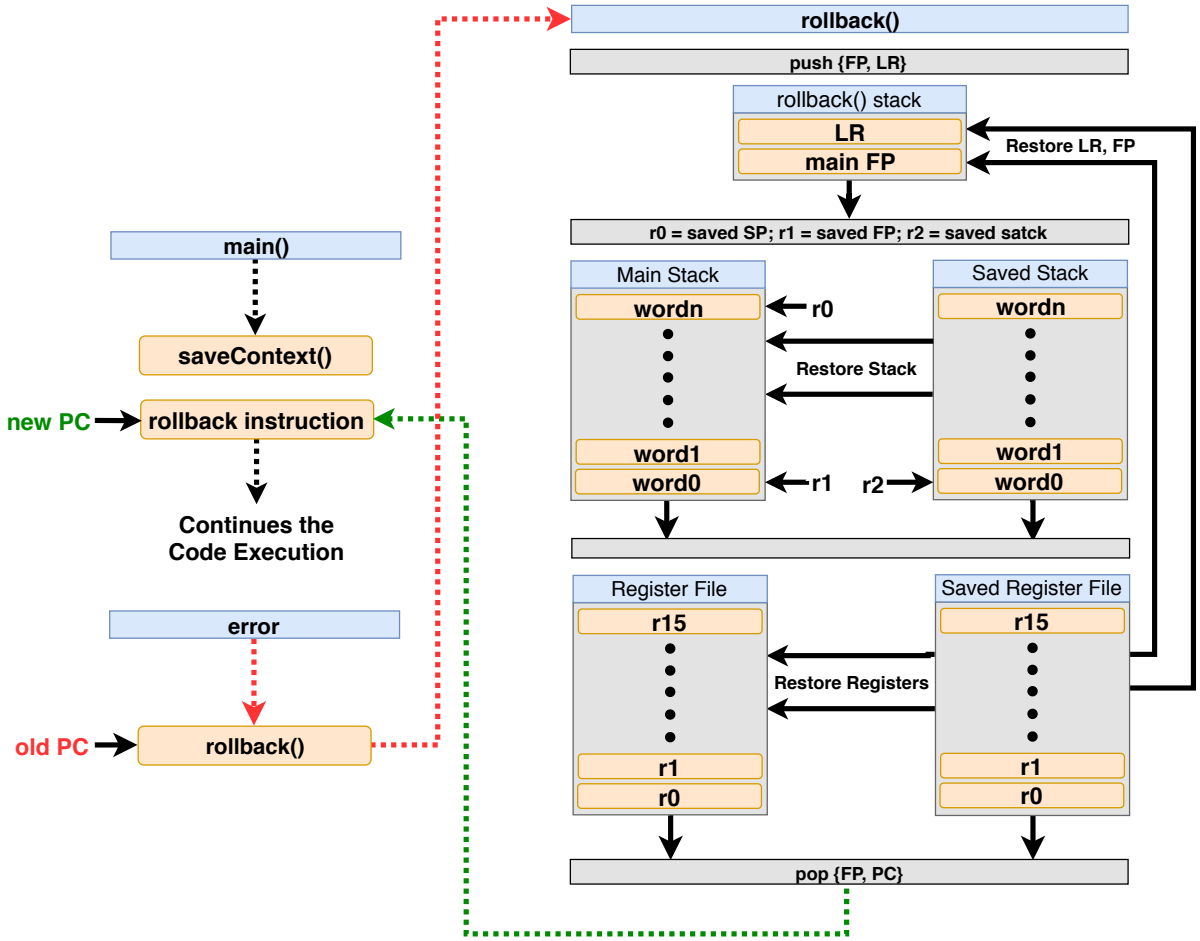


Figure 5.4: Example of the Arm rollback processor's context. Although the architecture is different, the logic behind the RISC-V rollback processor's context is the same.

used anywhere, the **saveContext** tool and its service just can be used in the **main()**. These constraints exist because of two reasons: (1) the way that the **saveContext()** and **rollback()** it is being performed, the LR is lost; and (2) if the context saving is done inside a function, the stack saved is the stack of that function and never the full stack of the program. However, saving the **main()** stack represents saving all the program stack at that point.

The checkpoint tool can be used anywhere, but it always needs to return to the **main()** in case of error or not. Whenever the checkpoint is performed, if there is no error, the context of the processor must be saved at that point of the program. Once the context saving can only be used in the **main()**, when there is no errors, then that information has to be passed to the **main()**.

The rollbacks due to lockstep errors are implemented in both cores, but the **rollbackAbort**, once that can be used alone, was only implemented in the Arm processor, for proof of concept. So, the fault-injection for testing the system can only be applied to the Arm processor. What is enough to prove that when errors that trigger exception occurs the proposed rollback mechanism may work perfectly.

The last constraint is about which applications are appropriate for deployment under Lock-V framework. Since the only memory protected beyond the register file is the memory allocated for the program stack,

the other has to be carefully used. To prevent errors in the register from propagating to the non-protected memory, all the variable that access memory out of the stack should be "read-only". This means that global variables can only be used as constants or variables initialized before the runtime (design time). So, for coding or writing an application compliant with Lock-V framework: (1) only local variables must be used; (2) global variable cannot be written, just read; (3) static variables cannot be used, since they are like a global variable; (4) the heap cannot be used because is not protected too.

6. Evaluation and Results

This chapter evaluates the proposed fault tolerance architecture, Lock-V, as well as its framework. Although endowing a system with Lock-V capabilities increases its reliability and safety, it does incur some overheads. The amount of hardware resources spent, the application code size and execution time are affected by the use of the Lock-V and its framework. In the first and second section, the resources used to implement the Lock-V as well as the Lock-V framework costs are exposed.

A case study is presented in the last section of this chapter. In that case study, a algorithm is implemented in the Lock-V and executed in an simulated harsh environment. This harsh environment will represent a real operational case where a system is susceptible to SEU. These SEU are emulated through a fault injection technique that causes some random bit-flips in the system. In this section, the effectiveness of the Lock-V will be tested. A side by side comparison between a hardened system with Lock-V and an unhardened without Lock-V is performed.

The Lock-V architecture, and its main components, was deployed and tested on a Zynq-7000 SoC, featuring a dual-core Arm Cortex-A9 and FPGA fabric used to host the RISC-V soft-core processor. In this implementation, the Arm Cortex-A9 is running at the frequency of 666 MHz and the lowRISC at the frequency of 25 MHz.

