POLITECNICO DI TORINO

Master Degree in Electonic Engineering

Master Degree Thesis

Design of configurable Fault Folerant RISC-V Instruction Fetch and automatization through Travulog templates

Based on cv32e40p core of OpenHW Group organization

Supervisor
Prof. Stefano Di Carlo

Co-supervisors:
Prof. Alessandro Savino
Prof. Maurizio Martina
Prof. Guido Masera

Candidate:
Elia Ribaldone

Luglio 2021

To my family and Anna

"Failure is simply an opportunity to begin again,
this time more intelligently."

Henry Ford

Acknowledgements

I thank my supervisors Prof. Stefano Di Carlo, Prof. Alessandro Savino, Prof. Maurizio Martina and Prof. Guido Masera for their technical support and guidance. I would also like to thank my thesis teammates Luca Fiore and Marcello Neri for making the days and evenings spent working on our thesis so much enjoyable, I am very happy to have shared this experience and some previous projects with them. My thoughts also go to Christian Fabiano and Andrea Trufini from whom I learned a lot during courses, in particular I thank Andrea for all the days spent studying together and the crazy evening releases in Turin.
I want to thank my whole family, the best I could hope to have, I feel really lucky when I think of my parents and how they have believed in me during these years. A special thought goes to my two sisters, they have put up with me and supported me throughout my journey. I will always love them.
Finally I want to thank Anna, we have given each other so much in these five years, without her my life would have been very different, certainly sadder and less fun. In this thesis I have used the passion that I have in all the projects I do, but I got the strength and tenacity to complete this work from her.

Abstract

The miniaturization of the microelectronic components together with the use of integrated circuits in increasingly more application lead to an increasing use of Fault Tolerant (FT) systems. For this reason new techniques are required to automatize the transformation of systems from base to FT. This thesis investigates the automatic application of FT techniques inside the stage of a core written in SystemVerilog (SV). The aim of this project is to create a Toolchain able to apply different templates to an SV architecture in order to automatize the conversion of an architecture from base to Fault Tolerant. The templates are written in a new metalanguage called Travulog and they can be applied to an architecture using directly Python or HTravulog metalanguage inside the SystemVerilog.
The Toolchain is used to convert the Instruction Fetch stage of the RISC-V cv32e40p to Fault Tolerant. Anyway the Toolchain can be used for whatever SystemVerilog architecture with the great advantage to be open source and extendible to new uses.

Summary

A **FT (fault tolerant) system** continues to work properly even if some of the internal components are broken; this feature is necessary when a failure may cause damage to people, dangerous destruction, military upset or loss of data. FT systems are essential in aerospace, transport, medical and utility industries and they are usually composed by a power source, an hardware system and a software system, each of these parts are fault tolerant depending on application.
This work concerns the hardware system and in particular the chip architecture design. In this context the faults are **transient**, **intermittent** or **permanent** and they are generated by manufacturing defects, system degradation or particles strikes. **Manufacturing** defects generate permanent faults and they reduce the yield with an increase in the cost per piece since the affected chips are discarded during quality control process. **System degradation** produces permanent faults and it is a life-limiting phenomenon that brings the chip to wearout phase. Finally, **particles strikes** produce both transient or permanent faults. These problems rely on the application: system degradation generally depends on chip temperature, clock speed and the workload, otherwise particle strikes rely on sources of alpha particles or neutrons, which are generated by the cosmic rays or radioactive materials. For these reasons an ideal fault tolerant system should be protected against transient faults caused by particle strikes and it should manage permanent faults in order to increase the yield and chip life. A complete protection against faults creates drawbacks in speed, area and power budgets. This is the reason why we create a configurable architecture where faults coverage can be changed according to the specific application and the project constrains.

In this Master Thesis the **Instruction Fetch of cv32e40p** core is converted in a **configurable fault tolerant stage** in order to reduce failures in fetching instructions. The core used was designed by the "OpenHW Group" organization and it can be integrated in PULPissimo platform in order to create a complete microcontroller architecture.
Before the design of the architecture we study the cv32e40p core and then we proceed with the creation of the simulation environment, building the script used for all simulations. These tools born in the cv32e40p *core-v-verif* repository with the purpose of automatize compilation of testbenches and their simulation using QuestaSim and Modelsim.
The most important feature is the **optimized fault injection method** used to simulate faults in the architecture. This tool is able to inject faults in a list of signals set in the tcl script, in this way we are able to use both a worst case approach injecting faults only in sequential parts (FF and memory) or inject fault also in combinatory path increasing

simulation time and precision. The first method is faster since we inject faults in less signals but in this way we consider that all faults injected in the combinatory path (for example after a particle strike) reach a FF and are sampled. This is not always true because some bits can be logical masked in the following cases: if they are don't care bits when fault occurs, if they can be electrical masked due to attenuation before latch or if the fault don't have the time to reach a FF (latch-windows). Knowing this we can decide to inject faults in sequential parts or in all circuits to vary the trade off between accuracy and simulation time.

Apart from this first considerations the tool optimizes simulation time also takes advantage of *vcdstim* feature in the case of single stage simulation. Indeed the whole core is initially simulated using a benchmark firmware, meanwhile the input and output data of a specific stage are saved into .vcd and .wlf files, finally a stage-specific simulation started using .vcd as input (*vcdstim* feature). In this way only one stage is simulated and we reduce working times. **Stage-specific injection simulation** can be repeated a specified number of times and each time the output of the stage is compared with .wlf output file in order to find failures.

Using our tool we first simulate fault injection in the reference IF stage of cv32e40p core and we find a fault tolerance equal to 87%. Later we use simulation results to understand the fault masking for each signals and then we use this information to understand which are the most critical blocks of the IF stage.

In the meanwhile we begin to apply FT techniques to the Compressed Decoder obtaining a FT Compressed Decoder that has been tested and verified, using this new FT block we create a template written in the new metalanguage Travulog. This template can be applied to a whatever reference architecture with our Toolchain written in Python to obtain a new FT architecture. We use this Toolchain, the Travulog template and HTravulog (a hidden version of Travulog) to divide the IF stage in six parts and apply FT model to each of them. The model that we applied is the same for each block.

The Travulog template applies a transformation to the reference design that can be controlled with some System Verilog parameters, for these reason once we apply the template to each internal block of the IF stage we only enable the possibility to protect each block. In this way we create an *Automatic Transformation Toolchain* with which we create a *Configurable FT IF stage* from the reference one. This conversion can be redone every time you change something to the reference IF stage, in this way we speed up FT design and decrease maintainability costs. Using this method during simulations and synthesis we can enable FT for a set of blocks and depending on these settings we can manage area, speed, power and FT trade-off.

In the design of FT architecture we use a FT technique that is able to detect and correct transient faults using **TMR** (Triple Modular Redundant), this methods works only if each

of the three identical blocks compared don't have permanent faults, and if we assume to neglect multiple particle strikes. Although multiple strikes is improbable, permanent faults is already present after manufacturing process and increase during time, for this reason we implement a technique to detect this faults.
In the end looking at the fault injection results on the reference IF stage we have identified the Prefetch Buffer as the most critical block, we protect only that block and run fault injection simulation obtaining a fault protection of 96%, if we want to further increase protection we can also protect Aligner. As we have seen these changes can be done easily changing some SV parameters. Finally we protect each block obtaining a fault protection of 99%.
Summarizing the main results of this thesis is the creation of a *Fault injection tool for CV32E40P core*, the creation of a *Configurable FT IF stage* and the parallel design of the *Automatic Transformation Toolchain* for Travulog/Htravulog (TV/HTV). Using the Toolchain and TV/HTV code we are able to recreate the Configurable FT IF stage each time we apply changes to the reference design. The TV/HTV code can be used in other project to speed up FT design solution increasing maintainability, even if the TV/HTV metalanguage has been designed for FT application it can be used in other application whenever can be defined a transformation template, the metalanguage can also be extended to enlarge field of use.

Contents

List of Figures

Listings

Chapter 1

Introduction

Electronic is pervasive in our life; each year are sold about 1530 million of smartphones [1], 75 million of computers [2] and 65 million of cars [3]. Nowadays smartphone users reach 5.22 billion which represent the 66% of worldwide population [4], about two out of ten people own a car [5] and there are 5774 orbiting satellites [6].

These are only few data which show how many electronic devices are used today worldwide. If you think that each of those devices contains one or many CPUs you have only partially achieved the number of worldwide CPU used nowadays.

Processors are also used in industrial, networking and data processing fields. In each application a system can have different requirements that should be applied also to CPU used. The most restrictive is the *Safety-Critical Application* field, in this system the designers should complain different levels of reliability. Examples of Safety-Critical Application are the Automotive, Space Mission field and Medical devices which are three type of systems strongly embedded in our everyday life (e.g., the Cars, the GPS, pacemakers [7]).

For these reasons the study and development of more reliable and secure CPUs are really important for VLSI industries. An improvement to CPUs development come from RISC-V Instruction Set Architecture, nowadays this ISA is increasingly used by industries because it enables a frozen interface between hardware and software stack. Indeed, RISC-V ISA provides four base ISA versions described without implementation details. Starting with these versions, as a foundation, the designers can add many standard or custom extensions. Accelerators and co-processors can be also added in order to create complex and high-performance CPUs.

1.1 Objectives

This Master thesis lies on a larger project aimed to create a fault tolerant RISC-V processor for pulpissimo processor starting from CV32E40P core. In order to achieve this result the first step is the creation of a Fault Tolerant CV32E40P, for these reasons the core has been divided between me and my two teammates in this way:

- **Instruction Fetch (IF):** Elia Ribaldone;
- **Instruction Decode (ID):** Marcello Neri [8];
- **Execution Stage (EX):** Luca Fiore [9];

In each thesis the reference stage has been converted in a Fault Tolerant one.

The objective of this thesis is the creation of a Fault Tolerant (FT) Instruction Fetch for CV32E40P core. In particular the automatization of FT transformation has been investigated and a metalanguage has been created to speed up this operation.
Indeed, nowadays an HDL able to automatically apply a transformation to an existing architecture don't exist. Such language can dramatically decrease the time to market and the maintaining efforts of a fault tolerant architecture because FT cores are usually created starting from a previous design. Even if the final result can be completely different from the original design, many pieces of the starting architecture are used and modified applying a template. A typical example is the TMR which consists in triplication and voting of a working not FT architecture, these techniques correspond to an architectural template that can be applied to many designs, anyway at today each time we should apply this technique we should manually create redundancy and connect voters in a HDL language such as System Verilog. These is the motivation that has pushed this work to a metalanguage direction.

1.2 Thesis structure

The Thesis structure is divided into seven chapters:

- **Chapter 2 - Technical Background:** This chapter explains the IEC61508 standard, the fault tolerance vocabulary and metrics, the cause of bit flip in VLSI circuits and the hardening techniques to increase reliability.
- **Chapter 3 - RISC-V and CV32E40P core:** Here the history, motivations and the structure of RISC-V ISA are explained, then the Instruction Fetch of the CV32E40P is studied in order to apply the FT techniques in the next chapter.

- **Chapter 4 - Fault Tolerant Compressed Decoder:** In this Chapter a FT Compressed decoder for CV32E40P core is designed and verified, the new stage is protected to transient faults and can detect permanent faults.
- **Chapter 5 - Travulog and HTravulog:** Starting from the FT Compressed Decoder (CD), a Travulog template and a toolchain, able to transform the original CD in the FT CD have been developed. In this chapter there is also a description about Travulog/HTravulog metalanguage that is used to transform each block inside the Instruction Fetch in a FT block.
- **Chapter 6 - Results:** In this chapter the complete FT IF stage, created using Travulog metalanguage, are presented and the results of simulations are reported.
- **Chapter 7 - Future Works:** In this chapter some improvement to FT architecture and Travulog are proposed.

Chapter 2

Technical Background

This chapter describes the technical background needed to understand what is a Fault Tolerant system, where the thesis lies inside the FT system, which are the causes of faults and how to avoid that faults become errors.
We also briefly explain some Safety standards used in Safety Critical applications and we deeply investigate the origins and causes of particle strikes.
The chapter is organized in three sections:

- **Safety critical application system:** In this section we give an overview on the safety critical systems at high level; firstly we analyze the dependability model in which are described all metrics to measure the dependability of a system, then we describe the position of this thesis work inside a generic electronic system and finally we give an introduction to the IEC61508 standard for safety critical applications;
- **Dependability of Integrated Circuits:** Here we analyze all causes of error inside an integrated circuit, firstly we investigate internal factors of faults which are caused by manufacturing processes and internal mechanisms, then we investigate external factors of faults such as Solar, GCR and package radiations. For each factor of faults we describe the effect on the IC and in the last section we give some definition about Soft Errors;
- **Hardening techniques for digital circuit architecture:** In the last section of the chapter we describe some techniques to protect digital circuits at RTL level, all these techniques use redundancy to achieve this result.

2.1 Safety critical application systems

A system is defined "safety critical" when a failure may cause damage to people, dangerous destruction, military upset or loss of data. Through the next three sections we explore the parameters to measure the dependability of a system, how an Safety Critical electronic system is structured and which are the main standards used in the Safety Critical Applications.

2.1.1 Dependability Model

Dependability is the ability of a system to provide a predetermined level of service to the user [10]. This capacity depends on the system application, for example a wrong use or high workload lower the level of service offered. From the designer's point of view, the dependability of a system must be verified through tests and simulations, in order to verify the correct functioning of the system in various environments.
For system that works in critical applications, in addition to the functional tests, tests must be made to verify the level of service required despite environmental conditions.
For example in satellites it is not possible to do maintenance but it is necessary to avoid the fall of the asset. In these cases, when the Dependability required to the system is high, many stress tests must be done to have a complete technical testing. For these reasons to guarantee the dependability in a given application the main factors are how the system is designed and which kind of tests are performed on it.

Dependability is characterized by: Metrics, Attributes, Impairments and Means. These four categories allow us to completely define the dependability in a system and they are explained below:

Dependability Metrics Dependability metrics are used to measure the dependability of a system and they are used to verify Dependability Attributes. The Metrics are experimentally measured or estimated through various techniques. These are the main metrics used:

- ***TTF* :** Time To Failure (TTF) is the time required to have an error in a specific system [11]. For example a device with TTF equal to 1 year will probably have an error after one year of correct work.
- ***MTTF* :** Mean Time To Failure (MTTF) is the mean time between two failures in a system. Under certain conditions (e.g., formula 2.6) we can combine the MTTF of various parts to find the MTTF of overall system, to do this we should use the

following formula:

$$MTTF_{system} = \frac{1}{MTTF_{part1}^{-1} + MTTF_{part2}^{-1}} = \frac{1}{\sum_{i=0}^{n_{parts}} \frac{1}{MTTF_i}} \tag{2.1}$$

- ***FIT*** **:** Failure In Time (FIT) is the number of errors in a billion of hours. The relation between MTTF (expressed in year) and FIT is:

$$FIT = \frac{10^9}{MTTF_{year} \cdot 365\,days \cdot 24\,hour} \tag{2.2}$$

The FIT metric is used instead of MTTF because it makes calculation easier, in fact system FIT can be easily calculated in this way:

$$FIT_{system} = \sum_{i=0}^{n_{parts}} FIT_i \tag{2.3}$$

- ***MTTR*** **:** The Mean Time To Recover (MTTR) is the time needed to a system to repair an error once it has been detected [11].

- ***MTBF*** **:** The Mean Time Between Failure (MTBF) is the mean time between the start/restart and an error detection, for this reason we have:

$$MTBF \;=\; MTTF + MTTR \tag{2.4}$$

Dependability Attributes Attributes are the properties which are expected from a system that experiencing faults to be dependable [10]. These attributes are evaluated from Dependability Metrics according to a faults model. The most used Attributes are Reliability, Safety and Availability, they are defined below:

- ***Reliability*** **:** it is the probability that a system will operate without failures in a given time interval. This type of Attribute is widely used for example in space applications, where it is necessary to guarantee operation for certain period of time. At the integrated circuit level many techniques have been adopted over time to increase reliability by improving production processes, usually are used old processes experiences to predict the reliability of a new product, this is done on all ICs but especially on memories **An_Extended_Building-In_Reliability_Methodology_on_Evaluating_SRAM**. Reliability can be expressed according to *exponential failure law* :

$$R(t) = e^{-h(t)\ t} \simeq e^{-\lambda\ t} \tag{2.5}$$

Where $h(t)$ is the *Instantaneous Error Rate* considered as the probability that the system has an error in a certain interval Δt which start at instant t, so it is the

probability of error in the time interval $(t,\ t\ +\ \Delta t)$. To simplify calculation $h(t)$ is usually approximated with the constant error rate λ, that is equal to $1/MTTF = FIT$ [11].

For these consideration when we have the FIT of each part of a system we can use formula 2.5 to find total reliability, in this case we consider to have n independent parts each with a certain failure rate h_i:

$$R(t)_{system} = \prod_{i=0}^{n-1} R_i(t) = e^{-\left(\sum_{i=0}^{n-1} h_i\right)} \tag{2.6}$$

This model is valid if we consider the failure rate constant. From formula 2.6 we can states that the FIT of a system is equal to the sum of the FIT of each part.

- ***Availability* :** It is the percentage of time the system remains active and it can be used. This Attribute is employed a lot in the IT field, for example to characterize servers or a communication network [12] [13]. It is therefore required in areas where it is expected that the system may not work for some periods, so in this case we are interested to know how long it will actually work properly.

