POLITECNICO DI TORINO

MASTER OF SCIENCE THESIS

Design of a fault tolerant instruction decode stage in RISC-V core against soft and hard errors

Supervisor:
Prof. Stefano DI CARLO

Author:
Marcello NERI

Co-supervisors:
Prof. Maurizio MARTINA
Prof. Alessandro SAVINO
Prof. Guido MASERA
Prof. Luca Maria CASSANO

Master of Science degree in

Electronics Engineering
Department of Electronics and Telecommunications

A.Y. 2020/2021

Declaration of Authorship

I, Marcello NERI, declare that this thesis titled, "Design of a fault tolerant instruction decode stage in RISC-V core against soft and hard errors" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 04/04/2021

"Stay hungry. Stay foolish."

Steve Jobs

(Stanford Commencement Address, 2005)

POLITECNICO DI TORINO

Abstract

Electronics Engineering
Department of Electronics and Telecommunications

Master of Science in Electronics Engineering

Design of a fault tolerant instruction decode stage in RISC-V core against soft and hard errors

by Marcello NERI

Failures in electronic devices caused by radiation is one of the most challenging issues arising in the last decades. Nowadays, radiation effects are crucial not only in the space environment, but also at the sea level, since transistor downscaling is affecting the characteristics of integrated circuits. When operating in hostile environments, solid-state devices and integrated circuits may be directly struck by photons, electrons, protons, neutrons, heavy ions, or alpha particles causing alteration of their electrical properties. This puts at risk the reliability and integrity of those devices, leading also to possible catastrophic consequences if they occur in safety critical applications. The international standard IEC 61508 sets the requirements that a safety-related system must meet in order to be classified and certified according to its reliability level.

For what concerns the hardware design, the mitigation against radiation effects is possible by applying the concept of redundancy to all the components in the system. Processors, which are general purpose hardware devices, are of very common usage in many fields of application, sometimes operating also in hostile environments. This is the reason why they can be considered really critical components that need to be made fault tolerant.

In this thesis project, the fault tolerant design of the Instruction Decode (ID) stage of the CV23E40P core, which is a RISC-V core, implementing the RV32IMC instruction set, is presented. The work developed in this thesis is included in a wider project that aims to make the entire CV32E40P core fault tolerant. The proposed design makes use of Error Correction Code (ECC) and N-Modular Redundancy (NMR) techniques which ensure fault tolerance against Single Event Effects (SEEs) to all the component included in the stage. Specifically, the Hsiao code is one of the most suitable ECC from the hardware optimisation perspective. So, it is used in the design with Single Error Correction and Double Error Detection (SECDED) capabilities. As regards the NMR technique, for the purpose of the thesis, triplication (TMR) is the best trade-off between hardware overhead and fault tolerance level. In fact, the TMR uses the minimum level of redundancy to get capable to detect and correct single errors without suspending the program execution.

However, in the state-of-the-art, some RISC-V cores already use these techniques to mitigate transient errors. The innovative side of this thesis work is the design of a specific partial solution against the permanent errors, beside the traditional techniques used against the transient ones. Specifically, the most critical component in the ID stage from the radiation perspective is the register file, being the most extend-

ed component of the whole core. Its design provides for extra "supply" locations to replace the permanently damaged registers and allow the correct execution of the program in any of the most hostile situations.

The effectiveness of the fault tolerant design of the ID stage is evaluated by means of the simulation-based approach. One transient fault is injected to each simulation of the core running the CoreMark benchmark program. The system implemented for the fault injection is based on TCL scripts which exploit special functions of QuestaSim simulator for bit-flipping the memory elements present into the stage. Under these conditions, the fault tolerance level reached by the architecture against the soft errors is included between the 95% and the 100% with a confidence level of the 99%.

Acknowledgements

First and foremost, I would like to express my deep and sincere gratitude to my supervisor and co-supervisors, Prof. Stefano Di Carlo, Prof. Alessandro Savino, Prof. Maurizio Martina, Prof. Maurizio Masera, and Prof. Luca Maria Cassano, for giving me the opportunity to work on this thesis project and for guiding me throughout it. Their help, advice, and support was essential to overcome all the challenges faced during the work and succeed in reaching the objective.

Secondary, but not as importance, I would like to thank the Politecnico di Torino and the Department of Electronics and Telecommunications (DET), including all the professors I have been involved with, for the excellent education provided and for the innumerable opportunities of cultural and personal growth offered to me. I am really grateful and proud of having been part of it. What I learned during these years has given me the chance to continue pursuing my dreams and never stop, and I am extremely thankful for that.

I would like also to thank all the people that somehow and sometime have contributed to make my academic journey easier and pleasant.

I thank my family, my parents and my sister Giulia, for their continuous support, their constant presence, and the endless love they give to me that never made me feel away from home. This goal would not have been possible without them and I will be forever grateful for this.

I need also to thank all the rest of my family, my uncles, my aunts, my cousins and my grandparents. They shared with me all my achievements and rejoiced in them more than me every time. This has always represented for me a good reason to carry on pursuing my next goals stronger and more determined. I am sure my nonno 'Nzino would be proud for this goal as well.

A thanks goes to my second family, the Emmolo's family. They are able to make me feel at home everywhere and represented a stable reference point in Turin. Their warmth made my days simpler and nicer.

Thanks to my "Avolese" friend, Paolo, for the great time spent together, chilling out and having a good time. He was also the best quarantine-mate ever, drinking together our "Old Fashioned" and cooking everything.

Thanks to my friends, Luca, Kevin, Gigi, Jessica, and Carolina, for their constant and warm presence and for all the memorable moments spent together, both in classes and outside.

Thanks to my friends, Giuseppe, Gianluca, and Fra, for the countless days and nights spent at the "Ostello Ferotti", doing everything except for studying.

Thanks to the best teammates ever, Luca (again), Kevin (again), Matteo, and Elia, for having made easier and less stressful every class project during these years. I spent with them the best "educational" moments of my university career. Working with them was a pleasure for me. I learned a lot from them.

Another thanks to my far friend Gianni, that, despite the distance, has always been close to me.

Finally, a special and huge thanks to Giuliana. She is not only my girlfriend, but also my adventures-mate, my travel-mate, my classmate, my teammate. She shared with me every single moment, good or bad, during the whole university course. She supported me in every situation. She gives me the strength and the courage to pursue all my objectives and helps me in setting new ones. I can never stop saying thanks to her.

Contents

List of Figures

List of Tables

List of Abbreviations

Electronics

IC	**I**ntegrated **C**ircuit
CPU	**C**entral **P**rocessing **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
MOS	**M**etal-**O**xide **S**emiconductor
CMOS	**C**omplementary **M**etal-**O**xide **S**emiconductor
SRAM	**S**tatic **R**andom **A**ccess **M**emory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DICE	**D**ual-**I**nterlocked **Ce**ll
SOI	**S**ilicon **On** **I**nsulator
ISA	**I**nstruction **S**et **A**rchitecture
RISC	**R**educed **I**nstruction **S**et **C**omputer
SoC	**S**ystem **o**n **C**hip
FPGA	**F**ield **P**programmable **G**ate **A**rray
IPC	**I**nstruction **P**er **C**ycle
IF	**I**nstruction **F**etch
ID	**I**nstruction **D**ecode
EX	**Ex**ecution
MEM	**Mem**ory Access
WB	**W**rite **B**ack
FIFO	**F**irst-**I**n **F**irst-**O**ut
RF	**R**egister **F**ile
MBIST	**M**emory **B**uilt-**I**n **S**elf **T**est
FSM	**F**inite **S**tate **M**achine
ALU	**A**rithmetic **L**ogic **U**nit
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
FPU	**F**loating-**P**oint **U**nit
APU	**A**uxiliary **P**rocessing **U**nit
LSU	**L**oad-**S**tore **U**nit
PMP	**P**hysical **M**emory **P**rotection
OBI	**O**pen **B**us **I**nterface
RTL	**R**egister **T**ransfer **L**evel
MSB	**M**ost **S**ignificant **B**it
LSB	**L**east **S**ignificant **B**it

Standard

E/E/PE	**E**lectrical/**E**lectronic/**P**rogrammable **E**lectronic
IEC	**I**nternational **E**lectrotechnical **C**ommission

CENELEC	**C**omité **E**uropéen de **N**ormalisation en **É**l**e**ctronique et en **É**l**e**ctrotechnique
SIL	**S**afety **I**integrity **L**evel
SFF	**S**afe **F**ailure **F**raction
SC	**S**ystematic **C**apability

Radiation

TD	**T**otal **D**ose
SEE	**S**ingle **E**vent **E**ffect
TID	**T**otal **I**onizing **D**ose
TNID	**T**otal **N**on-**I**onizing **D**ose
DD	**D**isplacement **D**amage
SER	**S**ingle **E**vent **R**ate
SEE	**S**ingle **E**vent **E**ffect
SET	**S**ingle **E**vent **T**ransient
SEU	**S**ingle **E**vent **U**pset
SBU	**S**ingle **B**it **U**pset
MBU	**M**ultiple **B**it **U**pset
MCU	**M**ultiple **C**ell **U**pset
SEFI	**S**ingle **E**vent **F**unctional **I**nterrupt
SEL	**S**ingle **E**vent **L**atch-up
SEGR	**S**ingle **E**vent **G**ate **R**upture
SEB	**S**ingle **E**vent **B**urnout

Fault Tolerance

FT	**F**ault **T**olerance
NMR	**N**-**M**odular **R**edundancy
TMR	**T**riple-**M**odular **R**edundancy
ECC	**E**rror **C**correction **C**ode
SEC	**S**ingle **E**rror **C**correction
DED	**D**ouble **E**rror **D**etection

Chapter 1

Introduction

The industry of integrated circuits (ICs) keeps on scaling transistor size, increasing devices density, raising clock frequencies, and lowering operating core voltages, according to the Moore's Law [1]. The effects of such progress may lead to the onset of some new mechanisms of soft errors in digital systems, causing an increase in the probability of system failures. The radiation represents one of the main causes of system failure and is today a real problem not only in the traditional hostile environment, such as space, but also in everyday environments. In fact, initially, the radiation effects were avoided in daily electronic devices because an older technology made use of larger transistors, which are more robust and tolerant against radiation effects. So, the radiation was a problem only in space equipment. Now, as technology improves, bit-flipping induced by radiation is increasingly likely since the amount of energy required by the smaller transistors to cause such bit-state alteration became smaller as well. This makes possible that such soft errors are experienced, for example, also at normal flight altitudes for civilian aviation [2].

This emerging problem pushes the research towards the necessity of developing fault tolerant systems capable to keep working even in presence of such phenomena. In the fault tolerance context, it is important to distinguish among three fundamental concepts: fault, error, and failure. Fault is defined as the abnormal working condition of a component in a system due to physical effects, which can lead the system to perform in an unintended manner. If the system performs in an unintended manner because of one or more faults, an error occurs. Finally, an error becomes a failure, if the system is not capable to deliver the specified external services anymore due to that error.

In order to reach fault tolerance goals, many different techniques can be used to protect an electronic system against radiation effects, mainly based on the concept of redundancy. Redundancy is simply the addition of information, resources, or time besides what is needed for the normal system operation. In hardware, the resources redundancy applied to entire blocks is called N-Modular Redundancy (NMR) and exploits the majority voting of all the replicas of the module to evaluate any possible error. The redundancy related to the information gives raise to Error Correction/Detection Code (ECC or EDC) which adds extra information to correct/detect errors, such as parity bit, and so on. The temporal redundancy, instead, exploits the re-execution of some operations to discern the correctness of the outputs.

One of the most interesting field of application for the fault tolerance concept is the general purpose processor (or Central Processing Unit, CPU), and the reason lies mainly on its heavy usage in several applications. It is clear that if a failure occurs during the normal operation condition of the CPU, and in particular if this happens in critical applications, dangerous consequences can follow, just think of applications like respiratory support devices, heart-rate monitors, and other life supporting

equipment in hospitals, or like control systems in nuclear plants.

Among the large variety of microprocessors existing in the market, particular focus can be posed on the RISC-V cores. In the last years, the employment of RISC-V processors is increasing markedly thanks to its great potential. The RISC-V is a free and open Instruction Set Architecture (ISA) and it is suitable to be used in a large variety of applications. In fact, only the ISA is provided, giving the possibility to the designer to add any kind of extra features to the core to satisfy any specific application requirements.

A fault tolerant version of the RISC-V core would increase its real potentialities. By adding extra fault tolerant capabilities, the RISC-V core could be used in a greater number of applications, which ranges from the traditional use as a CPU for everyday scopes, to the use in dangerous control systems, or even to the use in more sophisticated environments, like military experiments or aerospace missions.

1.1 Thesis objectives

The presented thesis work is part of a wider team project that aims to design a fully fault tolerant RISC-V core. Being a very demanding project, the fault tolerant design has been divided into three parts (as the number of the teammates), each one related to a part of the core. According to this approach, this thesis project puts its focus on the fault tolerant design of the Instruction Decode (ID) stage, while the remaining parts of the core are taken into account by the other two teammates, Luca Fiore and Elia Ribaldone.

The main objective of this thesis project is the development of a configurable ID stage in the RISC-V core where different levels of fault tolerance can be selected. The configuration can create up to 24 different architectures, from a fully protected version, with the highest level of fault tolerance, to a single-component protected version, with a reasonable level of fault tolerance according to the necessities. In fact, in some scenarios, it is advisable to have a fully protected core, but sometimes it is better (best trade-off) to protect only the Register File, which is a very critical component, avoiding the waste of power and resources.

From the main objective, another important sub-objective is derived, that is the design of a particular version of the Register File capable to manage also the permanent errors, besides the transient ones. In this way, the permanently damaged locations can be easily replaced run-time without stopping the execution. This version is also configurable. In fact, it is possible to set the threshold beyond which the permanent fault is detected. Moreover, the design aims to keep the information about the damaged locations into the non-volatile memory, through the Control and Status registers inside the core, so that at the power-on reset the machine is capable to work correctly starting immediately.

The other main objective is the implementation of a development environment for RISC-V cores which provides several features for the designer. The main features implemented are compilation of C-programs, simulation support, comparison with the reference architecture, fault injection simulation, fault tolerance benchmark, results analysis, and other minor options which facilitate the development of the project.

1.2 Thesis overview

This thesis is organised in a book-style way, so for each chapter a different argument is treated and detailed. Overall, it consists of eight chapters, which specifically can be grouped into three subsets: the first one (chapters 1, 2, 3, and 4) relates to the background knowledge, the second one (chapters 5, and 6) to the development of the proposed design, and the third one (chapters 7, and 8) to the presentation and discussion of the overall results.
In the following, the complete list of the chapters flanked by a brief description is reported.

- Chapter 2 - the International Standard IEC 61508 is briefly described, highlighting the requirements for the design of hardware components in a safety-related system;
- Chapter 3 - the effects of radiation on digital circuits are reported, and a summary comparison of terrestrial and space environment is given, providing the proper terminology;
- Chapter 4 - the most used fault tolerant techniques for hardware design are reported;
- Chapter 5 - the RISC-V specifications are listed and the core used in this thesis project is illustrated;
- Chapter 6 - the design of the fault tolerant version of the Instruction Decode stage is explained, justifying the choice of fault tolerant techniques applied;
- Chapter 7 - the fault tolerance levels reached by the Instruction Decode stage, and by each component in it, are reported and compared with respect to the state-of-the-art levels;
- Chapter 8 - final comments on the entire work of this thesis project are given.

The chapter 6 and 7 are the ones concerning the personal work developed during the thesis project, so they can be considered the most important ones. However, for a good understanding of the overall context, the first five chapters are essential.

Apart from the chapters, the thesis provides an appendix where the relevant parts of some algorithms or codes are reported. This makes the understanding of some algorithms easier. Anyway, the complete codes and the entire project files can be found on the following GitHub repositories (links below), which have been used for the development of the project.

- Reference Core repository: `https://github.com/RISKVFT/cv32e40p/tree/master`
- Fault Tolerant Core repository: `https://github.com/RISKVFT/cv32e40p/tree/FT_Marcello`
- Development Environment repository: `https://github.com/RISKVFT/core-v-verif/tree/FT_verif_Marcello`
- General Documentation repository: `https://github.com/RISKVFT/RISKV_FT_Docs`

Chapter 2

International Standard

A fault-tolerant system, or in general a safety-system, aims to prevent dangerous failures or to control them when they arise with a certain probability of success. In industrial and commercial contexts, it is necessary to observe some standards in order to determine the safety level of components involved in a process, or in an equipment, and to ensure a tolerable level of risk.

As regards electronic devices and software, functional safety is one of the main aspects of the overall safety of the system. It is related to "the function of a device or system and ensures that it works correctly in response to commands it receives or fails in a predictable (safe) way". For example, functional safety regards: fire and gas control systems, car's airbag protection systems, automated flight control systems, etc.

In 1998, the International Electrotechnical Commission (IEC) published a series of standards, the IEC 61508, for electrical, electronic and programmable electronic (E/E/PE) safety-related systems. Currently, its second version, published in 2010, is effective and it is used by a wide range of manufacturers, system builders, designers and suppliers of components and subsystems and serves as the basis for conformity assessment and certification services [3].

Moreover, the IEC 61508 standards were approved by CENELEC (Comité européen de normalisation en électronique et en électrotechnique) as a European Standard EN 61508

2.1 IEC 61508

IEC 61508 series are the International Standards consisting of methods on how to apply, design, deploy and maintain automatic protection in E/E/PE systems, called safety-related systems. In other words, IEC 61508 regulates the entire life-cycle of products and systems related to safety [4]. In this context, functional safety measures the risk by evaluating the probability that a given event will occur and how severe it would be. IEC 61508 applies to a safety-related system when one or more of E/E/PE devices are incorporate in the system itself. It covers possible hazards caused by failure of the safety functions to be performed by the E/E/PE safety-related system, as distinct from hazards that may derive from the E/E/PE equipment itself (for example electric shock, etc). The standard is compliant with all E/E/PE safety-related systems regardless of the application [5].

The level of risk due to a failure of the system or equipment under control is assessed with a probabilistic approach, i.e. as a function of the frequency or probability of occurrence of the failure and the severity of the consequences of the failure itself, that could affect the safety of persons and/or the environment. Four safety levels,

called safety integrity levels (SILs), are defined according to the risks involved in the system application, which give a quantitative measure (order of magnitude) of the necessary risk reduction and therefore the degree of reliability that the safety-system must achieve. SIL1 has the lowest level of risk reduction (the least safe), SIL4 has the highest level of risk reduction (the most safe).

2.1.1 Safety life cycle

The objective of the safety life cycle is to define the safety functions of the safety-related system, firstly defining the scope of the system, secondly assessing the potential system failures and estimating the risks they can cause. Then, safety requirements need to be specified and met by different components of the system. So, it is required to develop and document a safety plan, execute that plan, document its execution (to show that the plan has been met) and continue to follow that safety plan through to decommissioning with further appropriate documentation throughout the entire life of the system [6]. The life cycle's flow is shown in Figure 2.1.

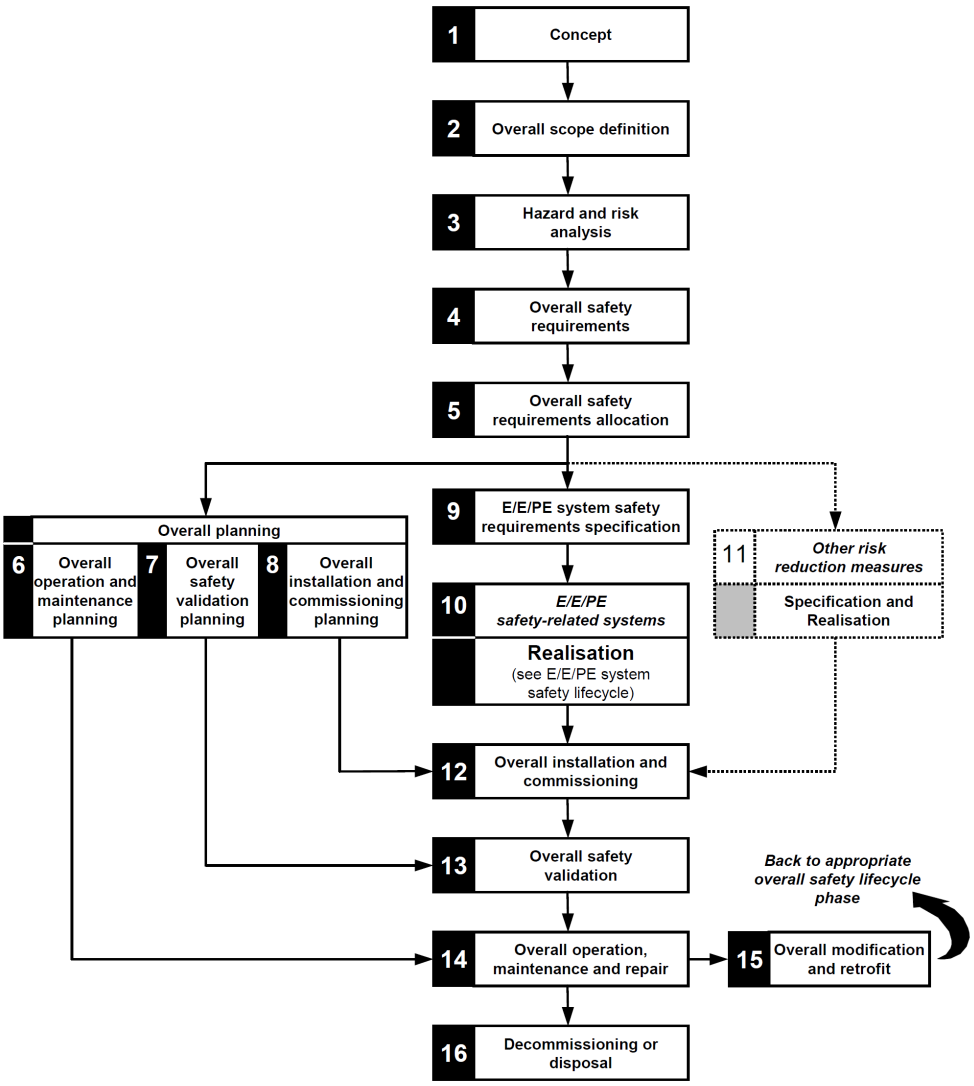

FIGURE 2.1: Safety life cycle [6]

In a simpler way, the safety life cycle can be seen as a continuous "identify-assess-design-verify" loop.

2.1.2 Safety Integrity Level (SIL)

IEC 61508 defines four levels of safety (SIL), from 1 to 4. Note that a safety integrity level is a property of a safety function rather than of a system or any part of a system and it is determined depending on the mode of operation of the safety function.

The standard describes two modes of operation for a safety function: low demand mode and high demand (or continuous) mode of operation. A safety function operating in low demand mode is only performed when required in order to keep the system into a safe state. While, a safety function operating in continuous mode continuously controls the state of the system. According to this, the SIL of a safety function is evaluated in two different ways:

- the average probability of a dangerous failure on demand (in the case of low demand mode) (Table 2.1);
- the average frequency of a dangerous failure per hour (in the case of high demand or continuous mode) (Table 2.2).

Safety Integrity Level (SIL)	Average probability of failure on demand (Low Demand mode)
4	$\geq 10^{-5}$ *to* $< 10^{-4}$
3	$\geq 10^{-4}$ *to* $< 10^{-3}$
2	$\geq 10^{-3}$ *to* $< 10^{-2}$
1	$\geq 10^{-2}$ *to* $< 10^{-1}$

TABLE 2.1: SILs for Low Demand mode of operation [6]

Safety Integrity Level (SIL)	Average frequency of dangerous failure per hour (Continuous mode)
4	$\geq 10^{-9}$ *to* $< 10^{-8}$
3	$\geq 10^{-8}$ *to* $< 10^{-7}$
2	$\geq 10^{-7}$ *to* $< 10^{-6}$
1	$\geq 10^{-6}$ *to* $< 10^{-5}$

TABLE 2.2: SILs for Continuous or High Demand mode of operation [6]

At first glance, the continuous mode appears to be far more stringent than the demand mode. It is important to notice that the continuous mode is evaluated per hour, while the demand mode is evaluated in a time interval of roughly one year per the definition. If it assumed that about 10,000 hours are in a year (actual 8,760), the modes are approximately the same in terms of safety metrics [6]. In fact, the average frequency of dangerous failure is evaluated starting from the average probability of failure on demand.

For example, if an E/E/PE safety-related system operating in continuous mode is used in a mission with a certain time duration during which no repair can take place, the required SIL for a safety function can be derived simply by: determining the required probability of failure of the safety function during the mission time, and dividing this value by the mission time, to obtain the required frequency of failure per hour. Then, the Table 2.2 can be used to derive the SIL.

Any device, sensor, control & command unit, and in general any component being part of a safety-related system, needs to be classified according to its related SIL. All these subsystems, elements and components, when combined to implement the safety function (or functions), are required to meet the safety integrity level target of the relevant safety functions. Also subsystems and components supplied by other producers need to be assessed again even if they have been already quoted as "suitable for the required safety integrity level target". The standard requires to assess all the components of the system throughout all the phases of the lifecycle under the functional safety perspective. Finally, the compliance of the E/E/PE safety-related system with the IEC 61508 standard needs to be demonstrated.

2.1.3 IEC 61508 overview

The concepts explained above are the two fundamental concepts the IEC 61508 standard is based on: the safety life-cycle (defined as "an engineering process that includes all of the steps necessary to achieve required functional safety") and safety integrity levels. Fundamental concepts apart, the total standard is divided into seven parts which define the requirements to satisfy in order to be compliant with the standard. They are:

Part 1 - General requirements (required for compliance);

Part 2 - Requirements for E/E/PE safety-related systems (required for compliance);

Part 3 - Software requirements (required for compliance);

Part 4 - Definitions and abbreviations (supporting information);

Part 5 - Examples of methods for the determination of safety integrity levels (supporting information);

Part 6 - Guidelines on the application of parts 2 and 3 (supporting information);

Part 7 - Overview of techniques and measures (supporting information).

Describing all parts of the norm is beyond the scope of this study. However, part 2 is relevant and of particular interest for the design of a fault tolerant hardware system compliant with the standard, and it will be treated more in detail. While part 3 would be important if the safety-related system makes use of software.

2.1.4 IEC 61508 - Part 2

Part 2 [7] covers the hardware requirements for safety-related systems. Specifically, it specifies: how to refine the E/E/PE system safety requirements specification into the E/E/PE system design requirements specification; the requirements for activities involved in the design and manufacture processes (i.e. establishes the E/E/PE system safety lifecycle model), including the application of techniques and measures that are graded against the safety integrity level, for the avoidance of, and control of, faults and failures; and the information necessary for the final safety validation of the E/E/PE safety-related systems.

As previously explained, in this part the requirements for activities involved in the design process are specified. Pursuing this, a detailed V-model (shown in Figure 2.2) is proposed for the development lifecycle for the design of ASICs. The model starts with the activity related to the ASIC safety requirements specification and ends with the validation test, passing through behavioural modelling, synthesis, simulation and verification activities.

In general, requirements specification activity requires the individuation of all the subsystems and elements composing the E/E/PE safety-related system and of all the safety functions. As a consequence, the requirements for the integration of the subsystems to meet the safety functions requirements need to be specified. In this way, hardware (and software) specification are set. Safety functions requirements may regard: how safe state is achieved, response time, operator interfaces, operating modes of equipment under control, required E/E/PE behavior modes, start-up requirements, etc.