6.1 Lock-V PL Resources Utilization

Table 6.1 shows the hardware resources needed, after implementation, for the lowRISC soft-core, the *xLockstep* modules and the context saving IP. The results are expressed in terms of LUT and FF. The lowRISC module is the most costly in terms of needed hardware, representing around 81% (34138 out of 42124) of LUT and nearly 46% (16324 out of 35850) of FF. This is due to the deployment of a soft-core RISC-V processor, rather than a hard-core implementation, which is one of the tradeoffs of the proposed solution. The solution provides flexibility and the possibility to customize the RISC-V architecture, but it comes with the cost of FPGA resources. Regarding the needed resources by the *xLockstep* accelerator (441 LUT and 672 FF), it is possible to conclude that the *xLockstep* has a lightweight implementation. It is only responsible for 1% of the used LUT and around 2% of the used FF. Despite the error detection (*xLockstep*) requires fewer resources, in contrast to the error recovery logic which consumes a lot of the FPGA fabrics. The *ContextSave* IP is very costly, consuming 18% of the used LUT and around 53% of used

Table 6.1: Post-Implementation results obtained from Vivado 2016.2.

HW module	LUT	FF
lowRISC	34 138	16 324
AXI_RISCV_Slave	135	267
AXI_ARM_Slave	122	269
TopxLockstep	25	40
Checker	148	90
Synchro	6	3
Synchro_to_Resume	5	3
Subtotal	441	672
ContextSave_Arm	1 513	3 799
ContextSave_RISCV	6 032	15 055
Subtotal	7 545	18 854
Total	42124 (79.2%)	35850 (33,7%)

FF. These number are huge and not so far from the resources consumed by the lowRISC soft-core. A future solution for this context saving IP that uses less resources should be explored. In short, if both processors were available in the SoC in a hard-core implementation and the context saving logic is implemented in external memories, the solution could resort a FPGA with less resources, once the *xLockstep* has a very lightweight implementation.

6.2 Lock-V Framework Costs

In order to estimate the impact in using the Lock-V framework, its execution and memory footprint is measured. In doing so, an application is written for running in both cores. The memory footprint was measured with the application executing with and without Lock-V services. The chosen application implements a Fibonacci function because its execution time is easily scalable. The scalability aspect is very important to test the execution overhead of the Lock-V regarding the number of checkpoints used and the overhead added to a system with different application execution sizes.

6.2.1 Memory Footprint

In order to get the information about the memory footprint two tools were used. To measure the memory footprint of Arm and RISC-V applications, **arm-none-eabi-size** and **riscv64-unknown-elf-size** were used, respectively. These tools are similar in their functionality. Both give information about an .elf file in terms of: (1) number of code bytes (.text); (2) number of bytes allocated to the initialized global variables (.data); and (3) number of bytes allocated to the uninitialized global variables (.bss). Tables 6.2 and 6.3 presents the memory footprint of an application used in four different scenarios as well as the overhead in using the Lock-V framework. The scenarios taken in consideration for measuring the application size are

Table 6.2: Arm memory footprint in bytes.

Arm	**.text**	**.data**	**.bss**	**total**
Application without Lock-V	19552	1152	22580	43284
Application with Lock-V	22096	1216	22580	45892
Lock-V Overhead	2544	64	0	2608 (6%)

Table 6.3: RISC-V memory footprint in bytes.

RISC-V	**.text**	**.data**	**.bss**	**total**
Application without Lock-V	45864	97	647	46608
Application with Lock-V	49212	616	660	50488
Lock-V Overhead	3348	519	13	3880 (8.3%)

the following: (1) the application compiled targeting the Arm processor without using the Lock-V framework; (2) the application compiled targeting the RISC-V processor also without using the Lock-V framework; (3) the application compiled targeting the Arm processor, but this time using the Lock-V framework; and (4) the application compiled targeting the RISC-V processor also using the Lock-V framework.

Having the application without Lock-V as a baseline benchmark, it is possible to see, as expected, an increase in the memory footprint. Adding Lock-V capabilities to the application, increases its memory footprint in nearly 6% in Arm and 8.3% in RISC-V. The Arm and RISC-V Lock-V framework are practically the same. Therefore, the difference in the Arm and RISC-V Lock-V overhead are due to the difference in processors' ISAs. The final machine code are different, because the Arm and RISC-V compilers generate different amounts of instructions for the same C function. Additional factors that increases RISC-V Lock-V overhead are the higher number of registers (32) and the bigger word (the RISC-V is a 64 bits processor, 8 bytes word, and the Arm is a 32 bits one, 4 bytes word). The rollback mechanism in Arm uses 64 bytes (16×4) while the same rollback mechanism in RISC-V needs 256 bytes (32×8). Despite the differences between the used processors the measured overhead is not significantly. To endow an application with Lock-V fault tolerance capabilities, it is necessary to do some tradeoffs. The Lock-V memory print overhead is one of them.

6.2.2 Execution Footprint

The execution overhead was performed only in one processor. The RISC-V processor was the chosen one because of two reasons. First, it is the bottleneck of the Lock-V architecture. The lowRISC soft-core runs at 25 MHz while the Arm runs at 666 MHz. Second, the lowRISC is the main core while the Arm is the checker core. As said in 2.2.4.1, a dual redundant system must have a component that outputs an result and another redundant one that checks if that output is correct. This means that the Arm is used for checking the integrity of the application that runs in the lowRISC processor. The output of the system comes from the lowRISC processor. To measure the latency of a specific function or program, the **rdcycle**

RISC-V instruction is used. This instruction returns the current number of elapsed cycles. Measuring the cycles in two points of the program execution and subtracting the first from the second gives the latency in executing a certain piece of code. All the taken results presented in this topic, are the mean value of a hundred samples.