  Availability is usually expressed as a percentage or by the downtime at a certain instant. For example, a system with Availability of 99.999% will have a mean downtime of 5 minutes over a year. The common expression for Availability is:

$$Availability = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF} \tag{2.7}$$

- ***Safety* :** For this attribute, two types of failures are considered : *fail-safe* if the fail does not cause danger or damage, while *fail-unsafe* if the fail causes safety problems.

  A simple example is a RADAR that detects airplanes, if an airplane that doesn't exist is detected there is no serious damage and therefore we consider this failure as fail-safe, instead if an airplane is not detected we have a fail-unsafe failure. The safety of a system is the probability that it remains fail-safe over a certain period of time. It is used in critical sensing, safety and control systems.

Dependability Impairments Dependability Impairments are used to communicate that something in the system has gone wrong [10]. There are three types of Impairments and each indicates a problem at a different level:

- ***Faults* :** They indicate a problem at the physical level. For example in a PCB circuit a fault can occur when a component desoldered due to incorrect manufacturing process. In the field of integrated circuits a fault is usually due to a bit flip caused by external particles, by a manufacturing defect or a bug in the microcode or software.

Any failure of a system always starts with a fault, this fault may or may not cause a problem depending on how the design was done. In integrated circuits faults can be masked by certain architectural design techniques and their number can be limited by special layouts and processes. However, they cannot be eliminated entirely.

- ***Errors* :** They indicate a problem at computational level caused by a Fault. Errors are caused by Faults that are not masked by the system, for example if there is a bit flip in an input register of the ALU, there will be an Error in the output register because the operation has a wrong result.

- ***Failures* :** They indicate system failure due to an Error. The failure of the system is an Impairments that you never want to have in a critical application since the behavior of the circuit is unpredictable and so unsafe.

To summarize a Fault can cause an Error and this in turn can cause a Failure. For these reasons the designers of a critical application system should have the ability to mask Faults and Errors in order to avoid Failure.

Dependability Means Dependability Means are a set of techniques and methods needed to create a Dependable system[10]. Fault Tolerance is the method that is used in this thesis but it is normally followed by other techniques, these are the most important ones:

- ***Fault Tolerance (FT)* :** Fault Tolerant systems continue to work even in the presence of Faults, this result is achieved through redundancy and a set of processes: The first is called Fault Masking and consists in avoiding the propagation of a fault by correcting the values in the system.

  In fact Fault Masking consists both in the reduction of errors and in their masking to avoid failures. Common examples of Fault Masking techniques are TMR (Triple Modular Redundancy) and ECC (Error Correcting Code) that allow to reduce Errors in memories and circuits.

  The second process is the Fault Detection that allows to recognize the presence of an error in the system, for example using the TMR in order to detect a Fault we can just verify that there is a module with different results from the others. This technique is also used in systems without redundancy where you want to understand if the system is working properly.

  When a fault is detected in a FT system, you can decide to correct it and continue with the execution, or you can disable the system part from which the fault started, in the case of permanent fault. This mode of performances decay of a system is called Graceful Degradation.

- ***Fault Prevention (FP)*** **:** FP is a very broad field because it is the set of processes that allow to reduce the introduction of faults in the system. This goal is achieved by controlling all processes from specification to manufacturing.
- ***Fault Forecasting*** **:** Fault Forecasting is the set of techniques that allow to predict the trend of the number of Faults and their effects in a system.
- ***Fault Removal*** **:** Fault Removal is the set of techniques used to eliminate errors already present in the system. This is done through verification of circuit operation and maintenance.

We have seen the basic vocabulary used in dependable system design and maintenance, in **Figure 2.1** are summarized all concept explained in order to give a graphical overview of the design of a Dependable System.

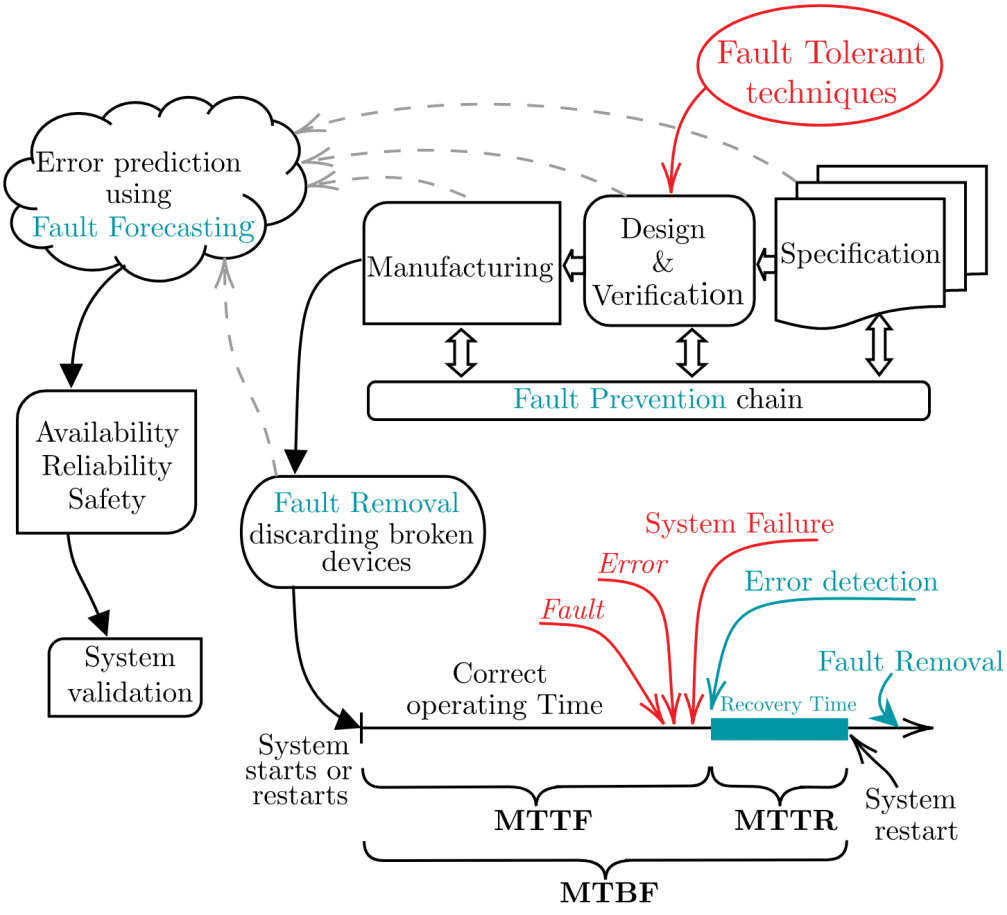

Figure 2.1: Design and life of a Dependable System

The block diagram in **Figure 2.1** start with the specification of the system, then the

designer use Fault tolerant techniques to design and verify the system, finally the product is manufactured, in these tree steps is applied Fault Prevention in order to reduce unwanted errors.
After manufacture, the manufacturer apply a selection in order to discard broken devices and finally the systems is sold and it begins to be used. Meanwhile we gather data from all production chain in order to use Fault Forecasting to predict MTTF,MTTR and MTBF. Then using predicted data are evaluated required Dependability Attributes and finally system is validated and can be sold.

When the system begins to be used there are some periods of correct operations (estimated as MTTF), then at a certain instant a fault occur, this fault can propagate in an Error and this can became a System Failure. If the Failure is detected the system begins the Recovery Time (estimated as the MTTR) in which the failure is fixed. In the diagram we select a time interval in which fault is propagated but in a Dependable system this should happen rarely. It is also indicated the removal of defected parts using Fault Removal, this techniques can be also applied during Recovery time.

In the next section we contextualize this thesis work analyzing the parts of a critical electronic system.

2.1.2 Electronic system parts

This section describe how this Thesis is positioned in a complete dependable electronic system.
In **Figure 2.2** we give an example of electronic system, it receives information from *sensors* and it controls some *actuators* according to their specification.
The circuit is powered by a battery or by power network and this energy should be converted inside the board to be used. For this reason there is a part of the PCB dedicated to *voltage conversion*, this block is composed by analogue and digital components that together create the Power Conversion and Distribution system.

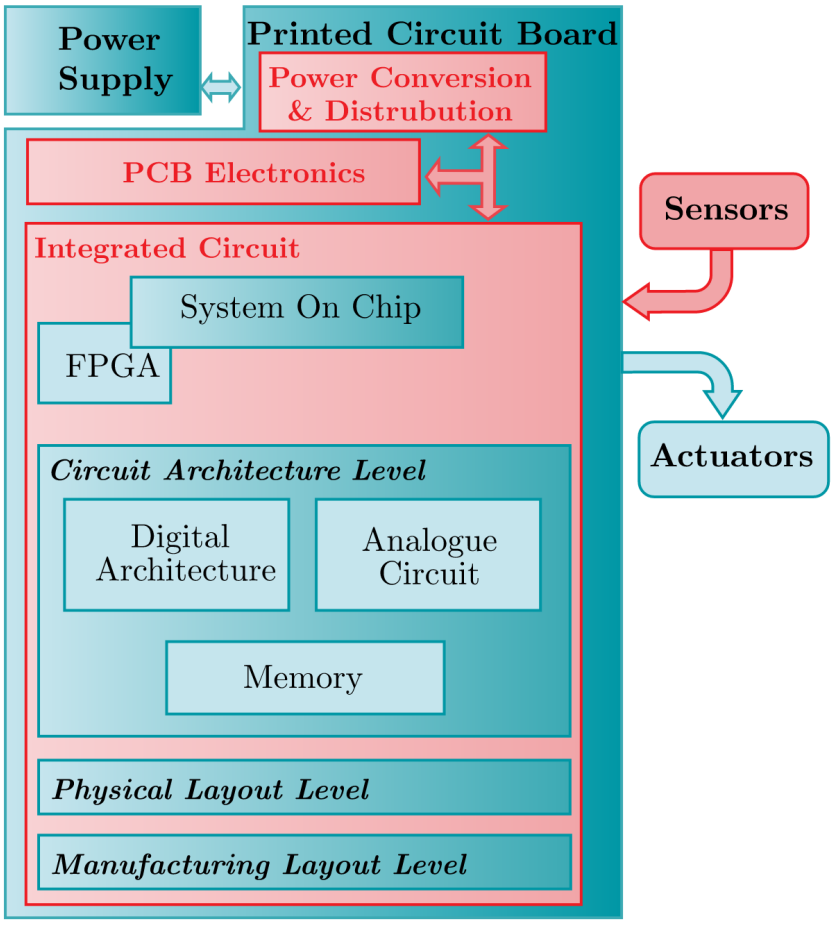

Figure 2.2: Example of Electronic System

The elaboration part instead is composed by integrated circuits that analyze the data received from analog and digital sensors and they use this data to decide how to control the actuators. This elaboration is done by a microcontroller or an FPGA and the design

of these ICs have four main design level [14] as you can see in **Figure 2.2**:

- **Manufacturing Process Level** (lev. 4) : This is the level of manufacturing processes, in this step are defined all techniques to create the die from a silicon wafer. In the case of hardened chip the manufacturer apply fault tolerant and fault prevention techniques in order to improve system dependability.
- **Physical Layout Level** (lev. 3) : It is the set of techniques used to place transistors properly. In the case of robust systems the layout is improved in order to decrease the sensitivity of the circuit to radiation.
- **Circuit Architecture Level** (lev. 2) : At this level circuits design is carried out at the RTL level; the circuits may be digital, analogue or a mixed signal. Generally to make this level robust are used fault tolerance redundancy and error correction techniques.
- **Electronic System Level** (lev. 1) : In this case we can still work at the RTL level using components previously created at the architectural level, or at the unit level (e.g., cluser computers). In the case of robust systems is used processor redundancy (e.g., lockstep technique) or redundancy of computers.

As we have seen, an electronic system is made up of many parts which must all be dependable in order to have a dependable system. *This Master Thesis will deal with the second design level, which is the architectural one.* In order to be able to use the proposed rtl project correctly, it is necessary to use hardening techniques in all the lower and higher levels.
In fact what is important for the final application is the dependability of the system, so it would be almost useless to use a hardened processor in a device where the power supply part is not dependable.
Anyway this consideration should be done case by case by designers. In the next section we analyze the main standards used in Safety Critical Application.

2.1.3 IEC61508 and ISO26262 Standard

In 1971 Intel produce the first microprocessors and after only ten years the decreasing cost of processors made possible the spread of electronic devices. In only ten year an huge amount of consumer devices was computerized but the first critical use of electronic devices was in machinery, industrial plant and robots, for these reason in 1985 was born the IEC61508 standard created by a IEC technical task forces created by the International Electrotechnical Commission Advisor Committee Of Safety (IEC ACOS), in order to create a guideline for those industries that needs to meet some safety requirements.

The IEC ACOS is the committee which manages all Technical Committee of IEC and it reports directly to Standardization Management Board (SMB).

Despite some criticism the IEC61508 began to be used in many industries and it evolves in other more specific standards such as IEC61511 for process industries, IEC62061 for machinery safety , IEC61513 for nuclear plants and many others. Today IEC61508 is used manly in machinery, chemical plants, nuclear plants and robots, instead for all on-road vehicles or the so called Automotive world is used the ISO26262 created by the International Organization for Standardization.

IEC61508 is composed by seven parts and published with the title *Functional safety of electrical/electronic/programmable electronic safety-related systems*, for this reason the standards refers to any devices from electronic valves, relays, and switches to industrial PLC that need to have a certain safety level.
To fulfill this safety level the designers needs to reach the *Functional Safety*, It is the full program to ensures that the safety-related E/E/PE system brings to a safety state when a safety issues occurs. Functional safety is a really important concept that is also used in ISO26262 and any following safety-critical standards.
As already mentioned, IEC61508 is composed by seven parts, each part has it related document:

- **Part 1:** This part covers General Requirements for compliance with the standard, here are defined Safety Integrity Levels and the overall Safety Life cycle of the system. It is composed by 68 pages.
- **Part 2:** This part covers requirements for E/E/PE system related systems. It is composed by 96 pages.
- **Part 3:** This part covers Software requirements for compliance with the standard. It is composed by 118 pages.

- **Part 4:** This part covers definitions and abbreviations. It is composed by 42 pages.
- **Part 5:** This part contains many example of methods for the determination of Safety Integrity Levels. It is composed by 54 pages.
- **Part 6:** This part contains some guidelines on the application of IEC61508 part 2 and IEC61508 part 3. It is composed by 118 pages.
- **Part 7:** This part contains an overview of techniques and measures in order to support the application of the standard. It is composed by 150 pages.

IEC61508 part one

The first part is about the **General Requirements**, this part is the introduction to the standards and it could be divided in some parts or arguments:

- **Scope:** As already mentioned IEC61508 refers to any E/E/PE devices, these include electro-mechanical devices, solid-state non programmable electronics and and electronic devices based on computer technology (microprocessors, micro-controller, programmable controllers, ASIC, PLC , smart sensors, transmitters, actuators and so on). Anyway the standard doesn't cover E/E/PE system which with a System Integrity level lower then 1 or an availability lower the 90% of time.
- **Conformance:** In order to be compliance to the standards the designers should create a documentation in which is explained how each requirement is fulfilled according to the decided Safety Integrity level. The documentation required is usually considerable and you should be detailed in fulfilling the requirements.
- **Documentation:** The main argument of the documentation is the Safety Life Cycle which represent the life of the product from specifications to decommissioning. This life cycle should be described in order to enable the Functional Safety verification and assessment, the documentation also should be a support for the life cycle process in order to guarantee that each step is executed.
- **Management of functional safety:** Functional safety should be achieved and maintained, for these reason you should write all activity that make it possible in a document called Functional Safety Management (FSM) plan. This plan should contain: all strategy used, all people responsible of safety , all about Safety Life Cycle and Functional Safety, the procedures to ensure that the personals are competent and to enable the analysis of hazardous incidents, the maintenance operations, the procedures to change something in the system and some other minor documentations. It is also very important that each Safety Life Cycle part should have a responsible for that phase.

- **Safety Life Cycle Requirements:** Safety life cycle is a closed loop in which the system goes through identification, analysis, design and verification. This life cycle came from the analysis of the Health Safety Executive (HSE) about the industrial control systems classified accident causes shown in **Figure 2.3**.

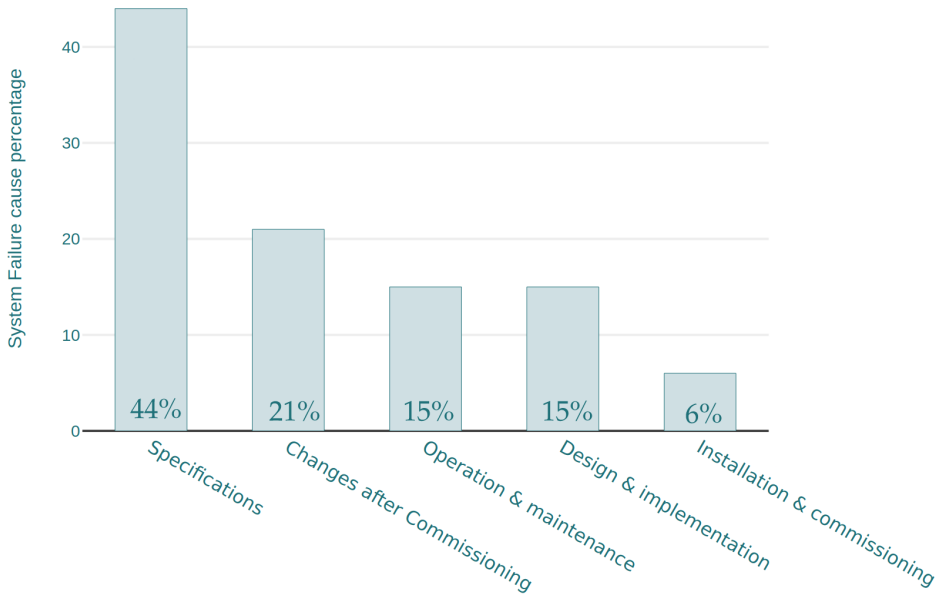

Figure 2.3: Results of system failure cause study from HSE

In **Figure 2.3** there is the overall safety life cycle of a safety-critical system composed by E/E/PE device and software, this is the life cycle to follow the system should accomplish to the IEC61508 standard. The block diagram in **Figure 2.3** is only a brief summary of the safety life cycle, in order to completely understand this part read the IEC61508 part one clause 7.