The next activity concerns the planning for the validation of the safety of the E/E/PE safety-related system. This task shall consider the requirements of the E/E/PE system previously defined, the procedures to validate the implementation and the safety integrity of the safety functions, the testing environment and other testing criteria.

Parallel to the E/E/PE system safety validation planning, the design of the system (including ASICs) takes place considering it shall meet many requirements for safety. In particular for integrated circuits (ICs), on-chip redundancy is a special requirement to be taken into account, as well as requirements for systematic safety integrity (systematic capability). However, requirements for hardware (and software) are set

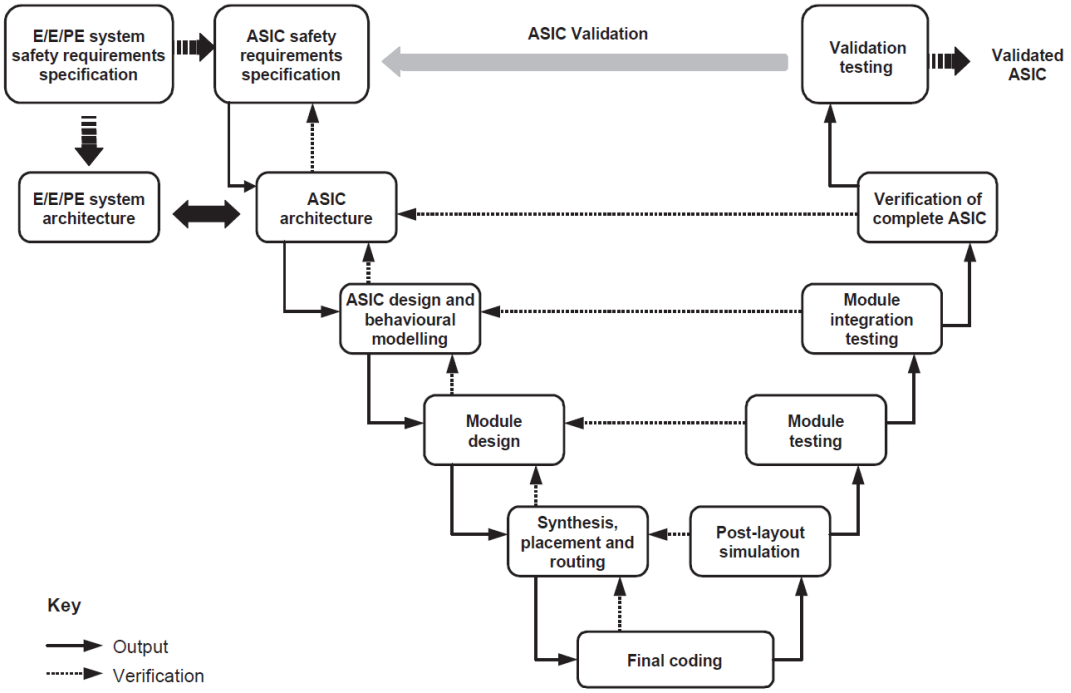

FIGURE 2.2: ASIC development lifecycle [7]

on the basis of the highest safety integrity level associated to any safety functions implemented by that hardware.
When the initial design has been completed, an analysis shall be undertaken to determine whether any reasonably foreseeable failure of the system could cause a hazardous situation or require a risk control measure. If any of these effects occur, the design of the E/E/PE safety-related system should be changed to avoid such failures. In the case a new design cannot be done, then the likelihood of such failure modes needs to be reduced to a level commensurate with the target failure measure. Note that there may be cases where the failure rate of the specified failure modes cannot be reduced and either a new safety function will be required or the SIL of the other safety functions reconsidered taking into account the failure rate.
Also the systematic faults that lead to a failure of the safety functions should be taken into account in order to determine the systematic capability (SC) of each element composing the system. With SC is intended a confidence level which defines the maximum SIL level achievable by the element/subsystem in terms of systematic integrity.
As regards hardware safety integrity, the safety integrity level referred to a safety function is limited by the constraints deriving from implementing one of two possible routes (to be implemented at system or subsystem level for hardware safety integrity requirements):

- Route 1_H based on hardware fault tolerance and safe failure fraction concepts;
- Route 2_H based on component reliability data from feedback from end users, increased confidence levels and hardware fault tolerance for specified safety integrity levels.

The choice of the route is application and sector dependent.
However, hardware fault tolerance is classified with an integer value N which means that "N+1 is the minimum number of faults that could cause a loss of the safety

function". This requirement is defined according to the security level of integrity required for each safety function, and according to the type of subsystem. There can be two types:

- type A: if the mode of failure of any single component of the subsystem is well defined, so it is possible to determine its behaviour in any situation;
- type B: if the mode of failure of any single component of the subsystem is not well defined and it is not possible to completely determine its behaviour.

Route 1_H is based around the Safe Failure Fraction (SFF) calculation approach. The SFF is "the fraction of the overall failure rate of a device that results in either a safe fault or a diagnosed (detected) unsafe fault" [6]. This parameter is evaluated together with the SIL to determine the minimum level of hardware fault tolerance required to meet the target. The minimum level of hardware fault tolerance is determined differently depending on the type of subsystem.

Safe Failure Fraction (SFF)	Hardware Fault Tolerance		
	N=0	N=1	N=2
$< 60\%$	SIL 1	SIL 2	SIL 3
[60%, 90%[	SIL 2	SIL 3	SIL 4
[90%, 99%[	SIL 3	SIL 4	SIL 4
$\geq 99\%$	SIL 3	SIL 4	SIL 4

TABLE 2.3: SIL required for type A subsystem with N hardware fault tolerance [6]

Safe Failure Fraction (SFF)	Hardware Fault Tolerance		
	N=0	N=1	N=2
$< 60\%$	Not allowed	SIL 1	SIL 2
[60%, 90%[	SIL 1	SIL 2	SIL 3
[90%, 99%[	SIL 2	SIL 3	SIL 4
$\geq 99\%$	SIL 3	SIL 4	SIL 4

TABLE 2.4: SIL required for type B subsystem with N hardware fault tolerance [6]

At this point, the route 1_H sets some rules to follow in order to determine the SIL of the E/E/PE safety-related system. The rules refer to the theoretical hierarchy of the

system. In fact, a system implementing a safety function is composed by subsystems which in turn can be composed by one or more channels (parallel elements) each one composed by serial elements. So, first of all, it is important to determine the SIL of each element. This is done by evaluating for each channel the SIL for the achieved SFF for a hardware fault tolerance of 0. Then, the SIL of the channel is determined by taking the lowest SIL reached by all the elements composing that channel. Then, the subsystem is considered and its SIL is determined by taking the highest SIL that has been achieved by the related channels and by adding N (level of hardware fault tolerance for the subsystem) SIL to it. Finally, the maximum SIL that can be claimed for the complete E/E/PE safety-related system is determined by the subsystem that has achieved the lowest SIL.
Route 2_H, instead, does not consider SFF at all. The minimum hardware fault tolerance for each subsystem of a E/E/PE safety-related system should be determined according to the following table (Table 2.5), that is based on the mode of operation, and the SIL required for the safety function implemented by the system.

Safety Integrity Level (SIL)	Hardware Fault Tolerance (N)	
	Low demand mode	High demand mode
SIL 4	N=2	N=2
SIL 3	N=1	N=1
SIL 2	N=0	N=1
SIL 1	N=0	N=0

TABLE 2.5: Hardware fault tolerance required for SIL [7]

Sometimes, for type A elements only, if a hardware fault tolerance greater than 0 is required, probably it is advisable to change the architecture with a safer one with reduced hardware fault tolerance.

Apart from the routes, part 2 of the standard provides also some techniques and measures that shall be used in order to prevent the introduction of faults during the design and development of the hardware and software of the E/E/PE safety-related system. In the case a fault is detected, the part sets the requirements for the behaviour of the system.

After the design and development phase, an adequate documentation must be produced in order to prove that the E/E/PE safety-related system has reached the required SIL for the safety functions taken in consideration. The documentation should include: conditions of use during the test, impact analysis on the difference between the intended operation and the previous operation experience with related demonstration of equivalence, satisfactory evidence that elements not covered by previous demonstration cannot affect the SIL of the system.
Then, the E/E/PE safety-related system shall be integrated according to the specified E/E/PE system design and shall be tested according to the specified E/E/PE system integration tests. The next task requires to test and evaluate the outputs of a

given phase to ensure correctness and consistency with respect to the products and standards provided as input to that phase. The verification process shall be planned concurrently with the development, for each phase of the E/E/PE system safety lifecycle, and shall be documented. Lastly, it is necessary to validate that the E/E/PE safety-related system meets all the requirements for safety in terms of the required safety functions and safety integrity. The validation process (referred to both hardware and software) is carried out following a prepared plan and providing simultaneously an appropriate documentation. To facilitate this task, a consistent group of suggested techniques and measures is provided in Annex B of Part 2 of the standard.

Once the E/E/PE safety-related system is validated, the standard requires to provide procedures to ensure that the required functional safety of the system is maintained during operation and maintenance, and also procedures to make corrections, enhancements or adaptations to the system, always ensuring that the required safety integrity is achieved and maintained. Every modification shall be documented adequately.

In annexes of this part of the standard, many techniques and measures to adopt during the development and maintenance of E/E/PE safety-related systems are listed. In particular as regard hardware safety integrity, in Annex A (not reported in this documentation), Table A.1 provides the requirements for faults or failures that shall be detected by techniques and measures to control hardware failures, instead Tables A.2 to A.14 support the requirements of Table A.1 by recommending techniques and measures for diagnostic tests. Also techniques and measures recommended to reach a certain SIL are reported in other tables. In Annex B, techniques and measures to avoid systematic failures during the different phases of the lifecycle are presented, categorising the failures in before or during system installation and in after system installation. Annex C is useful to determine the diagnostic coverage and to calculate the SFF of a hardware element. Finally, Annexes E and F set some requirements and provide techniques and measures for the development of ICs with on-chip redundancy and ASICs.

Chapter 3

Radiation environment

Failures caused by radiation is one of the most challenging issues arising in the last decades. In fact, nowadays electronic systems are operating in a number of hostile environments. When operated in these environments, solid-state devices and ICs may be directly struck by photons, electrons, protons, neutrons or heavy ions, causing alteration of their electrical properties and consequently failure of its operation. Such a scenario may be a danger in some circumstances, such as in aerospace and avionic applications, high energy physics experiments, nuclear plants, and military environments. Moreover, this is relevant also for medical diagnostic imaging and therapy, industrial imaging and material processing [8].

Radiation that affect electronic systems come from Sun (solar wind and solar particle events), galactic cosmic rays (and related interactions with atmospheric elements), radiation belts (planets with magnetic fields), and radioactive impurities and materials [9]. They are present both in space and in Earth, obviously in different doses and arising different kinds of problem.

Regardless of the environment, radiation effects in electronic devices can cause several kinds of problem (or even damage), both reversible or irreversible, leading to transient or permanent errors. In very critical scenarios, it is crucial to mitigate and solve both of them. Anyway, the objective of this thesis work is to be tolerant to transient errors, so major focus is posed on them and the related causes. Other kinds of error are rapidly described.

3.1 Radiation effects

The effects of radiation on electronic components can be divided into two main classes: Total Dose (TD) effects (also called cumulative effects), which lead to a progressive degradation of semiconductor devices characteristics (e.g. electron/hole mobility, oxide properties, etc), and Single Event Effects (SEEs), which can lead to destructive (e.g. latch-up) or non-destructive (e.g. bit flip) damages due to charge deposition induced by a single particle. So, SEEs occur in a stochastic way and are related only to a small part of the device since caused by single particles, while TD is cumulative and may become visible only after some time since it depends on the duration of exposure to radiation which uniformly affect the whole device, because it results from the effect of several particles randomly hitting the device for the entire exposure interval [8]. It is evident that TD is more dangerous than SEE, in fact the first one causes hard errors (permanent physical damages), while the latter can cause soft (loss of information) and hard errors [10].

3.1.1 Cumulative effects

Cumulative effects are those ones deriving from the progressive exposure of electronic devices to radiation, so they are usually related to long-term changes in devices. The consequences, at device level, can be the generation of trapped charges in insulators, interface traps, and defects in crystal lattice. These are effects which lead to a permanent drift of device characteristics, resulting in malfunctioning of the entire electronic system that needs to be repaired (or substituted). Generally, TD effects are serious concerns in harsh radiation environment such as in the space, and they are almost negligible in human living environment [10].
This kind of effects can be subdivided again into two categories: Total Ionizing Dose (TID), and Total Non-Ionizing Dose (TNID), also called Displacement Damage (DD), effects. Both of them, as previously mentioned, lead to hard errors.

Total Ionizing Dose (TID)

This effect takes place thanks to the ionising energy transferred by radiation to component materials. The energy is exploited to generate electron-hole pairs (free charge carriers), which diffuse or drift to other locations of the device where they may get trapped (typically in dielectric layers), leading to unintended concentrations of charge and parasitic fields. This produces several effects on the device characteristics, such as flat-band and threshold voltage shifts, leakage currents and timing skews [11].
TID affects mainly devices based on surface conduction, so it is of concern for ICs, since they are mainly fabricated using MOSFET devices. This kind of damages is the primary effect of exposure to X- and γ-rays and charged particles [8].

Total Non-Ionizing Dose (TNID)

This effect is caused by the interaction of energetic non-ionizing particles, such as neutrons, protons, and electrons, with device component atoms. Incident energetic particles scatter off lattice ions, locally deforming the crystal lattice and creating permanent defects in it. This gives raise to changes in semiconductor electronic properties. DD effects are strictly dependent on the incident particle type, incident particle energy, and target material.
Also this effect is cumulative as TID, but differently this effect is mainly concerned to bulk conduction based devices (e.g. BJT, JFET, etc) [8].

3.1.2 Single Event Effects

Single event effects are caused by the charge deposited by a single ionizing particle. More in detail, the energetic incident particle travels through a device region and loses energy by ionizing the device material and creating electron-hole pairs (free charge carriers). If an amount of charges higher than a threshold is collected at a device junction, then a short-lived but intense current pulse that can modify the electric state of the nearby elements (circuit level effects) is produced [12]. In other words, a SEE is produced.
The SEE is produced on sensitive nodes. Sensitivity of nodes is strictly linked to the charge collection threshold for the single event, that is called the critical charge. So, if the critical charge for a device is reduced (more sensitive node), then its Single Event Rate (SER) is increased. The critical charge depends on several factors, such as power

supply values, temperature and also clock frequency (even if some experimental results disagree) [12]. Generally, the probability for a SEE to occur is measured with the cross-section parameter (σ_{SEE}), expressed as:

$$\sigma_{SEE} = \frac{N.\,of\ events}{fluence}$$

where $fluence$ is expressed in $N_{particles}/cm^2$, so σ_{SEE} is expressed in cm^2.

However, SEEs are classified into destructive (hard errors) or non-destructive (soft errors). The first type of errors includes Single Event Latch-up (SEL), Single Event Gate Rupture (SEGR), and Single Event Burnout (SEB). The latter, instead, includes Single Event Transient (SET), Single Event Upset (SEU), and Single Event Functional Interrupt (SEFI).

SET (non-destructive)

A SET (Single Event Transient) is a temporary voltage excursion (voltage spike) at a node in an IC. In combinatorial digital circuits, this leads to erroneous data values propagation through gates. This event can occur also in analogue circuits.

SEU (non-destructive)

Similar to SET, in SEU (Single Event Upset) the voltage excursion induces the change of state of a storage element (e.g. flip-flop, latch, SRAM cell, etc). SEU produces bit-flip in memories and sequential logic.

SBU, MBU & MCU (non-destructive)

These errors are subcategories of SET and SEU. Single Bit Upset (SBU) produces a single bit-flip in the storage component. Multiple Bit Upset (MBU) includes two or more error bits in the same word of the storage component. Multiple Cell Upset (MCU) means that two or more error bits in different cells of the storage component occur. Typically, when MCU takes place, the corrupted cells are physically adjacent.

SEFI (non-destructive)

In complex devices, such as microprocessors or modern memories, SETs or SEUs may not be directly detected when occurred. This leads the device to continue to operate but in an unpredictable manner. At this point, the device (if well-designed) may go to recovery state, reset, lock-up, or otherwise malfunction in a detectable way. So, a SEFI (Single Event Functional Interrupt) is an SEE that places a device in an unrecoverable mode, often stopping the normal operation of the device. This kind of errors does not damage the device, but may cause loss of data and complex recovery actions.

SEL (destructive)

SEL (Single Event Latch-up) error provides for the activation of parasitic bipolar structures (mainly existing in CMOS circuits), with all the negative consequences

that may result also destructive for the device because of thermal effect (high-current flows through the circuit). An SEL can only be solved by power cycling the device.

SEGR (destructive)

SEGR (Single Event Gate Rupture), or Single Event Dielectric Rupture (SEDR), is barely the rupture of gate oxide (or any dielectric layer) occurring especially in power MOSFETs caused by a single ion strike. The energy transfer and damage induced by energetic heavy ions in dielectrics is so fast in comparison to the time response of any electrical protection (e.g. filtering), that there is no possible protection against this kind of error [11].

SEB (destructive)

SEB (Single Event Burnout) is the triggering of the parasitic bipolar structure in a power transistor (typically n-channel), accompanied by regenerative feed-back, avalanche and high current condition. SEB is potentially destructive unless suitably protected. This error is not so frequent in ASICs and FPGAs [11].

3.1.3 Fault attacks

Besides SEEs caused by physical causes, SEEs can also be caused on purpose. At the beginning, this kind of fault happened accidentally. But in a short time, fault attacks have been exploited to disturb the functioning of a device or retrieve secret information intentionally. The most common techniques to inject a fault are: introduction of clock glitches or voltage spikes, underpowering the system, temperature-based attacks, and light, radiation or magnetic attacks [13].

3.2 Space vs Terrestrial environment

The Sun, with its radioactive events such as solar flares, coronal mass ejections, and solar winds, is one of the main source of radiation. Other sources of radiation are radiation belts of planets with magnetic fields, or come from outside the solar system and produce galactic cosmic rays (their true nature is still under investigation).

Surely, space environment is much more bombarded by radiation than the Earth which is protected by its atmosphere (particles flux in Earth is many orders of magnitude lower than the one in space). Anyway, while in the past radiation effects were a concern only for applications devoted to operate in space vessels or particle accelerators, today they are considered a critical problem also for standard systems operating in normal condition at sea level. In fact, energetic charged particles, mainly electrons, protons and heavy ions, are encountered in interplanetary space and also in the magnetosphere of planets, including the Earth's one. Cosmic radiation bombards the Earth like a rain of charged particles, but only the most energetic ones reach sea level, the less energetic ones are reflected by the earth's magnetic field or absorbed in the atmosphere, creating in turn a continuous rain of secondary particles. Actually, the main type of energetic particles overcoming the atmosphere is neutron. Even though these particles have no electric charge (so no ionizing effects), they are capable to ionize materials by means of secondary mechanisms, e.g. hitting atoms that are then ejected from the lattice becoming the new ionizing agents.

The first reported occurrence of SEE on Earth due to cosmic rays (95% neutrons) [9] was in 1979 and was reported by J. F. Ziegler & W. A. Lanford [14]. Moreover, altitude and latitude influence the level of neutrons flux [12]. The flux assumes values ten times higher every $3000m$ of altitude with respect to the value at sea level, and values 5 times lower on the Equator than on the Poles.

Another source of radiation is the device itself. During the manufacturing process, traces of radioactive elements can contaminate the production chain and the device itself becomes a source of α-particles. One of the first occurrences of this kind of event was reported in 1978 by T. C. May & M. H. Woods [15]. The α-particles were emitted by the radioactive decay of uranium and thorium which were present in parts per million levels in packaging materials and solders of DRAMs. When the particles penetrate the die surface, they can create enough electron-hole pairs near a storage node causing random SEEs.

However, the probability of a basic device experiencing an SEE at sea level is very low. Nevertheless, there are some particular cases where the probability is not negligible anymore or cases where the probability of errors must be null. For example, servers, workstations, extended systems, or medical devices such as pacemakers that cannot allow any error [12]. This reflects on the possibility for electronic devices at sea level to incur only in SEEs, since they are caused by single particles, and not in TD effects, which are related to cumulative events. For achieving high reliability of electronic components (e.g. processors), so designers need to pay attention to SEEs. Only for particular medical devices, TD tolerance is required, for example for chip in medical electronic tags because they are sometimes sterilized by γ-rays .

As regard space environment, electronic equipment on board spacecrafts is exposed to a multitude of highly energetic particles (protons, heavy ions, etc) coming either from the Sun activity or from the galaxy core. In this scenario, radiation is capable to penetrate spacecrafts and ICs packaging causing any of the possible effects previously described (both destructive or non-destructive). Space is a very hostile environment for electronic devices, both for the high levels of radiation and for the fact that generally spacecrafts live in space for very long time. So, TD effects have a not negligible probability to occur. Moreover, radiation in space are highly dynamic, therefore design and test of electronic components requires much effort, since high uncertainty is present [11].

3.3 Solutions against radiation effects

In order to face the problems caused by radiation effects, several solutions can be adopted, ranging from technology to circuit and from design to system level. Some of them aim to decrease the sensitivity to radiation, while others aim to reduce the probability of error caused by radiation effects.

TD effects are not of concern for this thesis work, so related solutions will be only touched upon. For example, as regard TID effects occurring in MOS transistor, device parameters affected by this event are threshold voltage, that is a key parameter related to circuit power consumption and speed, and leakage current. To overcome the problem of the increase of power consumption, usually designers may add significant margin to their power requirements to allow the digital circuit to work still properly despite TID [16]. A solution against TNID, instead, could be the use of

shields to reduce the impact of electrons and low-energy protons, but generally this does not reduce the SER caused by high-energy cosmic rays. On the contrary, thick shields can increase the SER rate because of the creation of multiple secondary particles due to interactions between the cosmic rays and the shield material [16].

As regard SEEs, many solutions can be applied at different levels. In the following subsections, some of them are presented classified on the basis of the application level.

3.3.1 Technology-based solutions

A first general solution at technology level may be the avoidance of using elements quite sensitive to thermal neutrons (such as Boron) during the fabrication process of ICs. In fact, devices without such elements seem to have SER about ten times lower than those incorporating the elements [12].

Another solution exploiting material properties aims to reduce the initial electron-hole density so that few charge can be generated and collected at device nodes. This is reachable by using material with a large bandgap [9].

Increasing the node capacitance so that more charge needs to be collected to produce a dangerous voltage/current spike could be another solution but with circuit performance penalty [17].

The use of Silicon-on-insulator (SOI) technology instead of the bulk one may be another solution that gives raise a consistent reduction of SER. More in detail, the use of fully-depleted SOI technology leads to a SER 50 times lower than that of the bulk technologies [12]. This big reduction mainly results from the fact that SOI technology is immune to destructive latch-up thanks of the insulating layer.

3.3.2 Circuit solutions

Focusing on SRAM cells, a solution to mitigate SEEs is the introduction of resistive paths to slow the regenerative feedback response and restore the memory cell content in case of error [12]. Other solutions involve the use of extra transistors (spatial redundancy).

Similar solutions have been implemented to be applied to other storage elements (e.g. DRAM, latch, etc). For example, a popular technique is the dual-interlocked cell (DICE) topology for latch. The DICE has four tri-state inverting stages connected in a loop, resulting in four internal nodes that store the data. If one of the nodes is affected by the SEE, then the rest of the circuit takes action to solve it. [16]

3.3.3 Design solutions

The solutions presented in this subsection are the most interesting ones for the scope of the thesis, since implementable by hardware design. More focus will be posed on them in the next chapter.

In the case of digital circuits, a typical and simple solution exploits logic redundancy. Replicating (two, three, five times) components or entire modules, it is possible by means of a majority voting to detect and eventually correct the induced error. When a replica of the whole device or another device with the same functionalities is used to realize redundancy, the solution is called lock-step.

Another kind of redundancy to reduce SEEs is the temporal redundancy. Temporal redundancy is based on the multiple execution of the same operation and the subsequent majority voting.

A solution used to mitigate errors in memory elements, instead, involves the use of extra bits to add extra information to the stored data (e.g. parity bits, Hamming code, etc). Extra information give the possibility to detect and/or correct one or more errors occurring at the same time. More extra bits are added, more errors can be detected and/or corrected.

3.3.4 System solutions

Solutions at system levels involve the use of software to mitigate errors. The idea of these solutions consists in modifying the program running on the device (e.g. microprocessor) to self-detect potential errors. This goal can be achieved by adding check and correction capabilities, such as the duplication of data and instructions, temporal redundancy, etc. For example, a solution may requires to periodically save the context and restore the last save when an error is detected.

Chapter 4

Fault Tolerance hardware techniques

The fault tolerant techniques used in hardware rely on the concepts of hardware redundancy, information redundancy and time redundancy. The use of redundancy can provide additional capabilities within a system, but can have very important impact on the system's performance, especially as regard area occupation, power consumption, and computation time. For these reasons, a number of different variants of fault tolerant techniques exist in order to focus better on some aspects rather than others, according to the requirements of the design.

4.1 N-modular redundancy

The most common technique exploiting the hardware redundancy is the N-modular redundancy, that is the N-times replication of the targeted module, e.g. ALU, decoder, combinatorial blocks, or also sequential blocks. Basically, this technique is used with a passive approach, which means that faults are masked when possible, but it can be used also with an active (actually hybrid) approach, in the sense that the system needs to activate another unit to mask the fault after its detection. Regardless of the approach, this technique requires a majority voter downline of the replicas, in order to vote the correct output. Depending on the number of replicas, the system is provided with different fault tolerance capabilities: detection or both detection and correction. An example of NMR with N=3, that is TMR, is shown in Figure 4.1.

If N is equal to two (Dual-Modular Redundancy, DMR), then the system is capable only to detect when errors occur. If N is equal or higher than 3, instead the system is capable to detect and possibly also correct errors when they occur. In fact, taking into account the TMR configuration, if the error occurs only in one replica (single error case), the correction surely takes place. But, if errors occur differently in all the replicas, only error detection is allowed. So, TMR ensures 100% error correction capability in case of single or multiple error in only one replica. In order to reach tolerance against errors occurring in more replicas at a time and reach an higher overall reliability, an higher value of N needs to be chosen. Obviously, an higher N means higher costs, especially in terms of area and power consumption. So, it is important to choose accurately the N replicas to use, taking into account both correction/detection capabilities and overall hardware performance.

A lot of variants of NMR technique for reducing the overhead can be implemented, optimizing different points. A common version of NMR tries to avoid having all replicas online concurrently. This exploits an hybrid approach since some

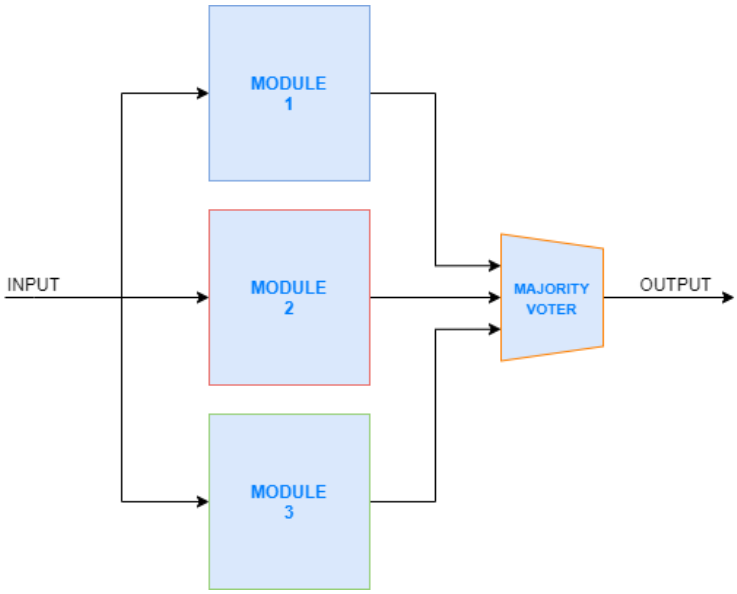

FIGURE 4.1: TMR general implementation

replicas are always used to detect a fault, while the spare ones are only powered on when necessary, such as to resolve a discrepancy seen in the output of the active systems. For example, the TMR system could be implemented as a DMR system that activates a spare unit only when the outputs of the active units are not in agreement. This solution reduces the cost in terms of power consumption, and increases the lifetime of the units that are powered down. As a cons, the proposed technique probably affects the computation time, since the spare unit is activated only after having detected an error.