In first step, the latency of the three Lock-V functions were measure. Table 6.4 presents the latency in executing the services or functions of **saveContext** and **rollback** Lock-V tools. The Lock-V framework takes around 125 microseconds for saving the processor context and around 114 microseconds to restore all the processor context and stack, when a rollback it is performed. The processor context saving takes more 10 microseconds than its restoring pair. Afterward, the latency of the **checkpoint** service was measured in two different scenarios. When has occurred an error and when the system operates in a normal state without errors. The measured results are presented in Table 6.5. For testing the impact of different outputs' vector sizes, 11 testes/measures were performed. The number of data was varied from 1 to 100 units. Because lowRISC is a 64-bit processor, its word is 8 bytes. Therefore, the amount of data vary between 8 bytes until 800 bytes. Each test uses more 10 units of data (80 bytes) than the previous one. From the measures it is possible to see that when an error is active, the checkpoint takes in average more 49 microseconds that when the system is free from errors. This happens because the checkpoint has to inform the system that an error was detected. It is also possible to see that each extra 10 units of data verified by the checkpoint increases the overhead in 987 microseconds.

In the second step of the execution footprint measurement, the latency of an application was calculated without Lock-V, i.e., the application running in a normal environment, and with Lock-V. When the Lock-V was used three scenarios were tested and measured:

- the application runs once with one 1 checkpoint and another with N checkpoints, but in absence of errors;
- the application runs once with one 1 checkpoint and another with N checkpoints, but this time in the presence of an error in the first compared element;
- the application runs first with one 1 checkpoint and another with N checkpoints, in the presence of an error in the last compared element.

The results presented in Table 6.6 are taken running the application in order to calculate the Fibonacci of 10, 15 and 20. When the system has no errors, in the worst case scenario (N checkpoints), the overhead passes from 112,7% to 1.8%. When the system has an error and the error is in the last element, the overhead goes from 324,8% to 103.6%. With the increase of the application execution times, the overhead of the Lock-V framework reduces drastically, demonstrating that the Lock-V is suitable for application with

Table 6.4: *saveContext* and *rollback* execution footprint.

	Latency	@25MHz
saveContext()	3 128	125.1 μs
rollback()	2 852	114.1 μs

Table 6.5: *Checkpoint* execution footprint in clock cycles.

Nº Data	**Checkpoint**					
Integers	Normal	@25MHz	W/ Error	@25MHz	Overhead	@25MHz
1	10 420	416.8 μs	11 698	467.9 μs	1 278	51.1 μs
10	32 851	1314.0 μs	34 118	1364.7 μs	1 267	50.7 μs
20	57 508	2300.3 μs	58 783	2351.3 μs	1 275	51.0 μs
30	82 206	3288.2 μs	83 494	3339.8 μs	1 288	51.5 μs
40	106 906	4276.2 μs	108 098	4323.9 μs	1 192	47.7 μs
50	131 552	5262.1 μs	132 763	5310.5 μs	1 210	48.4 μs
60	156 229	6249.2 μs	157 423	6296.9 μs	1 194	47.8 μs
70	180 901	7236.0 μs	182 203	7288.1 μs	1 302	52.1 μs
80	205 573	8222.9 μs	206 743	8269.7 μs	1 170	46.9 μs
90	230 250	9210.0 μs	231 398	9255.9 μs	1 148	45.9 μs
100	254 921	10196.8 μs	256 079	10243.2 μs	1 158	46.3 μs
Avg. Increment	24 675	987.0 μs	**Avg. Overhead**		1 226	49.0 μs

high execution time. Another curiously fact happens when the error occurs in the first elements. The overhead in using the Lock-V with N checkpoints is fewer then when just one checkpoint it is used. This phenomenon occurs because the execution granularity of the error detection is smaller. So when an error occurs it is faster detected and processed. When just one checkpoint is used, the verification can only be done at the end of the program. So, if an error occurs, the system can only be rollbacked when the program finishes its execution, resulting always in an overhead greater than 100%. In contrast, when N checkpoints are used, the error can be detected faster and the system is recovered earlier.

Table 6.6: Lock-V execution footprint with and without an error in clock cycles.

Application	**Fibonacci(10)**		**Fibonacci(15)**		**Fibonacci(20)**	
Without Lock-V	95 121		1 063 543		11 782 912	
No Errors						
[a] With Lock-V [1 checkpoint]	108 543	12.4%	1 074 376	1.0%	11 805 556	0.2%
[b] With Lock-V [N checkpoints]*	202 283	112.7%	1 216 577	14.4%	11 999 361	1.8%
Difference *b* - *a*		+100.3%		+13,4%		+1.6%
Error in 1st element						
[a] With Lock-V [1 checkpoint]	218 026	129.2%	2 152 372	102.4%	23 582 287	100.1%
[b] With Lock-V [N checkpoints]*	236 280	148.4%	1 254 465	18.0%	12 028 680	2.1%
Difference *b* - *a*		+19.2%		-84.4%		-98%
Error in last element						
[a] With Lock-V [1 checkpoint]	218 336	129.5%	2 152 372	102.4%	23 582 277	100.1%
[b] With Lock-V [N checkpoints]*	404 044	324.8%	2 434 529	129.0%	23 988 473	103.6%
Difference *b* - *a*		+195.3%		+26.6%		+3.5%

* Application running with 10,15 or 20 checkpoints [n° checkpoints = **N**, Fibonacci(**N**)].

6.3 Case study

In this section it is presented a case study that was made in order to test and evaluated the Lock-V capabilities (error detection and error recovery). An application was executed with and without the Lock-V. This application was chosen to be easily scalable in terms of used data. The application is a calculation of prime numbers under a specify number. For example, if the application calculates primes number of 100, a vector with all these numbers is generated. After the applications is developed, a fault injection mechanism, based on the previously proposed in [85], was performed to test the Lock-V architecture. This fault injection aims to emulate bit-flips that occur in harsh environments due to SEU. First, the test setup will be explained, then the fault injection mechanism will addressed and at the end the results will be presented.