- **Safety Integrity Level:** IEC61508 define four level of Safety Integrity Level, they are the order of magnitude levels of risk reduction. The SIL is defined by two tables, a table of Low Demand Mode **Figure 2.5** of operation and Continuous mode of operation **Figure 2.4**. When a system work in Low Demand mode the safety function is executed only when it is needed, otherwise in Continuous Mode of operation the safety function is executed continuously. For these reason when the safety function is in Continuous mode the SIL is calculated as failure per hour, instead when the safety

function works in Low Demand Mode the SIL is calculated as dangerous failure on demand.

Safety Integrity Level	*Probability of dangerous failure per hour*
SIL 4	$10^{-9} \geqslant P < 10^{-8}$
SIL 3	$10^{-8} \geqslant P < 10^{-7}$
SIL 2	$10^{-7} \geqslant P < 10^{-6}$
SIL 1	$10^{-6} \geqslant P < 10^{-5}$

Figure 2.4: SIL definition for continuous mode

Safety Integrity Level	*Average probability of failure on demand*
SIL 4	$10^{-5} \geqslant P < 10^{-4}$
SIL 3	$10^{-4} \geqslant P < 10^{-3}$
SIL 2	$10^{-3} \geqslant P < 10^{-2}$
SIL 1	$10^{-2} \geqslant P < 10^{-1}$

Figure 2.5: SIL definition for demand mode

Looking at the table you should notice that continuous mode seems to have lower value of probability for the same SIL, anyway we should consider that SIL is evaluated in one year for definition and that the continuous mode probability is referred to failure per hour. For these reasons, and considering that there are 8760 hour in one year, the two SIL table are comparable in terms of safety metric.

- **Functional Safety Assessment:** Basing on the Safety Integrity Level the failure in the system could cause a certain amount of injury. For this reason the standard states that there should be independent person, department or organization that do a Functional Safety Assessment of the system.

  This naturally increase the safety of the final system since this procedure is a redundancy in verification. The people who apply the Functional Safety Assessment should

be competent and they will have the responsibility of the system.

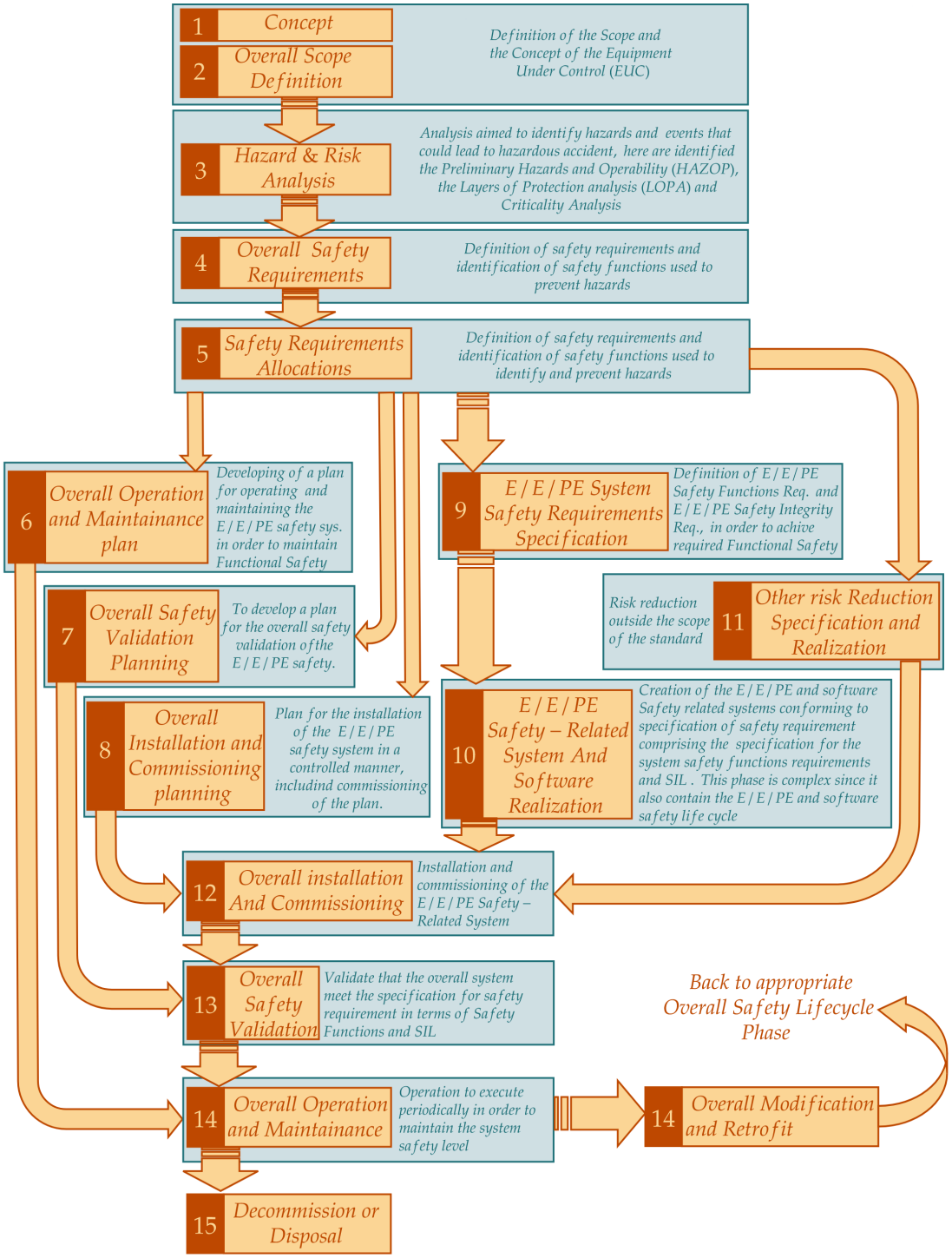

Figure 2.6: Overall Safety Life Cycle

IEC61508 Part two Hardware Requirements

Part two of IEC61508 covers the requirements of safety-related hardware. In particular this part discusses and expands phase 9 of the main safety life cycle (figure 2.6).
Inside this part is described the methods used to determine the SIL for a safety-related system. In order to understands this procedure we should define some terms:

- **Safe Failure Fraction (SFF):** From part 4 of IEC61508 it is defined as "property of a safety related element that is defined by the ratio of the average failure rates of safe plus dangerous detected failures and safe plus dangerous failures". The "safe plus dangerous failures" is the overall number of failures, since in this thesis we will work with fault injection this number is the number of fault injected.

  Instead "average failure rates of safe plus dangerous detected failures" are the overall number of failures injected that don't cause an error in the output of the stage. For these reasons in this thesis:

  $$SFF = 1 - \frac{\text{Errors at the output of our stage}}{\text{Total injected error}}$$

- **Element Type:** It is a classification of the elements used in the system, for the standard exist two type of element, Type A and Type B. Element Type A: they have well defined failure modes and they have failure behaviour completely determined and "there is sufficient dependable failure data to show that the claimed rates of failure for detected and undetected dangerous failures are met".

  For these reason Type A are those element which are enough simple to be completely defined and which behaviour is predictable.

  Element Type B: They have not well defined failure mode or they haven't failure behaviour completely defined or "there is insufficient dependable failure data to support claims for rates of failure for detected and undetected dangerous failures".

  Therefore Type B are usually complex elements which general behaviour is not completely predictable, anyway it is important to notice that for Type A is used "and" statement instead is used "or" statement for Type B, for these reasons it is more common to have a Type B element rather than a Type A, for our scope ASICs can be defined as Type B since not all event inside the chip can be modeled in order to predict failure behaviour.

- **Hardware Fault Tolerance (HFT):** From part 2 of IEC61508 it is defined as: "HFT of N means that N+1 is the minimum number of faults that could cause a loss of the safety function", This means that in a system with HTF equal to 1 we need at

least two fault to cause a loss in the safety function. So a system with HFT of one is designed to tolerate only one dangerous failure, in this thesis we use triplication that is a way to create a system with HTF of one.

In order to claim a certain level of SIL we should follow **Route** 1_H or **Route** 2_H explained in the coming paragraph.

Route 1_H This approach takes in consideration the Safe Failure Fraction (SFF) of the elements. Route 1_H starts from the calculation of SFF and from the required SIL we find the required HFT. This connection between SFF, SIL and HFT change from Type A and Type B as we can see in Table 2.7 for Type A and in Table 2.8 for Type B.

Safe Failure Fraction of an element	*Hardware Fault Tolerance*		
	0	1	2
< 60%	SIL 1	SIL 2	SIL 3
60% – < 90%	SIL 2	SIL 3	SIL 4
90% – < 99%	SIL 3	SIL 4	SIL 4
⩾ 99%	SIL 3	SIL 4	SIL 4

Figure 2.7: Maximum SIL for a safety function carried out by a type A element.

Safe Failure Fraction of an element	*Hardware Fault Tolerance*		
	0	1	2
< 60%	Not Allowed	SIL 1	SIL 2
60% – < 90%	SIL 1	SIL 2	SIL 3
90% – < 99%	SIL 2	SIL 3	SIL 4
⩾ 99%	SIL 3	SIL 4	SIL 4

Figure 2.8: Maximum SIL for a safety function carried out by a type B element.

Route 2_H Route 2_H don't consider the SFF of a system but only specify the level of Hardware Fault Tolerance according to required SIL and the operation mode.

ISO26262

ISO26262 is the standard used in Automotive Industries and in whatever device that should be used in on-road vehicles. It is born in order to answer to the increasing amount of safety-related electronic used in vehicles, for these reason it has been created much later IEC61508, in 2011.
This standard is radically different from the IEC61508. Anyway ISO26262 was created after the IEC61508 and so many aspect of the two standards can be connected, for example the Safety Integrity Level in ISO26262 is replaced with Automotive Safety Level (ASIL). A complete analysis of ISO26262 is outside the scope of this thesis, anyway the use of the designed architecture in Automotive field typically imply an ISO26262 certification.

2.2 Dependability of Integrated Circuits

Faults in integrated circuits are due to both bit flip or electrical problems such as broken interconnects. The origins of these problems are due both to the aging of integrated transistors and their susceptibility to charge injection by external particles, such as cosmic rays.

These two phenomena are influenced by the field of use of the IC and by the working conditions. For example, aging is accelerated by high temperatures and high workloads, which wear out the interconnections. On the other hand the influence of external particles increases in space applications due to the increased cosmic ray flux, as well as in nuclear power plants or where some radioactive materials are present.

The understanding of these phenomena is essential to improve fault tolerance techniques applied to integrated circuits also at RTL level in different application, therefore the causes and mechanisms of faults are now investigated by dividing them into *internal factors* (graceful degradation) and *external factors* (e.g., particle flux).

2.2.1 Internal Factors of Faults

As already mentioned, the internal factors of faults are due to intrinsic electrical problems of transistors, which can be caused either by the *breakage of the interconnections* or by problems related to the *gate oxide failure*.
As far as *interconnections* are concerned, there are two origins of failure:

- **Electromigration** (EM) : EM is a phenomenon known since 1966 [15], whereby the electrons generating the electric current in the interconnections impart a momentum to the atoms of metal. This momentum transfer can create void in the very small interconnections of ICs. The phenomenon is directly proportional to the square of the charge density (j_e; $[A/cm^2]$) and depends exponentially on the *activation energy* of the material (E_a; $[eV]$) and on the temperature (T $[K]$). These relationship are condensed in the Median Time To Failure calculated according to the Black's formula [11]:

$$MeTTF_{system} = \frac{A_0}{j_e^2} e^{\frac{E_a}{kT}} \tag{2.8}$$

  Where A_0 is a technology dependent constant and k is the Boltzmann constant.

  The opposite effect to EM is due to mechanical stress which tends to compensate for the displacement of metal atoms, this principle is the basis of the Blech effect for which below a certain length (called the Blech length) EM has no effect because the two forces are balanced.

Normally the length of the interconnections is greater than the Blech length and for this reason EM should be reduced by various techniques. Two of these techniques are the use of metal alloys (Al+Cu, Al+Pd) or the creation of *Bamboo Structures* that reduce the number of metal grains. In fact, the creation of a void in a connection starts at the interface between two or more grains of metal. Here the mobility of the atoms is greater respect to normal mobility, for this reason metal atoms are able to move and they leads to an avalanche effect which creates the final void.

Electromigration create both permanent or intermittent faults and leads the chip in the wear-out phase. As we have seen this phenomena is related to current density that normally depends on workload, hence architecture and system fault tolerant strategy for EM reduction lead with resource multiplexing and oversizing.

- **Metal Stress Voiding** (MSV) : The MSV is due to the difference in expansion ratios between the metal of the interconnection and the surrounding material. The phenomenon is closely related to temperature and the formula 2.9 gives a quantitative evaluation in terms of MTTF [11]:

$$MTTF_{system} = \frac{B_0}{(T_0 - T)^n} e^{\frac{E_b}{kT}} \tag{2.9}$$

  Where: B_0, n and E_b are material dependent constants, k is the Boltzmann constant and T is the temperature in Kelvin. According to the equation the larger the temperature the lower the MTTF, this is a further reason why heat dissipation is important for system dependability. Another important methods to reduce the influence of this phenomenon is the use of stronger metals, with expansion constants similar to the interfaces.

  MSV related faults are very similar to those caused by EM and can be either intermittent or permanent.

As far as *Gate Oxide Failure* is concerned, there are three main physical mechanisms that cause faults:

- **Negative Bias Time Instability** (NBTI) : NBTI is the process that causes short-channel pMOS (hence the term Negative Bias) subjected to high temperature or negative gate voltages, to degrade the maximum frequency of the circuit and to create faults. These phenomena is due to charges being trapped under the gate of the pMOS [11] [16]. These charges slow down the switching process, decreasing the speed of the circuit and creating Timing Faults. Timing Faults happen when the propagation time of the critical paths no longer respects the sampling conditions according to the circuit's clock.

The physical effect related to this phenomenon is the decrease in mobility under the gate due to the bombardment of charges during normal operations. This causes the pMOS threshold voltage to increase (hence the term instability) and the maximum current to decrease, leading the logic gates (which use pMOS) to slow down and fault [11].To reduce the contribution of this effect are used Dynamic Voltage Scaling and the power gating [16] [17] [18].

- **Hot Carrier Injection** (HCI) : HCI leads to a reduction of the f_{max} of the circuit but in this case this is due to the charges trapped in the gate. In fact, during the acceleration along the channel, the ionization effect produces electron-hole pairs, if these charges have sufficient energy they can inject themselves in the gate and get trapped [11]. This creates a variation of the threshold voltage that lead to faults as in the case of NBTI.

  Unlike the other effects, HCI get worse at lower temperatures due to the increase in charge mobility in the material. The first consequence of HCI is the degradation of the threshold voltage that decreases the maximum saturation current, this lead to a reduction of the maximum frequency from 1% to 10% [11].

  Again, duty cycle reduction is a way to reduce the effect of HCI. Despite technological advances, HCI is still present in recent Tri-Gate Nanowire [19], FLASH memories [20] and general CMOS electronics.

- **Time Dependent Dielectric Breakdown** (TDDB) : Continuously applied voltages in the transistors create defects in the gate material, which can lead to the creation of conductive paths between the channel and the gate, knocking out the transistors. In thicker gate this effect is more pronounced.

  To reduce this phenomenon, attempts are made to reduce the gate voltage and to use stronger gate materials, anyway many work has been done on this problem [21] [22].

All these effects added to the manufacturing defects lead to: an infant mortality phase of the components which are discarded before being sold, a life phase with a certain fixed value of failure rate and finally a wear-out phase which causes the final failure of the integrated circuit. This variation of the failure rate over time is shown in the **Figure 2.9**.

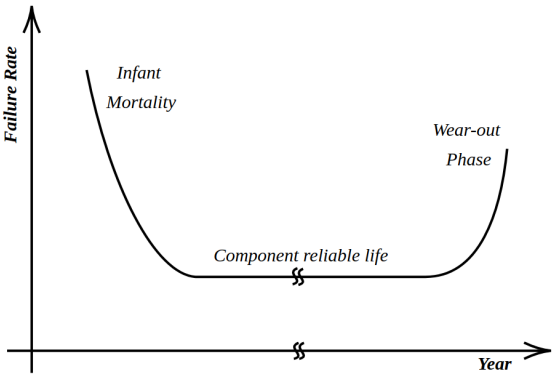

Figure 2.9: Example of Electronic System

2.2.2 External Factors of Faults - Radiations

For External Factors of Faults we mean all those external factors that can cause ICs to malfunction. In the next paragraphs we first analyze the different sources of radiations, and then the radiation effects on ICs.