As regards the majority voter, two basic implementations are possible. The first one is based on one instance of the voter, so only one output is voted and sent to the next block in the circuit. The other implementation provides N instances of the voter, because the majority gate itself could fail so it needs to be protected applying redundancy as well. In this way, N outputs are generated and sent to the next block, as shown in Figure 4.2. This implementation with redundant voter is called Full NMR.

The last way of implementing the voting block makes particular sense if the following block in the circuit is redundant as well. By means of this fault tolerant technique, an even higher reliability, with respect to the basic implementation, is reached at the expense of area and power. An example of multi-stage full TMR is shown in Figure 4.3.

4.2 Lockstep

The lockstep technique exploits the NMR under another perspective. It is based on the "replication" of systems which work independently and in parallel running the same program. In this case, with the term replication is meant the instance of different systems capable do the same operations. Usually, the Lockstep technique is applied to processors. The term “lockstep” arises from the military field, where it is used to indicate the “synchronized walking, in which marchers walk as closely together as physically practical” [18].

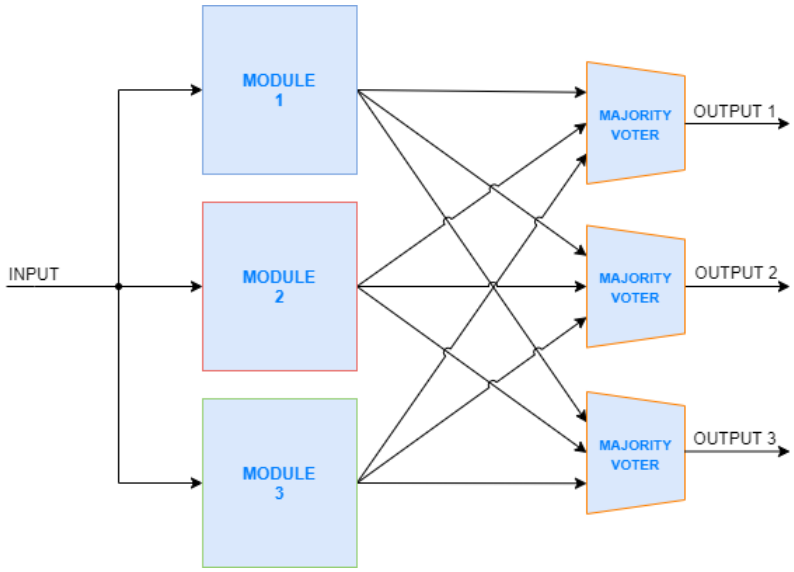

FIGURE 4.2: Full TMR implementation

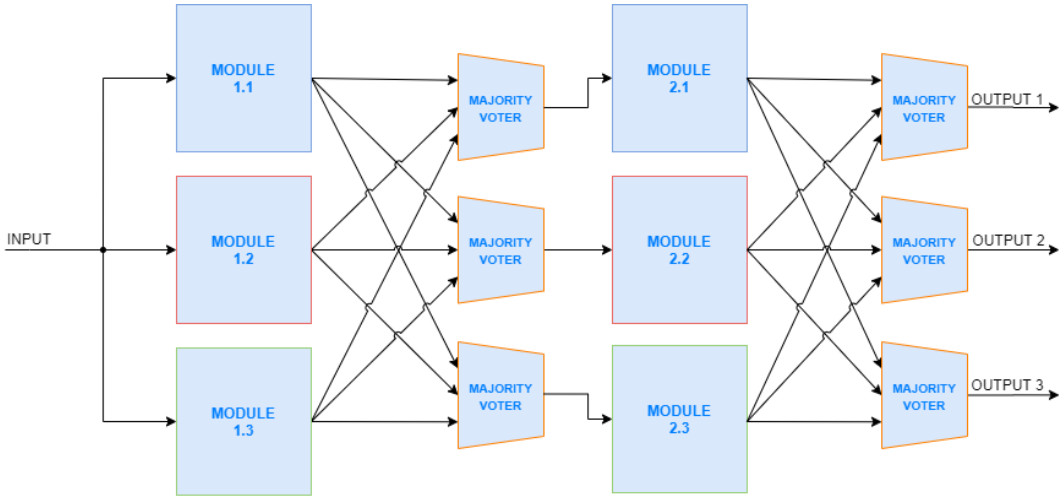

FIGURE 4.3: Multi-stage full TMR example

The approach to keep the system safe is a bit different from the NMR technique. Faults are always recognized by comparing results deriving from the units, but the correction is made in a different way. If the system is composed by two units (that is the most common case, since the processor is a very expensive unit), these are continuously interrupted at predefined points for a consistency check. In case the result differ, then the entire system is recovered restarting from the last correct checkpoint, that is the most recent state where the system was fault-free.
If there are more than two units, instead, the fault can be masked simply by voting the results, as in TMR for example, or restarting again from the last correct checkpoint.

The checkpoint is a concept really important for this technique. Context info are stored during this operation in order to be able to unequivocally force the state of the system in case of faults occurrence. Checkpoint properties strongly determine the performance of the system. For each checkpoint, in fact, context info need to be stored, typically requiring more memory area than the normal storage blocks. Moreover, each context saving slows down the program execution. Therefore, a trade-off

should be made between the number of contexts to be saved and the frequency of checkpoints insertion [19].

Applying the Lockstep technique, the operating clock frequency is poorly affected. What is mainly affected is the total execution time, since time is wasted performing checkpoint operations and eventually restoring the context.

4.3 Time redundancy

Time redundancy, differently from the techniques presented up to here, attempts to reduce the amount of extra hardware typically used (like applying the TMR) at the risk of spending additional time to perform the operation. In fact, this technique is based on the repetition of computations in time and not on the "repetition" (replication) of hardware. Here, only a small amount of extra hardware is required in order to store temporary results and to compare them.
This technique, obviously, heavily worsens the execution time, so it is suitable for application with no realtime constraints.
In Figure 4.4, a block scheme of this technique is depicted, highlighting the fact that the module is the same, while the input and the output are taken at different time instants.

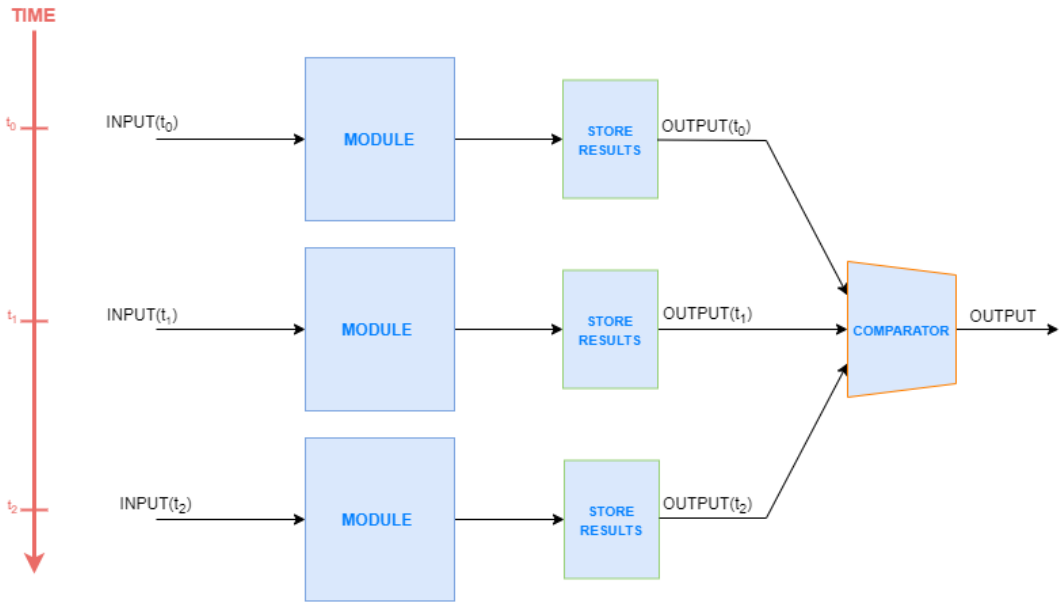

FIGURE 4.4: Time redundancy block scheme against transient errors

Time redundancy allows also to mitigate permanent faults. This version of the technique relies on different data encoding used to perform the operation. This means that a different encoding is applied to inputs at each operation repetition, so as to allow faults in the hardware to be detected. If the same encoding is applied in all the repetitions, the results will be all affected in the same manner and no error will be detected. Applying different encoding, permanent faults are revealed, if present, because bits assume a different meaning at each repeated computation.
In Figure 4.5, a block scheme of this variant is shown, assuming that the computation is repeated three times and, for each computation, input date are encoded in a different way.

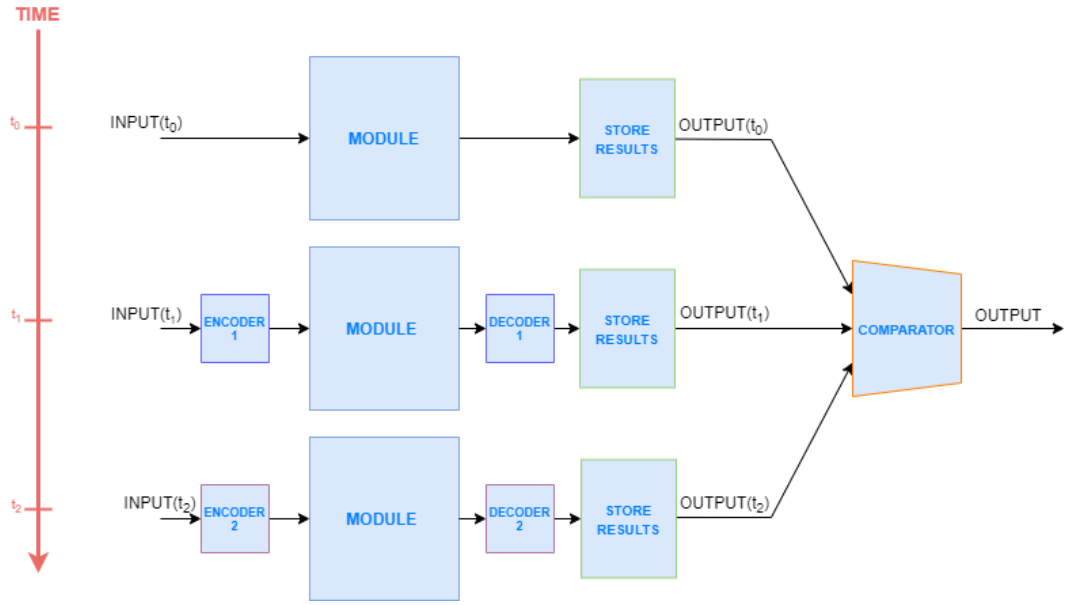

FIGURE 4.5: Time redundancy block scheme against permanent errors

4.4 Watchdog

Due to its very limited cost, timing checks by means of "watchdog" timers are the most widely-used concurrent error detection mechanism. The concept of a watchdog timer is based on the lack of an action intended as a consequence of a fault.
A watchdog is a timer that must be reset periodically within the timer expires. The basic idea is that the system is fault free if it is capable to repetitively carry out a function, such as setting a timer. The frequency to reset the timer is application dependent. If the timer is not reset in time, it is very likely that the system went in a wrong state because of a fault and so it is incapable to perform the reset operation.

This is a very low-cost technique, so it can be applied in any system to improve the reliability with a minimum hardware overhead. Moreover, the watchdog timer is applicable both in hardware and in software.

4.5 Error Detection and Correction code

Error Correction Code is a general fault tolerant technique based on information redundancy and used to protect storage blocks, such as memory and registers. It exploits additional info added to the data in order detect or correct corrupted bits because of one or more faults. There exist several different techniques of ECC each one with different properties that basically can be divided into two categories [20]:

- Block codes: these codes work on fixed-size blocks (packets) of bits or symbols of predetermined size and are generally decoded in polynomial time. The different blocks are independent of each other.
- Convolutional codes: these codes work on bit or symbol streams of arbitrary length.

Convolutional codes are out of scope for this thesis project. The most useful ECCs for ASIC design are the block codes since the data width is predetermined at the design stage. The most used are Reed-Solomon Code, Golay, BCH, Multidimensional parity, and Hamming codes. The Hamming code is the first ECC invented in 1950 by the American mathematician R. Hamming [21].

The notation used to express properties of ECC is the following:

$$(n,\ k,\ m)$$

where n is the length of the code word, k is the length of the data word and m is the length of the check word. Beside these variables, another important parameter for ECC is the minimum Hamming distance (d_{min}) required. The Hamming distance between two data of equal length is defined as "the number of bit positions at which the corresponding bits are different". In other words, it measures the minimum number of substitutions required to make one data equal to the other, or the minimum number of errors that could have transformed one data into another one still valid [22].

Moreover, ECCs are characterized by detection and correction capabilities, according to the maximum number of faults that it is possible to detect and correct. These properties depend on the minimum Hamming distance between any two codewords and an higher number of detectable/correctable faults implies a wider check word, so more hardware.
The maximum number of detectable faults is equal to $d_{min} - 1$, while the maximum number of correctable faults is equal to $\lfloor (d_{min} - 1)/2 \rfloor$. For example, with $d_{min} = 0$ no errors are detectable/correctable, with $d_{min} = 3$ two errors can be detected and one can be corrected, and so on.

4.5.1 Parity bit

The use of the parity bit [23] allows to detect only a single fault and no correction is possible. This technique requires a $d_{min} = 2$ and requires just a single additional bit as check word regardless of the data width. The technique has two versions: Even parity or Odd parity.

In the case of even parity, for a given data word, the occurrences of bits at the logic status 1 are counted. If that count is odd, the parity bit value is set to 1, making the total count of occurrences of 1s in the whole code word (data word + parity bit) an even number. If the count is already even, the parity bit is set to 0.

In the case of odd parity, the coding is reversed. If the count of bits with a value of 1 is even, the parity bit value is set to 1 making the total count of 1s in the whole code word an odd number. If the count is odd, the parity bit is set to 0.

4.5.2 Hamming code

The Hamming code is an ECC with the property to correct up to one error and detect up to two errors, so it has three variants: SED, SEC and SECDED.

The SED version is simply the use of the even parity bit. So, a single bit is added to the data word.

The SEC version is more complex and requires the use of m bits, on the basis of the data word width. The m parity bits are placed at the positions with an index equal to a power of two, and the rest of the positions are used for the bits of the

data word. In this way, the code word is composed by a mix of data and parity bits, placed from the first to the k-th position. For the calculation of parity bits, the binary representation of the position number is considered, remembering that position numeration starts from 1 in this case. To calculate the i-th parity bit, all the bits of the code word in a position whose binary representation has a 1 in the i-th least significant bit are taken into account. As an example, for the 1st parity bit, bits in positions 1, 3, 5, 7, and so on are evaluated since their binary representation has a 1 in the least significant position. For the second parity bit, bits in positions which have a 1 in the 2nd least significant bit position of the related binary representation are considered and they are bits in positions 2, 3, 6, 7, and so on. For clarity, for the third parity bit, bits in positions 4, 5, 6, 7, 12, 13, and so on, are evaluated since they have the third least significant bit in their binary representation at 1. At the decode stage, parity bits are evaluated against the parity of the related bits in the code word. If the parity check matches, the result is a 0, otherwise it is a 1. The results of these checks are called syndrome bits and form a code (error code), whose value indicates where the error has occurred. Finally, the correction takes place by flipping the bit in the code word at the position stated by the error code.
According to the algorithm described up to now, syndrome bits must be sufficient to determine the position of the error (that can have $k + m$ positions) and to state that no error has occurred. In total, $k + m + 1$ values need to be represented by the syndrome bits, and being the syndrome bits as wide as the parity bits, this means that the following condition must be satisfied to properly select the number of parity bits:

$$2^m \geq k + m + 1$$

The SECDED version is a mix of SEC and SED, in the sense that parity bits are used like for SEC, and an extra even parity bit is added like for SED. So, for this version of the Hamming code, the code word is one bit wider with respect to the one of SEC version ($m_{SECDED} = m_{SEC} + 1$). The capabilities of this technique are single error correction and double error detection, so three cases need to be distinguish: no error, single error corrected, and double error not corrected.
No error is given if all the check bits plus the parity bit are correct, so the error code is zero. Single error corrected is given if at least a syndrome bit is not zero and the extra parity bit fails. Double error detection, instead, is given if at least one of the syndrome bits is not zero and the extra parity bit is satisfied.
In this case, considering that m includes the parity bits and the overall parity bit, the total number of parity bits must meet the following condition:

$$2^{m-1} \geq k + m$$

From a mathematician analysis of this ECC, the Hamming code is a linear code and so it is possible to perform encoding and decoding operations by exploiting the multiplication of the input (the data word for encoding, the code word for decoding) by the so called generator (or parity-check) matrix or by a part of it. The generator matrix is composed by m rows and $k + m$ columns. More in detail, the first k columns are referred to the data word, the last m ones to the parity bits.
Parity bits, in the encoding stage, are generated by multiplying the data word (k bits wide) by the sub-matrix of the generator matrix, composed by m rows and the first k columns. The result of this operation is a vector of m bits which are the parity bits to insert in the code word. In the decode stage, instead, the code word ($k + m$ bits wide) is multiplied by the entire generator matrix producing a vector of k bits as well, that

are the syndrome bits.

The generator matrix, obviously, needs to satisfy some properties. Firstly, as stated before, it is important to see the Hamming parity-check matrix as composed by two matrices. The first one is referred to the parity bits and it is an identity matrix. The identity matrix, actually, contains for each column the binary representation of a power of two (1, 2, 4, 8, and so on). The second matrix is referred to the data bits and has for each column the binary representation of the remaining ascending numbers (3, 5, 6, 7, 9, and so on). In the case the overall parity bit is used, an additional rows with all the elements at 1 is added to the generator matrix.

4.5.3 Hsiao code

In 1970, Hsiao proposed its variant of the Hamming code, which aimed to optimize the number of 1s in the parity-check matrix. In this way, less bits need to be evaluated to calculate and check the parity, resulting into faster calculations.

This code works in a slightly different manner with respect to the Hamming code. The first difference is about the parity-check matrix that must satisfy different properties. For the sub-matrix related to the parity bits, the identity matrix is used here too. For the part related to the data bits, instead, a different logic rules. In fact, in this case, the columns of this sub-matrix must contain only an odd number of 1s and must be all different. Columns are filled in order to minimize the number of 1s, so are filled with combinations of 3-out-of-m, 5-out-of-m, 7-out-of-m, and so on until all the k columns are full. If the column-vectors are selected in a way that the number of 1s in each row is kept close to the average number of 1s in a row, then it is possible to reduce the number of required logic gate levels of the hardware [24].
The other difference is that the overall parity check is not required. This difference together with the properties of the parity-check matrix means that the three cases (no error, single error corrected and double error detected) are distinguished in another way. No error is given always if all the syndrome bits are zero. Single error corrected is given if the syndrome bits are not zero and the overall parity calculated on them is one. The position of the error is obtained by comparing the syndrome bits with the column-vectors in the parity-check matrix. The i-th position of the matching column gives the position of the wrong bit in the code word. Double error detection, instead, is given if the syndrome bits are not null and the overall parity calculated on them is equal to zero.

The Hsiao code is considered an optimized version of the Hamming code from the hardware's point of view. In fact, Hsiao's encoder and decoder involve fewer logic gates (with respect to the Hamming's ones) leading to a lower occupation area and possibly also less gate levels, which means lower delay.

In Table 4.1 and Table 4.2, some hardware characteristics resulting from applying Hamming and Hsiao codes are presented in order to have a quick comparison of them. These values are extracted from the paper published by G. Tshagharyan et al. [24], where the two ECCs have been compared through an experimental analysis. The Table 4.1 report values related to the number of logic levels used to implement the encoder and decoder of the ECC. This gives an approximate idea of the delay caused by applying the ECC. The Table 4.2, instead, reports the area occupied by the encoder and decoder for each ECC.

ECC type	Data Word Width	Code Word Width	Encoder Logic Levels	Decoder Logic Levels
Hamming SEC	8	12	2	6
	64	71	5	14
	310	319	12	17
Hamming SECDED	8	13	2	6
	64	72	6	15
	310	320	14	18
Hsiao SECDED	8	13	2	8
	64	72	5	13
	310	320	10	15

TABLE 4.1: Hamming vs Hsiao codes: number of logic levels [24]

ECC type	Data Word Width	Code Word Width	Encoder Area [μm^2]	Decoder Area [μm^2]
Hamming SEC	8	12	7.78	16.38
	64	71	70.81	148.21
	310	319	350.28	573.92
Hamming SECDED	8	13	8.92	19.39
	64	72	82.06	152.57
	310	320	407.26	582.06
Hsiao SECDED	8	13	9.02	20.03
	64	72	80.92	137.49
	310	320	402.64	535.90

TABLE 4.2: Hamming vs Hsiao codes: area occupation [24]

The results of this comparison show that Hamming code is more efficient for smaller data words, while Hsiao code has better performance for larger data sizes despite its complexity. This means that the proper ECC code should be selected depending on the specific application scenario.

4.5.4 Cyclic Redundancy Check code

The Cyclic Redundancy Check (CRC) is an error-detecting code commonly used in digital networks and storage devices and it is based on cyclic codes. The data protected by CRC code have a short check value appended, based on the remainder of a polynomial division of their contents, that adds only redundancy but not information. On retrieval, the calculation is repeated and, in the event the check values do

not match, specific action are taken against data corruption. CRCs can be used also for error correction [25].

The algorithm exploits the repetitive division by a binary polynomial (called generator polynomial) to generate a check code, that is the reminder of the division. The generator polynomial is represented in binary format, in the sense that the coefficients can be only 1 or 0. The algorithm starts with the data (k bits wide) with the check code initialized to 0 (m bits wide) appended in the less significant positions, that is the dividend, and the (constant) generator polynomial (n bits wide), that is the divisor. The widths of the divisor and of the check code depend on the order of the polynomial itself. Assuming x the order of the polynomial, n is equal to $x + 1$, while m is equal to x. In this way, the division of the dividend ($k + m$) by the divisor ($m + 1$) is always possible. At the end of the multiple divisions, the output is a string of 0s in correspondence of the data bits position and the reminder in correspondence of the check code. Now, the calculated check code is appended to the original data ensuring error detection capability. In fact, at the decoder side, the same operations are performed taking into account the data and the received check code (not initialized to 0). If the reminder after the multiple divisions is equal to 0, then the data is valid, otherwise an error has occurred [26].
The most commonly used polynomial are CRC-8, CRC-16, CRC-32, and CRC-64, respectively 9, 17, 33, and 65 bits wide [25].

Chapter 5

RISC-V core

RISC-V (pronounced "risk-five") is “a free and open ISA (Instruction Set Architecture) based on RISC (Reduced Instruction Set Computer) principles enabling a new era of processor innovation through open standard collaboration” [27]. The RISC-V ISA is provided under open source licenses and this is its major strength. In fact, no patents were filed related to RISC-V, as the RISC-V ISA itself does not represent any new technology, but it is based on computer architecture ideas that date back at least 40 years [28].
The RISC-V ISA project was originally developed in 2010 by researchers from the Computer Science Division at the Electrical Engineering and Computer Science department of the University of California, Berkeley, along with many volunteer contributors not affiliated with the university. Despite other academic designs are generally optimized only for simplicity of exposition, the RISC-V designers aimed to make the RISC-V ISA usable also for practical computers [29]. Now, the ISA is governed by the RISC-V Foundation.
Processors based on RISC-V ISA are very versatile. The ISA supports three word-widths, 32, 64, and 128 bits, so it can be used in several ways and exploited for different purposes, from supporting basic or complex operating systems (such as Linux) to supporting supercomputers with vector processors. This makes RISC-V cores well-liked also from an industrial point of view and it is the reason why it is spreading more and more worldwide.

5.1 RISC-V overview

The most common architectures of RISC-V core are characterised by five stages pipeline to improve the performance, but there are also other versions with a lower or higher number of stages of pipeline. The traditional five stages are:

- IF: instruction fetch from memory
- ID: instruction decode and register read
- EX: execute operation or calculate address
- MEM: access memory operand
- WB: write result back to register

Cores basically implement the Base Integer Instruction Set (I) plus some optional extensions [30]. The Base Integer Instruction Set contains integer computational instructions, integer loads, integer stores, and controlflow instructions.
Actually, RISC-V is a family of related ISAs, of which there are currently four base

ISAs according to the width of the integer registers and the corresponding size of the address space and by the number of integer registers. They are [30]:

- RV32I: which provides 32-bit address space
- RV64I: which provides 64-bit address space
- RV128I: which provides 128-bit address space
- RV32E: (to support small microcontrollers) which provides 32-bit address spaces and has half the number of integer registers

For each family, which basically implements the "I" instruction set by constraint, additional standard extensions are defined to provide further functionalities. For instance, the most used ISA extensions are Integer Multiplication and Division (M), Atomic Instructions (A), Single-Precision Floating-Point (F) (IEEE 754-2008 compatible), Double-Precision Floating-Point (D) (IEEE 754-2008 compatible), Compressed Instructions (C), and so on.
By convention, RISC-V instructions are on 32 bits (4 bytes o 1 word) divided in fields of regular sizes to make the hardware simpler. According to the type of instruction, each field of the instruction assumes a particular function (explained in Figure 5.1). Types of instruction implemented in the RISC-V ISA are:

- R-type: arithmetic instruction format
- I-type: load and immediate arithmetic
- S-type: store instruction format
- SB-type: conditional branch format
- UJ-type: unconditional jump format
- U-type: upper immediate format

Name (Field size)	Field					
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
R-type	funct7	rs2	rs1	funct3	rd	opcode
I-type	immediate[11:0]		rs1	funct3	rd	opcode
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode
U-type	immediate[31:12]				rd	opcode

FIGURE 5.1: RISC-V instruction formats [31] ("opcode": operation code – "rd": destination register number – "funct3": 3-bit function code (additional opcode) – "rs1": the first source register number – "rs2": the second source register number – "funct7": 7-bit function code (additional opcode) – "immediate": constant operand, or offset added to base address)

In Figure 5.1, it is easy to notice that the fields in each instruction type try to keep the identical meaning as much as possible. This is a strength of RISC-V cores because it leads to many improvements for what concerns area, power and delay, as well as

ease of design.

Another important aspect of RISC-V ISA regards the register file. This essential component requires 32 (or 16 for RV32E ISA family) integer registers of width dependent on the ISA family (32-, 64-, or 128-bits) [32] and at least one write port and two read ports. All the registers (named from x0 to x31) are defined general purpose registers, even if some of them are used to serve specific functions, by convention.
Besides the "integer" register file, if "F", "D", or "Q" instruction sets are implemented, 32 additional "floating-point" registers are required to store floating-point values.
In the standard RISC-V calling convention [33], each integer and floating-point register (named from f0 to f31) assumes a particular role described in Table 5.1.