6.3.1 Setup

Figure 6.1 depicts the setup used for evaluating the system. The setup is composed by three main parts: (1) the processor that read the stats of the fault injection; (2) the extra hardware used for the fault injection, a hardware timer; and (3) the host environment to where is sent the data of the tests. The hardware timer is used to count the time to inject a fault. It is set with time to inject, and when the timer expired an interrupt is generate and a fault is injected. This injected faults can result in three different things: (1) in one ineffective fault. The injected fault does not provoke any error; (2) in one Silent Data Corruption (SDC). The injected fault provokes an error in the outputs; and (3) in one hang. This occurs when a fault originates a system crash or an infinite loop. Three unused and reserved register of the *xLockstep* are used for collect data of the fault injection. Whenever that a fault is injected a counter of faults is incremented. If that fault originates an SDC or a hang, a specific register for each type of error

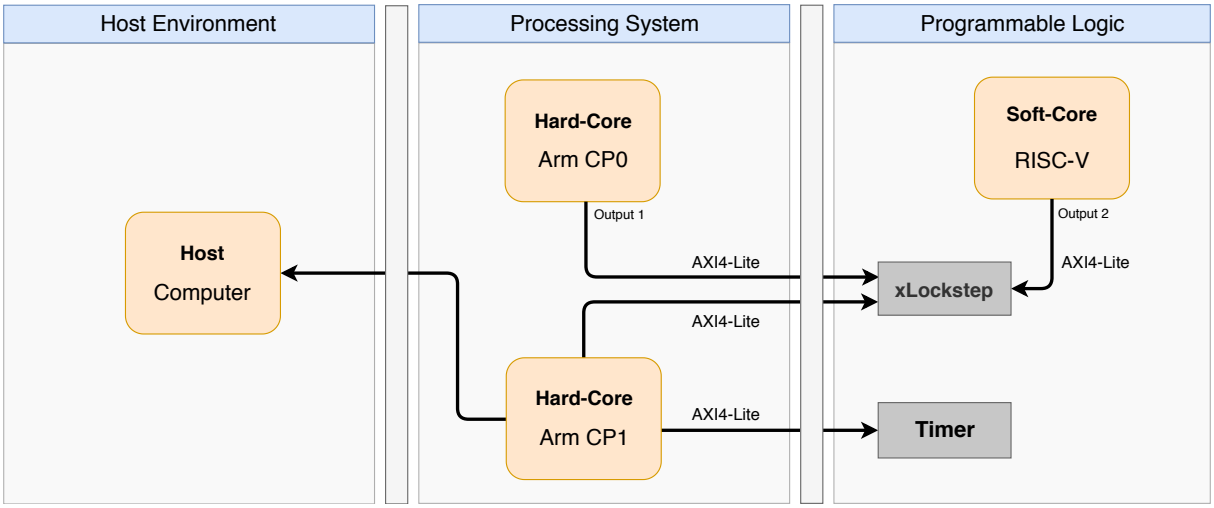


Figure 6.1: Fault injection setup.

is incremented. This information stored in the *xLockstep* is later read by the second Arm processor. After reading the statistics, the processor sends them to the host computer.

6.3.2 Fault Injection

The fault was injected in register file randomly in terms of time and location. The injection was done in the Arm processor, because it is the faster Lock-V processors and so the overhead to perform the fault injection have almost no impact in the application performs. The fault injection mechanism works the following way. In each time the application is executed, a timer is started with a random value. This random value contains an interval of time between zero and the time of the application execution. Each time the timer expires, it is reloaded with another random time value. This time is called the time to inject. After the time to inject elapses, an interruption is triggered and the fault is injected. Figure 6.2 depicts the mechanism behind that injection. First, the register and bit of the register to be flipped are randomly chosen. So, one out of the 16 Arm registers and one out of its 32 bits will be flipped. The faults are injected following three steps: (1) the register file is copied; (2) the fault is injected in the replicated register file performing an XOR with the register and bit to flip; and (3) the register file is restored with the fault already injected. This loop of events for injecting faults are continuously done until a system crash occurs.

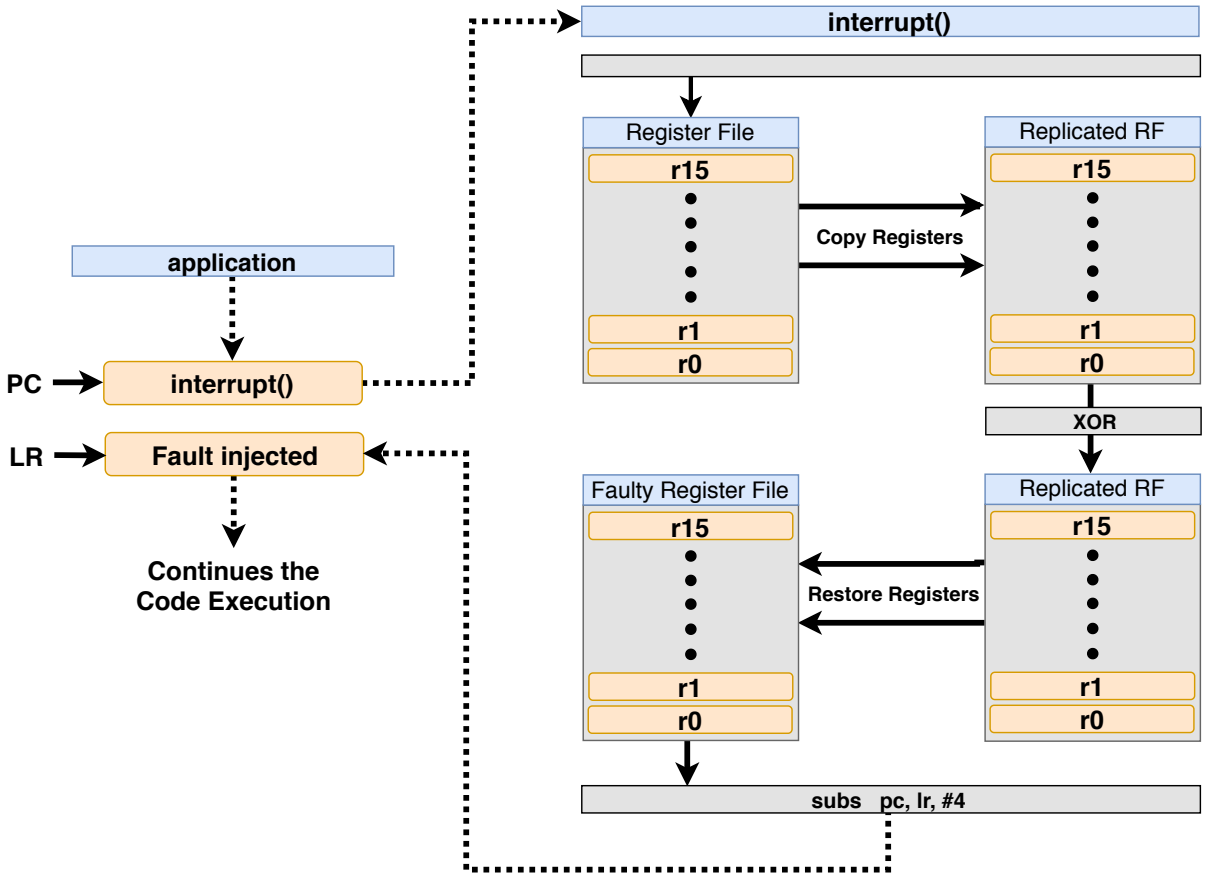


Figure 6.2: Fault injection mechanism.