Radiation Levels and Sources

There are essentially four sources of radiation: supernovae and celestial explosions that create Galactic Cosmic Rays, the Sun that generates Solar Cosmic Rays, terrestrial radioactive materials (e.g., ^{238}U), and finally nuclear weapons and reactors. The characteristics and radiation levels of these sources are described in the following paragraphs.

Galactic Cosmic Rays (GCRs) In order to understand how GCRs arrive on earth, we need to know the structure of the heliosphere.
As described in **Figure 2.10**, the Sun emits particles in all directions, mainly protons and alpha particles that form the Solar Wind at $400 - 700 km/s$. The Solar System moves through the local interstellar medium (LISM) composed mainly of helium and rarefied hydrogen. For this reason the solar wind collides at supersonic speed with interstellar dust (at a relative velocity of about $26 km/s$ respect to the Sun) and is slowed down to subsonic speeds at the so-called 'Termination Shock' (75-100 AU from the Sun).

After the Termination Shock, moving away from the Sun there is a zone where the LISM and solar rays are compressed to form plasma, this zone is called Heliosheath (pink filled at the right of the sun in **Figure 2.10**). At the end of the Heliosheath there is the limit beyond which the solar rays cannot go, this is called the Heliopause ($\simeq$ 121-150AU from the Sun). Beyond the Heliopause there is probably the Bow Wave, the shock wave of the LISM with the heliosphere, such as water does on the bow of a ship.
In this environment, the Galactic Cosmic Rays are the isotropic flow of energetic particles from outside the solar system that try to pass through the solar wind and magnetic field shields into the Earth's atmosphere as shown in **Figure 2.10**.
GCRs are created by stellar explosions such as supernovae and gamma-ray bursts, active galaxies or quasars, they reach the Earth isotropically and so they hit it uniformly in more or less all directions. In fact, unlike the LISM, these rays have an energy that can reach $100\,000\,TeV = 10^{20}\,eV$. Anyway considering that GCRs need to have an energy of at least $50 MeV$ to pass the Termination Shock, only 35% of them reach the Earth's atmosphere.

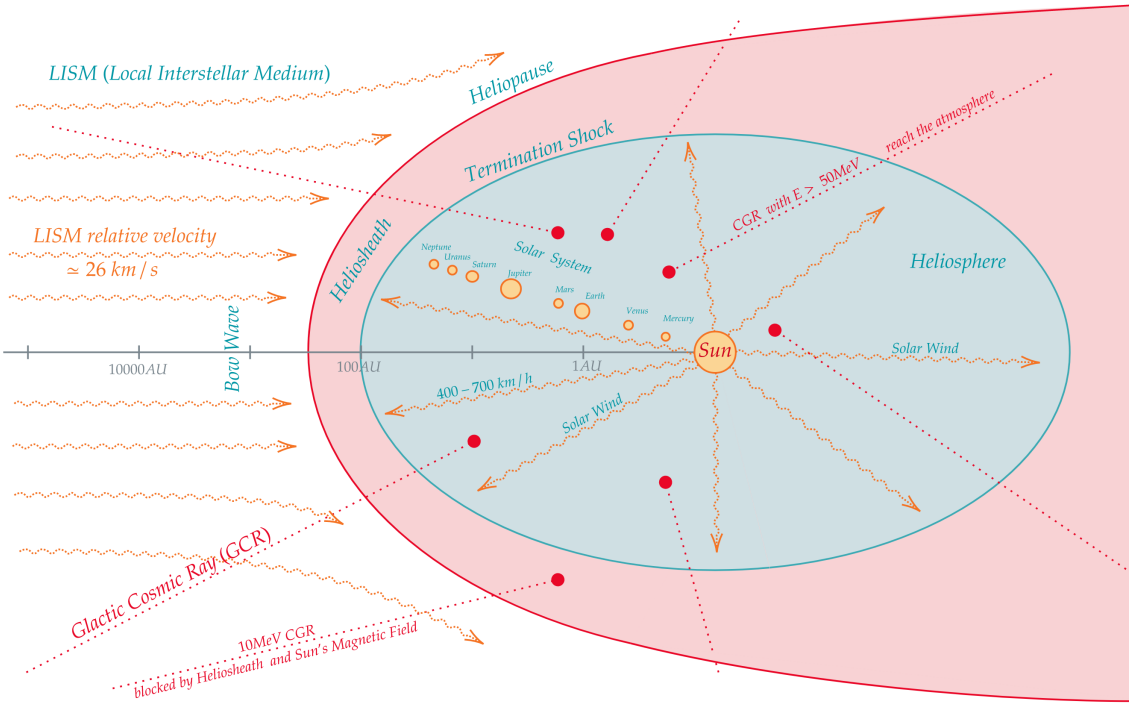

Figure 2.10: Solar system moving within the LISM while it is hit by GCRs, the planets are not to scale and the distance is logarithmic.

.

GCRs are composed for 89% of protons (p^+), 9% of alpha particles (He^+) and 2 % of heavy ions (mainly Lithium, Beryllium and Boron). Their effect on the Terrestrial Cosmic Rays varies according to the variation of the Earth's magnetic field, when the Sun's peak occurs the Earth's magnetic field is maximum, consequently there is a minimum in the radiation induced by the GCRs. On the contrary, when the Sun is at a minimum, there is a maximum of radiation on the Earth.

The flux of cosmic rays depends on their energy, as can be seen in **Figure 2.11** the flux is measured in $\frac{particles}{m^2\ sr\ GeV\ sec}$, where steradians refer to the centre of the earth while m^2 is the distance of the area to be measured

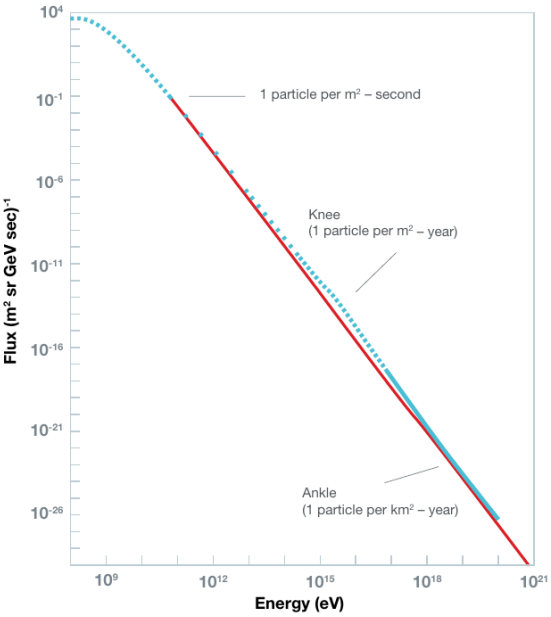

Figure 2.11: Galactic Cosmic Rays [23]

from the centre of the earth.
For these reasons, the value $m^2 * sr$ corresponds to the area over which we want to calculate the number of particles. Therefore to calculate the flux of 1GeV particles in a $cm^2 = 0.0001\,m^2$ using **Figure 2.11**, we have:

$$Flux_{part} = 10^3 \frac{particle}{m^2\,sr\,GeV\,sec} \cdot 1GeV \cdot 0.0001m^2 = 0.1\,\frac{particle}{cm^2\,sec} = 6\,\frac{particle}{cm^2\,min} \quad (2.10)$$

Solar Cosmic Rays The Sun is a star that continuously converts hydrogen into helium through nuclear fusion, ejecting more than $60MW/m^2$.
Externally it is composed of a visible proton emitting photosphere and a corona composed by plasma. The solar magnetic field is manifested by sunspots, relatively cold spots where there is a concentration of magnetic field, unlike the Earth, the Sun has multiple magnetic poles **Figure 2.12** .

The appearance of new sunspots is a prelude to a period of high solar activity leading to Coronal Mass Injection (CMEs), solar flares, prominences and coronal rings. These activities in turn depend on the sun's 11-year cycle; during the first 4 years we have an inactive sun with a minimum number of sunspots and in the remaining 7 years we have an increase of the activity with many sunspots.

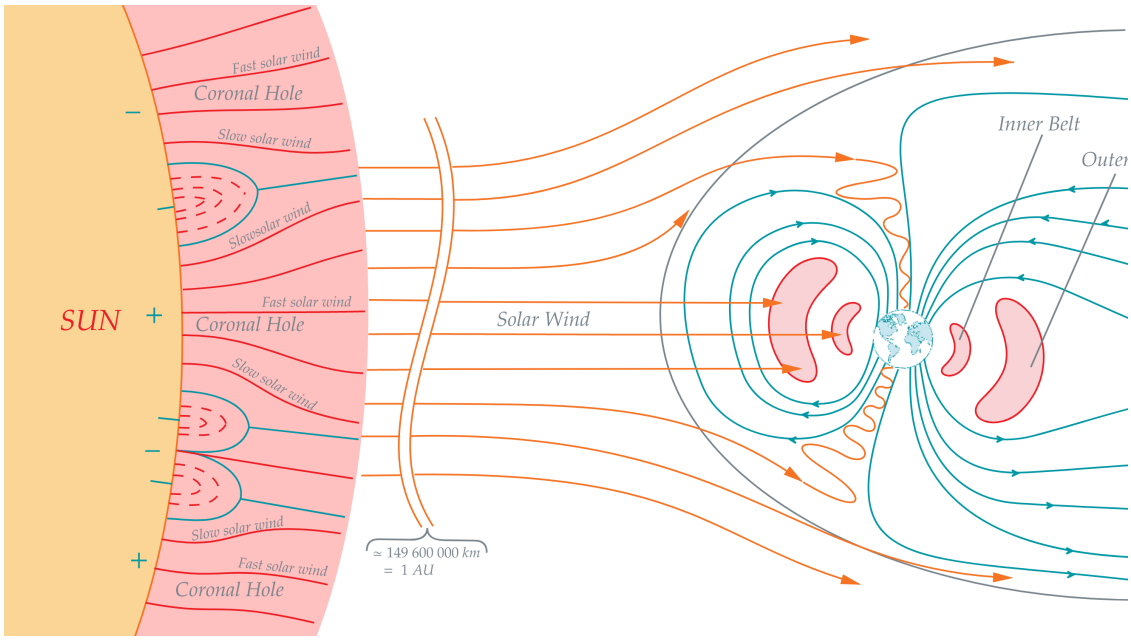

Figure 2.12: The figure shows the variation of the Earth's magnetic field due to the solar wind. The Van Allen Belts are also highlighted in light red on either side of the Earth. The Corona of the sun show how magnetic field changes particle ejection.

The emitted particles are mainly photons, protons, electrons, alpha particles and a small number of heavy ions, all of which are energized and ejected at about 400-700km/s out of the sun. The ejection is due to the high temperatures (6000K) and the inability of the Sun's gravitational force to hold the particles because it is too weak at that distance from the nucleus.

The solar wind at a distance of 1AU from the Sun ($149\,597\,900\,km$) strikes the Earth with $500 \cdot 10^6\ particles/(cm^2\,sec)$ at a speed of 300-450 km/s, the average kinetic energy of protons is about $1keV$, while for electrons it is $10eV$ [24]. Because of the low kinetic energy of the solar wind, it is normally trapped in the Van Allen Belts or deflected by the Earth's magnetic field. But when phenomena such as solar flares, CMEs and prominences occur the energy of the ejected particles is higher and particles with $E > 1GeV$ can be detected on the ground, these energized particles are called Solar Energetic Particles (SEPs) and have energy of $1\,MeV$ to $1\,GeV$.

The main problem with SEPs is that over a period of a few hours or days they have a very high flux of up to an excess of $500\,000/(cm^2\,sec)$, so solar activity can create serious problems for space mission electronics.
When the solar wind reaches the Earth, it changes the Earth's magnetosphere as shown in **Figure 2.12**. The Earth's magnetic field then deflects much of the solar wind, and the particles that manage to enter the atmosphere (the SEPs and GCRs) collide with hydrogen and oxygen to form a cascade of particles that make up the secondary cosmic rays.

As they descend into the atmosphere, Secondary Cosmic Rays continue to collide with nuclei in the air, until they arrive attenuated on earth as Terrestrial Cosmic Rays. The same thing happens to GCRs that normally have higher energies and flux.

Manufacture and Package materials Radiations The materials used to build the die and package have radioactive impurities that release alpha particles due to natural decay to more stable atoms. For example, ^{232}Th decays by emitting 6 alpha particles from 4MeV to 8MeV, while ^{238}U releases 8 alpha particles with similar energy. For example, in the solder bumps there are some isotopes that create a flux from 7 to 0.002 $alpha\ particles/(cm^2\,hr)$ [23].

For these reasons, the primary source of alpha particles for a circuit is the package; in fact, any particle emitted by radioactive impurities can be the source of a SEE, since it is ionized and cannot be shielded.

Today the limit reached by ULA (Ultra Low Alpha) materials is $\sim 0.001\,\alpha/(cm^2\,hr)$, if each alpha particle generated a SEE we would have about one million FITs, but in reality we have only from 1000 to 100 FITs in common chips since not all alpha particles generate a SEE. This is why in terrestrial applications the main cause of error is due to the isotopes impurities of the package since neutrons are few and rarely create SEE. As the altitude rises, the effects are reversed because the neutrons are increasingly energetic and they cause more SEEs.

Medical Radiation Radiation in medicine is used in exams (X-rays) and sterilization (X-rays, gamma rays, e-Beams), normally the maximum observable dose in an examination is 20mSv (millisievert) equal to $2\,rad_{Si}$, this dose is normally harmless even for commercial electronics.
On the other hand for sterilization the radiation is much higher ($\sim\ 5\,Mrad$) making it impossible for even military electronics to survive, so normally if you have electronics in a device to be sterilized, you either use other techniques or switch it off in order to reduce the damage.
In fact the TID depends very much on the electric field, which is absent if the circuit is switched off [23].

Nuclear Power Plants In nuclear power plants and industrial environments, there are sources of X-rays, gamma rays, e-beams and neutrons.
TID effects are the main effects on electronics, although in particular applications (such as measuring the temperature of the cooling ponds of nuclear reactors) the electronics are subject to too much radiation (even for hardened circuits) and must therefore be replaced periodically to prevent deterioration.

Nuclear Weapon The effects of a nuclear explosion depend on the location of detonation and on the power of the bomb. Many nuclear bomb experiments are carried out in the air, the Hiroshima bomb itself detonated at an altitude of 580m, and some explosions have occurred in water and soil.
For an air blasting, immediately after the explosion is formed a fireball filled with strong radiation and temperatures of $10\,000°C$, it expands over a 1km radius for Megaton. The fireball in turn creates a pressure wave that reaches 5 to 10 psi and speeds up to $1000km/h$, reaching a distance of 5-7km for Megaton. Thus about 50% of the bomb's energy is converted into the explosion, while 3% becomes thermal energy which heats the explosion site and can explode fuel reserves up to 10km away per Megaton.

The initial radiation in the fireball makes up about 5 % of the bomb's energy and is composed of gamma particles (at the speed of light (c)), X-rays and neutrons (at 15 % of speed of light with $12.14MeV$). After the initial explosion, 35% of the energy is converted into Fallout, a residual radiation composed of secondary fission products and neutron-activated products that fall out of the atmosphere for weeks after the explosion [23].

Another very important effect for the circuits is the EMP generated immediately after the explosion, in fact the radioactive emission reacts with the atmosphere, the ionosphere and the magnetic field in three different phases: in the first phase there is a short pulse of a few nanoseconds caused by the hydrogen and oxygen ionized by the Gamma particles, followed immediately by the second phase with a pulse of about 1sec produced by the reflected Gamma rays and by the reactions of the neutrons with the atmospheric nuclei in the air. Finally, the last pulse is formed by the radiation ionizing the upper ionosphere and distorting the magnetic field. This variation in the magnetic field couples with the energy transmission lines, creating strong pulses in the distribution network, which destroys devices and transformers and causes extensive damage [23].

The circuits involved in a nuclear explosion, depending on the distance of the epicentre, may suffer all or some of the above effects.

Radiation Effects on ICs

As far as *nuclear radiation* and *Cosmic Rays* are concerned, there is a bombardment of the IC with Alpha or Neutron particles, which penetrate the material and release energy in the form of electron-hole pairs. Depending on the energy of the colliding particles and the sensitivity of the circuit the generated charges can cause a bit flip or soft error.

The sensitivity of the circuit is expressed in *Critical Charge*, which is the charge required in a circuit to create a bit flip. The energy of the particle is referred to as *Stopping Power (SP)* that is the energy lost per unit length by the trace left in the material by an Alpha Particle, it is measured in $eV/\mu m$.
The *interaction mechanism of Alpha particles and Neutrons* is different: Alpha particles directly generate electron-hole pairs (this is why the SP refers to Alpha particles), while Neutrons interact with the atoms of the material in an elastic or anaelastic mode.

The most dangerous interaction is the anaelastic one, because Neutrons decay into other particles (Alphas, Pions, Muons, Neutrons, Deuterons and Tritons) which in turn generate charges in the material. Normally the particles generated by Neutrons have a higher Stopping Power than Alpha particles and lower penetration ranges. Because of this, Neutrons

generate a high charge for a short time (hence high current pulses) while Alpha particles create a charge streak that lasts longer (creating low but prolonged currents)[11].