Register	Description	Saver
x0	Hardwired to zero 0	—
x1	Return address	Caller
x2	Stack pointer	Callee
x3	Global pointer	—
x4	Thread pointer	—
x5-x7	Temporaries	Caller
x8	Frame pointer	Callee
x9	Saved registers	Callee
x10-x11	Function arguments/return values	Caller
x12-x17	Function arguments	Caller
x18-x27	Saved registers	Callee
x28-x31	Temporaries	Caller
f0–f7	FP temporaries	Caller
f8–f9	FP saved registers	Callee
f10–f11	FP arguments/return values	Caller
f12–f17	FP arguments	Caller
f18–f27	FP saved registers	Callee
f28–f31	FP temporaries	Caller

TABLE 5.1: RISC-V calling convention register usage

The last specifications regard the memory. It is fundamental to know that RISC-V applies Little-Endian rules (least significant byte at least address of a word) and does not require words to be aligned in memory. Moreover, memory should be byte addressable.

Finally, a basic architecture (including the controller) of the five stages pipelined version of the RISC-V core is presented in Figure 5.2. Advanced versions with hazard detection unit and forwarding unit are not described since they are out of the

scope of the thesis work. Anyway, the presence of these units in RISC-V cores is crucial to reach better overall performance.

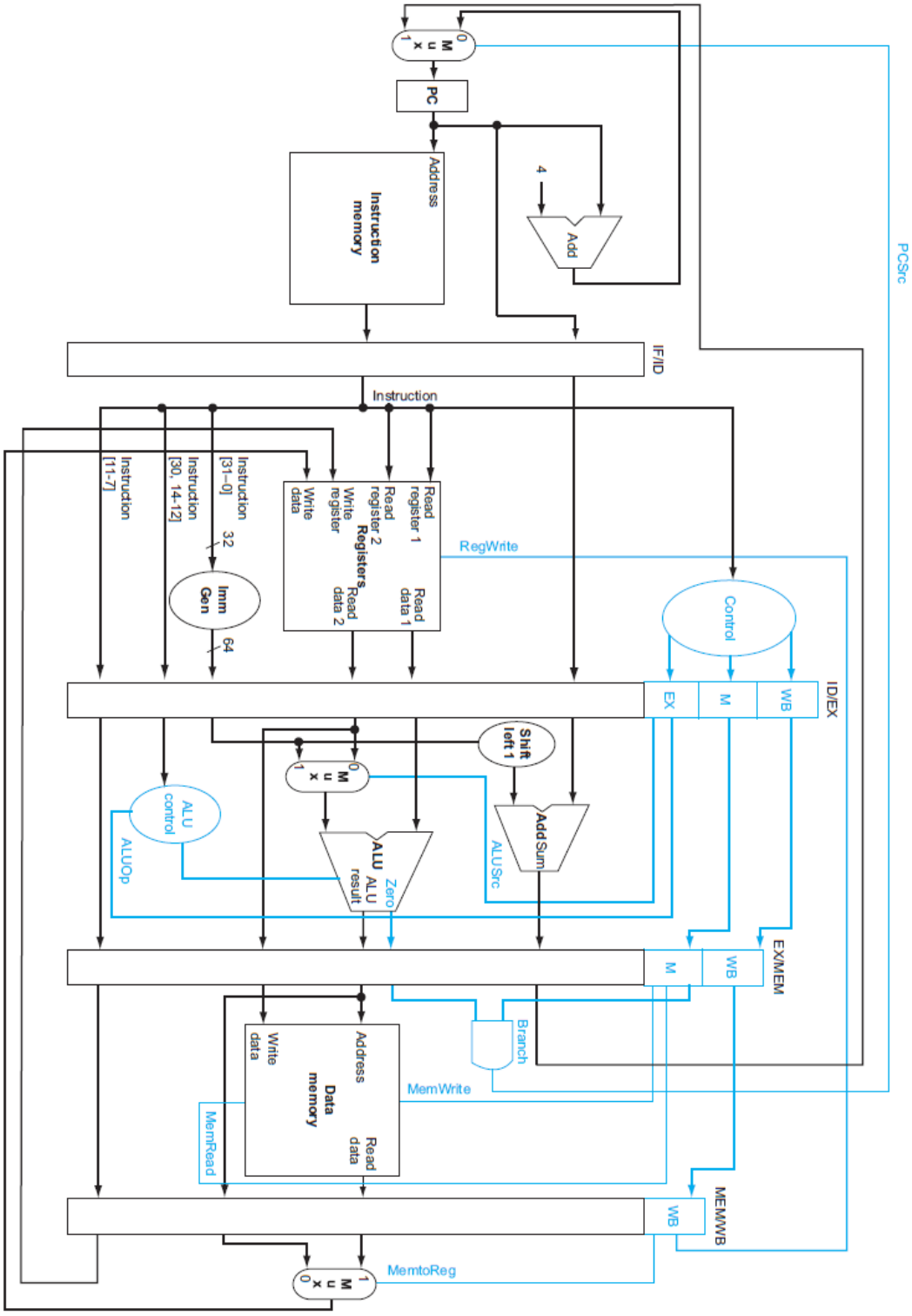

FIGURE 5.2: Basic architecture of pipelined RISC-V core [31]

5.1.1 RISC-V cores

Nowadays, several companies and universities have developed their own versions of RISC-V core and SoC platforms. Some of them have even managed to fabricate their chip, either available for sale or just for personal purposes [34].
A RISC-V core of great interest is the one developed by ETH Zurich and Università di Bologna, the "RI5CY" core. It is provided also with SoC platforms (named PULPino [35] and PULPissimo [36]), developed by the two universities, which make the core more interesting and suitable for many purposes. It is now (since February 2020) contributed to OpenHW Group [37] and named "CV32E40P".

CV32E40P is the core which is subjected to all the fault tolerant techniques that will be developed in this thesis project. The choice of the core is motivated also by a project previously developed in the scope of a university course in which PULPissimo platform with the RI5CY core has been mapped on the fpga Xilinx ZCU102 [38].

5.2 CV32E40P (RI5CY core)

CV32E40P [39] is a small and efficient, 32-bit, in-order RISC-V core with a 4-stage pipeline written in SystemVerilog. It implements the RV32IMC, and optionally F, ISA and Xpulp custom extensions [40] for achieving higher code density, performance, and energy efficiency. Xpulp custom extensions support multiple additional instructions, such as hardware loops, post-increment load and store instructions, and additional ALU instructions that are not part of the standard RISC-V ISA.

In Figure 5.3, the block diagram of the core is depicted, where the main components for each stage of the pipeline are highlighted, as well as the interface with memory.

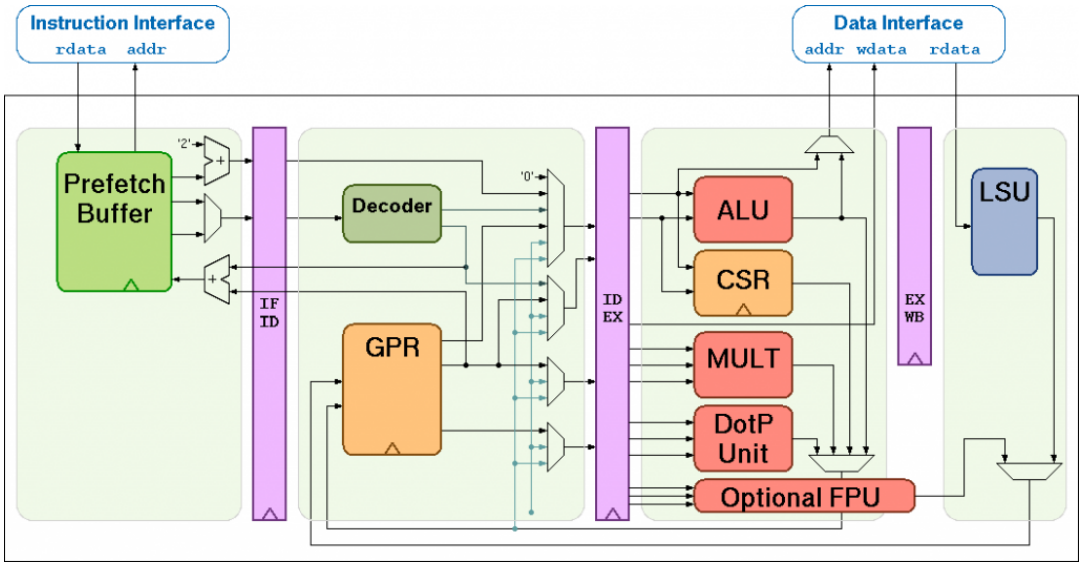

FIGURE 5.3: CV32E40P core block diagram [40]

The first obvious difference with other RISC-V cores is the number of pipeline stages, that is four instead of five. Stages are IF, ID, EX, and WB. The MEM stage is avoided since data memory is already addressed in the EX stage. The number of

pipeline stages used to implement the processor core is one of the key design decisions. A higher number of pipeline stages takes the advantage of obtaining higher operating frequencies, increasing the overall throughput, but also increasing the data and control hazards, which, in turn, reduce the IPC (Instruction Per Cycle) [41]. To reduce the impact of hazards, additional components (such as branch prediction, speculation, etc) would be required, but increasing the overall power consumption. So, this is another reason why four staged pipeline has been designed, en fact a key characteristic of this core is the ultra low power target.

The main features and hardware components of the core are explained in detail in the next subsections, dividing them by stage.

5.2.1 Pipeline

Instruction Fetch stage

In the IF stage, the instruction is fetched from the instruction cache or instruction memory, as usual, with the peculiarity that the instruction address must be half-word-aligned (a word is 32-bits wide) due to the support for compressed instructions.

Compressed instructions need an ad-hoc decoder, called "compressed_decoder", which extracts and extends all the compressed information from the compressed instruction. In case the instruction is not compressed, it lets the input instruction pass through unchanged.

To achieve optimal performance, a prefetcher, called "prefetch_buffer", is instantiated in this stage. This component fetches 32-bits instruction or 128-bits cache line from the instruction memory, stores the information into a FIFO memory (made up of four 32-bits wide registers, and called "fetch_fifo") and sends it to the compressed decoder. All the operations related to fetching instructions are managed by the "fetch_controller", that is also responsible for detecting branch instructions and acting consequently.

Instruction Decode stage

The ID stage is the stage where the instruction coming from the IF stage is decoded and, according to the opcode, signals towards the EX stage are enabled.

This stage is composed by three main components, that are register file, decoder, and main controller. Obviously, also other components are instantiated to support more functionalities.

The register file is available in two versions, the flip-flop based and the latch based, both with the same structure (for the purpose of this thesis work, the flip-flop based register file is used). As the RISC-V ISA requires, it is composed by 32 locations of 32-bits for integer values, plus other 32 locations for floating-point values if the "F" instruction set is implemented. So, the address is 5-bits wide, plus, if "F" is implemented, a 6th bit to select the register file (integer or floating-point) to use. Differently from the usual structure of RISC-V register file, this register file is provided with two synchronous write ports (A and B) and three asynchronous read ports (A, B and C). In this way, since the write ports are dedicated separately to ALU and WB stage, structural hazards are avoided. The last information about the register file is the presence of the MBIST (Memory Built-In Self Test) interface for debugging and testing operations.

The decoder is a combinatorial block described in SystemVerilog by means of a large case statement. In the case statements, all the combinations of opcode (7 bits) are taken into account, and for each of them specific fields of the instruction are also evaluated, in order to generate the proper signals towards the EX stage.

The last and most important block is the "core_controller". It is described as a 18-states FSM (Finite State Machine) responsible for managing the whole core. Synchronization among the stages is ensured thanks to this controller together with other controllers even instantiated in this stage,as well. The other controllers are "stall_controller", "forwarding_controller", and "interrupt_controller".

Execution stage

The EX stage is the main computational stage of the architecture. Here, the ALU (Arithmetic Logic Unit) is instantiated. The ALU supports four different operations: bit manipulation, fixed-point operations, iterative division, and packed SIMD (Single Instruction Multiple Data) operations. As regard the multiplier, it is instantiated beside the ALU.

The floating-point unit (FPU), instead, is instantiated outside the core, if the "F" instruction set is implemented, and it is accessed via the APU (Auxiliary Processing Unit) interface.

At this stage, the data memory can be accessed for a write operation (by means of a store instruction) or can be simply pointed by the address (calculated by the ALU or arrived from the ID stage) in order to read its content at the WB stage.

Write Back stage

The WB stage is the one responsible for writing back the result of the EX stage or the data read from the memory to the register file at the ID stage. To manage these operations, two units are instantiated, one of which is optional. Respectively, they are the Load-Store Unit (LSU) and the Physical Memory Protection (PMP) unit.

The LSU takes care of accessing the data memory, which supports "load" and "store" instructions on words (32-bits), half words (16-bits) and bytes (8-bits). It exploits the OBI (Open Bus Interface) protocol to communicate with the memory.

The PMP unit provides a filtering operation of data read from memory. It allows the core to possibly run in USER MODE. Every fetch, load and store access executed in USER MODE are first filtered by the PMP unit which can possibly generated exceptions.

5.2.2 Hardware characteristics

The characteristics of the core (in terms of area, frequency, and power consumption) have been analysed after synthesis, place and route targeting the UMC 65nm technology [42]. The energy consumption, obviously, changes with respect to different workloads and operating frequencies and voltages [43].

Area

Area is evaluated in kgates-equivalent (kGE), that is the equivalent minimum-size NAND2 gate area. In UMC 65nm, one gate equivalent (GE) is $1.44\mu m^2$. The core area is $40.7kGE$ [43], distributed as listed in Table 5.2.

Component	Area [kGE]	Percentage [%]
Prefetch Buffer	5.2	12.8
Decoder and Controller	5.6	13.6
Register File	10.0	24.7
ALU, LSU, CSR	7.8	19.2
Multiplier and Division unit	11.3	27.7
Debug unit	0.8	2
Total	40.7	100

TABLE 5.2: RI5CY area distribution [43]

Frequency

For the operating frequency two netlist have been generated, in order to see the effects on power consumption. The first netlist has been constrained with a relaxed clock period of $10ns$, whereas the second netlist has been synthesized with a tighter clock period of $3ns$. In typical conditions of the targeted technology at $1.2V$, the slow-netlist can reach $185MHz$ and the fast-netlist $560MHz$, whereas at $0.8V$, the slow-netlist can reach $55MHz$ and the fast-netlist $160MHz$. The clock frequencies are summarized in Table 5.3.

Clock constraint	Frequency [MHz] @ 0.8V	Frequency [MHz] @ 1.2V
10 ns	55	185
3 ns	160	560

TABLE 5.3: RI5CY clock frequency with different time constraints at different supply voltages [43]

Power consumption

The power consumption contributions are estimated for all the four combinations of clock frequency and voltage supply and are reported in Table 5.4.

5.3 State-of-the-art of fault tolerant RISC-V core

During last years, the fault tolerance question has become of big interest in the world of electronic devices. Identifying the potentialities of the RISC-V ISA and the ease of implementation, together with the need of using optimized processors (also from an area point of view), fault tolerant RISC-V cores can become a huge resource for this scope. Anyway, still few models of fault tolerant versions of RISC-V cores have been implemented.

Clock Frequency @ Voltage Supply	Dynamic Power [μW]	Leakage Power [μW]	Total Power [μW]
55 MHz @ 0.8 V	77	0.22	77
160 MHz @ 0.8 V	336	2.96	339
185 MHz @ 1.2 V	937.95	1.91	940
560 MHz @ 1.2 V	3752	24.9	3777

TABLE 5.4: RI5CY power consumption at different clock frequencies and supply voltages [43]

Wietse F. Heida [19] proposed an hybrid fault tolerant design against SEUs and SETs using NMR to protect the pipeline and combinatorial blocks and ECC (Hsiao SECDED) to protect memory elements. The core he used is a classic five staged pipeline core implementing the RV32I instruction set.

Another solution against SEUs and SETs has been designed by D. A. Santos et al. [44], employing Hamming code to protect the memory elements and TMR to protect the ALU and the control unit (disregarding the instruction and data memories). The processor they used implements a limited version of RV32I.

L. Blasi et al. [45] developed an enhanced version of the Klessydra F03_mini core (RV32I), adding a Hardware Thread (HART) full-weak protection (a particular version of TMR) and a thread-controlled Watch-Dog Timer module against SEEs.

An optimized solution, from a resource utilisation perspective, is proposed by A. Ramos et al. [46] with a selective TMR on ALU, based on the most executed ALU operations, against SETs. The core used is the lowRISC.

A fault tolerant approach different from the others is pursued by C. Rodrigues et al. [47] with a Dual-Core Lockstep solution, using the Arm Cortex-A9 processor and the lowRISC RISC-V processor.

Another fault tolerant version has been developed by L. A. Aranda et al. [48] which have made the Rocket RISC-V core [49] fault tolerant. This core has been simply protected by means of the DTMR (Distributed Triple Modular Redundancy) technique.

Basically, ECC and TMR are the fault tolerant techniques most used to make the RISC-V core fault tolerant. These techniques allow customization in terms of level of reliability (correction/detection) and resources optimization (resources overhead as low as possible).

However, the existing fault tolerant versions of RISC-V cores aim to mitigate only the SEEs, which are the most common errors, disregarding permanent errors.
A fault tolerant version of RISC-V core, which implements a complete RISC-V ISA and which is capable to mitigate both SEEs and cumulative effects, can be considered an innovative project in this field of study.

Chapter 6

Fault tolerant design of the Instruction Decode stage

The objective of this thesis project is to develop a fault tolerant version of the ID stage of the RISC-V core CV32E40P. Since the larger the components, the higher the probability that the radiation produces its negative effects, only the main components (that are also the most extended in terms of area) inside the stage are taken into account. They are: the decoder, the register file, the controller, and the pipeline registers between IF and ID stage. For the purpose of the thesis project, the other components are not considered to be at risk from a radiation point of view, and so they are used in their original versions.

The fault tolerant techniques used for designing the fault tolerant version of the components are selected from the ones listed in chapter 4 and customized depending on the purpose.
In particular, the register file occupies almost the 25% of the entire core (as reported in Table 5.2). This means that it is one of the most critical components of the core and it needs even an higher level of protection with respect to the others. To accomplish this need, two different fault tolerant versions are proposed for the register file, depending on the severity of protection required. The first one protects against soft errors (transient), the second one against hard errors (permanent). The latter has been designed by integrating the technique used for the soft error version with some specific observations. Obviously, this version allows the component to be, in general, more robust and more suitable for very harsh environments.

The design of the fault tolerant version of the ID stage aims to build a configurable system (at the pre-synthesis stage) that makes it possible to activate the fault tolerant version of a component independently on the others. This mechanism allows to reach different levels of fault tolerance in order to achieve the best trade-off between the hardware performance and the level of protection of the core required for the specific application. In total, 24 combinations are possible deriving from the fact that 2 versions (the basic and the fault tolerant one) are available for decoder, controller, and pipeline, while 3 versions (the basic, the soft, and the hard error one) are available for the register file. This means that 24 different levels of fault tolerance can be selected.

In Figure 6.1, an RTL (Register Transfer Level) view of the ID stage is presented highlighting the main components taken into account in the fault tolerant design. For simplicity, only few signals (the most relevant ones like the Program Counter

signal, the Instruction signal, the Register File's addresses and operands signals, and other generic ones just to show the link among the components) are illustrated. They are sufficient to give an idea of the connections inside the stage. As regards the pipeline blocks, the one of interface between the IF and the ID stages is considered part of the ID stage for the design scope, while the one of interface between the ID and the EX stages is not considered part of it, but only depicted to show a complete circuit between two banks of registers.
Starting from this basic RTL schematic, the fault tolerant versions of the components are gradually added and illustrated more in detail in the following subsections of the chapter.

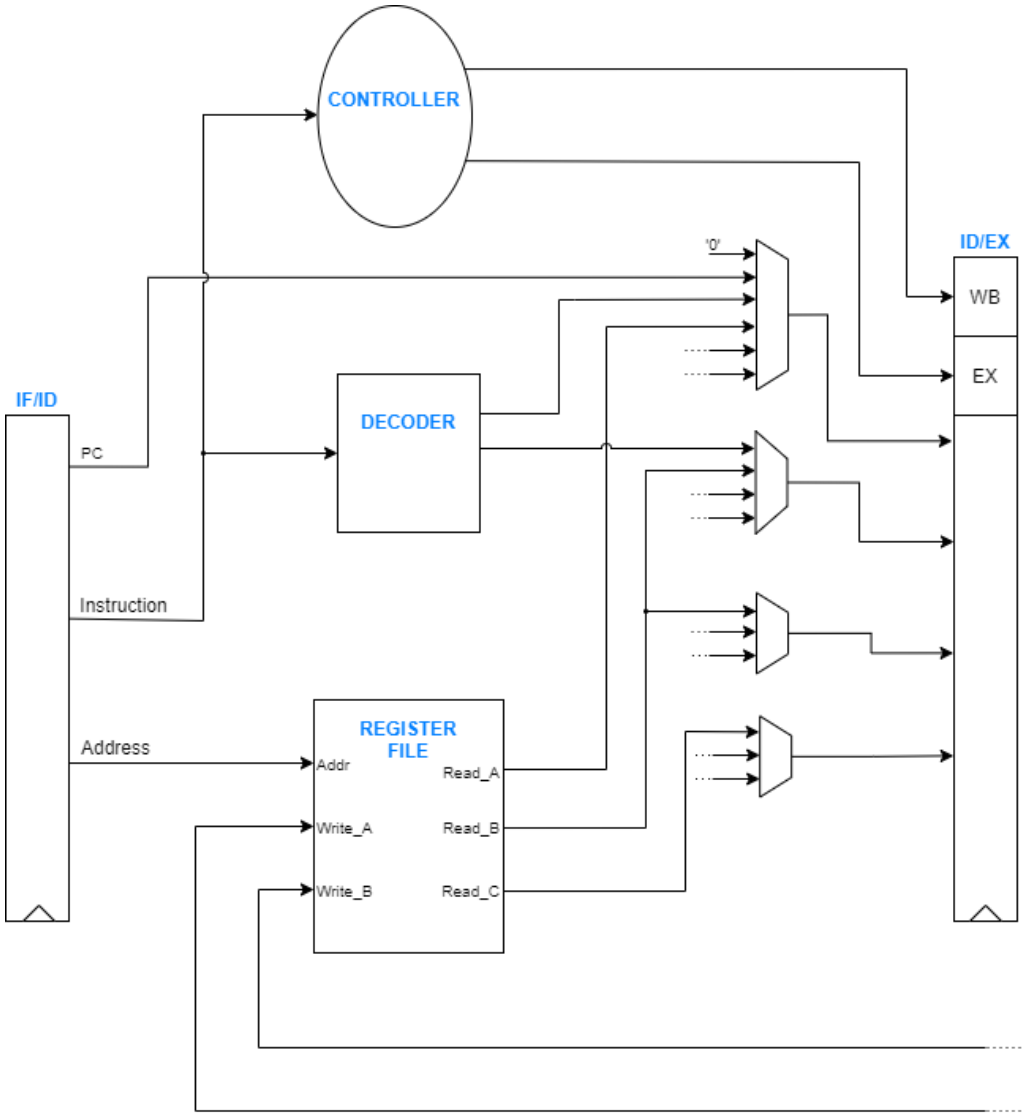

FIGURE 6.1: Block diagram of the ID stage unprotected

6.1 Fault tolerant version of the Pipeline Registers

The pipeline registers are simply a set of registers which store some signals going from the IF stage to the ID stage. The signals' width varies depending on the kind

of information that the signal transports. Some of them are 32-bit wide, like the instruction signal, while others are one single bit signals, for a total of six registers. More in detail, two registers have a width of 32 bits to store the instruction and the program counter, while the others are simply flip-flops to store some flags.
This implies that a generic solution is required to protect both the two kinds of registers.
The most generic one is the NMR (N-modular redundancy). The technique must have detection and correction capabilities for a SEE, so the triple redundancy (TMR) has been taken into account. All the registers are triplicated, and for each triplet a majority voter is added.
Another generic technique is the ECC (error correction code). This technique requires an encoder and a decoder for each register of the pipeline, as well as additional bits in the register to store the redundant information. For example, the Hsiao code featured with detection and correction capabilities would require 7 and 2 additional bits, respectively for the registers 32- and 1-bit wide.

The choice of the FT technique to apply has been done mainly comparing the two techniques mentioned above in terms of area and delay. The power consumption is not taken into account since it may depends largely also on the technology adopted. To proceed with the comparison, the designs of the two techniques have been synthesized by means of Synopsys, which allows to obtain an early estimation of the hardware characteristics of the circuit. For the synthesis, the library UMC 65nm is used, but any other library could be used because the objective is simply to show the differences of the circuits built both in CMOS logic with the same technology's node. In the following table (Table 6.1), the features (area and delay) related to the synthesis of the majority voter of the TMR, and of the encoder and the decoder of the Hsiao ECC are listed. Also the area of the register is reported, in order to better estimate the total area occupied by applying a specific technique. All the components taken into account are designed to support 32-bit wide signals.

Component	Area [GE]	Delay [ns]
Majority voter	480.75	2.17
Hsiao encoder	160.25	1.35
Hsiao decoder	362.25	3.19
Register	240	0.31

TABLE 6.1: Synopsys' estimations of the majority voter, the Hsiao encoder, the Hsiao decoder, and a generic register (every component is 32-bit wide)

From the table above, it is easy to notice how the majority voter and the Hsiao decoder affect in the same way the delay of the signals from the registers. Being these components placed just after the registers, the two techniques may be considered equivalent.
The big difference regards instead the total area occupied to protect the registers. In fact, considering the TMR applied to a 32-bit register, the total area occupied is 0.481 kGE (majority voter) plus 0.720 kGE (three copies of a single register), that is 1.201 kGE. As regards the Hsiao code, the total area occupied is 0.522 kGE (encoder

and decoder) plus about 0.293 kGE (39-bit register, because of redundant bits), that is 0.815 kGE. The TMR occupies an area about 47% larger than the one required for the Hsiao ECC.
If the analysis ended at this point, the choice would be easy (Hsiao code is better than TMR), but there are other parameters to be taken into account. First of all, it is extremely important to observe that the Hsiao technique affects not only the path downstream the register (going towards the ID/EX pipeline register), but also the path upstream the IF/ID register because of the encoder. This is a not negligible drawback. Apart from this negative side related to the Hsiao code, it is important to take into account a feasible improvement deriving from applying the TMR. That is the removal of the majority voter if the same technique is applied also to other component downstream the register. In this case, it is possible to connect the three outputs of the three copies of the pipeline register directly to the three copies of the other triplicated component. This means that the delay is not affected anymore, and also the area is reduced and becomes about 14% less than the area occupied by applying the Hsiao technique.
In conclusion, in this case it is more advantageous to apply the TMR technique, as explained before. For this reason, for the final fault tolerant design, the TMR is the technique chosen to protect the pipeline registers against the SEEs.
In Figure 6.2, the final design of the FT version of the pipeline registers is shown. Actually, the image shows a generalized case. The real schematic of the IF/ID pipeline registers is composed in total by six registers: two registers 32-bit wide and one register 1-bit wide.

CONFIGURATION	Area [GE]	Max Delay [ns]
normal	510	0.31
fault tolerant	2543	2.48

TABLE 6.2: Some relevant hardware characteristics for the three different versions of the Register File.