6.3.2.1 Results

The fault injection was performed in two scenarios. In an application without Lock-V and with the Lock-V. The aiming of these tests is measuring the percentage of faults that is detected and corrected by the Lock-V mechanism. The faults were injected in two different manners, full time injection and half time injection. In the first manner, just one fault is injected during each application execution. This emulates a real harsh environment, where is unlikely that more than one faults hits the registers during the execution. In the second manner, more than one faults are injected during the application execution, what leads to more errors. This heavy fault injection is more unrealistic but still useful for the Lock-V error recovery capabilities testing purposes. Figure 6.3 and 6.4 depicts the errors detected in the application when it runs in a normal fashion or with Lock-V. Looking at the graphs is possible to see an huge difference between using a system hardened by the Lock-V and one that has no fault tolerant mechanism. It is also possible to see that the half time injection test generates as expected, more errors both SDC and hang ones. However, the effectiveness of the Lock-V is not affected by the increase of faults during the application execution. This makes the Lock-V architecture well suitable for being used in harsh environments.

Table 6.7 summarizes all the tests done to the system. It was injected in total 45543 faults. Among those injected faults 137 of them generate a hang and 796 a SDC. When the Lock-V is applied to the application, the number of errors reduces drastically, around 97%. Out of the 933 faults, only 31 of them were undetected by the Lock-V mechanism. The Lock-V has proved to be an effective option and a suitable solution for endow a system with fault tolerant capabilities.

Table 6.7: Fault injections testes with and without Lock-V.

Tested Application	**Faults**	**Without Lock-V**		**With Lock-V**		**Error Correction**
		Hang	SDC	Hang	SDC	Percentage
Primes under 100*	18 312	16	164	7	1	95.6%
Primes under 300*	13 416	18	109	3	1	96.9%
Primes under 1000*	14 158	13	193	6	1	96.6%
Primes under 100**	13 815	90	330	12	0	97.1%
Total	45 543	933		31		96.7%

* Application tested with the maximum time equal to the application execution time.
** Application tested with the maximum time equal to half of the application execution time.

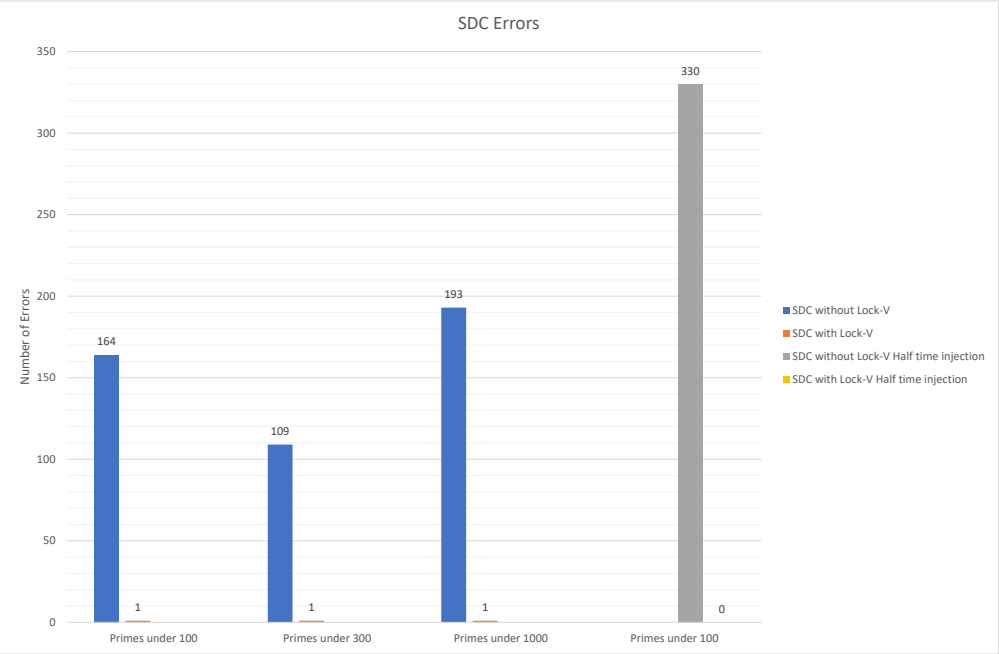


Figure 6.3: SDC errors after the injection of faults with and without the Lock-V.

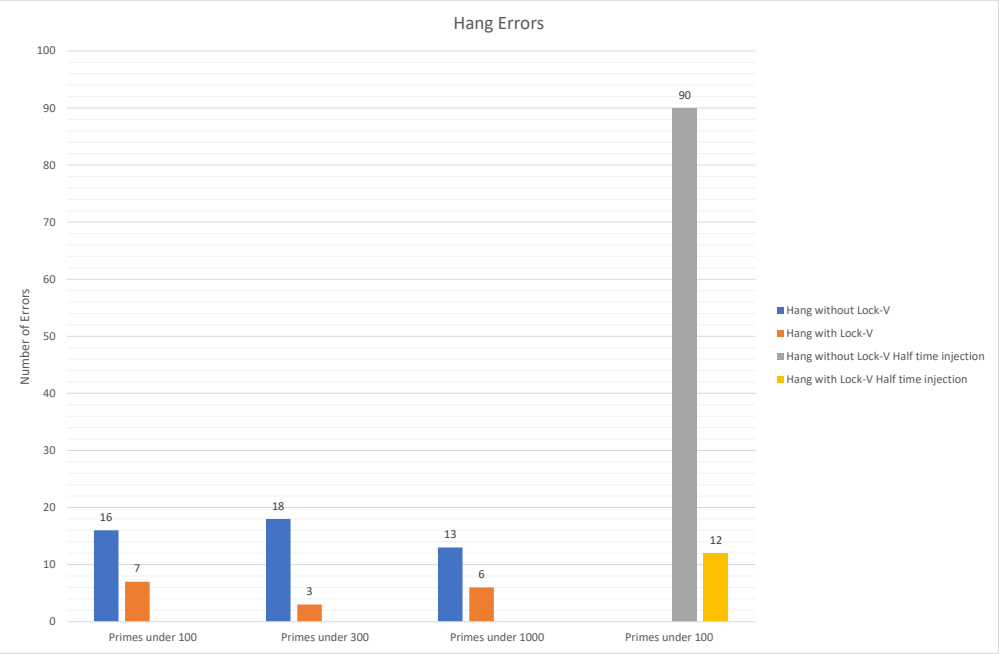


Figure 6.4: Hang errors after the injection of faults with and without the Lock-V.