In the case of Neutrons impact, an example of how the Soft Error Rate can be modelled is the following [25]:

$$SER_{circuit} = K\,\phi_{Neutrons}\,A\,e^{\frac{Q_{crit}}{Q_{coll}}} \tag{2.11}$$

Where K is a constant depending on the technological processes, $\phi_{Neutrons}$ is the Neutron flux, A is the area of the IC involved, Q_{crit} is the Critical Charge and $Q_{coll} = collected\ \ charge\ /\ generated\ \ charge$ (the ratio between the collected and generated charge per unit volume). From the formula 2.11 it can be seen that as the critical charge decreases, the SER of the circuit increases; there is also a linear dependence with the area and the neutron flux.

The effects of radiation on the components concern the various types of problems that generate the physical mechanisms explained above. They can be divided into Cumulative Effects and Single Event Effects, the former is caused by continuous exposure to energized particles and the latter is due to the effects of a single particle collision. The effects of each group are described in detail below.

Cumulative Effects CEs The cumulative effects of radiation cause progressive degradation of the components, in fact the exposure to primary and secondary cosmic rays generates long-term changes in the ICs, these defects lead initially to component degradation and subsequently to faults.
There are three main cumulative effects:

- **Surface Charging Damage Effect** (SCDE) :The charges generated by an energy particle can accumulate inside an insulating material in the IC and if the phenomenon continues, they generate electrostatic discharge (ESD). Normally an ESD create noise, bit-flip, latch-up and false signals [26]. The more energized the particles, the more frequent this phenomenon occur.
- **Total Ionizing Dose** (TID) : In this case the charges created by the particles are deposited in the bulk or other active parts of the IC such as the gate, these charges lead to degradation of the V_{th}, Leakage currents and timing skew. The TID is expressed in Gray (Gy) or rad ($100rad = 1Gy$) where $1Gy = 1j/kg$, normally in space or avionics missions the typical received TID varies from 1 to 100 $krad_{Si}$ [14]. It usually depends on the orbit, shielding and many other factors that vary the incident radiation on the chip.

- **Total Non Ionizing Dose** (TNID) : TNID is that portion of particles that do not create electron-hole pairs but instead directly apply a momentum to the semiconductor material. This energy applied to the lattice crystal is transformed into defects and variations from the crystal shape. In turn the degradation of the crystal structure leads to degradation in the parameters of the component, especially in optoelectronic systems [14].

These effects occur mainly in the avionics and space environment where the particles are more energetic and their flux is orders of magnitude higher than on earth.

Single Event Effects Single Event Effects are due to the charges deposited by the particles, SEEs can be either temporary or permanent effects. They are divided into Destructive SEEs that generate permanent damage in the circuit and Non Destructive SEEs that cause damage reparable with fault tolerance mechanisms or by a system reboot.

There are four principal Non Destructive SEEs:

- **Single Event Transient** (SET) : This event is a temporary voltage change in a node of an integrated circuit, it is caused by a single particle releasing charges as it penetrates the material. SEUs, SEFIs and other spurious phenomena can be generated by a SET.
- **Single Event Upset** (SEU) : The SEU is an event that corresponds to a bit-flip of a memory element: a latch, a Flip-Flop or e.g., the cell of a flash . If the corrupted memory is not used or is corrected by ECC, it is called a Silent SEU. The probability of a SEU depends very much on the critical circuit charge, the supply voltages and the size of the transistors.
- **Multiple Cell/Bit Upset** (MCU, MBU) :Both MCU and MBU are caused by the corruption of the value of two or more adjacent cells by a particle. The difference is that MCU occurs between cells of different words while MBU occurs between cells of the same word. This difference is substantial because in a memory with ECC that can correct only one bit, MCUs are corrected while MBUs can't be corrected because they cause two or more errors in the same word.

  These phenomena are increasing in new generations of memories because the proximity between cells continues to grow [14].
- **Single Event Failure Interrupt** (SEFI) : This event is defined as the soft error that causes a reset or stall of a circuit component or the whole system [14]. It is usually

caused by corruption of control memory or program memory, by communications disturbances and internal control signals [27].

There are also three different types of SEFI, some can be repaired with a software reset, other need power cycling due to a stall and some need partial reprogramming due to corrupted program data.

Destructive SEEs are more technology dependent and they is divided in [14]:

- **Single Event Latchup** (SEL) : This event occur when the parasitic PNPN or NPNP thyristor of the CMOS structures are turned on. When this happens and the power supply is on, the component can be destroyed by thermal effects. This mechanism don't exists in SOI systems because there are no parasitic thyristor.
- **Single Event Snap Back** (SESB) : This event occurs when NPN or PNP parasitic bipolar structures in CMOS circuits are activated. These parasitic transistors can self-sustain a current that can be destructive. SOI technology also suffers from this effect because parasitic transistors are present in these systems.
- **Single Event Hard Error** (ESHE also Stuck-bit) : The ESHE or Stuck-bit is a permanent or intermittent modification of a memory element. This applies to both memories and digital circuits. It differs from an ESHE because it is permanent, in the sense that that memory cell can no longer be used by the system after the event.
- **Single Event Gate/Dielectric Rupture** (SEGR, SEDR) : This event indicates the breakdown of a gate oxide or dielectric by a single particle. SEGR and SEDR are dangerous events because they have much faster dynamics than SEL, SESB and SEHE. For this reason, there is no protective circuitry against these events. In any case they are rarer events and occur mainly in the space environment where there are very energetic particles.

2.2.3 Soft Errors

All the possible transient errors analyzed in the previous chapters are Soft Errors, these errors remain in the memory elements (e.g., flip-flops, latch) only until a new value is written.
When fault detection and correction systems are applied to a system, two categories of errors are created at system level:

- **SDC (Silent Data Corruption) :** a faulty bit without detection is read and it modifies the final result of the program.
- **DUE (Data Unrecoverable Error) :** when a faulty bit with only error detection is read. At this point if the bit changes the final result is True DUE, otherwise it is a False DUE.

SDC errors are dangerous because they occur on bits for which errors cannot be detected or corrected, these errors can lead to a system crash and must be transformed into DUE errors by error detection or corrected.

The advantage of converting SDC errors into DUE is that DUE are detectable and they lead the system in fail-stop mode. In fact once a DUE is detected, the system stops and evaluates how to continue execution. At the operating system level, if the error is inside a process we can kill only that one and we talk about process-killer DUE, otherwise we say that DUE is system-killer because the OS has to restart the machine to avoid the propagation of the error.

DUE and SDC errors have different effects on dependability; DUE causes Availability penalties because the system has to recover, while SDC lowers Reliability, Safety and Availability because it can crash the system. For these reasons normally there is a budget of TDC and DUE expressed in FIT, for example 228FIT of SDC (500 years of MTTF) and 57000FIT of DUE (2 years of MTTF) by specification.

Time Vulnerability Factor (TVF) The TVF is the fraction of time in which the circuit is vulnerable to errors. It is calculated using the window of vulnerability (WOV), which is the time within the clock period in which the circuit can be subject to SEE, for example in edge-triggered flip-flops the WOV is equal to half the clock because only in that interval the FF change state if it is struck by a particle (only in the high/low phase is the data sampled and held). The TVF is therefore the ratio between the vulnerability window and the clock period, so for an edge-triggered FF the TVF is equal to 50%.
Actually, the calculation of the TVF is more complicated because the propagation delay of the circuit has to be taken into account, assuming in fact a period of 1ns and an average

combinatorial delay of 700ps, in this case the TVF will be lower than 50% since some faults injected in the first 500ns can be masked by the logic delay, for more details [28].

In this section we have seen the effects of particle strike on ICs now we explore the known methods to avoid system failures due to faults at RTL level.

2.3 Hardening techniques for digital circuit architectures

Hardening techniques are used to improve the dependability of a system and they can be applied at each design layer as described in section 2.1.2. We now focus on digital circuits architecture hardening technique that are used in order to increase the Safe Failure Fraction (SFF) and so increase the SIL of the architecture.

2.3.1 Spatial redundancy

Spatial redundancy consist in use N parallel system which output are voted, this structure decrease the failure rate of the overall system. Indeed when a cosmic ray collide with one of the parallel system and cause an error the other system continue to work properly. Since the probability of a multiple strike is lower than particle flux we have a reduction in cosmic ray sensibility.

Another advantage in the use of redundant system is the increase of MTTF. Indeed using N parallel system each with the same $\lambda = \frac{1}{MTTF}$ we have this new MTTF [16]:

$$MTTF_{Overall\ \ System} = \sum_{k=1}^{N} \frac{1}{\lambda k} \tag{2.12}$$

This formula highlight the increase in MTTF due to the use of parallel systems, this means that it is able to protect also from permanent fault in an ideal case. Anyway not all spatial redundancy techniques increase the MTTF, this normally is not a problem since in safety-critical application since the system is replaced much earlier then its MTTF.

All Spatial Redundancy techniques can be used both in low level architecture for example an FSM and in high level architecture such as cores. The first use is called *Fine Grain Redundancy* while the second one is called *Coarse Grain Redundancy*.
We have shown that general parallel system is able to protect mainly by transient faults and secondarily by permanent faults. In the next paragraph we analyze the main kind of spatial redundancy techniques from architectural point of view. Any of this systems is based on the general M-of-N system that is composed by N modules and needs at least M of them to work properly.

Double Modular Redundancy (DMR)

An example of Double Modular Redundancy is shown in **Figure 2.13**, the basic idea is the use of two identical blocks. When there aren't faults OUT1 and OUT2 are equal and so

the output will be OUT1, instead when the two replicas have different outputs is selected the Safe Output and is asserted an Error Detection.
The assertions of the Error Detection signals are normally handled at software level, here should exist a routine that enable recovery of the previous state and a new execution of the faulty step.

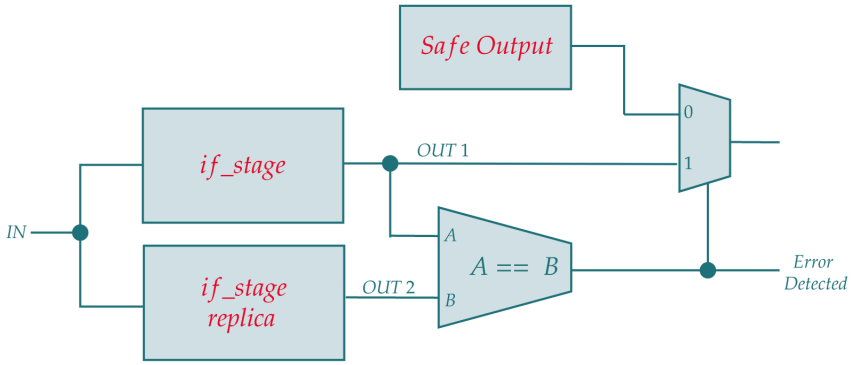

Figure 2.13: Example of DMR using an if stage of a core

DMR add time penalty when a fault occur because normally the execution should be repeated to obtain the correct output, indeed DMR is only able to detect a fault.
In **Figure 2.13** we use the if_stage of a processor but DMR can be used at whatever level, e.g., at core level using two identical core in parallel.

Triple Modular Redundancy (TMR)

In **Figure 2.14** there is an example of TMR using the if stage of a core, as you can see are used three replicas of the same stage working in parallel, the outputs from the stages are voted to find correct values using majority voters.

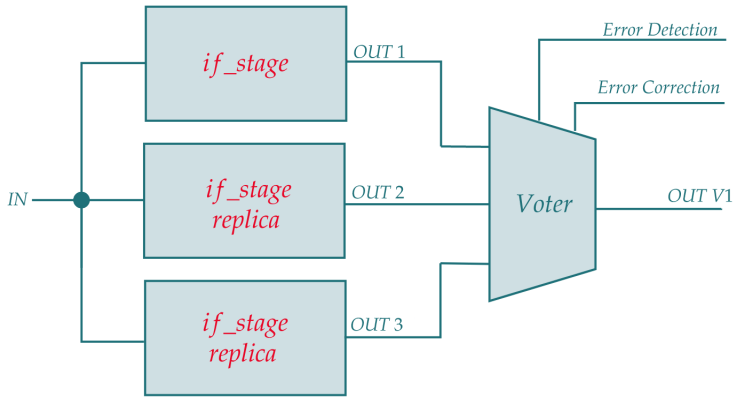

Figure 2.14: Example of TMR using an if stage of a core

This resilient structure is able to detect and correct only one fault, when two error occur in two replicas at the same time this structure isn't able to correct the faults but only to detect it. These are some possible cases of errors:

- **One error in one replica:** Assume that the fault is in the first replica and so OUT1 is wrong, the voter sees $OUT2 = OUT3 \neq OUT1$ and so it asserts $OUTV1 = (OUT2 = OUT2)$. At the same time is asserted *Error Detected* and *Error Corrected* signals. So in this case the error is detected and corrected.
- **Two parallel errors in different replicas:** In this case $OUT1 \neq OUT2 \neq OUT3$ and so the voter is not able to detect and correct the output, anyway it asserted *Error Detected* and negates *Error Corrected* signal, in this way high level software or architecture sees that an error occurs. Normally these types of errors are rare in terrestrial environments, since particle flux is low, anyway in the case there is already a permanent error in one replica a SEE would be able to stress such an error.
- **One error in majority voter:** In this case the output will be wrong due to voter errors, this raises questions about voter contribution in the final Safe Failure Fraction. Normally voter is smaller respect to the stage area and so SEEs are low in absolute number, anyway when we use fine grain redundancy we should consider the voter contribution.

The Reliability of a TMR system can be calculated using this formula [16]:

$$R_{TMR}(t) = R_{voter}(t)\left(3R^2(t) - 2R^3(t)\right) \tag{2.13}$$

Where $R_{voter}(t)$ is the reliability of the voter and $R(t)$ is the reliability of the stages (each stage have the same reliability). As you can see in **Figure 2.15** the reliability of the voter

is important such as stage reliability, as we already mentioned this difference in reliability is assured by the area and complexity difference.

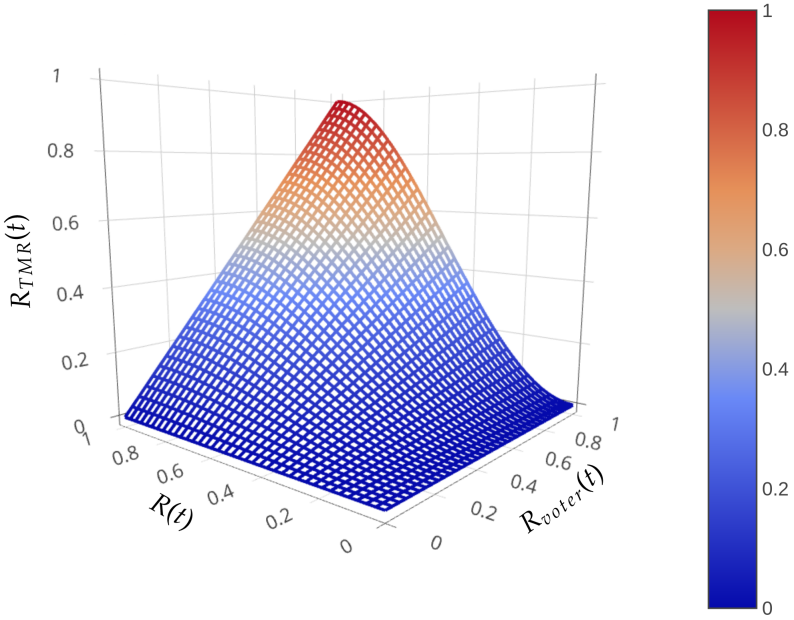

Figure 2.15: Plotting of equation 2.13

Now integrating equation 2.13 between 0 and ∞ we obtain the MTTF [16] (we consider R_{voter} about 1):

$$MTTF_{TMR} = \frac{5}{6\lambda} < MTTF_{simplex} = \frac{1}{\lambda} \tag{2.14}$$

The $MTTF_{TMR}$ is lower then $MTTF$ of simplex system, this is due to the fact that is enough for a stage to break to cause an error. However the MTTF difference is very small, we should also consider that in safety critical application is usually more important the reliability respect to the MTTF since the system is replaced far before R(t) < 0.5 [16]. Usually the real MTTF and the Reliability are higher then calculated since we should take into account fault compensation and errors that are masked by the system [16].

We have seen some advantage and limitation of basic TMR, for the large part of the application TMR is a good solution but sometimes we need an higher level of reliability, in these cases are used TMR with voter triplication or diversity redundancy.

Voter Triplication At first glance voter triplication seems to be useless, indeed looking at **Figure 2.16** we have three output from the voters, these three output should be newly

voted to find the correct output. Anyway voter triplication is used to connect two TMR stages as depicted in **Figure 2.17**, in this case the use of three voter is useful.
Indeed when one of the voters have a fault the error propagates only in the next stage replica, this error is then deleted by the voter at the output of the system. It is clear here that the use of triplication is useful in *fine grain* application in which many stage are protected and connected together. In the architecture of **Figure 2.18** an error in the middle voter propagates in all next stages without any possible correction, this case is avoided by voter triplication.

The main difference between architecture of **Figure 2.17** and **Figure 2.18** is the occupied area, in fact if you have a high number of output from each stage the triplication of voter imply an high number of interconnections and area, this reduce the speed of the circuit and increase the power consumption.