6.2 Fault tolerant version of the Decoder

The decoder instantiated inside the ID stage is a purely combinatorial block. It receives in input the instruction fetched and some flags, and produces in output all the signals necessary to activate the multitude of units required to correctly execute that instruction.
According to the Table 5.2, the decoder is one of the most relevant components of the core which occupies together with the controller about the 13% of the total area, so it needs to be protected against SEEs.

Being a combinatorial block, the simplest and most efficient fault tolerant technique to apply is the NMR, in particular, the TMR to reach detection and correction capabilities.
Moreover, the TMR allows to achieve the improvement explained in the previous paragraph, that is to remove the intermediate majority voters downstream the pipeline

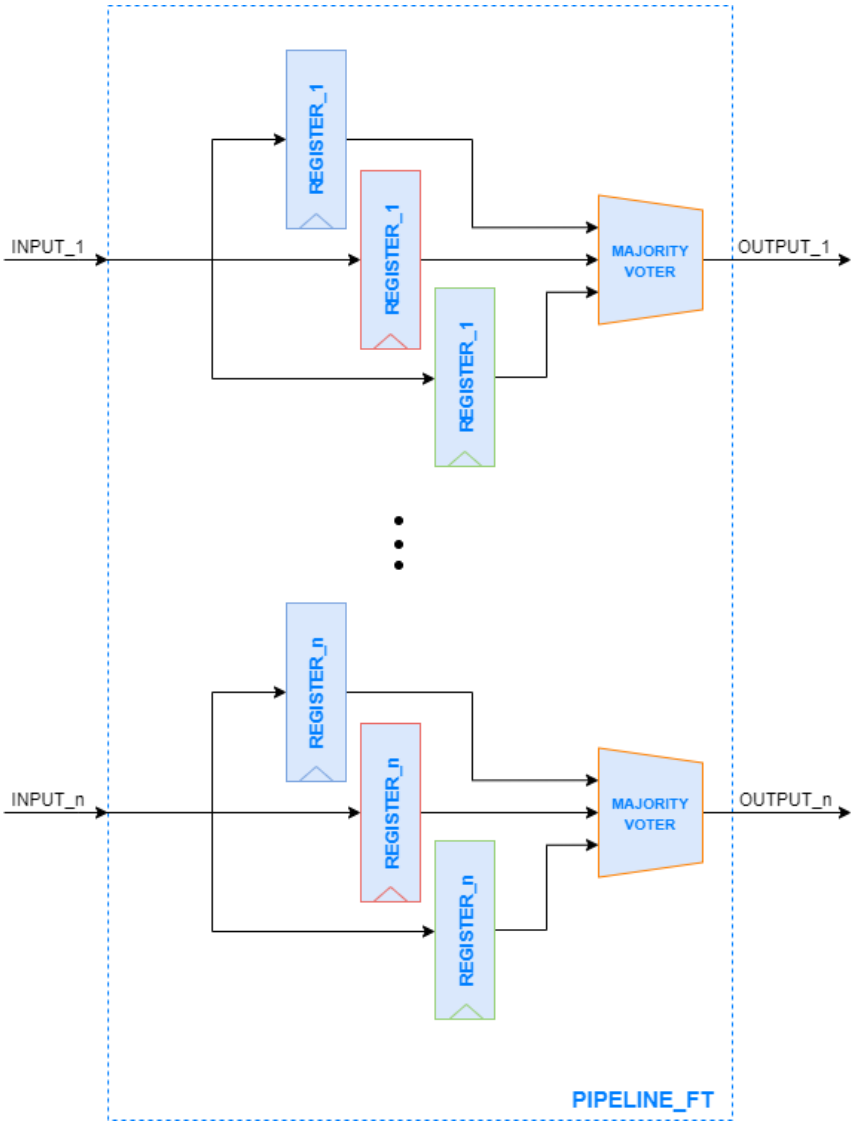

FIGURE 6.2: RTL design of the FT Pipeline registers

registers. In this way, the triplicated outputs from the pipeline registers go directly to the three copies of the decoder, leading to a reduction of delay, power consumption, and area.
Time redundancy could be another solution which allows to reduce the area (with respect to the TMR, that triplicates the decoder), but its main drawback is the increment of the execution time (which is triplicated).

In Figure 6.3, a simplified version of the design of the FT version of the decoder is shown. The input is the ensemble of all the inputs, as well as the output. The majority voter is actually decomposed into as many voters as the number of the outputs.

6.3 Fault tolerant version of the Controller

In the CV32E40P core, the main controller is placed in the ID stage and is responsible to generate the control signals for all the units inside the ID and the next stages. It

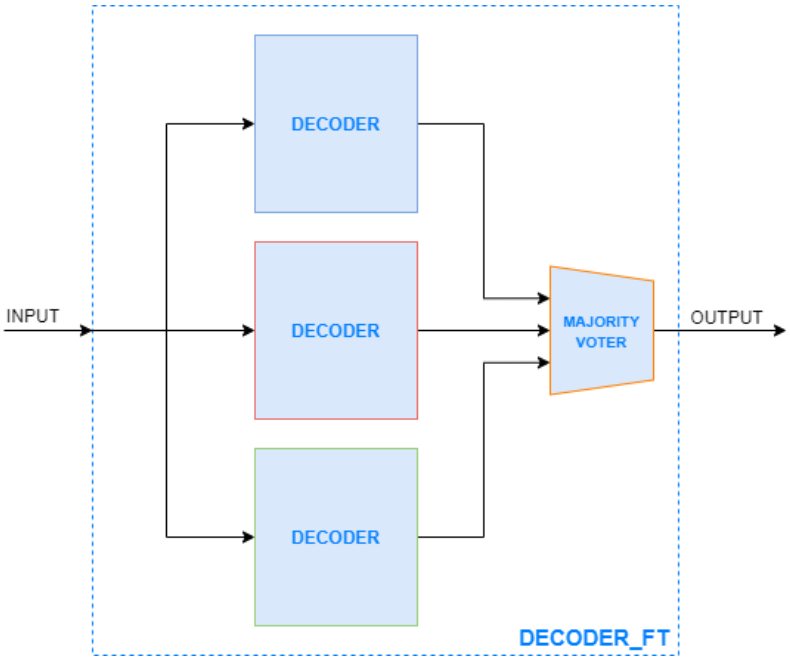

FIGURE 6.3: RTL design of the FT Decoder

CONFIGURATION	Area [GE]	Max Delay [ns]
normal	849	3.30
fault tolerant	4346	5.52

TABLE 6.3: Some relevant hardware characteristics for the three different versions of the Register File.

can be considered one of the most important components of the core from a functional point of view.

The controller is described as a 18-states FSM, so it is a mix of combinational and sequential blocks. In every FSM, the "present state" and the "next state" signals play a key role, the first one is the input of a combinational block responsible to generate the output signals, the last one, instead, is generated by another combinational block. Such a component needs to be treated differently with respect to a pure combinational (like the decoder) or pure sequential block (like the pipeline registers) even because of its feedback loop. For this reason and for a better understanding of the FT technique to apply, a clear schematic of the structure is given in Figure 6.4.

The structure can be seen as two combinatorial blocks divided by a register, which performs the transition from the next state to the present state. The main characteristics is the fact that the present state signal is used in feedback to generate the next state, so particular attention needs to be paid to this path when applying a FT technique.

Also in this case, the TMR has been chosen as the most suitable FT technique to

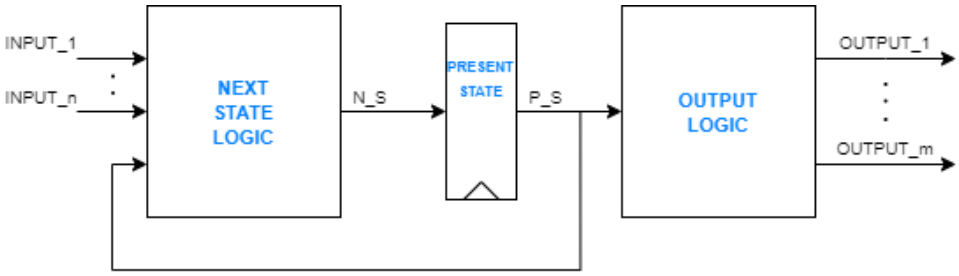

FIGURE 6.4: RTL schematic of a generic FSM

protect the component against the SEEs. In this way, both the combinational and sequential blocks can be protected without complex solutions. Basically, this solution requires simply to triplicate the entire controller (disregarding the type of components), and then apply a majority voter downstream the three replicas to filter out the correct outputs.
This technique applied in this way protects the component against the SEE, but it is not the safest way. In fact, analysing the working principle, if a SEE occurs in the output logic block, it is easily filter out, but the behaviour is completely different if that occurs in the next state logic block or in the present state register. A fault in any of those blocks may lead to the generation of the wrong present state, which in turn leads to the wrong generation of all the outputs. Anyway, this is not a problem, because the wrong outputs are filtered out by the majority voter. The real and considerable problem is the wrong present state. Storing the wrong present state means that the copy of the controller will work following a different evolution of the states. This represents a problem since the SEE is filtered out only at the moment it occurs, but its effects remain silent in the component for the remaining time making that controller's replica completely useless from the TMR perspective.
For example, a SEE occurs in one of the three copies (copy 1) of the controller causing the problem described above, after some time another SEE occurs in one of the two other copies (copy 2). In this scenario, the outputs of both the copy 1 and copy 2 will be incorrect, possibly even different among themselves. The correct outputs would be those from the copy 3, but, being wrong all the outputs from the three copies, the majority voter will not be capable to select the correct ones, and so the error will be only detected and not corrected. This leads the entire machine to operate in an unpredictable manner.
The solution against this problem is the application of a majority voter downstream the present state register as well. The voter lets the correct present state pass through, discarding the wrong one. In this way, the present state signal, which goes in input to the next state logic, is always valid and so the next state logic can produce the correct next state. This means that the effects of the fault occurred are completely eliminated because it is sure that the controllers evolve towards the correct state in any case. All the three copies will always work in the right manner and the related outputs can be always considered valid for the comparisons carried out by the voters.
In conclusion, the problem is solved simply by means of an extra majority voter which brings the faulty controller back to the right state.

In Figure 6.5, the complete RTL view of the enhanced TMR technique applied to the controller is presented. The generic input and the generic output are reported just as an example to show the signals flow.

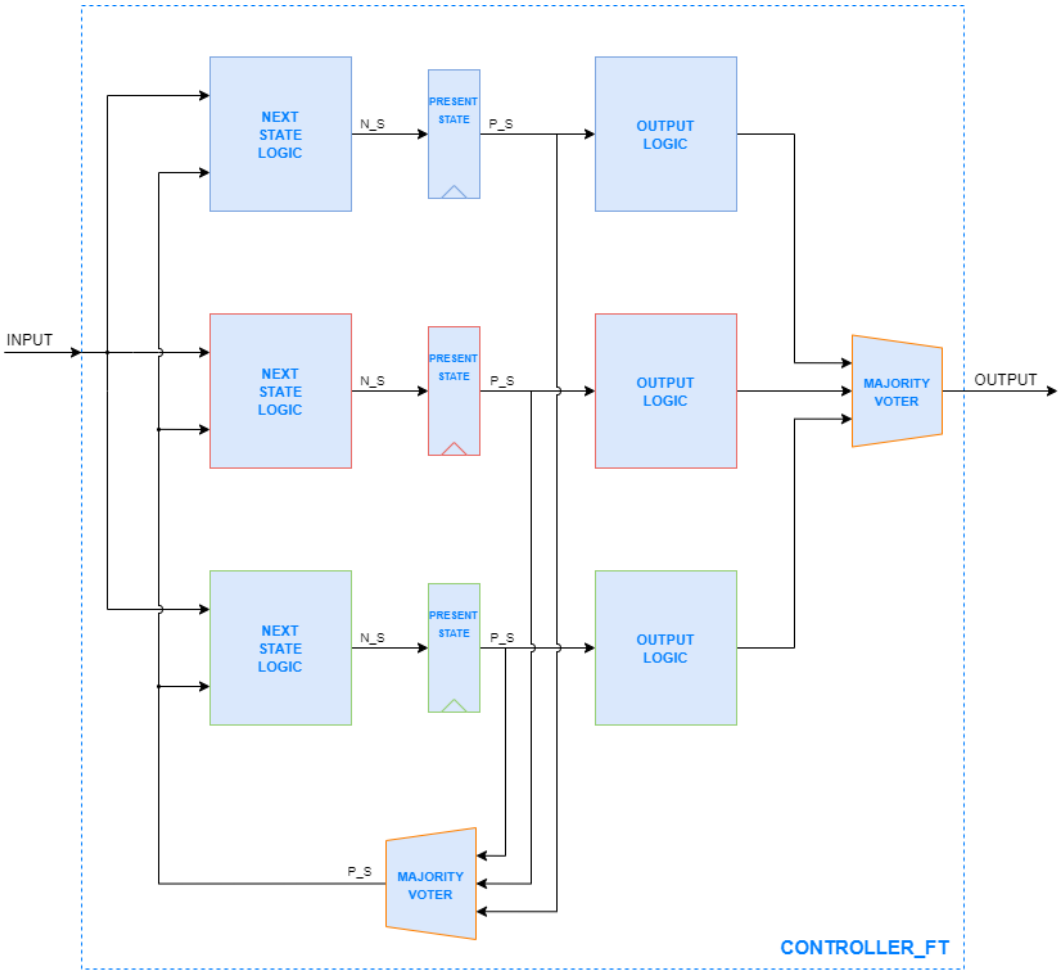

FIGURE 6.5: RTL design of the FT Controller

6.4 Fault tolerant version of Register File

The register file, as explained before, is probably the most important component of the whole core and not only of the ID stage. Its area occupies about a quarter of the entire core, so it is a very susceptible component from the radiation point of view. For this reason, the design of this component will provide two different variants (apart from the unprotected one) in order to make it tolerant against both soft and hard errors.

To have a complete understanding of the solutions applied to the register file, it is advisable to explain in detail its structure.
The register file is composed by 32 locations 32-bit wide (like every RISC-V core), each one used for a specific purpose, but this is not of interest for the design goal. The variant of the CV32E40P core provides 3 reading ports and 2 writing ports. As regards the reading ports, they require an address in input to put in output the data read. The address ports are named "Addr_RX", the data ports "Read_X" (where X goes from A to C). Analogously, the writing ports require both the address and the data to write. The address and data ports, in this case, are named respectively "Addr_WX" and "Write_X" (where X goes from A to B).
According to the specifications, the address signals need to be 5-bit wide, in order to be able to select any of the 32 locations of the register file, while data signals are

CONFIGURATION	Area [GE]	Max Delay [ns]
normal	1641	7.89
fault tolerant	6026	9.94

TABLE 6.4: Some relevant hardware characteristics for the three different versions of the Register File.

32-bit wide, since 32-bit is the parallelism of the CV32E40P core. Actually, if the "F" ISA extension is enabled, the additional floating point register file is instantiated, and the address signal requires an additional bit in order to be able to select one of the two variants (integer or floating point). The floating point register file, however, has the same characteristics of the integer register file.

6.4.1 First FT version (soft errors)

Among the multitude of FT techniques that can be applied to the register file, the most suitable are the Hamming code and its derived variant the Hsiao code. From the comparison of the two techniques carried out by taking into account Table 4.1 and Table 4.2 and by considering the analysis performed by Hao Li et al. [50], the Hsiao code results the best candidate to protect the register file against SEEs.
In fact, this technique allows to reduce delay, area, and also power consumption with respect to the other ECCs. Moreover, with respect to the TMR, the area is considerably reduced because only 7 bits (parity bits) are added for each location of the register file instead of triplicating all the locations. Just to give some numbers, the Hsiao code implies an increase of area of about 22% plus the overhead due to encoders and decoders, while the TMR leads to an increase of 300% plus the overhead due to the majority voters. The only advantage probably would be the reduction of the delay, but just for the writing operations, since for the reading operations the delay is, more or less, the same, because the Hsiao decoder and the majority voter have equal delay. This is a minimal advantage to take this technique into account, considering the enormous drawbacks of area and, consequently, of power consumption (all the reading and writing operations are triplicated, as well).
In conclusion, the Hsiao code is the FT technique selected to protect the register file.

The Hsiao ECC technique is composed by an encoder (to encode the data to write into the RF), a decoder (to decode the data read from the RF), and an extension of the data width from 32 to 39 bits (data + code are now stored). The general scheme of the RF with the Hsiao's technique applied is reported in Figure 6.6. Two encoders and three decoders are instantiated because of the two write and the three read ports implemented for this version of the register file.

The design of the encoder and the decoder is a tricky task. Their design is based on the definition of the so called parity-check matrix (also known as H-matrix), which outlines the combinations of the input data bits to produce the related parity bits. For this purpose, MATLAB comes into help. A custom MATLAB code has been developed to generate the H-matrix and test the algorithm both for encoding and decoding data. Another utility of the Matlab code is the possibility to validate

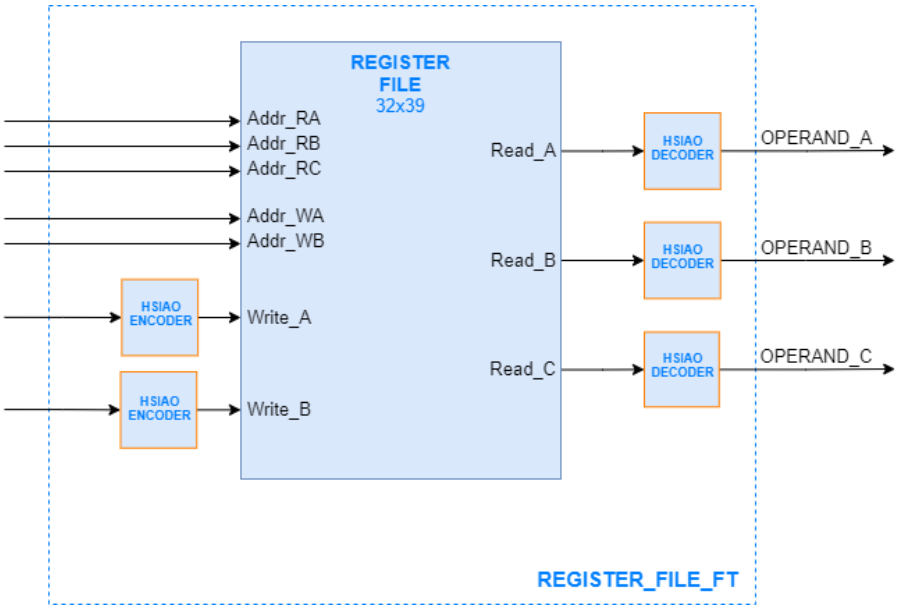

FIGURE 6.6: RTL design of the FT Register File

the future hardware design in a simpler way. The code developed is listed in Appendix A.

To simplify the understanding of the Hsiao's algorithm, it is important to clarify the role and the properties of the H-matrix. First of all, the Hsiao ECC requires that specific rules must be followed in the generation the H-matrix. In fact, these rules, different with respect to the Hamming ECC, allow to reduce the number of connections and logic gates for the hardware implementation, leading to more efficient performance. The H-matrix must respect the following rules:

1) every column contains an odd number of 1's;

2) no two columns are the same;

3) the difference of the number of 1's in any two rows is not greater than 1;

4) the total number of 1's reaches the minimum.

Secondly, the matrix has conceptually a unique meaning both for the encoder and for the decoder, that is the definition of the sets of input data bits which are used to generate a certain parity bit. In this case where the data width is 32-bit, the H-matrix is composed by 7 rows and 39 columns. The number of rows is equal to the number of parity bits required, and so each row is referred to a parity bit. While the number of columns is equal to the amount of bits of the data plus the number of parity bits, and so each i-th column is referred to the i-th bit of the data in input. Indexes from 0 to 31 refer to the data bits, while those from 32 to 38 refer to the additional parity bits. At this point, the parity bit is computed by carrying out the XOR operation among all the bits of the data correspondent to the elements at '1' in the row of the matrix. The XOR operation is the basis for the algorithms of encoding and decoding, which differ from each other in some particular conditions to be assumed when performing that operation.
The encoding algorithm requires that the parity bits are initialised to '0', so that the XOR operation is like if it were performed only among the bits of the dataword (input data). The result of that operation is a parity bit.

The decoding algorithm, instead, requires to carry out the XOR operation among all the bits of the whole codeword (including the parity bits). In this case, the result of the operation is a syndrome bit, which is then used to find the position of the error inside the codeword.

The content of the H-matrix generated by means of the Matlab code is visible in the following table (Table 6.5). It respects the above-mentioned rules. According to the previous explanation, the header column reports the indexes of the 7 parity bits, and the header row reports the indexes of the 32-bit data plus the 7 additional parity bits to compose the complete codeword (indexes are ordered accordingly to the standard adopted by IEEE for a 32-bit microprocessor architecture [51], MSB at the left-end, LSB at the right-end).

PARITY BIT	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	0	1	1	0	0	1	0	1	0	0	1	0	1	0	0	1	1	0	0	0	1
2	0	1	0	0	0	0	0	1	0	1	0	0	1	0	1	0	0	1	1	0	0	1	0	1	0	0	1	0	1	0	0	1	1	0	0	1	0	1	0
3	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	1	1	0	0	1	0	1	0	0	1	0	1	0	0	1	1	0	0	1	0	0	1	1	0
4	0	0	0	1	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	0	1	1	0	0	1
5	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	0	1	0	0	1	0	1	0	0	1	1	0	1	0	0	0	1	1	0	0	1	0	1
6	0	0	0	0	0	1	0	0	1	0	0	1	0	1	0	1	0	0	1	0	1	0	0	1	1	0	0	1	0	0	1	1	0	0	1	0	0	1	0
7	0	0	0	0	0	0	1	0	0	1	0	1	0	1	0	0	1	0	1	0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0

TABLE 6.5: H-matrix to generate the parity bits of the Hsiao ECC

At this point, it is possible to show the connections required for the encoding and decoding operations.

As regards the encoder, as a proof of concept, the first parity bit is theoretically generated by means of a XOR operation among the 0th, 4th, 5th, 8th, 10th, 13th, 15th, 18th, 19th, 22nd, 24th, 27th, 29th, and 31st bit of the input data (the columns from 32 to 38 are not considered). Translated in hardware, this operation is performed by means an equivalent tree of XOR gates with fewer inputs (up to 4, generally). The SystemVerilog code which describes the hardware for the encoder has been developed in a behavioural way, so that the Simulator (ModelSim) or the Design Compiler (Synopsys) can optimise it as much as possible, according to eventual constraint of delay. No other functionalities are required for the encoder.

As regards the decoder, the logic is quite similar for the generation of the syndrome bits, but now the XOR operations are performed taking into account also the columns from 32 to 38. For example, the first syndrome bit is generated by carrying out the XOR operation among the 0th, 4th, 5th, 8th, 10th, 13th, 15th, 18th, 19th, 22nd, 24th, 27th, 29th, 31st and 38th bit of the input codeword.
The main difference with respect to the encoder, that makes this component more complex, is the task of correcting the eventual error occurred due to the fault. This functionality implies much more combinational logic. Also the decoder has been described in SystemVerilog in a behavioural way to let the simulator define the connections more efficiently.
Another minor difference is the outputting of some flags to state if the error has been detected or corrected. These flags are combined with each other and given in output in the form of a signal of 2 bits called “errors_vector[1:0]” (almost equivalent to the “SECDED” signal which states if an error has been corrected or if a double error has been detected). The meaning of the signal is described in the following:

- if the value is '00', no error has occurred;
- if the value is '01', an error has been detected and corrected at least in one decoder;
- if the value is '10', a double error has been detected but not corrected at least in one decoder;
- if the value is '11', the two previous cases have occurred simultaneously.

If either the first or the second case happens, the machine does not need to be stopped and the operations can go on. Instead, if any of the remaining cases occurs, the machine needs to be stopped since the decoder has not been capable to correct the error.

6.4.2 Second FT version (hard errors)

The register file, as stated before, is the most critical component of the entire core, and this may be a valid reason to equip it with an additional level of protection against the permanent faults, apart from the transient ones.
The design of such a version requires the implementation of ad-hoc solutions deriving from the critical aspects to take into account to better protect the component.

The most important aspect which requires particular attention regards the utility of all the locations of the register file, because they are extremely important for the correct execution of the operations. The locations of the register file have different meanings according to the RISC-V standard, and so some of them may be considered more important than others. Anyway, for the solution proposed, they are all treated with the same weight basically for one reason. In fact, a solution that takes into account the different importance of the locations would have required an expensive control circuit to manage their priorities, with the pros of having a more efficient system in terms of total number of locations employed. For example, this kind of solution may require to properly handle the temporary registers, which are considered the less important locations. When a temporary register location is permanently damaged, then the system should try to reuse some unused locations conceived for other purposes so that no additional locations are needed. Anyway, this system should be able to operate without affecting the working flow or without recompiling the program for the custom hardware optimising, for example, the use of the locations. The solution developed, instead, aims to be a generic solution adaptable in several scenarios and extendable to an even higher number of locations.

Another consideration regards the way to protect the register file against the soft errors. In fact, the component must be able to face both the soft and the hard errors without stopping the execution. In this case, the same technique used before (for the first FT version of the RF) has been adopted, that is the Hsiao ECC.

The third and last main consideration made is about the threshold over which the permanent fault should be detected. Generally, a permanent fault is signaled when a certain number of transient faults occurs consecutively. That quantity is a very variable parameter, which depends mainly on the severity of the environment, or simply on the probability of a fault to occur. To satisfy any level of severity, the design proposed provides a pre-synthesis configurable threshold to better meet the specification. The default value is set to 31, this means that if 32 faults occur on the

same location, then the permanent fault is signaled.

The solution proposed takes into account all the considerations made previously, and relies on a "supply" copy of the register file (called "second" register file), which is conceived to replace the locations damaged in the "main" register file. The second register file has exactly the same organisation of the main one, and so it has 3 reading ports, 2 writing ports and 32 locations 39-bits wide.
The mechanism is really simple. Near the two register file, a 32-bit register is placed, called "location_damaged_register", which stores the information about the status of the 32 locations. Each bit of the 32 is related to the correspondent location (the 0th bit to the 0th location, the 1st bit to the 1st location, and so on until the 31st bit). If the bit is set to '0', then the location on the main register file is valid, otherwise, when the bit is set to '1', the related location on the main register file is damaged and so the operation needs to be executed using the correspondent location on the second register file.
Keeping this mechanism in mind, the working principle of the design can be divided into four parts:

1) how the "location_damaged_register" is used at the input ports;

2) how the "location_damaged_register" is used at the output ports;

3) how the "location_damaged_register" is updated;

4) how the "location_damaged_register" interfaces with the CSR.

1 - At the input ports, or more precisely at the address ports, the content of the "location_damaged_register" is first filtered by means of a multiplexer which has as the selector the address for the reading or writing operation. Since there are three reading ports and two writing ports, five 32-to-1 multiplexers (1-bit wide) are instantiated in total. The output of each multiplexer is called "RF_YX", where Y can be R or W (respectively for the reading or the writing operation), and X can be A, B, or C according to the name of the port of the register file.
Then, the signal "RF_YX", whose value is '0' or '1' depending on the register file to use, is exploited to let the address reach either the "main" or the "second" register file, by means of two AND gates 5-bits wide. The first AND gate, linked to the main register file, receives in input the address and the signal "RF_YX" (extended on 5 bits) negated, while the second AND gate is linked to the second register file and is equal to the first one with the exception of the signal "RF_YX" that is not negated. According to this scheme, only one register file will receive the real address, the other one will receive the address '00000' (that points to the only-read location containing the data '00...00'). This is equivalent to not perform any operation in the register file not used.
As regards the two input data ports for the writing operations, an Hsiao encoder is used for each input data. It is placed upstream the duplication of the data, so that only an Hsiao encoder is necessary for each writing port. In this way, the encoded data arrives to both the main and the second register file, but it is actually used only in one register file. This solution implies less area and delay than the other possible solution composed by two filtering gates, like for the address ports, and an Hsiao encoder for each fork of the signal.