7. Conclusion

The evolution of the processors technologies has been push the operations boundaries forward towards low power consumption, higher transistors' density as well as better performance. This comes up with some new challenges and some drawbacks. Computing systems have been becoming more vulnerable and susceptible to faults and consequently to system failures.

To mitigate this problem, the concept of fault tolerant was created. This concept can be implemented by applying redundancy and many other different techniques. The fault tolerance has two phases, the error detection phase and the system error recovery phase. The first phase detects an active error and the second one eliminates them. Another concern that the fault tolerant system designers have been aware of is protecting the fault tolerant systems against the CMF using design diversity. Although several implementations of fault tolerant have been proposed in the academics, they do not apply of design diversity to protect redundant processors. In opposite side, commercial processors for safety-critical applications implement time diversity, however, they do not promote design diversity at ISA level.

This thesis proposes the Lock-V, a heterogeneous fault tolerance architecture using DCLS built upon an Arm Cortex-A9 and the lowRISC processors. This architecture beyond the fault tolerance capability — that applies redundancy at the processor level — it also applies design diversity at the ISA level. Lock-V is composed by three main components. A hard-core with Arm ISA plus a soft-core with RISC-V ISA which are running in an lockstep fashion. Moreover, there is an hardware accelerator, *xLockstep* that implements the lockstep mechanism. The *xLockstep* offers two main services. The first is responsible to synchronize the processors and the second is responsible for comparing their outputs to detect errors. The *xLockstep* proves that is possible to successfully implement the lockstep detecting errors either by synchronization timeout or mismatching in the processors' outputs.

Built upon the Lock-V architecture, a framework was developed to allow an easy use of the whole architecture as well as to implement error recovery by software. The framework provides three tools that give some freedom to the user choice in terms of the lockstep granularity to be applied. To increase the error detection rate, the user should identify more critical code points while using the checkpoint tool more often. This tool is responsible to perform the lockstep in cooperation with the *xLockstep* module for error detection purpose. When an error is detected it must be mitigated. For this purpose, the framework provides two tools. One for saving a safe and healthy system state and another for recovering the system from an error to the previously saved state. The framework working alongside with the Lock-V has proved

to be efficient in detecting and fixing errors.

Summing up, by developing an application which adopts the proposed framework while targeting the deployment in the Lock-v architecture, the application is able to safely perform its intent operations. An application running in a Lock-V fashion owns fault tolerance and design diversity capabilities that protect it against SEU faults and the CMF. This thesis hardened two different processors and gave error recovery capabilities to a system, which owns a high error coverage.

7.1 Future Work

This thesis applies lockstep mechanism to two different processors architectures, Arm and RISC-V proposing a new architecture, Lock-V. This provides to an application fault tolerance capabilities while enabling design diversity. However, since it is in an initial developing phase, the Lock-V and its framework can be improved in terms of design and implementation as well as adding extra functionalities. The following topics addresses possible improvements in a future work:

FPGA partial reconfiguration The FPGA is vulnerable to SEU, once it is configured through a bit-stream file that is stored in memory. In harsh environments, the Reconfigurable Computing Unit (RCU) can be hit by radiation and suffer from bit-flips. Under such occurrence, both *xLockstep* and lowRISC soft-core can be affected and their functionalities changed. These kind of faults can trigger a system failure, which in the current Lock-V implementation is unrecoverable. For the RCU to be tolerant to faults, a mechanism of PR needs to be implemented. This mechanism in the presence of a fault in the FPGA allows dynamically changing of faulty design modules to a configuration without faults.

Protect all the memory Due to the tradeoffs made in the implementation of the error recovery presented in Section 5.3, the hardened memory in the Lock-V is only on registers and the stack. It will be interesting an extension of the protection memory from register and stack to the whole memory and compare the gain in performance (better error recovery) versus the increase in the resources usage. The protection of the .text, .rodata, .data and .bss could be done by implementing a redundant copy of that memory sections in an extra memory.

Implement system soft reset When an application executes in the Lock-V architecture, it is protected against the majority of the errors that are triggered, i.e., around 97% as presented in Section 6.3. However, in some cases (3% of the triggered errors) the Lock-V recovery mechanisms are ineffective. Under such scenarios, the system crash and only with human intervention, e.g., performing a hard reset, the system can be recovery. The current implementation is not the most desired. So, in a future iteration of Lock-V a soft reset should be implemented to recover the system when the whole other Lock-V mechanism fails.

Refactoring the save context The RCU resources used, presented in Section 6.1, show that the technique implemented for the processor context saving is not the most suitable. It requires a lot of LUT and FF, when compared with the *xLockstep*, i.e., it demands for huge hardware resource footprint. To decrease this footprint, the context saving has to resort to external memory to store the context data while reducing its introduced huge overhead.

Implement rollbackAbort() in RISC-V For prof of concept and due to the time constraints and chosen setup to perform the fault injection, the rollback executed after an internal processor's exception was implemented only in the Arm processor, as was addressed in Section 5.4. This one side **rollbackAbort()** implementation was enough to prove that the system can recovery from processors' exception successfully. However, for real-world deployment, the system is incomplete, lacking exception errors recovery capabilities in the lowRISC core. Since the logic behind the response at exceptions and the execution of the **rollbackAbort()** are the same for both cores, in a future refactoring of Lock-V framework, the **rollbackAbort()** should be also implemented in the RISC-V processor.