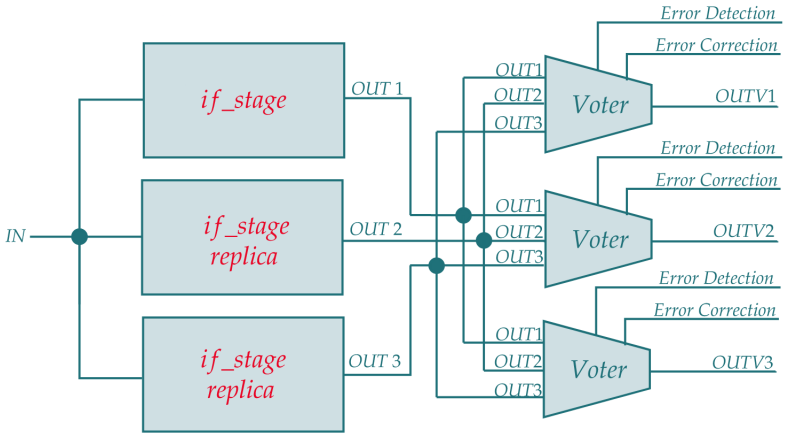

Figure 2.16: Example of TMR with voter triplication in only one stage

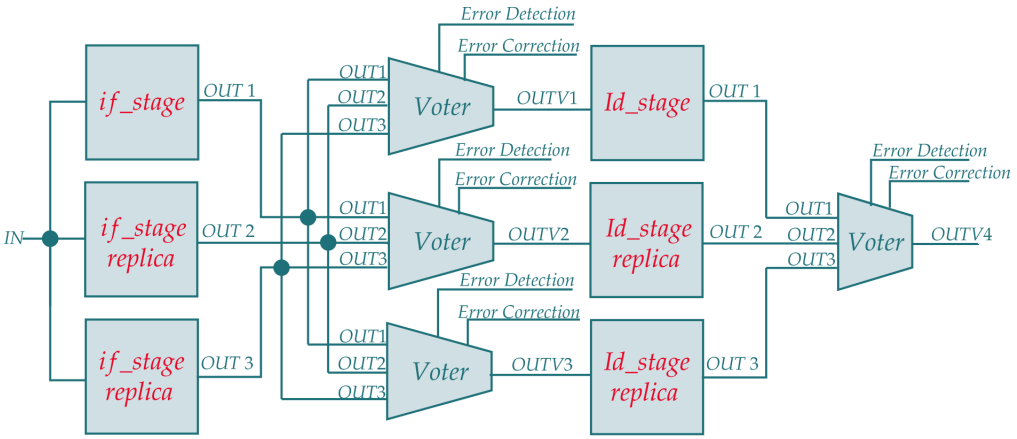

Figure 2.17: Example of TMR with voter triplication used in stages connection

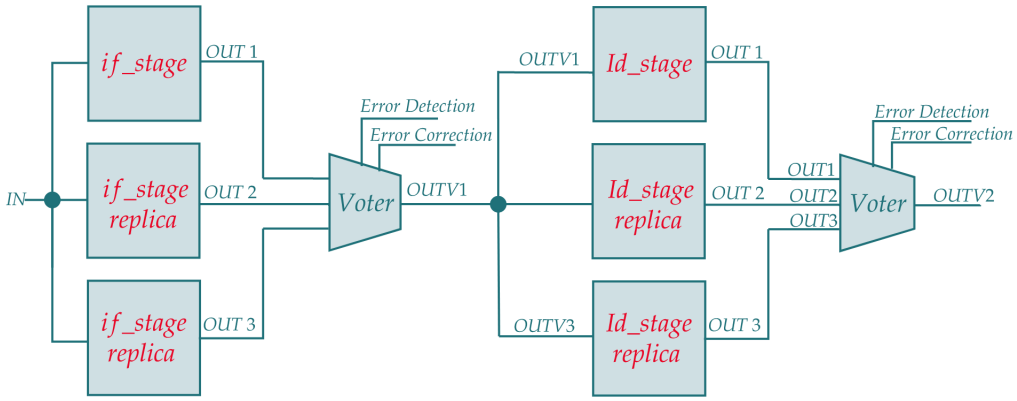

Figure 2.18: Example of TMR used in stages connection, the voter is the weak link of the chain.

Another application of the triple voter is writing into memory, indeed when we have a triplicated memory we could use triple voter, in this way a faulty voter write only a wrong value while other memory are correct. The same argument can be done for memory read, in this case the use of three voter avoid a wrong reading.

Diversity Redundancy Apart from SEEs and permanent faults there are implementation errors. These types of errors are due to designer implementation mistake but they can be reduced using Design Diversity in redundancy [29]. An architecture that implements TMR and uses design diversity is shown in **Figure 2.19**, as you can see the three parallel stages are designed independently by different teams, this decrease the implementation

errors because the probability that two independent teams commit the same mistake is low.

Diversity redundancy is also able to increase the reliability as shown in this reaseach [30], the paper evaluates the DTMR (Diversity Triple Modular Redundancy) using an FPGA hit by a neutron flux of $3.98 * 10^4 n/cm^2/s$ (standard deviation of $3.74 * 10^3 n/cm^2/s$) with an energy of 10MeV for 1268 minutes. The average number of upsets detected was about one per minute. It was found that the DTMR approach is better than TMR because each redundant block has a different reliability.

This technique increase the cost of the design. Indeed are needed three independent teams that work at the same architecture, this is an extra cost that may be paid for high reliability request, for example a system that should have the maximum SIL.

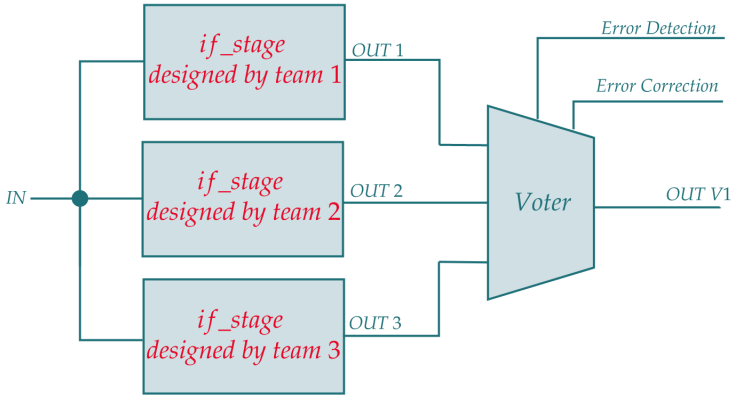

Figure 2.19: Example of design diversity used in conjunction with TMR technique in order to protect an IF stage.

N Modular Redundancy (NMR)

NMR is a more general set of all techniques that use many parallel block to increase reliability, these system are also called N-of-M where M is the number of redundant system and N is the number of system needed for proper operation. In order to avoid a deadlock M should be an odd number. As we have seen TMR is a typical NMR system but exist also NMR system with M=5 and upper number, anyway these type of architecture are rarely used due to the high power and area increase, indeed it is known that an increase in the power consumption increase temperature which in turn increase the wearout and probability of SEEs due to particle strike.

Assuming that all system used in NMR are statistical independent, the general formula of

reliability for a NMR system is [16]:

$$R_{N\ of\ M}(t) = \sum_{i=M}^{N} \binom{N}{i} R^i(t)[1 - R(t)]^{N-1} \tag{2.15}$$

The independence of the parallel blocks is essential and lead to an high reliability, for these reasons common mode failures are critical for these system. Integrating the reliability we find the MTTF of system, as we have seen for TMR the value of MTTF is slightly lower then for simplex architecture and this is also true for all NMR system. For these reason NMR system are really good in terrestrial application where human is able to maintenance devices and replace end-of-life parts. When we design an architecture for long space mission we can opt for an Hybrid Redundancy.

Even if for NMR is generally used an odd N number, a 2020 research [31] investigate the use of Quadruple Modular Redundancy (QMR). The paper proposes an approximate QMR redundancy method where are used three instances that approximate the true result and one that finds the exact one. A vote on the 3 approximate architectures is done first and then the result is voted with the exact architecture. The difference with the other works of approximate redundancy is in the use of approximators equal to each other while in the other works were used all different approximators.

Hybrid Redundancy

As we have seen TMR and NMR increase reliability but slightly decrease the MTTF of the system respect to simplex architecture. This can be a problem in space application where maintenance and replacement are not possible, for these application we can use Hybrid redundancy. As you see in **Figure 2.20** this technique starts from a NMR architecture adding k spare blocks. During normal operation the N output of the replicas are voted, when a primary block become faulty it is switched off and replaced with a spare block.

Compare block compares active blocks outputs with correct output and implements a faulty block detector for example using an Alpha Counter for each block. When the Alpha Counter of a block exceeds the critical level, the faulty block is disconnected and replaced with a spare working block. This mechanism implements a self-maintenance system in which end-of-life blocks are replaced, in this way MTTF and reliability of the system are both increased.

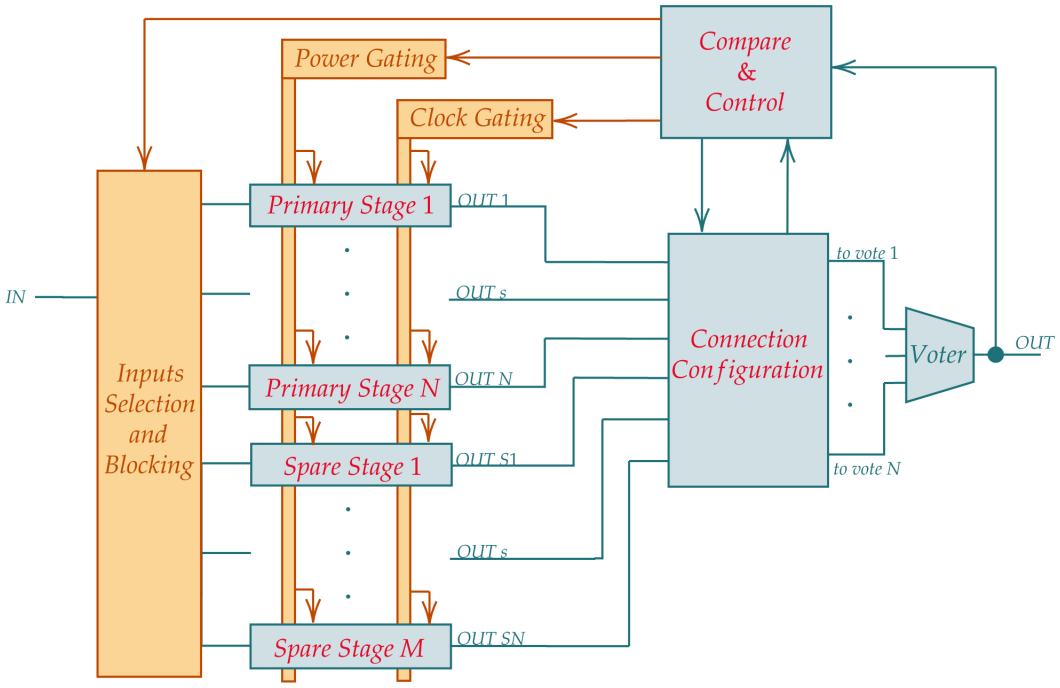

Figure 2.20: Example of generic Hybrid Redundancy with Power gating, clock gating and input selection and blocking.

Spare and primary blocks create a large number of elements to power and control. The increase of power consumption increase the probability of faults [32] and this leads a performance degradation. To reduce the power consumption designers use clock gating , power gating and input blocking in unused blocks, in this way are reduced useless switches. These methods reduce TID effects since without an electric field the Vth variation is low, this leads to an higher spare block life and an higher system MTTF.

Clock gating, power gating and input blocking are controlled by compare and control block that should maintain the should also maintain the history of faulty blocks. Problems arise when the system is reset, in these cases there are few methods to recover informations about block status:

- **Common and Status Registers:** These registers are used to store information about hardware and they are readable by software. When a new faulty block is detected the information should saved in one of these registers, then a software level routine can move this data in a non volatile memory. When the system is rebooted data from non volatile memory are loaded into CSR and then uploaded into Compare and Control block of each system we have;
- **Test Vector:** CSR methods is a little complex when we have a high number of Hybrid

architecture, it also needs a software procedure. For this reason some designers prefer to avoid CSR methods using a test vector after reboot [33]. The system works on this test vector and finds faulty blocks, indeed each Compare and Control blocks correctly configure each hybrid system looking for faulty blocks, at the end of this procedure the architecture is ready to start in safe mode.

2.3.2 Information Redundancy

Inside a whatever core are used memories to store information, unfortunately also memory experience errors. For these reason there are coding techniques used to detect or/and correct data errors, coding is the most common technique information redundancy. According to this technique are added some check bits to the original data, this extra bits are stored with the data and are used to verify the correctness of data.

The information redundancy process is summarized here:

- **Code creation:** The process start when a certain data with A-bits should be written into memory, it is encoded into B-bits codeword. the codeword can be *separable* if check bit and data are not mixed together, so a separable codeword will have A-bits for data and C-bits for check bits where A-bits + C-bits = B-bits;
- **Data degradation:** The data stored may be modified by SEEs, wear out process and any kinds of possible faults. Statistically the longer the data are not read, the easier they became corrupted due to some faults. This corruption change with memory, technology, temperature, environment etc. and can't be precisely calculated;
- **Data reading:** When data are read their correctness should be verified. At this step a certain block uses the codeword written in memory in order to detect and/or correct existing faults.

The most common code is the Parity Bit, the Hsiao code, the Hamming Code and the Cyclic Redundancy Check (CRC).

2.3.3 Temporal Redundancy

The idea of temporal redundancy is to repeat an operation N times and compare results. This decrease the area of a safety architecture but also decrease the performance, as you can see in **Figure 2.21** a certain operation is executed three times, the outputs are stored and at the end of computation they are compared to find correct output. This techniques is effective at processors level, where spatial redundancy adds too much overhead, using

this techniques can be used only one core plus some external logic.

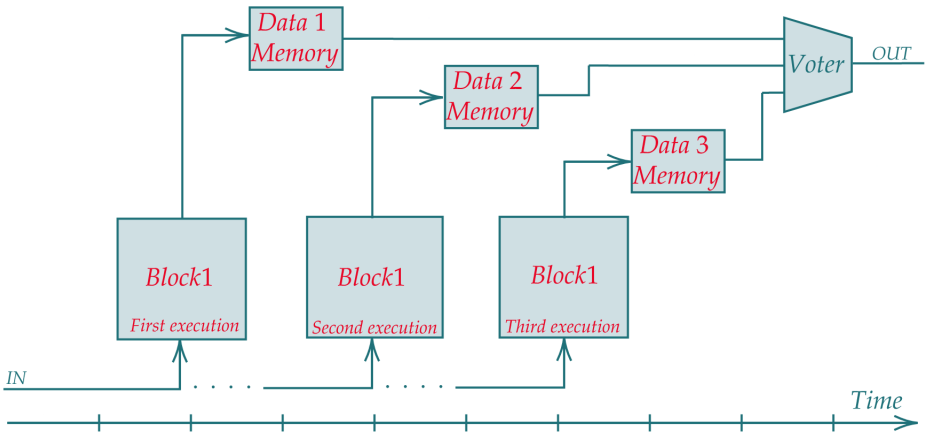

Figure 2.21: Example of generic Temporal Redundancy.

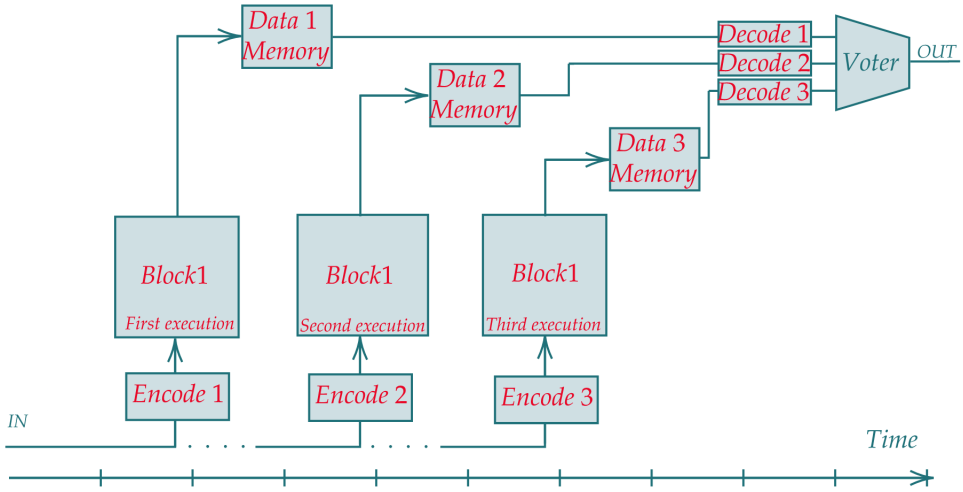

Figure 2.22: Example of generic Temporal Redundancy with data encoding.

All previous techniques can be applied together and in mixed way in order to increase Reliability and MTTF of the overall system. The implementation of these techniques should be done considering power, area, performance and safety requirements. This trade off led to refine the techniques used to decide where and when use a specific protection, for example a 2012 research [34] investigate the use of genetic algorithm to find the best TMR configuration, in this case the triplication of blocks or/and voters is done in order to optimize area

and reliability.