2 - At the output ports, the mechanism is really simple. In fact, to properly manage the data coming from the main and the second register file, only an extra 2-to-1 multiplexer (32-bits wide) is needed. The signal "RF_YX" (the same used for the input ports) is used as the selector of the multiplexer, while the data read from the main and from the second register file are the two inputs. The multiplexer lets only the actual read data pass through the downstream Hsiao decoder before being outputted. In total, this scheme is repeated three times, one for each read port.

3 - The mechanism to update the "location_damaged_register" is the most complex part of the design. It is based on the three signals "SECDED_X" (where X is the name of the reading port), generated by each Hsiao decoder, which activate the counters in case of error occurred. There is a counter for each location of the register file, so in total 32 counters are instantiated. To properly activate the counter referred to the location used for the current reading operation, an ad-hoc component has been implemented, called "enable_counters_generator". It receives as inputs both the "SECDED_X" and the "RADDR_X" signals, and produces in output thirty-two 1-bit signals, each one connected to a counter. These 1-bit signals represent the enable of the downstream faults counters. Then, the component raises at '1' the signal correspondent to the location addressed by "RADDR_X" only if the "SECDED_X" signal states that at least one error has occurred in that location. Obviously, the component is capable to raise at '1' up to three 1-bit signals concurrently, since in the worst case an error may occur in all the three locations used for the reading operations.
Once the enable signals are generated, the thirty-two "faults_counters" are ready to eventually increase their count at the rising edge of the clock. When the count has reached the threshold value set for the detection of a permanent fault, a flag is raised and a '1' is written into the correspondent element of the "location_damaged_register". In this way, starting from the next clock cycle, the damaged location in the "main" register file will not be used anymore and will be replaced by the correspondent one in the "second" register file.

4 - The last essential mechanism of the design concerns the support for updating the content of the "location_damaged_register" directly from the CSRs. The CSRs are particular registers that can be accessed (read or written) by means of special instructions. This means that the CSRs can store an information coming from the non-volatile memory present in the system. This feature is extremely important because it is essential that a damaged location in the register file will not be used anymore for the rest of the life of the device, and so the status of the locations needs to be stored permanently and not only into the "location_damaged_register" that is a volatile memory component.
In this way, for example at the power-on reset, that is the event which prepares the system to properly operate, it is possible to load into the CSR the last updated information about the status of the locations, and then load that information into the "location_damaged_register".
Obviously, also the inverse mechanism needs to be implemented in order to keep the content of the CSR always updated. For this scope, an enable signal has been instantiated which will raise as soon as any of the faults counters raise its flag because a permanent fault has been detected. More precisely, the enable signal is generated by an OR operation performed among all the flags coming from the counters, and it is delayed by one clock cycle to be synchronised with the updated content of the "location_damaged_register". So, the enable signal states when the CSR needs to be written with the new content of the "location_damaged_register".

In Figure 6.7, the RTL schematic view of the second FT version of the Register File is reported, showing the most important components, sometimes described at logic-gate level and sometimes described as a black-box.

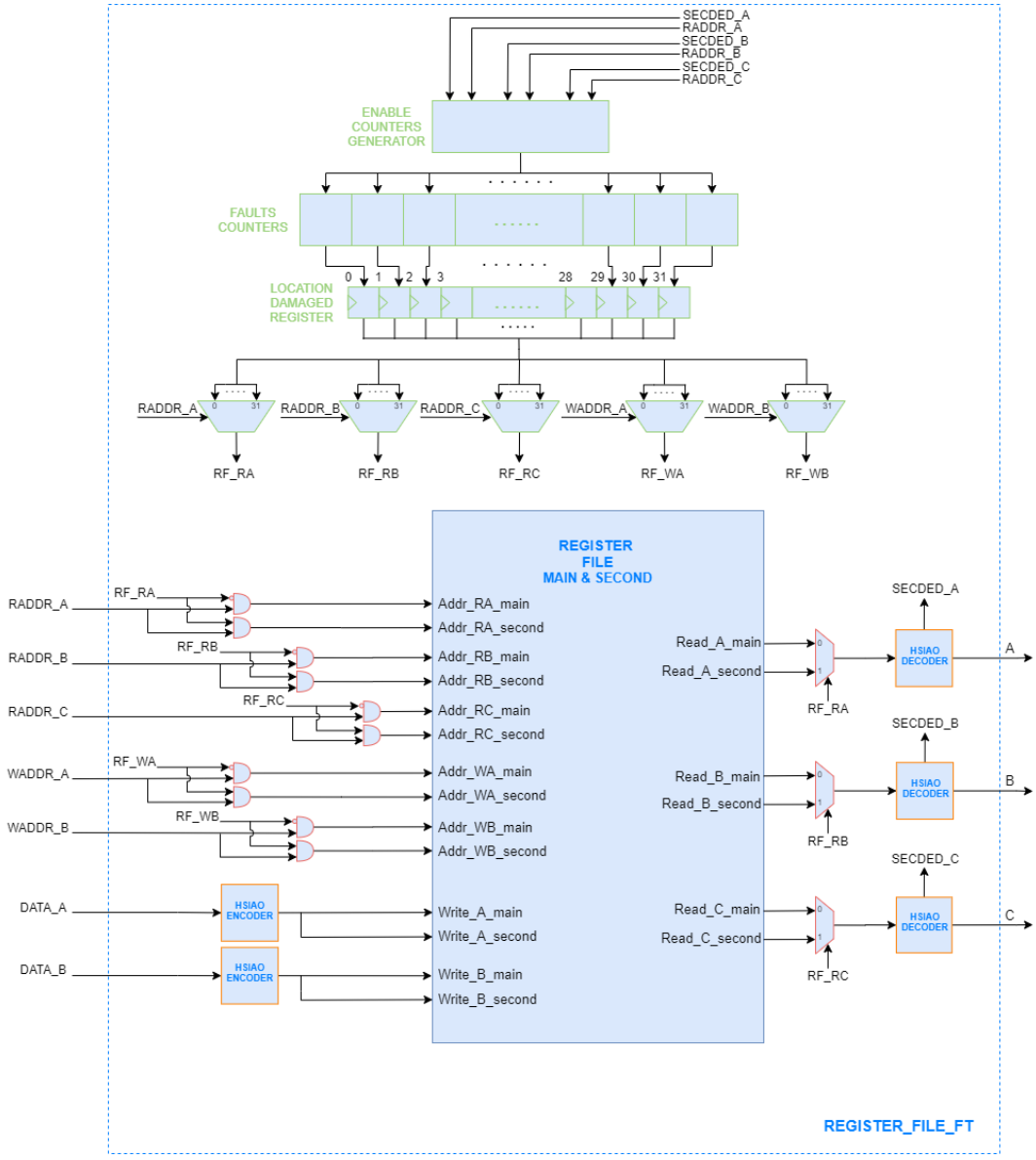

FIGURE 6.7: RTL design of the FT Register File

A relevant detail not reported in the above schematic is the support for updating the "location_damaged_register" with the content of the CSR, and the related mechanism. This choice comes from the fact that the hardware used for implementing that part is really basic and easy to understand, and its representation would have made the schematic unclear.
However, it is interesting to observe how the interface has been kept the same as the original one, with the exception of the three signals necessary for interfacing with the CSRs (not reported for simplicity). This is important to understand how the different versions of the Register File can be selected simply by setting a parameter in the design.

CONFIGURATION	Area [kGE]	Max Delay [ns]	Read Delay [ns]
normal	11.55	1.45	1.45
soft	15.49	5.24	4.39
hard	31.13	9.25	8.36

TABLE 6.6: Some relevant hardware characteristics for the three different versions of the Register File.

6.5 Configurations

After having described the protection techniques applied to each component, the complete scheme of the ID stage can be presented. The schematic should show each relevant component with the additional pieces of hardware instantiated to protect it against faults. In Figure 6.8, the complete RTL schematic of the ID stage is presented. The IF/ID Pipeline registers, the Decoder and the Controller make use of FT techniques based one the TMR, so some majority voters are placed downstream the three replicas as a proof of concept (also the signals taken into account are selected as a proof of concept).
The register file, instead, uses a technique based on the Hsiao ECC, so instances of the encoder and the decoder are placed, respectively, upstream and downstream the component (in their exact quantity in the schematic).

Looking at the schematic, the three replicas of each component have the contour coloured in blue, red, or green, while the additional components for the FT version have a yellow contour. These colours have been chosen to give a clearer idea of the organisation of the design and how it can be composed and decomposed. Depending on the configuration selected at the pre-synthesis stage, some components may be protected while others may be not. For what concerns the not selected components, the yellow contoured components are removed, and, in case, only one of the three replicas is instantiated.

As stated at the beginning of the chapter, there are 24 possible different configurations that can be chosen. The configuration to use can be selected by means of the design parameter, called “ID_FAULT_TOLERANCE”, representable on 5 bits. The first bit (the LSB) sets the FT version of the Controller, the second bit sets the Decoder, the third the Pipeline registers, and the last two set the three different versions of the Register File.
In Table 6.7, all the configurations of the ID stage are shown, reporting in the left column all the possible values of the parameter “ID_FAULT_TOLERANCE”, and in the remaining columns the correlated versions of the components that are activated. The term “normal” indicates the standard (unprotected) version of the component, “FT” or “FT_SOFT” indicate the FT version against soft errors, and “FT_HARD” indicates the FT version against hard errors.
From the table above, it is possible to notice that the binary code ’01111’ (’15’ in radix-10) corresponds to the most safe configuration against soft errors, while to activate the configuration that protects the register file also against the hard error it is necessary to set to ’1’ the MSB of the code (that means a number equal or higher

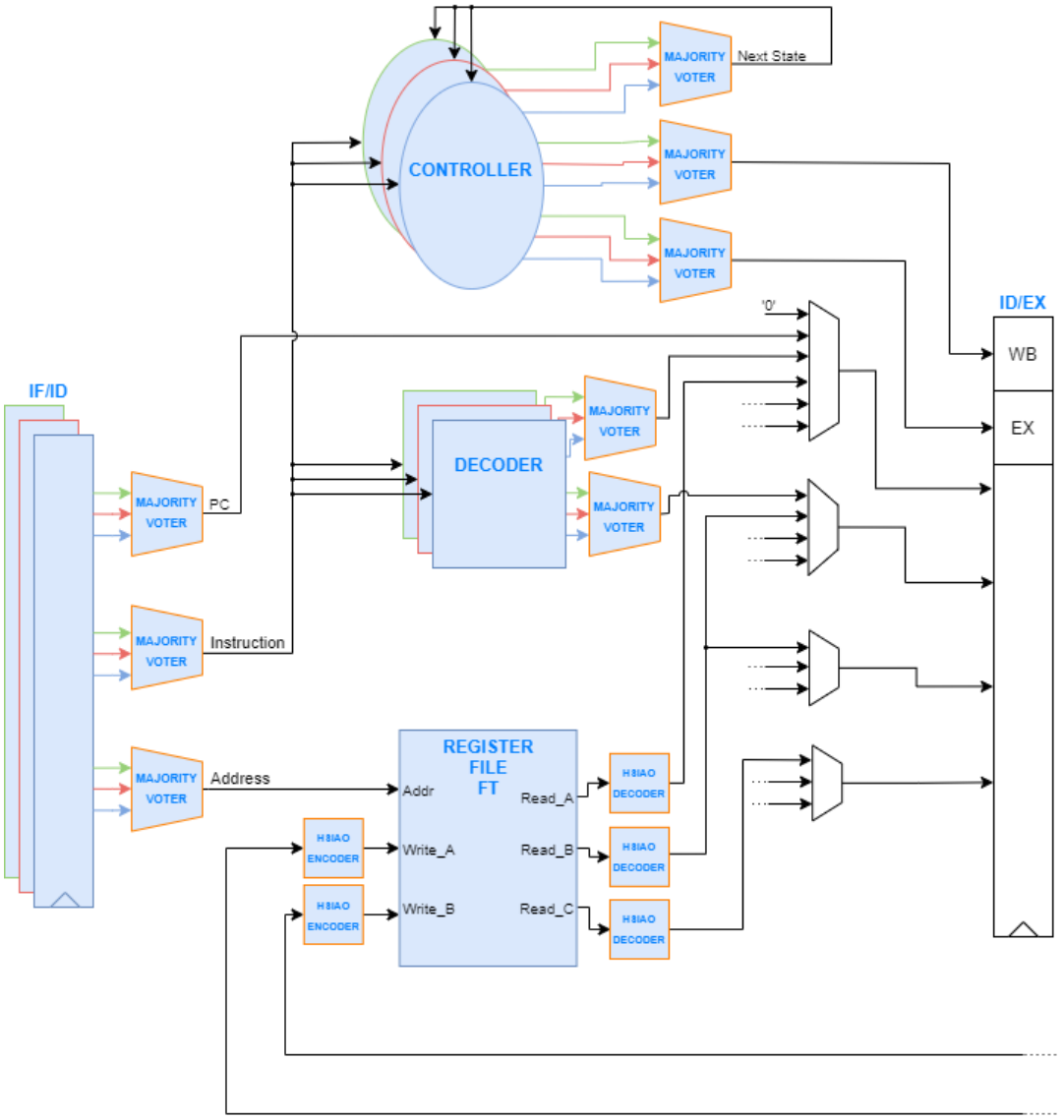

FIGURE 6.8: RTL design of the complete FT ID stage

than '16' in radix-10).

For each configuration, a certain level of fault tolerance can be reached, and different hardware characteristics are derived. The first characterisation is discussed in the next chapter, while the latter is presented below.

To derive the hardware characteristics and make some relevant comparisons, the synthesis has been carried out for each configuration of the stage (24 synthesis in total) by means of Synopsys Design Compiler. The library used is the same adopted by P. D. Schiavone et al. [43] to determine the characteristics of the core RI5CY, that is the UMC 65nm. Obviously, the results may differ since in this case the synthesis' process has been carried out without performing any high-effort optimization. Anyway, the focus should be posed on the comparisons of the various configurations expressed in percentage with respect to the original one (FT_CODE=0).
In Table 6.8, it is possible to observe how the different configurations have different values for area and delay, and how the application of the version of the register file against the permanent errors affects those parameters. The power consumption has

not been evaluated because of the low dependability and the high dependence on several factors, such as the workload, the clock frequency, the voltage supply, etc.
The results have been obtained without imposing any constraint to the Design Compiler, and with "exact_map" option for the compilation of the sequential elements.

The results show that the area for the fully protected version against the soft error (FT CODE 15) is increased but not doubled or triplicated (1.72x more extended than the original design), even though a lot of hardware redundancy is applied. This is reasonable since the most critical component, the register file, makes use of the Hsiao ECC that allows to save a lot of area with respect to the TMR. Also the delay gets worse, with an increment of about 41% (about 4ns). The components which mainly impact on the critical path are the pipeline and the register file. Overall, the hardware characteristics can be considered reasonable and acceptable considering further optimisations.
The version against the hard errors intuitively has even more limits. The fully protected version has an area 2.63x greater than the unprotected version (FT CODE 0) and 1.53x greater than the one against the soft errors (FT CODE 15). The delay, instead, is increased respectively of about 75% (about 8ns) and 24% (about 4ns).
As regards the delay, an additional consideration is necessary in order to better understand its implications. In fact, the delay reported in the table represents the delay of the most critical path of the ID stage and not of the entire core. This means that even if the delay gets worse, the maximum clock frequency reachable by the core may be higher.

FT CODE binary (int)	CONFIGURATIONS			
	Controller	Decoder	Pipeline	Register File
00000 (0)	normal	normal	normal	normal
00001 (1)	FT	normal	normal	normal
00010 (2)	normal	FT	normal	normal
00011 (3)	FT	FT	normal	normal
00100 (4)	normal	normal	FT	normal
00101 (5)	FT	normal	FT	normal
00110 (6)	normal	FT	FT	normal
00111 (7)	FT	FT	FT	normal
01000 (8)	normal	normal	normal	FT SOFT
01001 (9)	FT	normal	normal	FT SOFT
01010 (10)	normal	FT	normal	FT SOFT
01011 (11)	FT	FT	normal	FT SOFT
01100 (12)	normal	normal	FT	FT SOFT
01101 (13)	FT	normal	FT	FT SOFT
01110 (14)	normal	FT	FT	FT SOFT
01111 (15)	FT	FT	FT	FT SOFT
1X000 (16/24)	normal	normal	normal	FT HARD
1X- - - (>16)	—	—	—	FT HARD
1X111 (23/31)	FT	FT	FT	FT HARD

TABLE 6.7: The 24 possible configurations of the ID stage (where 'X' means "don't care"). In the last row, the symbol '-' means that the same pattern of the previous rows for that bit/component is followed.

FT CODE	Area [kGE]	Area variation [%]	Delay [ns]	Delay variation [%]
0	17.17	100.00%	10.49	100.00%
1	19.92	116.00%	10.49	100.00%
2	21.35	124.34%	11.20	106.76%
3	24.10	140.33%	12.13	115.63%
4	19.21	111.89%	13.00	123.92%
5	21.96	127.89%	13.00	123.92%
6	23.40	136.24%	13.84	131.93%
7	26.14	152.24%	14.89	141.94%
8	20.58	119.84%	12.35	117.73%
9	23.33	135.84%	12.35	117.73%
10	24.76	144.17%	12.32	117.44%
11	27.51	160.16%	12.32	117.44%
12	22.62	131.74%	14.73	140.41%
13	25.37	147.74%	14.73	140.41%
14	26.80	156.08%	14.78	140.89%
15	29.55	172.07%	14.82	141.27%
16 (24)	36.22	210.88%	15.99	152.43%
17 (25)	38.97	226.88%	15.99	152.43%
18 (26)	40.40	235.21%	15.96	152.14%
19 (27)	43.14	251.20%	15.96	152.14%
20 (28)	38.26	222.77%	18.34	174.83%
21 (29)	41.01	238.77%	18.34	174.83%
22 (30)	42.44	247.12%	18.38	175.21%
23 (31)	45.19	263.11%	18.38	175.21%

TABLE 6.8: The 24 possible configurations of the ID stage with the related values of area and max delay, expressed both in absolute value and in percentage with respect to the configuration with FT CODE equal to 0.

Chapter 7

Benchmark

Once the design of the ID stage is ready, the next steps concern the validation of the system and the evaluation of its fault tolerant performance. An ad-hoc environment has been developed to reach these goals. This environment creates the support to all the necessary operations regarding the validation of the model, the simulations in general, and the benchmark.
After having validated the fault tolerant version of the stage, the last aspect to take into account is the benchmark of the fault tolerance capabilities of the design. This is the most interesting aspect of the entire process of designing a fault tolerant system and also the most time- and effort-demanding task. It requires to reproduce those conditions that would lead to the generation of a fault (more precisely a SEU), and to reproduce all the possible scenarios that may occur while the core is running.
At this point is necessary to explain that, for the scope of the thesis work, the benchmark has been carried out simply to have a roughly estimation of the fault tolerance capabilities of the system and so an approximate assessment of the complete design. This is because a complete and exhaustive benchmark would have required several months.

To test the fault tolerance of an electronic system, it is necessary to create an environment where it is possible to reproduce the occurrence of faults. Such an environment should be based on a fault injection mechanism that forces the fault at a certain instant time and on some specific components of the design. In literature, there exist different methods based on fault injection strategies. Basically, they can be grouped into five categories [52]: hardware-based fault injection, software-based fault injection, simulation-based fault injection, emulation-based fault injection, and hybrid fault injection.
A brief description for each category is given below.

- Hardware-based fault injection: this method makes use of particular devices to induce a real fault into the target system's hardware at physical level. For example, J. R. Samson et al. [53] developed and demonstrated a Laser Fault Injection (LFI) technique to inject soft faults into VLSI circuits.

- Software-based fault injection: this technique consists of reproducing at software level the errors that would have been generated by a fault occurred in the real hardware.

- Simulation-based fault injection: this method is based on the development of the electronic system by means of an hardware description language so that the system can be simulated by means of an hardware simulator (i.e. ModelSim or QuestaSim) capable to force the fault in any point of the circuit and at any time instant.

- Emulation-based fault injection: this technique represents an alternative to the simulation-based one because it allows to speed-up the overall time spent to reproduce all the possible scenarios by using an FPGA emulating the circuit.
- Hybrid fault injection: this approach combines two or more categories of fault injection techniques exploiting the advantages of each considered technique.

All these techniques are more or less equivalent from the results point of view. What really changes are the tools and devices required and the time spent to execute the benchmark.
For the scope of this thesis work, the simulation-based method has been used for the benchmark, since it is the most low-cost one and does not require any special hardware, only a computer with a design simulator installed and some scripts coded to handle the data flow. Obviously, it has the main drawback of being very time-demanding because of the high number of simulations to perform, so the accuracy of the results will be limited because of the limited time. The other critical aspect to take into account is the organisation of all the scripts to synchronise and properly handle all the data and tools involved in.

7.1 Development Environment

Before explaining in detail the mechanism implemented for the benchmark, it is necessary to present the development environment that has been developed to facilitate all the operations concerning the validation and the simulation of the system. The development environment has been actually developed by the entire team (which aims to create a fully fault tolerant core) because it is useful for supporting the design process of all the stages (IF, ID, EX, and WB) of the core.

Basically, it extends the features provided by the verification environment developed by the OpenHWGroup [54] for some RISC-V cores. It has already testbench and UVM support. The features added to the environment are mainly implemented by means of a Bash script called “comp_sim.sh”, which in turn may call other scripts coded in Python, TCL [55], or other languages to complete that activity. The most relevant added features are listed below:

- compilation of any C-program by means of a generic Makefile;
- adaptation of the CoreMark [56] program to the CV32E40P core;
- simulation of the core, or any sub-module, running any C-program;
- support for saving the input and output signals of the simulation of the core, or any sub-module, to speed-up further simulations;
- automatic download of the reference architecture and the architecture under test from GitHub repositories;
- comparison between the simulations performed by the reference architecture and the one under test;
- simulation with fault injection on the core or any sub-module running any C-program;
- fault tolerance benchmark of the core, or any sub-module, running any C-program;

- support to obtain the benchmark result with a certain accuracy;
- support to efficiently handle the data of the benchmark simulations.

The support to these options has notably facilitated the entire benchmark process, but other requirements need to be defined to make the while process clear and valid, and make the results more understandable. For example, the workload needs to be defined to know which scenarios can be replicated with the simulations. Moreover, the mechanism to apply the fault injection needs to be clarified, and the number of simulations per benchmark must be reported and justified in order to give a clear idea of the meaning of the results.

7.2 Benchmark process

The benchmark process is composed by different steps can be grouped into three. The first one is the definition of the workload, that sets the scenarios covered by means of the simulations. The second one is the validation of the system as prior condition to validate the results deriving from the benchmark. Finally, the last one is the mechanism to inject the fault into the system and how to monitor the execution.

7.2.1 Workload

The workload for the benchmark process is important since it allows to understand which components are mainly used during the simulation and so which scenarios can be represented by running a specific program. For the scope of this thesis work, the programs used are taken directly from the verification environment (called "core-v-verif") for the core under test [54], discarding those related to the "PULP" options not taken into account in this project.
The programs used are coded in C-language and are:

- counters;
- csr_instructions;
- cv32e40p_csr_access_test;
- dhrystone;
- fibonacci;
- generic_exception_test;
- hello_world;
- illegal;
- interrupt_bootstrap;
- interrupt_test;
- misalign;
- modeled_csr_por;
- perf_counters_instructions;

- requested_csr_por;
- riscv_arithmetic_basic_test_0;
- riscv_arithmetic_basic_test_1;
- riscv_ebreak_test_0.

In total, they are seventeen programs, all coded in C-language, and compiled by using the RISC-V toolchain. These programs are provided with self-checking capabilities, that means they are capable to determine autonomously if their execution worked fine or not. Actually, some of these programs fail the self-check, but they have been used anyway. This is because the starting point of this thesis project is an existing core, the CV32E40P, that has been assumed to be the reference architecture correct by definition. The starting version of the core is the reference against which to compare the new fault tolerant version. So, even if some programs do not work as expected, they have been included in the workload because what really matters is that the new architecture works exactly as the reference one.
However, this choice comes also from the fact that the fault tolerant design has been developed so that it is possible to replace the base-component (i.e. the decoder or the register file) with another version having the same interface without altering the working principle. The fault tolerant design is applicable also to further releases of the core that will fix the current bugs.

The programs above mentioned are specific to test the RISC-V core functionalities, so they are not universally accepted (except for the Dhrystone) to compare the benchmark results. In order to have a benchmark of the core capable to be compared against other cores (not only RISC-V ones), the CoreMark benchmark program has been taken into account.
The CoreMark is a benchmark developed by EEMBC widely used to measure the performance of MCUs and CPUs [56]. It is very useful for the fault tolerance benchmark as well since it contains several algorithms implemented. This means that many scenarios can be simulated simply by running this program. The algorithms implemented are: list processing (find and sort), matrix manipulation (common matrix operations), state machine (determine if an input stream contains valid numbers), and CRC (cyclic redundancy check). Obviously, the CoreMark benchmark has many others advantages with respect to other benchmark programs, for example it ensures that the compilers cannot pre-compute the results at compile time, but they are not so relevant for this purpose since only the fault tolerance capabilities are evaluated and not the performance. For this reason, only one cycle of the CoreMark is executed, and not a number of cycle so that the total execution time is 10 seconds, as the standard for the CoreMark results would require. Being runned only for one cycle, it will be called “coremark_1” in this context.
The porting of the CoreMark program to the CV32E40P core has been done by starting from the version of the CoreMark files adapted for the 64-bit riscv core called “riscv-boom” [57]. This has facilitated the task since only the functions that manage the time inside the core needed to be adjusted for the CV32E40P.

However, among the programs used for the workload there is the Dhrystone benchmark program that may be universally accepted as the CoreMark, because it is used as well to benchmark the CPU performance. Unfortunately, this benchmark program is becoming obsolete because of some drawbacks that make its results not so clear and not so easily comparable. So, the CoreMark is preferred for benchmarking purposes. However, the Dhrystone code is dominated by simple integer

arithmetic, string operations, logic decisions, and memory accesses intended to reflect the CPU activities in most general purpose computing applications [58].
To sum up, the CoreMark and the Dhrystone are two benchmark programs that contain several algorithms to better evaluate the performance of the CPU. The main difference is that the manner is becoming even more used, while the latter disused. But there is another secondary difference, not so negligible from the simulation point of view. In fact, even if the CoreMark is more used, it requires a lot of simulation time to terminate its execution, so the benchmark using the CoreMark may be very time-demanding. On the contrary, the Dhrystone requires an acceptable amount of simulation time, so that will be preferred to carry out some quicker benchmarks (about one order of magnitude saved in the total time required.

In Table 7.1, the information regarding the number of clock cycles required by each benchmark program to complete is reported. Note the huge difference of workload between the "CoreMark" and the rest of the programs.