DCLS with Arm The Lock-V architecture uses the *xLockstep* for implementing the lockstep mechanism, which is loosely-coupled to the processors. That implementation allows its use in other architectures with fewer changes. Despite the implementation of Lock-V uses Arm Cortex A9 and lowRISC, it can be implemented with other processors. It will be interesting to implement Lock-V on another set of processors for comparing with the actual solution and for validating the portability of the *xLockstep* regarding processors used in Lock-V. This new DCLS is not very difficult to achieve, once that the used platform, ZedBoard, has a dual core Arm cortex A9. The second Arm processor can substitute the lowRISC in a new Lock-V architecture version.

Automate the Lock-V Framework Currently, the code regarding the framework has to manually be inserted in the application source code by the programmers. This can overwhelm the developers when they are adding Lock-V capabilities to the application. So, the framework can be optimized in order to provide code injection capabilities. This feature will allow the application code to be automatically analyzed by the framework, which will choose the best places to deploy the Lock-V checkpoints, and later generate the data to configure the chosen Lock-V architecture.

Implement a BIST To detect if an error is a hard or soft one, as was addressed in 2.1.2.2, a BIST must be implemented, as we saw in Section 2.3. Lock-V for now only provides protection against soft SEEs, but hard SEEs, i.e, SELs, can also be triggered when high energetic particles hit the system. The system can have errors due to different sources, which must be first identified and then properly fixed. In doing so, an identified error must be clearly associated to its root cause event, i.e., to a destructive or nondestructive kind of event. According to this classification, done by the BIST, some different actions should be taken.

References

[1] E. Dubrova, *Fault-tolerant design*. Springer, 2013.

[2] 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," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2017.

[3] A. 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, no. 8, pp. 1783–1790, Aug 2018.

[4] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, 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, no. 3, 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, no. 4, pp. 1992–2000, Aug. 2009.

[7] Á. 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*. Fortaleza, Ceará, Brazil: ACM Press, 2017, pp. 84–89.

[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)*. Fukuoka: IEEE, Oct. 2018, pp. 737–748.

[9] J. Han, Y. Kwon, Y. C. P. Cho, and H.-J. 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)*. Seoul: IEEE, Nov. 2017, pp. 313–316.

[10] 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.

[11] A. 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, no. 8, pp. 1783–1790, Aug. 2018.

[12] A. Hanafi, M. Karim, and A. E. Hammami, "Dual-lockstep microblaze-based embedded system for error detection and recovery with reconfiguration technique," in *2015 Third World Conference on Complex Systems (WCCS)*. Marrakech: IEEE, Nov. 2015, pp. 1–6.

[13] H.-M. Pham, S. Pillement, and S. J. Piestrak, "Low-Overhead Fault-Tolerance Technique for a Dynamically Reconfigurable Softcore Processor," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1179–1192, Jun. 2013.

[14] 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.

[15] 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)*. Como, Italy: IEEE, 2006, pp. 203–212.

[16] S. Pinto, A. Tavares, and S. Montenegro, "Space and time partitioning with hardware support for space applications," *Data Systems In Aerospace, European Space Agency, ESA SP 736*, 2016.

[17] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares, "Lightweight multicore virtualization architecture exploiting arm trustzone," in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, Oct 2017, pp. 3562–3567.

[18] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, "Towards a trustzone-assisted hypervisor for real-time embedded systems," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 158–161, July 2017.

[19] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on trustzone-enabled microcontrollers? voilà!" in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2019, pp. 293–304.

[20] Avizienis and Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, no. 8, pp. 67–80, Aug 1984.

[21] S. Mitra, N. R. Saxena, and E. J. McCluskey, "Common-mode failures in redundant vlsi systems: a survey," *IEEE Transactions on Reliability*, vol. 49, no. 3, pp. 285–295, Sep. 2000.

[22] J. Yiu, "Design of SoC for High Reliability Systems with Embedded Processors," p. 8, 2015.

[23] ARM, "Cortex-M33 Dual Core Lockstep," ARM Limited., Tech. Rep., 2017.

[24] S. Mitra, N. R. Saxena, and E. J. McCluskey, "A design diversity metric and reliability analysis for redundant systems," in *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, Sep. 1999, pp. 662–671.

[25] M. Berg and C. Michael, "FPGA Mitigation Strategies for Critical Applications, support of NASA/GSFC," Sep. 2018.

[26] 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, no. 20, pp. 6816–6824, Oct. 2017.

[27] 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, no. 1, pp. 11–33, Jan 2004.

[28] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[29] F. Afonso, "Operating system fault tolerance support for real-time embedded applications," Ph.D. dissertation, University of Minho, 2009.

[30] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[31] A. J. C. Lanot and T. R. Balen, "Fault mitigation strategies for single event transients on sar converters," in *19th Annual International Mixed-Signals, Sensors, and Systems Test Workshop Proceedings*, Sep. 2014, pp. 1–6.

[32] S. Mittal and M. S. Inukonda, "A survey of techniques for improving error-resilience of DRAM," *Journal of Systems Architecture*, vol. 91, pp. 11–40, Nov. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1383762118301693

[33] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran, "Software resilience and the effectiveness of software mitigation in microcontrollers," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2532–2538, Dec 2015.

[34] 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*, Nov 2013, pp. 2261–2266.

[35] 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)*, Nov 2017, pp. 313–316.

[36] A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, "Softerror mitigation for multi-core processors based on thread replication," in *2019 IEEE Latin American Test Symposium (LATS)*, March 2019, pp. 1–5.

[37] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran, "Robust duplication with comparison methods in microcontrollers," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 338–345, Jan 2017.

[38] E. Chielle, B. Du, F. L. Kastensmidt, S. Cuenca-Asensi, L. Sterpone, and M. S. Reorda, "Hybrid soft error mitigation techniques for cots processor-based systems," in *2016 17th Latin-American Test Symposium (LATS)*, April 2016, pp. 99–104.

[39] S. Mitra and E. J. McCluskey, "Design of redundant systems protected against common-mode failures," in *Proceedings 19th IEEE VLSI Test Symposium. VTS 2001*, April 2001, pp. 190–195.

[40] L. M. Kaufman, S. Bhide, and B. W. Johnson, "Modeling of common-mode failures in digital embedded systems," in *Annual Reliability and Maintainability Symposium. 2000 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.00CH37055)*, Jan 2000, pp. 350–357.

[41] C. Hernandez and J. Abella, "Live: Timely error detection in light-lockstep safety critical systems," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[42] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella, "High-integrity gpu designs for critical real-time automotive systems," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 824–829.