A 2010 work create a software supported frame-work that is able to improve the TMR application basing on the temperature change in the chip, the methods consist in the application of TMR to all processor, then the core is Place and Route to FPGA and the temperature distribution is analyzed. Since temperature distribution give guidelines about where it is most likely faults to occur, it is possible to eliminate redundancy from parts of the design that operate under low temperatures without affecting fault masking.

In this Master Thesis we focused on the automatization of the FT design and for this reason we use a basic TMR technique plus a permanent fault detection.

Chapter 3

RISC-V and CV32E40P Core

RISC-V is a free and open Instruction Set Architecture (ISA) with a small instruction set (Reduced Instruction Set Computer) at the heart of core of a System On Chip (SOC). An ISA is the abstract description of core instruction, registers, data types and extension, without design imposition. Many implementation of a CPU can rely on the same ISA using different design, in this way software used in these different implementation could be equal. RISC-V can be seen a first try to standardize the Instruction Set world without using new technology. The focus is on modular approach and extensibility in order to increase field of use of the ISA.
For these reasons the use of a free and open ISA worldwide unlock partially software from hardware implementation since the interface is always about equal. This subdivision speed up computer CPU progress with lower efforts in software stack support. Indeed companies efforts can be focused more on design and less on software support since the interface is the same.

RISC-V is maintained by the RISC-V foundation born in 2015, it is driven by open collaboration in order to improve RISC-V ISA. The use of free and open collaboration speed up bug resolution, reduce design risk and lead to speed up in design techniques. In order to be used worldwide the RISC-V architecture use a Reduced Instruction Set of 47 Base Instruction, the ISA is designed with a modular approach to easily add extensions. In this way the ISA can be used for whatever application since the core can be customized according to application and field of use. It is already used in computer, supercomputer, embedded application and it now supports by many Operative System like RTOS and Linux.

It is precisely the fact that it can be customized that increase the worldwide use. This increase the test done on the ISA implementation which lead to ISA improvement and increase the ISA dependability.

Starting from these ideas we could summarized some benefits of a open and free ISA [35] :

- **Greater Innovation:** More people working on the same ISA speed up innovation;
- **Shared Open Core Design:** with a free ISA the implementation (e.g., System Verilog design) can be shared as open source, this prevent malicious traps doors and increase transparency, it also allows the born of new industries that create products from open design;
- **Cheaper Processors:** the reduction of company work on software stack decrease the processors cost, this speed up widespread of IoT;
- **Longevity of Software Stack:** a standard ISA allow the creation of an endurance Software Stack since it don't depends to a company;
- **Architecture Education and Research closer to real application:** the academic world could work on open hardware and software.

At 2020 the 23% of ASICs and FPGAs project incorporate at least one RISC-V core and it is foreseen that in 2025 will be used 62.4 Billion of RISC-V Cores respect to 10 Billion of 2020. This is surely a positive sign for open ISA strategy, it means that many industries are rely on RISC-V ISA architecture and they use it to speed up their internal design.

3.1 History

RISC-V ISA design started in 2010 in a project for the Parallel Computing Laboratory (Par Lab) at Berkeley held by Prof. Krste Asanović and graduate students Yunsup Lee and Andrew Waterman. The idea of RISC-V ISA born to create a complete open hardware ecosystem, indeed until that moment the main ISA was proprietary and academic world work on non realistic architecture. Anyway the outcome of an open ISA also help industries since the main costs an complexity in the creation of a new chip is the development of software stack for new ISAs. Indeed any change to the ISA means redevelop some parts of software with high costs. Due to historical reason ISAs are proprietary, but in the last 40 years no meaningful development arise in this field and so there aren't no meaning to avoid ISA standardization. Starting from this ideas the RISC-V project begin and was developed up to now.

The first release of RISC-V ISA was in 2011 and it was under the Berkeley Software Distribution (BSD) license. After some year of RISC-V use and some publication [35] in 2011 was created the first chip in 28nm FDSOI and in 2015 was held the first RISC-V workshop, in the same year was founded the RISC-V foundation with 36 members [36].

In the following year the RISC-V ISA was put under Creative Common license in order to enable an easy use and open contribution.

In the year 2013-3018 in the UC Berkeley ASPIRE Lab was created many RISC-V compatible free processors, today RISC-V Foundation continue to support RISCV ISA standardization helping industries to use it, the versions of the ISA is now frozen at 2019 in order to simplify development of RISCV core. These are the official ISA standard: RISCV-privileged and RISCV-unprivilged. Instead in this page you can find all original standard documents: RISCV-spec.

3.2 RISC-V ISA

RISC-V ISA is described in two document:

- **riscv-privileged:** Or Kernel mode, in this document are described privileged instructions, any attempt to execute this instruction from User Mode will not be executed and it is considered illegal instructions. These are the instructions used in the Operative System to perform operations.
- **riscv-unprivileged:** Or Non privileged mode, it is made by all instructions that can be run only in user mode.

In these two documents the ISA is described avoiding implementations details as much as possible. At the beginning of the standards are defined some terms [37]:

- **core:** An architecture is defined a core if it contains an instruction fetch;
- **harts:** Are hardware threads that can be support by the ISA, a RISC-V compatible core can supports multiple harts;
- **coprocessor:** It is an instruction set extension used by the RISC-V compatible core, this coprocessor is considered as a separate unit that is controlled by a RISC-V instruction flux, but it have relative autonomy respect to primary RISC-V core, this is an example of RISC-V coprocessor programming for sensor reading RISC-V sensors;
- **accelerator:** This component are really useful to perform specialized complex task, e.g., I/O and AI accelerator, the first manage I/O processing task while the second could be a voice recognition AI.

As you can see RISC-V can have extensions and coprocessors, it can also have many system-level organizations; single-core, many-core shared memory and so on.

A RISC-V ISA implementation must contain base integer ISA, starting from this basis can be added optional extensions. The base integer ISA is enough to support a compilers, assemblers, linkers, and operating system, in this way it provide a starting point to custom implementations.

The RISC-V ISA is divided in four base ISA family and it is "designed to support extensive customization and specialization" [37], each family is distinguished by a different register width and the number of corresponding address space. The four family are RV32I, RV32E (with 32 bits) RV64I (64 bits) and RV128I (128 bits), naturally 128 bits is a huge number of address, anyway RISC-V standard want to be prepared for future huge computer address. These four ISA are considered four base ISA, in this way e.g., RV64I shouldn't be compliant with RV32I and so have separate implementation.

In addition to base ISA (marked with I suffix for Integer) there are some standard extension:

- **M:** It is the standard integer multiplication and division extension;
- **A:** It is the standard Atomic instruction extension that add some instruction able to "atomically read, modify, and write memory for inter-processor synchronization" [37];
- **F:** It is the standard floating point extension;
- **D:** It is the standard double precision floating point extension;
- **C:** It is the standard compressed instruction extension that provide the 16 bit form of common instructions.

The memory of a RISC is a single byte-addressable address space of 2^{XLEN} bytes, a word is 32 bits, an halfword 16 bits, a doubleword 64 bits and a quadword 128 bits. The memory accesses can be done using *explicit* and *implicit* mode, the first is used by load and store instruction while the second is used in instruction fetch to give the encoded instruction to execute. Only some section of the memory can be used, these sections create the *execution environment*, if an instruction try to read/write outside of this zone, an exception is raised.

Each base instruction for the RISC-V ISA have 32-bit aligned to the 32-bit boundaries, anyway are also supported variable length instruction like 16-bit compressed instruction aligned to the 16-bit boundary, for these instruction is needed the compressed decoder block in the Instruction Fetch stage. The term IALIGN refers to the instruction address alignment of the ISA, it could be only 32 or 16 bit. Instead the term ILEN refers to the maximum instuction lenght bit.

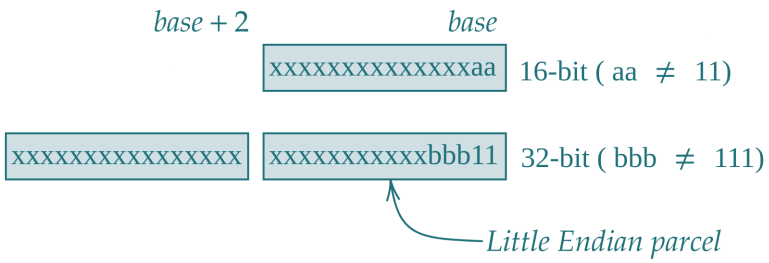

Figure 3.1: 32-bit and 16 bit instruction, frozen in the last release.

The lowest two bit [1:0] on the right are used to recognize a compressed instruction, if they are both 1 the instruction isn't compressed, othewise it is compressed.
Even if the RISC-V can have either Little-Endian and Big-Endian memory order, the instruction is divided in 16-bit Little-Endian parcels, the parcels of the same instruction is saved contiguously with the lowest-addressed parcel in the lowest-numbered bits in the instruction.

3.2.1 Base Instructions

We will discuss the RV32I base version since in this Master Thesis we use the CV32E40P core. For RV32I ISA we have 32 unprivileged registers where the first is hardwired to 0 and the others are general purpose, there is also the 32 bit Program Counter register.
The base instructions used are six R/I/S/B/U/J and are shown in figure 3.2, all 47 base instruction is encoded using these 6 formats. As you can see many parameters have the same place, this enable an easier decode of instructions and so increase low power behaviour of possible implementations.

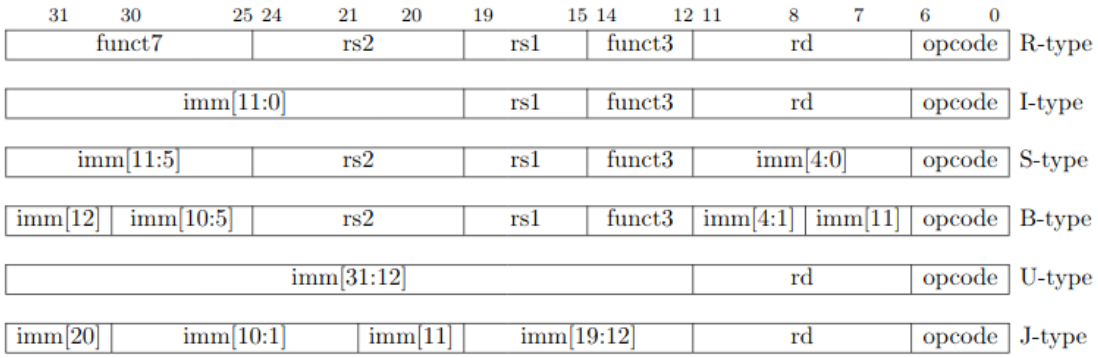

Figure 3.2: RISC-V ISA instruction format.

Inside the table 3.2 in the field you can see: *opcode* that is the operation code, *rd* that is

the number of register destination, *immediate* that is a constant value or the offset for an address, *func3* that is an additional opcode of 3 bit, *func7* that is an additional opcode of 7 bit, *rs1 & rs2* that are the first and the second source register number.

In figure 3.3 there is an implementation of the RISCV ISA.

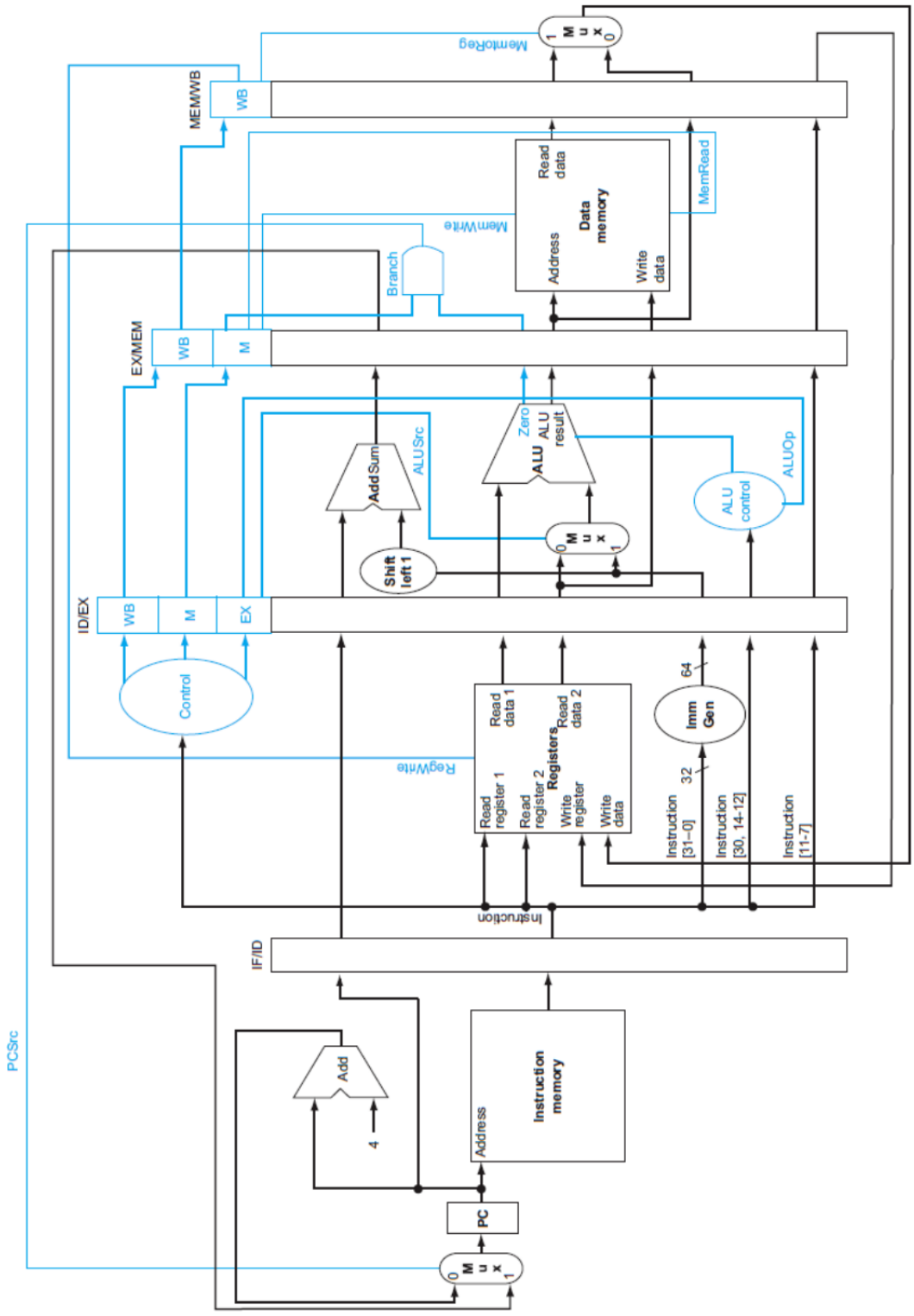

Figure 3.3: RISC-V architecture with four pipeline stage.

3.3 CV32E40P core

CV32E40P is a 32-bit RISC-V core that implement RV32IM[F]C ISA using a custom extension called Xpulp, with this extension the core is able to reach a high code density, performance and efficiency [38] [39]. The core architecture has been designed to work with a Near-Threshold voltage in order to increase the efficiency of the transistors.

For this core are created some instruction extensions and architectural optimization that increase computational density and speed up processing reaching a 3.5x faster elaboration and 3.2x more energy efficiency. This core has been designed also to be used in the Parellel Ultra Low Power (PULP) platform. PULP use a set of RISC-V core with shared memory to reach high efficiency, it contain a set of IP that enable this performances.

The original name of the CV32E40P was OR10N based on the OpenRISC ISA, then in 2016 it change name in RI5CY and it became a RISC-V compatible core, it has been maintained by PULP Platform until 2020 when it begin to be contributed by OpenHW Group.

In figure 3.4 there is the main diagram of the CV32E40P core, it is a 4-stage in order RISC-V processor, the MULT block correspond to the M extension while compressed decoder is the C extension, in the figure there isn't the floating-point unit because it is optional in CV32E40P core.

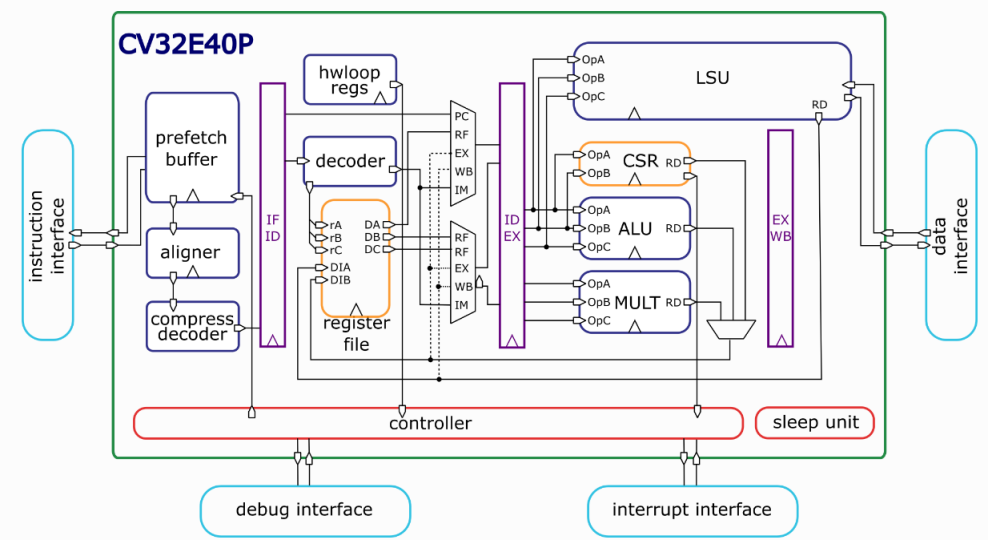

Figure 3.4: CV32E40P core diagram.