Program	N cycles
coremark_1	18141307
counters	49750
csr_instructions	10210
cv32e40p_csr_access_test	950
dhrystone	195050
fibonacci	98500
generic_exception_test	2193
hello_world	13432
illegal	10847
interrupt_bootstrap	5562
interrupt_test	8046
misalign	358798
modeled_csr_por	41410
perf_counters_instructions	80460
requested_csr_por	62795
riscv_arithmetic_basic_test_0	32223
riscv_arithmetic_basic_test_1	32446
riscv_ebreak_test_0	41770

TABLE 7.1: Workload in terms of number of clock cycles required to complete

7.2.2 Validation

The first operative step of the benchmark process is the validation of the new model with respect to the original one. This check is necessary in order to have a further confirmation about the equivalence between the original design and the fault tolerant one. Actually, because of the 24 possible configurations, all the 24 different design need to be validated.
In this context, with the term equivalence is meant that the two models taken into account should work in the same way, with the same timing, that means that all the signals in output from one stage should be equal cycle by cycle during the simulation.
Obviously, the first and main validation has been carried out by means of standard testbenches coded for the purpose of verifying the correctness of the new fault tolerant components. So, with the first validation the design has been evaluated considering its logical behaviour. Instead, this further validation is to be sure about the equivalence of the model and how to interpret the results of the benchmark discarding the possibility that they may be affected by design mistakes.

The development environment comes into help to accomplish this task in an efficient way. It provides the possibility to simulate with QuestaSim the core running any C-program, and meanwhile saving both the input and the output signals respectively in a “vcd” and a “wlf” file. Then, these signals can be reused to run the simulation of any fault tolerant version of the core and to compare simultaneously the outputs cycle by cycle.
This procedure has been carried out taking the original core (unprotected) as the reference version. This means that its inputs and outputs have been saved and used, respectively, to feed all the simulations (by means of the "-vcdstim" option of the "vsim" command of QuestaSim) and as the golden outputs against which to compare the outputs of all the simulations of the 24 fault tolerant versions of the core (by means of the "compare" command). This procedure has been repeated also for each program of the workload.
To clarify the flow of this process, a block diagram is reported in Figure 7.1, dividing the process into two sub-steps. In the diagram, with the symbol “X” is meant the name of any C-program in the workload.

The “comp_sim.sh” script is the main file in the development environment from which all the processes start. In this scenario, it is first used to launch the QuestaSim simulation of the unprotected core running the “X” program. Meanwhile, QuestaSim has been initialised by “comp_sim.sh” to run the “save_data.tcl” script. At the end of the simulation, the TCL script commands QuestaSim to produce two files related to the “X” program, one with the input signals, “input_X.vcd”, and the other with the output signals, “output_X.wlf”. In the second step, “comp_sim.sh” is used to launch the QuestaSim simulation of the fault tolerant core while using the signals stored in “input_X.vcd” as input stimuli. In this step, QuestaSim is initialised to launch the “compare.tcl” script that needs the open the dataset stored in “output_X.wlf” in order to perform the comparison with the outputs of the current simulation. If the comparison succeeds, then the two versions of the core are equivalent. If it fails, the benchmark cannot go on, and the design needs to be reviewed.

Once the validation step has reported all successful results, the benchmark can proceed to the next step.

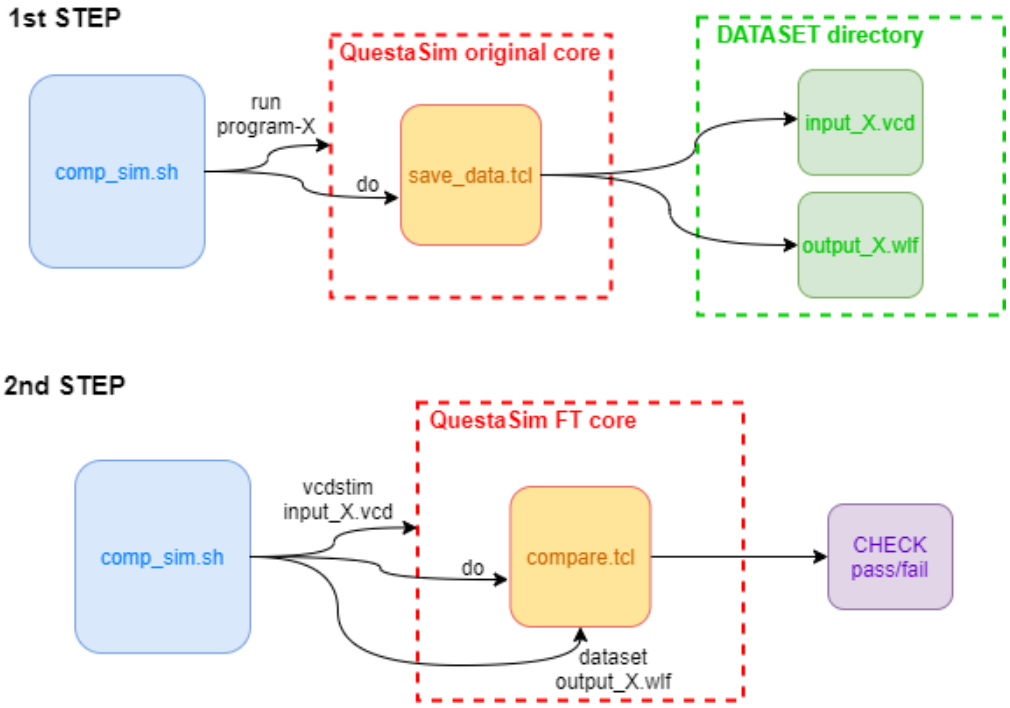

FIGURE 7.1: Validation steps in the development environment

7.2.3 Fault injection system

The simulation-based approach for the benchmark, but actually this is valid for any kind of approach, typically requires some useful components, such as fault injector, fault library, workload generator, workload library, controller, monitor, data analyzer, and so on [52]. Each component has a specific role in the benchmark process, and a brief description about its role is reported below for the components used in this context:

- fault injector: it is responsible to inject the fault into a specific signal of the target system while is running the workload;
- fault library: it stores the information related to the fault injections;
- workload generator: it generates the input stimuli for the simulation to run;
- workload library: it contains the data for the workload generator;
- controller: it controls the whole benchmark process;
- monitor: it is responsible to control when a fault becomes error and keep track of it;
- data analyzer: it elaborates the results deriving from all the simulations of the benchmark and gives the final level of fault tolerance reached by the design evaluated.

In the development environment, these components are implemented by means of Bash scripts, TCL scripts and Makefile files, but actually they are not separated entities as the previous list shows. In fact, some scripts implement several functions, eventually including more components. For example, the TCL script that QuestaSim

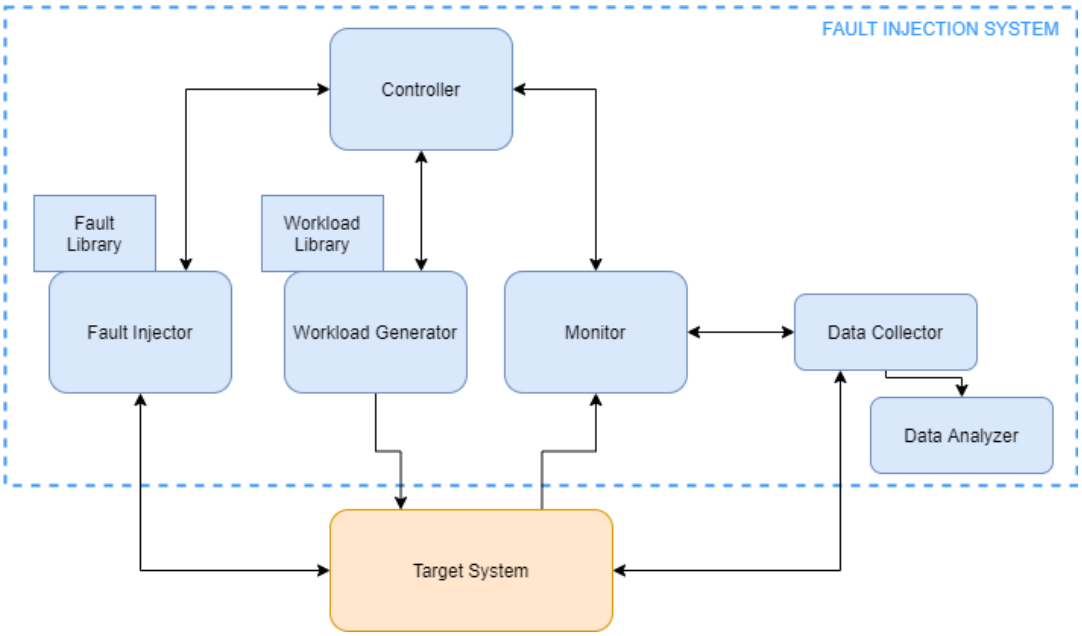

FIGURE 7.2: Fault injection system scheme [59]

runs during the simulation works as fault injector and monitor.

Before to further explain the mechanism of this step, it is important to focus on some key points that make this analysis clear and grounded. Actually, the most important ones are three. The first point (probably the most important one) regards the accuracy of the results. The second one concerns the method used to inject the fault, including all the requirements it respects. The third one is the way the data are analysed and elaborated.

Accuracy

The accuracy of the results strictly depends on the total number of simulations performed. More simulations means higher accuracy, but also more time required. The question is now how to choose the correct number of simulations to carry out to obtain a certain accuracy, or vice versa, which is the accuracy if the benchmark performs a certain amount of simulations. To solve this question, a statistical approach has been adopted by exploiting the formula derived by R. Leveugle et al. [60].

$$n_{cycle} = \frac{N}{1 + e^2 \cdot \frac{N-1}{t^2 \cdot p \cdot (1-p)}}$$

"N" is the initial population size given by multiplying the total number of signals where the fault can be injected by the total number of clock cycle required to terminate the execution of the program. "e" is the margin of error on the result. "t" is the cut-off point corresponding to the confidence level that is the probability that the exact value is actually within the error interval. "p" basically corresponds to an estimate of the true value being searched (in this case the percentage of faults resulting in an error) and since it is a priori unknown (but between 0 and 1) a conservative approach is to use the value that will maximize the sample size, that is obtained when "p" is equal to 0.5. "n_{cycle}" is the total number of simulations to perform to obtain

the accuracy defined with the previous parameters.
Analysing the formula, it results that what heavily influences "n_{cycle}" is the parameter "e", that makes the result rapidly increased if it is reduced below the value 0.03. "t" has a much lower impact on the formula, while the remaining parameters poorly affect the result.

For the scope of the thesis, the following values and methods have been used for those parameters:

- "N" is evaluated considering all the bits in the memory components inside the ID stage and considering the total number of clock cycle strictly dependent on the program to execute;
- "e" is set to 0.05, that means that the actual result of the benchmark will be the interval $[FT_{\%} - 5\%\,;\, FT_{\%} + 5\%]$;
- "t" is set to 2.5758, that corresponds to a confidence level of 99%;
- "p" is set to 0.5, as the study suggests.

By using these parameters, the number of simulations to perform is proven to be equal for all the benchmark programs, and it results $n_{cycle} = 663$. This is a reasonable value considering the characteristics of the hardware available to perform the benchmark and the time available. All the benchmarks for the soft errors will be carried out with this accuracy.
This calculation is implemented in the TCL script named "cycles_cov.tcl".

Fault injection

As stated previously, the fault injection mechanism relies on the use of built-in QuestaSim commands within TCL scripts. The fault injection should fulfill some criteria useful to clarify the meaning of the results and necessary to support the statistical approach above mentioned. Basically, they can be summarised explaining the simulation steps [61] that the implemented method follows. The steps can be divided into seven.

1) Choose a random flip-flop where to inject the fault (a SEU is reproduced). The bit is selected from a list containing all the signals that represent the content of the registers.
2) Choose a random time instant when to inject the fault. Any time instant can be selected except for the first 6 clock cycles necessary for the system to reset.
3) Run the simulation until the chosen time instant for the fault injection.
4) Inject the fault by forcing the bit flip in the selected bit. This task is accomplished by means of the "force" command of QuestaSim with the option "-deposit" to emulate the transient faults (soft errors) or "-freeze" for the permanent faults (hard errors).
5) Run the simulation for the next 10 clock cycles and compare the outputs. This is because it is highly likely that the fault becomes an error during the early clock cycles when it has been injected. If at least one error is occurred, the simulation is stopped.

6) If the first comparison does not reveal any error, the run and compare phase is repeated ten times, so dividing the rest of the simulation into ten blocks. If the comparison of a block shows at least one error, the simulation is stopped and so the other comparisons are not executed.

7) Collect the results into a file for later analysis.

An important property not listed above is that the fault cannot be injected more than once into the same bit at the same time instant. This fault injection mechanism is implemented in the TCL script named "fault_injection.tcl", and actually it is defined inside a loop where the number of cycles to perform is equal to the number of simulations required to obtain a certain accuracy.

Analysis

Finally, the last important point is the analysis of the results post simulations. The most interesting and relevant data is about the level of fault tolerance reached by the device under test. This is simply obtained by dividing the number of correct simulations by the number of total simulations and then multiplying by 100 to get the result as a percentage.

$$FT_{\%} = \frac{n_{cycles} - n_{errors}}{n_{cycles}} \cdot 100$$

As long as the fault tolerance level, another interesting analysis may be carried out on how the fault is actually perceived by the system. Generally, a classification of five categories [61] is done about the the fault effects:

1) No effects: if the program simulation terminates correctly, outputs and registers match the golden simulation. In this case, the fault may be overwritten or detected and corrected;

2) Latent: if the program simulation terminates correctly, but the content of the registers differs from the golden simulation;

3) Wrong result: if the program simulation terminates with wrong results;

4) Timed out: if the program simulation does not terminate within a certain time interval longer the the normal execution time;

5) Exception: if the system detects an unexpected event, aborting the execution.

This classification has not been implemented in the TCL script, although it is possible to state that any simulation has reported an abnormal behaviour, like "timed out" or "exception". The only category not evaluable is the "latent" one, so it would be collapsed into the "no effects" one.

Finally, another interesting feature to evaluate concerns the number of faults detected and corrected, detected but not corrected, and undetected. Even though the design provides the possibility to easily check these features, this classification has not been taken simply because of lack of time. In fact, the ID stage contains the "errors_vector_o" signal which collects the errors information coming from the different fault tolerant components instantiated inside the stage. This analysis can be

carried out by monitoring this signal cycle by cycle during the simulation.

However, additional information are saved for each fault injection performed, so that it is possible to show some stats about which signals are more sensitive to faults.

The key points explained above are the pillars of the fault injection system and of the entire benchmark process in general. By using these functions, it is possible to create a basic benchmark routine for the system. Actually, before to proceed with the final explanation, it is important to remember that the benchmark relies on the simulation-based approach, that is very time-consuming. For this reason, some smart ideas have been implemented to speed-up the simulations and consider an higher coverage for the benchmark.

The first optimisation done is the fifth point of the list in the "fault injection" paragraph. It compares the outputs of the simulation at the early clock cycles after the time instant when the fault has been injected. Being very highly likely that the fault becomes an error during those cycles, no time is wasted in case that the error has been detected since the rest of the simulation will not run. In Figure 7.3, the organisation of the files and how they interact along with the simulator is shown.

The second optimisation is probably the most efficient one. It relies on the fact that it is useless to simulate the entire core if the fault tolerant design to test regards only a single stage of the core (in this case the ID stage). So, a simple mechanism has been implemented so that all the simulations of the benchmark actually take into account only the ID stage. The mechanism is quite equal to the one used for the validation. While in the validation the inputs and the outputs of the core were saved, here the inputs and the outputs of the ID stage are saved. Moreover, since the fault tolerant version has been declared equivalent to the unprotected one, the inputs and the outputs are taken directly from the simulation of the fault tolerant version so that the interfaces of the stage already match. If they were taken from the original version, some ad-hoc functionalities needed to be implemented to accomplish the cases when the fault tolerant version has the triplicated pipeline enabled (and so the input interfaces differ).

Actually, one more optimisation could be implemented that would have lead to an even higher coverage with zero time consumption. That is the profiling of the hardware and the program [62]. This technique is crucial to recognize the parts of the system that benefit or suffer from a specific change (the fault injected). This helps to know a priori which components are nonsensitive to the faults injected because for example they are not used during the execution of the program. Identifying these cases allows to reach an higher coverage, and so an higher accuracy achievable.
A roughly and basic implementation of this technique may be the analysis of the permanent faults applied from the very first clock cycle to each single flip-flop inside the stage. In this way, the cases where the simulation terminates correctly demonstrate that those flip-flops are not used during the execution of the program.

At this point, the playground is clear and the key points are known, so the scheme emulating the mechanism of the entire benchmark process can be easily understood. In Figure 7.3, the scheme subdivided into four steps is reported.

The four steps are: generation of inputs and outputs for the benchmark program, calculation of the number of simulations to reach a certain accuracy, repeated simulations with fault injection, analysis of the results. Before these steps, actually some parameters are declared at the command entering stage. The stage, the benchmark program, and the accuracy are defined. Then, the process starts. First of all, the input and output files are generated by simulating the FT core running any benchmark program. The name of the files contains info about the program and the stage simulated. Then, the entire FT core running the program is simulated to find the number of flip-flops instantiated inside the stage and the total execution time (in terms of clock cycles). The "cycles_cov.tcl" script does the task and calculate the number of simulations (or cycles) to carry out. Once "N_cycle" is calculated and stored, the most time-consuming activity of the process can starts. The simulations of the stage are performed again and again until "N_cycle" times. Meanwhile, information about the simulations are saved into files, that later are processed to produce the stats and the fault tolerance level reached by the stage.

As it is easy to notice, the "comp_sim.sh" script is always present in each step. It plays the role of controller of the entire benchmark process, ensuring that each step performs correctly. Moreover, it is responsible also to analyse the data at the end of the process. The "fault_injection.tcl" script represents both the fault injector and the monitor. The "signals.txt" files store all the information about the faults injected, so it is the fault library. QuestaSim itself works as workload generator. The directory where all the benchmark programs taken into account is the workload library.

Actually, the organisation is a bit more complicated, but this represents a good simplification of the structure. Just to mention a main practice difference, the first step is in turn divided into two sub-steps, one to save input signals, one for output ones. The reason of this further division is the fact that the input stimuli need to be save into a vcd file by means of the command "vcd dumpports -vcdstim", while the output (and all in general) signals need to be stored into a wlf file by means of the command "dataset save" and with the GUI activated, otherwise it is not stored.

7.3 Results

At this point, several simulations have been carried out by means of the development environment implemented. In total, three kinds of benchmark have been performed:

1) complete benchmark (all the workload tested) of the fully protected version of the ID stage against soft errors (FT=15);

2) single benchmark (only the Dhrystone program) of all the versions of the ID stage (FT from 0 to 15);

3) single benchmark (only the Dhrystone program) of the fully protected version of the ID stage against hard errors.

For the first two categories the benchmarks have been carried out setting the accuracy to the confidence level of 99% and the error of 5%. While for the third category the coverage is calculated differently.

Obviously, these benchmarks need to be compared against the reference architecture in order to see how much the fault tolerance capability of the stage has been improved. So, the complete benchmark has been carried out also for the unprotected

stage, using the same level of confidence and error as the first two categories.

In Table 7.2, the results of the complete benchmarks for the unprotected (FT=0) and fully protected (FT=15) versions of the ID stage are shown.

Program	FT=0		FT=15	
	Errors	FT level	Errors	FT level
coremark_1	126	80.99%	0	100.00%
counters	125	81.14%	0	100.00%
csr_instructions	89	86.57%	0	100.00%
cv32e40p_csr_access_test	123	81.44%	0	100.00%
dhrystone	116	82.50%	0	100.00%
fibonacci	123	81.44%	0	100.00%
generic_exception_test	99	85.06%	0	100.00%
hello_world	88	86.72%	0	100.00%
illegal	98	85.21%	0	100.00%
interrupt_bootstrap	105	84.16%	0	100.00%
interrupt_test	107	83.86%	0	100.00%
misalign	106	84.01%	0	100.00%
modeled_csr_por	91	86.27%	0	100.00%
perf_counters_instructions	100	84.91%	0	100.00%
requested_csr_por	106	84.01%	0	100.00%
riscv_arithmetic_basic_test_0	107	83.86%	0	100.00%
riscv_arithmetic_basic_test_1	123	81.44%	0	100.00%
riscv_ebreak_test_0	91	86.27%	0	100.00%

TABLE 7.2: Benchmark results on 663 simulations of the architectures FT=0 and FT=15 (confidence level=99%, error=5%)

The unprotected version shows a fault tolerance level that ranges from 80.99% to 86.72%. The associated error intervals intersect each others, so it is very likely that the exact value of fault tolerance level will be in the interval between 80.99% and 86.72% (actually the real interval will be a bit smaller because of the intersection of all the error intervals). However, the average level of fault tolerance of the unprotected version is 106.83%.
The fully protected version instead shows a fault tolerance level of the 100% for all the benchmarks. Taking into account the error margin, the exact fault tolerance level will be between 95% and 100% with an high probability. Such an high value for the FT level is plausible because only the SEUs have been taken into account, and all the

sequential components in the stage have been protected by design.

In Table 7.3, the results of the benchmarks of all the FT configurations of the stage are reported. Only the Dhrystone program has been simulated for this purpose. For a better understanding the Table 6.7 is extremely important, that shows which FT components are enabled according to the FT code.

FT CODE	Errors	FT level
0	116	82.50%
1	93	85.97%
2	107	83.86%
3	100	84.92%
4	80	87.93%
5	94	85.82%
6	80	87.93%
7	82	87.63%
8	18	97.29%
9	11	98.34%
10	17	97.44%
11	10	98.49%
12	0	100%
13	0	100%
14	0	100%
15	0	100%

TABLE 7.3: Benchmark results on 663 simulations of all the architectures FT from 0 to 15 running the Dhrystone program (confidence level=99%, error=5%

From this table, it is possible to notice that the FT decoder and the FT controller actually do not impact on the fault tolerance capability of the stage. In fact, the FT levels are more or less equivalent in both the cases where their FT version is enabled and when not. This strange behaviour actually is explained by the fact that the faults are injected only into the flip-flops, and the decoder and the controller make a very low use of them. Only the controller uses a register 32 bits wide, while the decoder is purely combinational.
The most interesting point is that the FT level is evidently increased from the FT CODE 8 on (increment of about 10%). This is explained by the activation of the FT version of the register file, that is one of the most extended and the most critical component for the radiation effects. Its activation allows to reach FT level higher than 97%, even when the other components remain unprotected. Moreover, the FT level 100% is reached when the FT version of the pipeline is activated as well.

This result is perfectly inline with the prediction due to method used to benchmark the system. In fact, the fault injection has been carried out only on the memory components, and the register file and the pipeline are uniquely composed by flip-flops.

Finally, the single benchmark of the FT version of the stage against the hard errors has been carried out. For this benchmark it is important to clarify some points, since they are a bit different with respect to the other benchmarks. In fact, the fault has not been injected at a random time instant, but it has been injected always at the time instant 200ns in order to cover all the duration of the simulation and better evaluate its impact on the design. Moreover, 1842 simulations have been run, which would correspond to a confidence level of 99% and error of 3%. Actually, the time instant is not chosen randomly anymore, and this may lead to a biased result. By the way, since the fault injected covers approximately the entire duration of the simulation and the signal where to inject the fault is selected randomly, it is still possible to adopt the same statistical method previously used to evaluate the accuracy even if it does not fit perfectly. Injecting the fault at 200ns forces the benchmark to evaluate all the worst case scenarios. In fact, if a fault injected into a signal at a specific time instant leads to an error, injecting the fault at the beginning of the simulation ensures to cover that case and also others critical. This means that the result may be considered a worst case one.
Then, another important aspect regards the design parameter required for the FT version of the register file against the permanent faults which sets the threshold for the detection of a permanent fault. For this benchmark, this parameter has been set to 4.
The result of this final benchmark is reported in Table 7.4.

FT CODE	Errors	FT level
31	0	100%

TABLE 7.4: Benchmark results on 1842 simulations of the architecture FT=31 (or FT=23) running the Dhrystone program (confidence level=99%, error=3%

The analysis of the results lead to state that for the FT version of the ID stage to mitigate permanent faults the fault tolerance level ranges from 97% to 100% with a confidence level of 99%.
Obviously, this single benchmark is not so exhaustive, because it would be opportune to run other benchmarks and compare the results, or run a benchmark which uses a program with a very long execution time, like the CoreMark program. Running a program with a long execution time allows to have much more probability that the permanent faults are detected by the appropriate components.

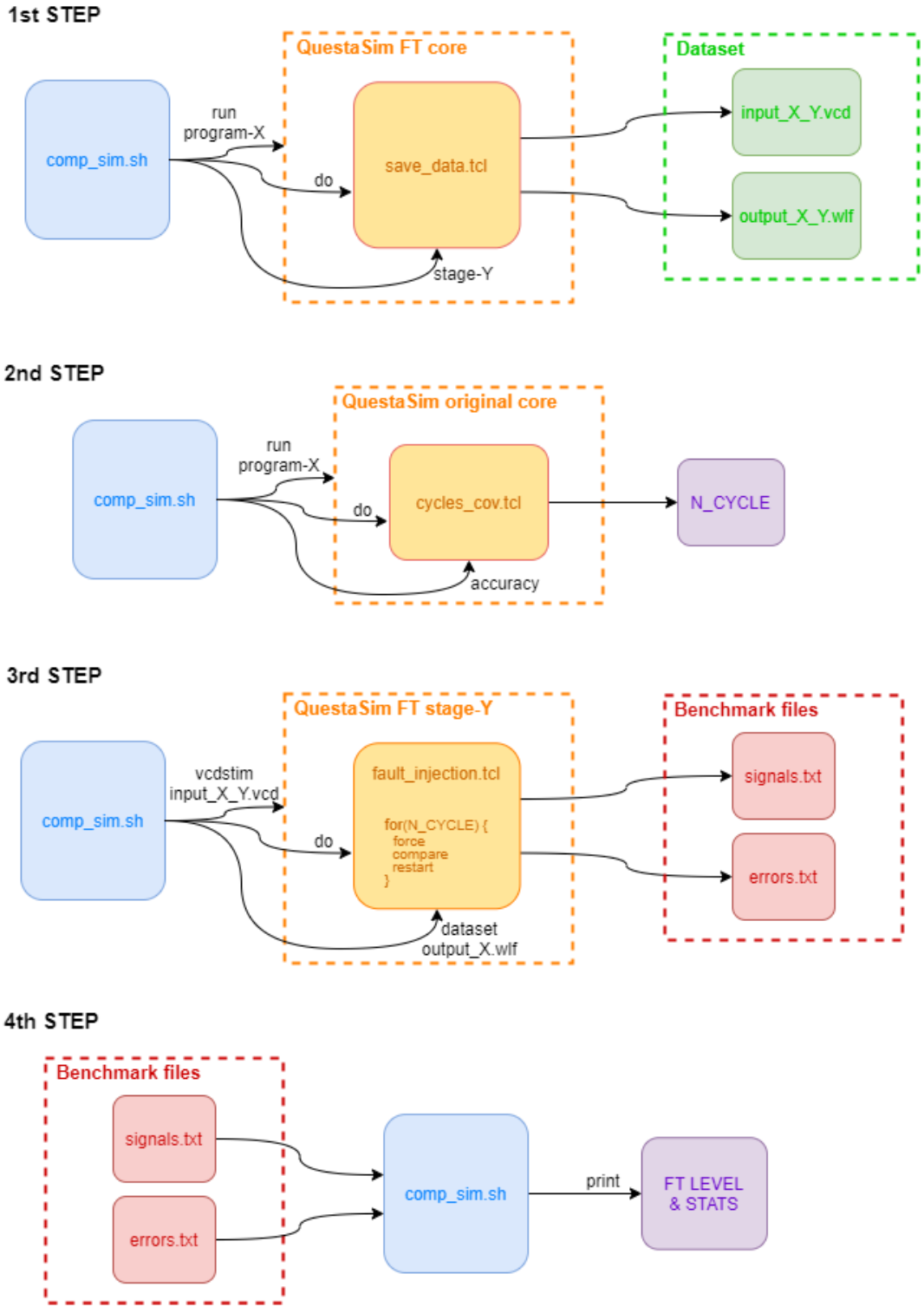

FIGURE 7.3: Benchmark steps in the development environment

Chapter 8

Conclusions

The work carried out in this thesis has given a general overview on the fault tolerance context, presenting briefly the standard IEC 61508 which set the rules, the phenomena that are involved in the real world, and the most common techniques used to mitigate these errors. Then, an introduction on the RISC-V specifications and on the CV32E40P's characteristics is presented. Finally, the entire design process for the fault tolerant version of the core, the environment developed to test the architecture, and the benchmark method are disclosed.