[43] C. M. Jeffery and R. J. O. Figueiredo, "A flexible approach to improving system reliability with virtual lockstep," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 1, pp. 2–15, Jan 2012.

[44] Infineon, "Highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications," Infineon Technologies AG, Tech. Rep., 2017.

[45] "S32s24: Safety microcontroller for automotive applications," https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/s32-automotive-platform, accessed: 2019-08-21.

[46] "Ultra-reliable mpc57xx 32-bit automotive and industrial microcontrollers (mcus),"

https://www.nxp.com/products/processors-and-microcontrollers/power-architecture-processors/mpc5xxx-55xx-32-bit-mcus/ultra-reliable-mpc57xx-32-bit-automotive, accessed: 2019-08-21.

[47] ARM, "Cortex-R4 and Cortex-R4f Technical Reference Manual," ARM Limited, Tech. Rep., 2011.

[48] ARM, "Cortex-R5 Technical Reference Manual," ARM Limited, Tech. Rep., 2011.

[49] ARM, "Arm® Cortex®-R52 Processor Technical Reference Manual," ARM Limited, Tech. Rep., 2018.

[50] ARM, "ARM Cortex-R7 MPCore Technical Reference Manual," ARM Limited, Tech. Rep., 2012.

[51] ARM, "Arm® Cortex®-R8 MPCore Processor Technical Reference Manual," ARM Limited, Tech. Rep., 2018.

[52] ARM, "Arm® Cortex®-A76ae Core Technical Reference Manual," ARM Limited, Tech. Rep., 2018.

[53] F. ARM, "Exploring the ARM® Cortex®-M7 Core: Providing Adaptability for the Internet of Tomorrow," ARM Limited, Freescale, Tech. Rep., 2018.

[54] 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)*, June 2016, pp. 246–249.

[55] X. Iturbe, B. Venu, J. Jagst, E. Ozer, P. Harrod, C. Turner, and J. Penton, "Addressing functional safety challenges in autonomous vehicles with the arm tcl s architecture," *IEEE Design Test*, vol. 35, no. 3, pp. 7–14, June 2018.

[56] A. Hanafi, M. Karim, and A. E. Hammami, "Dual-lockstep microblaze-based embedded system for error detection and recovery with reconfiguration technique," in *2015 Third World Conference on Complex Systems (WCCS)*, Nov 2015, pp. 1–6.

[57] 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*, Nov 2013, pp. 2261–2266.

[58] 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, no. 6, pp. 1179–1192, June 2013.

[59] 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, 06 2019.

[60] 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, no. 4, pp. 1992–2000, Aug 2009.

[61] A. 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)*, Feb 2017, pp. 1–4.

[62] 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 *2017 30th Symposium on Integrated Circuits and Systems Design (SBCCI)*, Aug 2017, pp. 84–89.

[63] ARM, "ARM: Media fact sheet (Sept. 1, 2016)," ARM Limited., Tech. Rep., 2016.

[64] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*, 1st ed. Strawberry Canyon, Nov. 2017.

[65] M. Nöltner-Augustin, "RISC-V — Architecture and Interfaces The RocketChip," *COMPUTER ENGINEERING*, p. 6, 2016.

[66] lowRISC, "Ibex: a 32 bit risc-v cpu core," https://github.com/lowRISC/ibex, accessed: 2019-08-22.

[67] U. d. B. ETH Zurich, "Pulpino: An open-source microcontroller system based on risc-v," https://github.com/pulp-platform/pulpino, accessed: 2019-08-22.

[68] C. Wolf, "Picorv32 - a size-optimized risc-v cpu," https://github.com/cliffordwolf/picorv32, accessed: 2019-08-22.

[69] U. d. B. ETH Zurich, "Ariane, a 6-stage RISC-V CPU capable of booting Linux," https://github.com/pulp-platform/ariane, accessed: 2019-08-22.

[70] VectorBlox, "Orca: Risc-v by vectorblox," https://github.com/vectorblox/orca, accessed: 2019-08-22.

[71] Microsemi, "Mi-v risc-v ecosystem," https://www.microsemi.com/product-directory/fpga-soc/5210-mi-v-embedded-ecosystem, accessed: 2019-08-22.

[72] U. B. Esperanto, "Boom: Berkeley out-of-order machine," https://github.com/riscv-boom/riscv-boom, accessed: 2019-08-22.

[73] SiFive, "Sifive's freedom," https://github.com/sifive/freedom, accessed: 2019-08-22.

[74] U. B. SiFive, "Rocket chip generator," https://github.com/chipsalliance/rocket-chip, accessed: 2019-08-22.

[75] lowRISC, "lowrisc project," https://github.com/lowRISC/lowrisc-chip, accessed: 2019-08-22.

[76] lowRISC, "Untethered lowrisc, memory mapped io and tilelink/axi," https://wsong83.github.io/presentation/lowRISC20150727.pdf, accessed: 2019-07-15.

[77] lowRISC, "Overview of the rocket chip," https://www.lowrisc.org/docs/untether-v0.2/overview/, accessed: 2019-07-15.

[78] J. W. Jonathan Bachrach, Krste Asanović, "Chisel 3.0 Tutorial," EECS Department, UC Berkeley, Tech. Rep., 2017.

[79] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings International Conference on Dependable Systems and Networks*, June 2002, pp. 389–398.

[80] 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, no. 8, pp. 2179–2187, Aug 2017.

[81] M. A. Abazari, M. Fazeli, A. Patooghy, and S. G. Miremadi, "An efficient technique to tolerate mbu faults in register file of embedded processors," in *The 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS 2012)*, May 2012, pp. 115–120.

[82] A. Ramos, A. Ullah, P. Reviriego, and J. A. Maestro, "Efficient protection of the register file in soft-processors implemented on xilinx fpgas," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 299–304, Feb 2018.

[83] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An experimental study of soft errors in microprocessors," *IEEE Micro*, vol. 25, no. 6, pp. 30–39, Nov 2005.

[84] G. Memik, M. T. Kandemir, and O. Ozturk, "Increasing register file immunity to transient errors," in *Design, Automation and Test in Europe*, March 2005, pp. 586–591 Vol. 1.

[85] R. Velazco, S. Rezgui, and R. Ecoffet, "Predicting error rate for microprocessor-based digital architectures through c.e.u. (code emulating upsets) injection," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2405–2411, Dec 2000.