The fault tolerant design of the ID stage is the main point of the thesis, so most of the attention has been posed on it. At the end of the design a completely configurable system is produced, which allows to select the fault tolerant components to enable according to the desired level of fault tolerance set by requirements. Then, the design needs to be tested by means of ad-hoc benchmarks that allow to show the fault tolerance level reached by the system.
Necessarily, the design and the benchmark processes have required the development of an environment capable to facilitate all the operations to execute during these phases. Such an environment has made automatic some important steps of both the design and the benchmark processes, reducing so the probability of making a mistake by doing those operations manually.

The benchmark has revealed that the maximum fault tolerance level reachable is of 100% (with an error of 5% and a confidence level of 99%) for the fully protected version against the soft errors. This result was reported for all the benchmark programs used. The same result has been obtained also for the version against the hard errors, even if it was tested only with the Dhrystone program.
These values are perfectly inline with the fault tolerance levels reached by the other architectures of the state-of-the-art. In fact, for example, the FT core developed by D. A. Santos et al. [44] reaches a FT level of 100% calculated with a benchmark on 100 simulations with the CoreMark and considering only SEUs. The architecture developed by L. A. Aranda et al. [48] shows as well a fault tolerance level of 99.95% on a benchmark carried out on 27000 simulations running the Dhrystone program.
Anyway, it is necessary to remember that in this thesis work only the ID stage is evaluated. For this reason, the comparisons are to be intended as a guide to know if the results can be considered trustworthy. For a 1-to-1 comparison, the fault tolerant version of the ID stage must be integrated with the other fault tolerant versions of the remaining stages developed by the rest of the team. As explained in the introduction, this thesis work is part of a wider project which aims to make the entire CV32E40P core fully fault tolerant.

The results obtained by means of the benchmark make it possible to classify the new fault tolerant architecture according to the standard IEC 61508. Taging into account the architecture fully protected against the soft errors, the fault tolerance level

reached is between the 97% and the 100%, with a confidence level of 99%. This implies that the Safe Failure Fraction (SFF) of the system (the ID stage) is included in the same interval. Considering the ID stage as a system of Type B (a complex system whose failure behavior is not fully known) and considering the Hardware Fault Tolerance (HFT) of 1 (this means that 2 faults may lead to the lowering of the SFF), the target system can get a Safety Integrity Level (SIL) equal to SIL 3 if the SFF is considered up to 99%, or to SIL 4 if the SFF is considered from 99% to 100%.
Analysing the worst case, it is possible to state that the fault tolerant ID stage reaches a SIL 3. A further improvement of this level may be achieved if a more accurate benchmark is carried out, or if the level of redundancy is increased. The last way would lead to an extremely high overhead in terms of hardware resources and performance, so it would be advisable to accomplish a more detailed benchmark with a reduced error margin.
Obviously, other safety tests need to be executed to guarantee the SIL reached by the system.

All kinds of results obtained from the synthesis and from the benchmark are summarised in the following table (Table 8.1). This is useful to give a general overview of the characteristics of each hardware configuration of the ID stage designed.

FT CODE	Area [kGE]	Area var. [%]	Delay [ns]	Delay var. [%]	FT level
00000	17.17	100.00%	10.49	100.00%	82.50%
00001	19.92	116.00%	10.49	100.00%	85.97%
00010	21.35	124.34%	11.20	106.76%	83.86%
00011	24.10	140.33%	12.13	115.63%	84.92%
00100	19.21	111.89%	13.00	123.92%	87.93%
00101	21.96	127.89%	13.00	123.92%	85.82%
00110	23.40	136.24%	13.84	131.93%	87.93%
00111	26.14	152.24%	14.89	141.94%	87.63%
01000	20.58	119.84%	12.35	117.73%	97.29%
01001	23.33	135.84%	12.35	117.73%	98.34%
01010	24.76	144.17%	12.32	117.44%	97.44%
01011	27.51	160.16%	12.32	117.44%	98.49%
01100	22.62	131.74%	14.73	140.41%	100%
01101	25.37	147.74%	14.73	140.41%	100%
01110	26.80	156.08%	14.78	140.89%	100%
01111	29.55	172.07%	14.82	141.27%	100%
1X111	45.19	263.11%	18.38	175.21%	100%

TABLE 8.1: Area, Area variation (with respect to FT CODE 0), Max delay, Max delay variation (with respect to FT CODE 0), and FT level, reported for each configuration of the system

Appendix A

Hsiao ECC design code

A.1 Matlab code

In this section of the Appendix A, the highlights of the Matlab code for the Hsiao ECC are listed. In particular, three parts are shown: the definition of all the variables required, the encoder algorithm, and the decoder algorithm.

This is the part related to the definition of the variables to implement the algorithm, such as the number of parity bits, the H-matrix, and other supporting variables.

```
%% VARIABLES USED FOR THE HSIAO (OR HAMMING) ALGORITHM
% k : WIDTH OF DATA
k = 32;
% r_tmp : TEMPORARY NUM OF PARITY BITS
r_tmp = ceil(log2(k));
% r : TOT NUM OF PARITY BITS
r = r_tmp + floor( (k + r_tmp) / 2^(r_tmp) ) +1;

% INITIALIZATION OF H-MATRIX AND PARTIAL DEFINITION
h_matrix = zeros(r,k+r);
h_matrix(1:r,1:r) = eye(r);
num_ones_per_row = 3*k/r;

r_fact = 1:r;
el_fact = 1:3;
r_el_fact = 1:r-3;
tot_comb = prod(r_fact)/(prod(el_fact)*prod(r_el_fact));
combs = zeros(3,tot_comb);


% GENERATION OF ALL THE ODD COMBINATIONS OF '1' (1, 3, 5) AS
    REQUIRED BY HSIAO'S ALGORITHM
% UP TO 5 '1' BECAUSE IT IS ENOUGH FOR ENCODING 32-BITS DATA
n_comb = 1;
for el_1 = 1:r
    for el_2 = el_1+1:r
        for el_3 = el_2+1:r
            combs(:,n_comb) = [el_1, el_2, el_3];
```

```
28              n_comb = n_comb +1;
29          end
30      end
31  end
32
33  % DEFINE H MATRIX
34  rep_el = zeros(r,1);
35  n_rep_min=0;
36  for j = r+1:k+r
37      trovato = 0;
38      n_rep_min = min(rep_el);
39      for i = 1:n_comb-1
40          el_1 = combs(1,i);
41          el_2 = combs(2,i);
42          el_3 = combs(3,i);
43          if(min(combs(:,i))~=0 && rep_el(el_1)==n_rep_min && rep_el(
    el_2)==n_rep_min && rep_el(el_3)==n_rep_min)
44              rep_el(el_1) = rep_el(el_1) +1;
45              rep_el(el_2) = rep_el(el_2) +1;
46              rep_el(el_3) = rep_el(el_3) +1;
47              h_matrix(el_1,j) = 1;
48              h_matrix(el_2,j) = 1;
49              h_matrix(el_3,j) = 1;
50              combs(:,i) = zeros(3,1);
51              trovato = 1;
52              break
53          end
54      end
55      if trovato == 0
56          for i = 1:n_comb-1
57              el_1 = combs(1,i);
58              el_2 = combs(2,i);
59              el_3 = combs(3,i);
60              if(min(combs(:,i))~=0 && rep_el(el_1)==n_rep_min+1 &&
    rep_el(el_2)==n_rep_min && rep_el(el_3)==n_rep_min)
61                  rep_el(el_1) = rep_el(el_1) +1;
62                  rep_el(el_2) = rep_el(el_2) +1;
63                  rep_el(el_3) = rep_el(el_3) +1;
64                  h_matrix(el_1,j) = 1;
65                  h_matrix(el_2,j) = 1;
66                  h_matrix(el_3,j) = 1;
67                  combs(:,i) = zeros(3,1);
68                  trovato = 1;
69                  break
70              elseif(min(combs(:,i))~=0 && rep_el(el_1)==n_rep_min &&
     rep_el(el_2)==n_rep_min+1 && rep_el(el_3)==n_rep_min)
71                  rep_el(el_1) = rep_el(el_1) +1;
72                  rep_el(el_2) = rep_el(el_2) +1;
```

```
73                  rep_el(el_3) = rep_el(el_3) +1;
74                  h_matrix(el_1,j) = 1;
75                  h_matrix(el_2,j) = 1;
76                  h_matrix(el_3,j) = 1;
77                  combs(:,i) = zeros(3,1);
78                  trovato = 1;
79                  break
80              elseif(min(combs(:,i))~=0 && rep_el(el_1)==n_rep_min &&
    rep_el(el_2)==n_rep_min && rep_el(el_3)==n_rep_min+1)
81                  rep_el(el_1) = rep_el(el_1) +1;
82                  rep_el(el_2) = rep_el(el_2) +1;
83                  rep_el(el_3) = rep_el(el_3) +1;
84                  h_matrix(el_1,j) = 1;
85                  h_matrix(el_2,j) = 1;
86                  h_matrix(el_3,j) = 1;
87                  combs(:,i) = zeros(3,1);
88                  trovato = 1;
89                  break
90              end
91          end
92      end
93      if trovato == 0
94          for i = 1:n_comb-1
95              el_1 = combs(1,i);
96              el_2 = combs(2,i);
97              el_3 = combs(3,i);
98              if(min(combs(:,i))~=0 && rep_el(el_1)==n_rep_min+1 &&
    rep_el(el_2)==n_rep_min+1 && rep_el(el_3)==n_rep_min)
99                  rep_el(el_1) = rep_el(el_1) +1;
100                 rep_el(el_2) = rep_el(el_2) +1;
101                 rep_el(el_3) = rep_el(el_3) +1;
102                 h_matrix(el_1,j) = 1;
103                 h_matrix(el_2,j) = 1;
104                 h_matrix(el_3,j) = 1;
105                 combs(:,i) = zeros(3,1);
106                 trovato = 1;
107                 break
108             elseif(min(combs(:,i))~=0 && rep_el(el_1)==n_rep_min+1
    && rep_el(el_2)==n_rep_min && rep_el(el_3)==n_rep_min+1)
109                 rep_el(el_1) = rep_el(el_1) +1;
110                 rep_el(el_2) = rep_el(el_2) +1;
111                 rep_el(el_3) = rep_el(el_3) +1;
112                 h_matrix(el_1,j) = 1;
113                 h_matrix(el_2,j) = 1;
114                 h_matrix(el_3,j) = 1;
115                 combs(:,i) = zeros(3,1);
116                 trovato = 1;
117                 break
```

```
118             elseif(min(combs(:,i))~=0 && rep_el(el_1)==n_rep_min &&
    rep_el(el_2)==n_rep_min+1 && rep_el(el_3)==n_rep_min+1)
119                 rep_el(el_1) = rep_el(el_1) +1;
120                 rep_el(el_2) = rep_el(el_2) +1;
121                 rep_el(el_3) = rep_el(el_3) +1;
122                 h_matrix(el_1,j) = 1;
123                 h_matrix(el_2,j) = 1;
124                 h_matrix(el_3,j) = 1;
125                 combs(:,i) = zeros(3,1);
126                 trovato = 1;
127                 break
128             end
129         end
130     end
131     if trovato == 0
132         for i = 1:n_comb-1
133             el_1 = combs(1,i);
134             el_2 = combs(2,i);
135             el_3 = combs(3,i);
136             if(min(combs(:,i))~=0 && rep_el(el_1)==n_rep_min+1 &&
    rep_el(el_2)==n_rep_min+1 && rep_el(el_3)==n_rep_min+1)
137                 rep_el(el_1) = rep_el(el_1) +1;
138                 rep_el(el_2) = rep_el(el_2) +1;
139                 rep_el(el_3) = rep_el(el_3) +1;
140                 h_matrix(el_1,j) = 1;
141                 h_matrix(el_2,j) = 1;
142                 h_matrix(el_3,j) = 1;
143                 combs(:,i) = zeros(3,1);
144                 trovato = 1;
145                 break
146             end
147         end
148     end
149 end
```

Here, the encoding algorithm is reported, keeping in mind that the variables related to the data are flipped because of the mirrored relation between data in hardware and data in Matlab.

```
1 %% ALGORITHM
2 % The variables "_flipped" have the bits ordered like in
3 % hardware (MSB to the left, LSB to the right), while the other
4 % variables have the bits mirrored w.r.t. hw because in
5 % Matlab the index starts from the first left element of the array
6
7 %%%% ENCODER
8 % DATA TO BE ENCODED (value taken for example)
9 data_in_enc_hw = '10110110011101100101110100011001'; %input data of
     the encoder HW
```

```
data_in_enc = fliplr(data_in_enc_hw); %input data of the encoder in
    Matlab
data_encoded = ''; %data encoded

k_data = strlength(data_in_enc_hw);

% LOOP TO INITIALIZE THE NEW VARIABLE WHICH CONTAINS DATA & PARITY
    BITS
if (k == k_data)
    data_encoded(1:r)=string(zeros(1,r)); %parity bits initialized
    to 0
    data_encoded(r+1:k+r) = data_in_enc(:);
end

% LOOP TO SET THE PARITY BITS
for i = 1:r
    parity=0;
    for j = 1+r:k+r
        if h_matrix(i,j) == 1
            parity = xor(parity,data_encoded(j)-'0');
        end
    end
    data_encoded(i) = num2str(parity);
end

data_encoded_hw = fliplr(data_encoded); %final data encoded in hw
```

```
%% ALGORITHM
% The variables "_flipped" have the bits ordered like in
% hardware (MSB to the left, LSB to the right), while the other
% variables have the bits mirrored w.r.t. hw because in
% Matlab the index starts from the first left element of the array

%%%% DECODER
% It's the inverse process of the encoder
% DATA TO BE DECODED (value taken from the prev example)
data_in_dec_hw = '10110110011101100101110100011001111011 1';  %input
    data of the decoder HW
data_in_dec = fliplr(data_in_dec_hw);  %input data of the decoder
    in Matlab

error_check = zeros(r,1); %array which contains the errors related
    to the parity (len=r)
error_position=0; %variable which states the index where the error
    occurred

% LOOP TO CHECK THE ERROR POSITION
for i = 1:r
```

```
18      parity=data_in_dec(i)-'0';
19      for j = 1+r:k+r
20          if h_matrix(i,j) == 1
21              parity = xor(parity, data_in_dec(j) - '0');
22          end
23      end
24      error_check(i) = parity; %this variable says where the error
        has occurred
25  end
26  for i = 1:k+r
27      if h_matrix(:,i) == error_check
28          error_position = i;
29      end
30  end
31
32  parity_calc = 0;
33  for i = 1:r
34      parity_calc = xor(parity_calc, error_check(i));
35  end
36
37  parity_check = parity_calc; %variable to state error on the overall
        parity
38  ecc_check = 0; %variable to state error on the data+code word
39
40  data_corrected = data_in_dec;
41
42  % CHECK IF THE ERROR HAS OCCURRED, AND CORRECT IT
43  if (error_check(:) == zeros(r,1))
44      ecc_check = 0;
45  elseif (error_position > 0)
46      ecc_check = 1;
47      data_corrected(error_position) = num2str(not(data_in_dec(
        error_position)-'0'));
48  else
49      ecc_check = 1;
50  end
51
52  NOERROR = 0; % everything is OK
53  SEC = 0; % single error detected and corrected
54  DED = 0; % double error detected and not corrected
55  PE = 0; % parity error detected
56  if (parity_check == 0 && ecc_check == 0)
57      NOERROR = 1;
58  elseif (parity_check == 1 && ecc_check == 1)
59      SEC = 1;
60  elseif (parity_check == 0 && ecc_check == 1)
61      DED = 1;
62  elseif (parity_check == 1 && ecc_check == 0)
```

```
63      PE = 1;
64  end
65
66  data_corrected_hw = fliplr(data_corrected);
67
68  NOERROR
69  SEC
70  DED
71  PE
```

Bibliography

[1] G. E. Moore. "Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]" In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 36–37.

[2] E. Normand. "Single-event effects in avionics". In: *IEEE Transactions on Nuclear Science* 11.3 (1996), pp. 461–474.

[3] IEC. "Functional safety: Essential to overall safety". In: *IEC Basecamp website* (Oct. 2019). URL: https://basecamp.iec.ch/download/functional-safety-essential-to-overall-safety/.

[4] Contributors of Wikipedia. "IEC 61508". In: *Wikipedia, L'enciclopedia libera* (May 2020). URL: https://it.wikipedia.org/w/index.php?title=IEC_61508&oldid=112895181.

[5] IEC. "Functional Safety and IEC 61508". In: *IEC website* (Sept. 2020). URL: https://www.iec.ch/functionalsafety/.

[6] Exida. "IEC 61508 Overview Report". In: *Exida website* (Jan. 2006). URL: https://www.exida.com/articles/IEC-61508-Overview.pdf.

[7] IEC-CENELEC. "Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 2: Requirements for electrical/electronic/programmable electronic safety related systems". In: (May 2010). URL: https://webstore.iec.ch/publication/5516.

[8] Lodovico Ratti. "Ionizing Radiation Effects in Electronic Devices and Circuits". In: *INFN Laboratori Nazionali di Legnaro* (Apr. 2013). URL: https://agenda.infn.it/event/5622/contributions/59843/attachments/43107/51149/Ratti_radiation_effects_on_electronics.pdf.

[9] S. Buchner and D. McMorrow. "Overview of Single Event Effects". In: *Naval Research Laboratory* (2015).

[10] Taiki Uemura. "A Study on Soft Error Mitigation for Microprocessor in Bulk CMOS Technology". In: *Osaka University Knowledge Archive* (Jan. 2015). URL: https://doi.org/10.18910/52020.

[11] ESA Requirements and Standards Division. "Techniques for radiation effects mitigation in ASICs and FPGAs handbook". In: *ECSS website* (Sept. 2016).

[12] R. Velazco and F. J. Franco. "Single Event Effects on Digital Integrated Circuits: Origins and Mitigation Techniques". In: *2007 IEEE International Symposium on Industrial Electronics* (2007), pp. 3322–3327.

[13] Josep Balasch. "Introduction to Fault Attacks". In: *IACR Summer School 2015* (Oct. 2015).

[14] J. F. Ziegler and W. A. Lanford. "Effect of Cosmic Rays on Computer Memories". In: *Science* (Nov. 1979).

[15] T. C. May and M. H. Woods. "A New Physical Mechanism For Soft Errors In Dynamic Memories". In: *16th International Reliability Physics Symposium* (1978).

[16] Richard H. Maurer et al. "Harsh Environments: Space Radiation Environment, Effects, and Mitigation". In: *Johns Hopkins APL Technical Digest* 28.1 (2008).

[17] Tang D. et al. "Soft error reliability in advanced CMOS technologies—trends and challenges". In: *Sci China Tech Sci* 57.9 (Sept. 2014).

[18] Wikipedia contributors. "Lockstep (computing)". In: *Wikipedia, The Free Encyclopedia* (Nov. 2020). URL: https://en.wikipedia.org/w/index.php?title=Lockstep%5C_(computing)%5C&oldid=984167647.

[19] Wietse F. Heida. "Towards a fault tolerant RISC-V softcore". In: *MSc thesis at Delft University of Technology* (2016).

[20] Wikipedia contributors. "Error correction code". In: *Wikipedia, The Free Encyclopedia* (Nov. 2020). URL: https://en.wikipedia.org/w/index.php?title=Error%5C_correction%5C_code%5C&oldid=979250247.

[21] R. Hamming. "Error Detecting and Error Correcting Codes". In: *Bell System Technical Journal* 29.2 (Apr. 1950), pp. 147–160.

[22] Wikipedia contributors. "Hamming distance". In: *Wikipedia, The Free Encyclopedia* (Nov. 2020). URL: https://en.wikipedia.org/w/index.php?title=Hamming%5C_distance%5C&oldid=986463848.

[23] Wikipedia contributors. "Parity bit". In: *Wikipedia, The Free Encyclopedia* (Nov. 2020). URL: https://en.wikipedia.org/w/index.php?title=Parity_bit&oldid=990437820.

[24] G. Tshagharyan et al. "Experimental study on Hamming and Hsiao Codes in the Context of Embedded Applications". In: *2017 IEEE East-West Design Test Symposium (EWDTS)* (2017), pp. 1–4.

[25] Wikipedia contributors. "Cyclic redundancy check". In: *Wikipedia, The Free Encyclopedia* (Nov. 2020). URL: https://en.wikipedia.org/w/index.php?title=Cyclic_redundancy_check&oldid=991338850.

[26] Alrkiyan Saleh. "Cyclic Redundancy Check CRC". In: *Research Gate* (Jan. 2017). DOI: 10.13140/RG.2.2.30201.06248.

[27] RISCV. In: *RISCV website* (2020). URL: https://riscv.org.

[28] RISCV. "RISCV history". In: *RISCV website* (2020). URL: https://riscv.org/about/history/#:~:text=The%5C%20RISC%5C%2DV%5C%20Foundation%5C%20(www,of%5C%20the%5C%20RISC%5C%2DV%5C%20ISA.

[29] Wikipedia contributors. "RISCV history". In: *Wikipedia, The Free Encyclopedia* (Oct. 2020). URL: https://en.wikipedia.org/w/index.php?title=RISC-V&oldid=985473669.

[30] Andrew Waterman and Krste Asanovi. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified". In: *RISC-V Foundation* 1 (Dec. 2019).

[31] David A. Patterson and John L. Hennessy. "Computer Organization And Design: the hardware/software interface, RISC-V Edition". In: *Morgan Kaufmann* (2017).

[32] User David. "Registers - RISC-V". In: *WikiChip website* (Oct. 2018). URL: https://en.wikichip.org/w/index.php?title=risc-v/registers&oldid=77434.

[33] Andrew Waterman and Krste Asanovi. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Chapter: Calling Convention". In: *RISC-V Foundation* 1 (2019), pp. 83–86. URL: https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf.

[34] RISC-V Foundation. "RISC-V cores list". In: *RISC-V repository on GitHub* (2020). URL: https://github.com/riscv/riscv-cores-list.

[35] PULP-platform. "PULPino platform". In: *PULP-platform repository on GitHub* (2020). URL: https://github.com/pulp-platform/pulpino.

[36] PULP-platform. "PULPissimo platform". In: *PULP-platform repository on GitHub* (2020). URL: https://github.com/pulp-platform/pulpissimo.

[37] OpenHW Group. In: *OpenHW Group website* (2020). URL: https://www.openhwgroup.org/.

[38] M. Neri, L. Fiore, and E. Ribaldone. "AutomatePULPissimo". In: *RISKVFT repository on GitHub* (2020). URL: https://github.com/RISKVFT/AutomatePULPissimo.

[39] OpenHW Group. "CV32E40P". In: *OpenHW Group repository on GitHub* (2020). URL: https://github.com/openhwgroup/cv32e40p.

[40] A. Traber, M. Gautschi, and P. D. Schiavone. "RI5CY: User Manual". In: *RISKVFT repository on GitHub* (2020). URL: https://github.com/RISKVFT/AutomatePULPissimo.

[41] M. Gautschi et al. "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25 (2017).

[42] UMC. "UMC technologis: 55, 65 and 90 nm". In: *UMC website* (2020). URL: https://www.umc.com/en/Product/technologies/Detail/55_65_90nm.

[43] P. D. Schiavone et al. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications". In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)* 25 (2017).

[44] D. A. Santos et al. "A Low-Cost Fault-Tolerant RISC-V Processor for Space Systems". In: *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)* (2020).

[45] Luigi Blasi et al. "A RISC-V Fault-Tolerant Microcontroller Core Architecture Based on a Hardware Thread Full/Partial Protection and a Thread-Controlled Watch-Dog Timer". In: *Applications in Electronics Pervading Industry, Environment and Society* (Mar. 2020).

[46] Alexis Ramos et al. "An ALU Protection Methodology for Soft Processors on SRAM-Based FPGAs". In: *IEEE Transactions on Computers* (Sept. 2019).

[47] Cristiano Rodrigues et al. "Towards a Heterogeneous Fault-Tolerance Architecture based on Arm and RISC-V Processors". In: *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society* (2019).

[48] L.A. Aranda et al. "Analysis of the Critical Bits of a RISC-V Processor Implemented in an SRAM-Based FPGA for Space Applications". In: *Electronics* (Jan. 2020).

[49] LowRISC. "Rocket core overview". In: *LowRISC's website* (2020). URL: https://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core/.

[50] Hao Li et al. "Low cost design of microprocessor EDAC circuit". In: *Journal of Semiconductors* (Nov. 2015).

[51] "IEEE Standard for a 32-bit Microprocessor Architecture". In: *IEEE Std 1754-1994* (1995). DOI: 10.1109/IEEESTD.1995.79519.

[52] Haissam Ziade, Rafic Ayoubi, and R. Velazco. "A Survey on Fault Injection Techniques". In: *Int. Arab J. Inf. Technol.* 1 (Jan. 2004), pp. 171–186.

[53] J. R. Samson, W. Moreno, and F. Falquez. "A technique for automated validation of fault tolerant designs using laser fault injection (LFI)". In: *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)* (1998), pp. 162–167.

[54] Mike Thompson et al. "core-v-verif". In: *OpenHWGroup's repository on GitHub* (2020). URL: https://github.com/openhwgroup/core-v-verif.

[55] John K. Ousterhout et al. "Tcl: An Embeddable Command Language". In: *University of California, Berkeley, Computer Science Division* (1989).

[56] "CoreMark: an EEMBC benchmark". In: *EEMBC's website* (2020). URL: https://www.eembc.org/coremark/.

[57] Cristopher Celio et al. "riscv-coremark". In: *riscv-boom's repository on GitHub* (2019). URL: https://github.com/riscv-boom/riscv-coremark.

[58] ARM Ltd Richard York. "Benchmarking in context: Dhrystone". In: *ARM White Paper* (Mar. 2002).

[59] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. "Fault injection techniques and tools". In: *Computer* 30.4 (1997), pp. 75–82. DOI: 10.1109/2.585157.

[60] R. Leveugle et al. "Statistical fault injection: Quantified error and confidence". In: *2009 Design, Automation Test in Europe Conference Exhibition* (2009), pp. 502–506. DOI: 10.1109/DATE.2009.5090716.

[61] R. Travessini et al. "Processor core profiling for SEU effect analysis". In: *2018 IEEE 19th Latin-American Test Symposium (LATS)* (2018), pp. 1–6. DOI: 10.1109/LATW.2018.8347235.

[62] H. Hubert and B. Stabernack. "Profiling-Based Hardware/Software Co-Exploration for the Design of Video Coding Architectures". In: *IEEE Transactions on Circuits and Systems for Video Technology* 19.11 (2009), pp. 1680–1691. DOI: 10.1109/TCSVT.2009.2031522.