In core repository cv32e40p you can find all System Verilog code, documentation and verification procedures. This Master Thesis is focused on the transformation of the Instruction Fetch (IF) of the CV32E40P in a fault tolerant stage, for this reason we will focus on IF stage architecture of the core.

3.3.1 CV32e40P Instruction Fetch

The instruction fetch read the instruction from instruction memory and provide them to the Instruction Decode (ID). As you can see in figure 3.5 the IF stage can be divided into six main block, anyway only the Prefetch Buffer, the Compressed Decoder and the Aligner are designed as separate System Verilog file.
The six block of the IF stage cover these functions:

- **Prefetch Buffer (PB):** PB is designed in order to fetch instruction from instruction memory/fetch via the OBI interface (cv32e40p_obi_interface.sv file), the instruction are then inserted to a four element FIFO (cv32e40p_fifo.sv file) and then they are sent to the ID stage. OBI interface and the FIFO is controlled by the Prefetch Controller (cv32e40p_prefetch_controller.sv file). The advantage of the use of Prefetch Buffer is a higher instruction fetch speed that affects the core performances.

- **Compressed Decoder (CD):** CD check the instruction in order to verify if is compressed, this identification is done using the first two bit of the instruction, when they are both 1 means that the instruction isn't compressed and it is copied to the output, otherwise the instruction is decoded. The stage is completely combinatory since it is described as a big case in System Verilog.

- **Aligner:** This block aligns instructions considering compressed instructions format. It contain an FSM and so FF components.

- **IF pipeline:** This block contain the pipeline of the if stage, it is a System Verilog (SV) block written inside the cv32e40p_if_stage.sv file.

- **IF FSM logic:** This block manage control signals of Prefetch Buffer and Aligner, it is a System Verilog (SV) block written inside the cv32e40p_if_stage.sv file.

- **Program Counter Definition (PCD):** PCD define the PC according to jump, branch, trap and exceptions. It is a System Verilog (SV) block written inside the cv32e40p_if_stage.sv file.

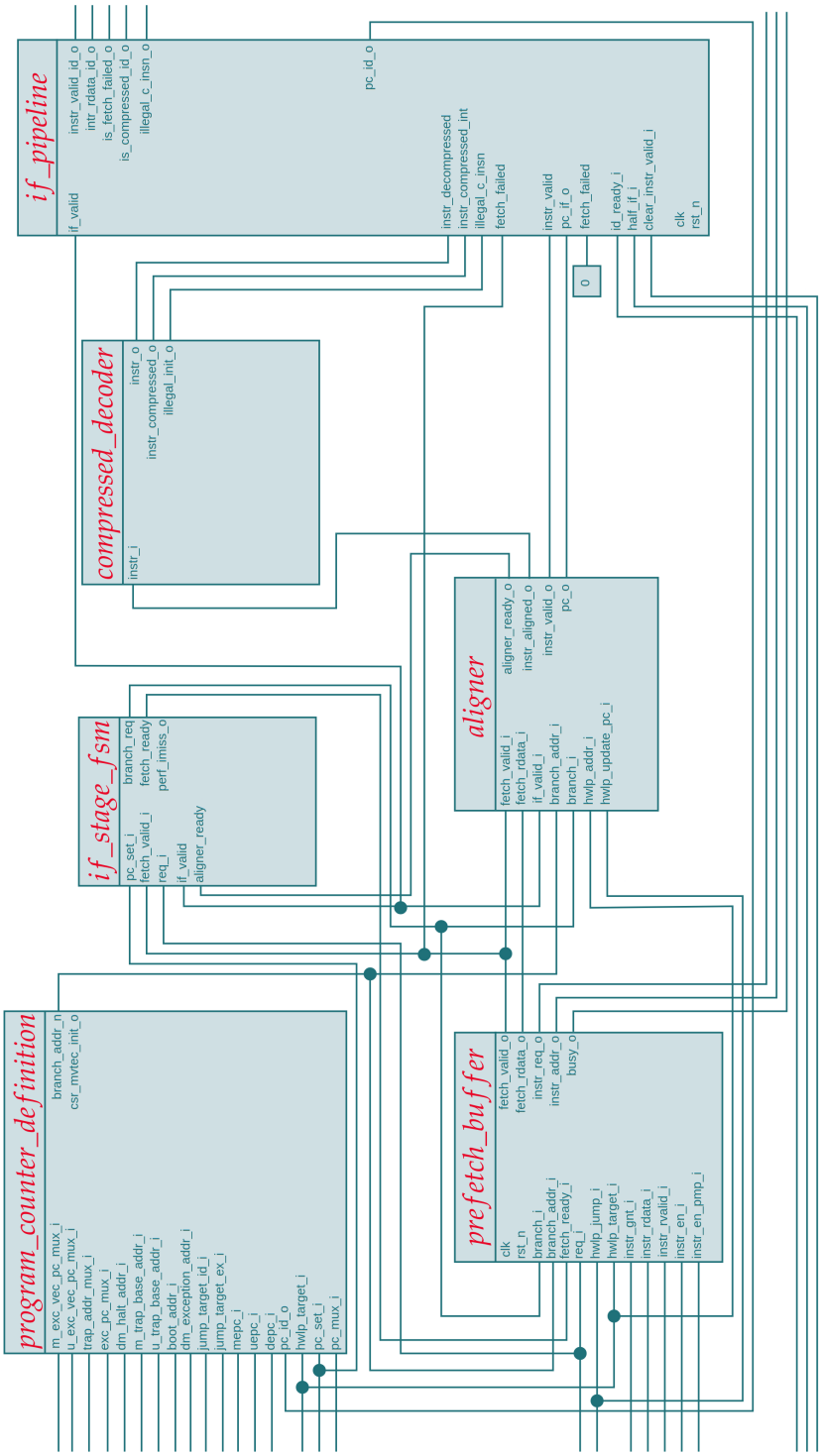

Figure 3.5: CV32E40P Instruction Fetch block diagram.

Chapter 4

Fault Tolerant IF Stage

We decide to begin the creation of the FT IF stage from the Compressed Decoder (CD). The idea is to create a FT Compressed Decoder manually, test it and then convert this architecture in a template. This generic template will be written in Travulog and used to automatically convert IF stage sub-blocks in fault tolerant architecture. The conversion is described in section 4.2 and 4.3, instead in the first section we focus on the design of the FT Compressed Decoder.

4.1 FT Compressed Decoder

The Compressed Decoder is a small, one input - three output, combinatorial sub-block of the IF stage. It is suitable to FT experimentations due to the low number of inputs and outputs, indeed for TMR technique we need to add a voter for each output, so lower outputs means faster design time.

The choice of the FT technique to use is linked to the final objective of the thesis: to create a POW (Proof Of Work) of the Travulog/Htravulog Toolchain and a FT IF stage at the same time. Therefore we decide to use the basic TMR technique plus a permanent error detector with alpha-counters. Another reason to implement TMR is that this is a full combinatorial block. In this way we implement a FT IF stage with the most used FT technique; this can be a good reference in the creation of Travulog template and it speed up the understanding of the architecture, by the simplification of the external approaches to the Travulog/Htravulog Toolchain.

4.1.1 Basic Voter

We created as first block the **cv32e40p_3voter**. As you can see in **Figure 4.1**, the voter compares three inputs data in_1_i, in_2_i and in_3_i, the output *voted_o* is the result of the majority vote. If all three inputs are different or in_2_i != in_3_i the output is equal to in_1_i, otherwise the output is in_2_i. This type of voting gives priority to in_1_i when all inputs are different, this is a non-significant choice for the final fault tolerance, indeed we can't know what input is correct if they are all different.

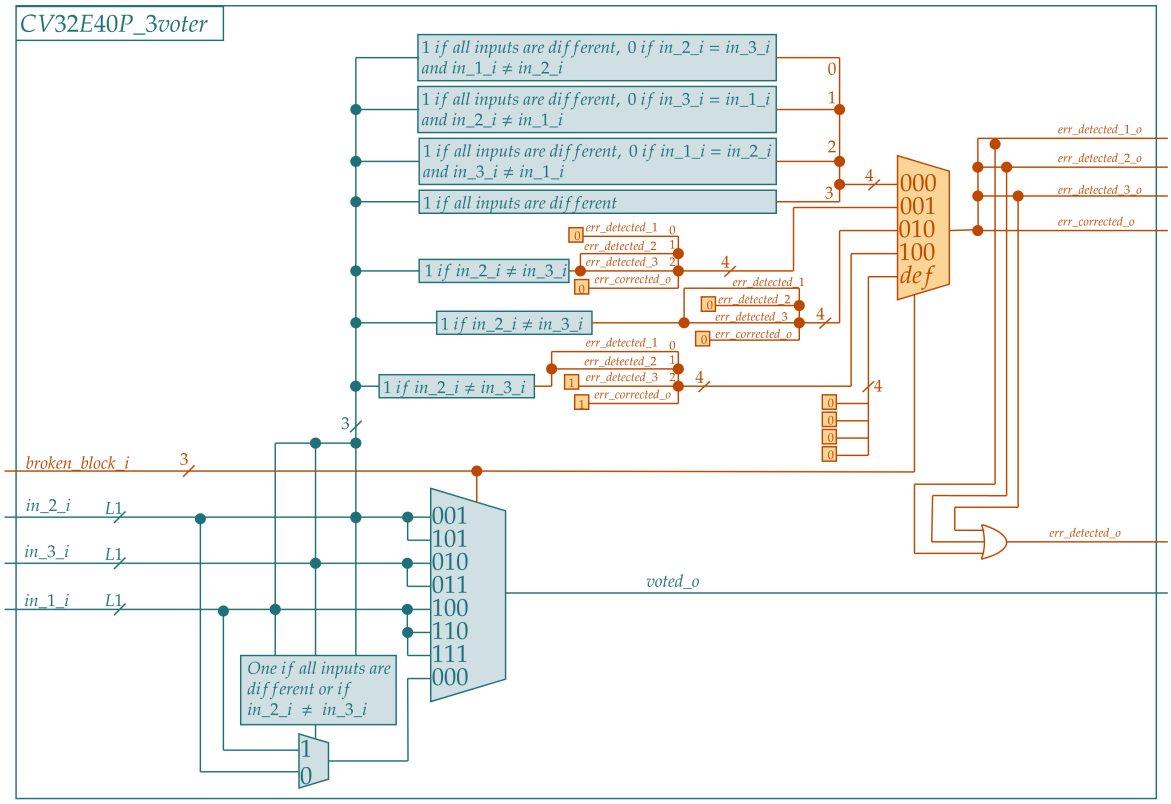

Figure 4.1: This is the voter used in the fault tolerant architecture.

The designed voter has also a three bit input called *broken_block_i*, each bit refers to a block, zero means that the block works properly while 1 means that it has a permanent error. For example [0,0,0] means that all input signals come from blocks without permanent errors, instead [0,1,0] means that the second block have a permanent error.
broken_block_i allows the discarding of input signals from permanent faulty blocks. Indeed, the multiplexer on the left is controlled by the broken_block_i, e.g., when in_1_i comes from a permanent faulty block, in_2_i is automatically selected as output, so the system becomes a duplexer.

In addition to the voting part there is some logic that indicates when there is a faulty input and when it is corrected by the voter. *err_detected_1_o*, *err_detected_2_o* and *err_detected_3_o* are one if the corresponding input signals in_1_i, in_2_i and in_3_i have a transient fault. When the input signal in_x_i is permanently faulty the corresponding output err_detected_x_o is zero, the remaining two output err_detected_y_o and err_detected_z_o will be both 1, if the corresponding inputs are different, because we can't know which is the correct input, otherwise they are zero.
If the detected error has been corrected, the err_corrected_o signal is one, instead it is zero when all inputs are different.

The cv32e40p_3voter block can be used for single dimension array by changing the L1 parameter; this parameter changes the dimension of in_x_i and voted_o signals, e.g.,if you have three 32-bit signals to compare, you have to set L1 to 32 to use the voter.

4.1.2 Configurable Voter

In order to enable triple voter configuration, we create the **cv32e40p_conf_voter**. This block can be used as a simple voter if TOUT is set to zero **Figure 4.2** or as a triple voter if TOUT is set to 1 **Figure 4.3** .
The external interface of the conf_voter is equal to the cv32e40_3voter apart for triple input called to_vote_i and triple output.
If TOUT is set to 0, the output of the voter is connected to the first element of voted_o, the other two elements aren't connected and they don't exist in the final layout.
The three err_detected signals of cv32e40_3voter are grouped in block_err_o, so basically if TOUT is set to 0 the conf_voter is approximately equal to the cv32e40p_3voter.
Instead if TOUT is set to 1, the conf_voter creates three instances of cv32e40_3voter and each voter votes independently the three inputs. The three outputs are connected to the voted_o signal. Finally the status signals are voted and sent to the output.
As you can see, the external interface of the conf_voter remains the same if you change TOUT, this allows a wide use of the block.

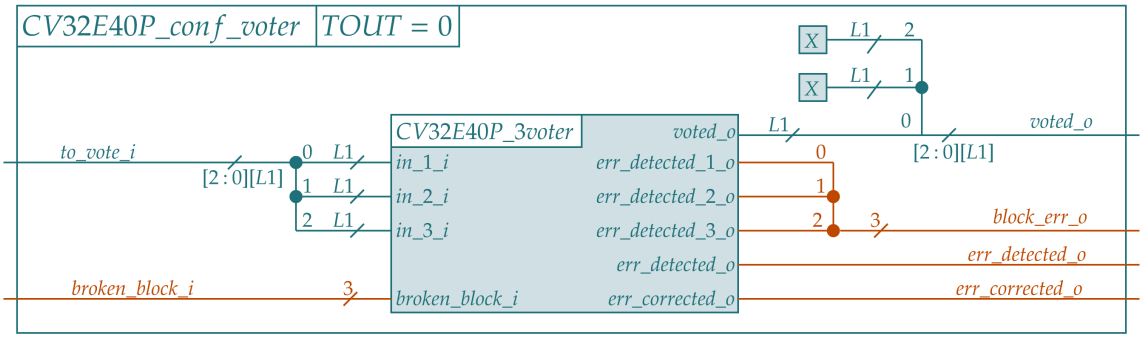

Figure 4.2: Configurable voter, with TOUT=0, a single voter.

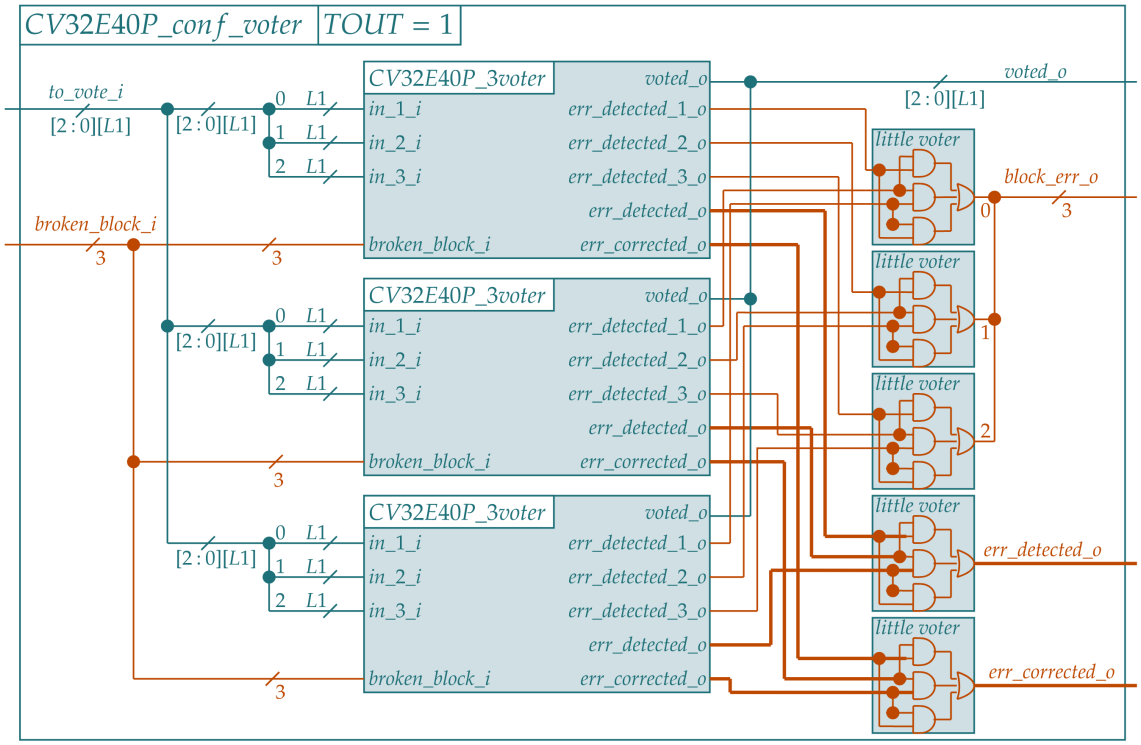

Figure 4.3: Configurable voter, with TOUT=1, a triple voter.

4.1.3 Breakage Monitor

The third block is the **cv32e40p_breakage_monitor** in Figure 4.4. Each time the Breakage Monitor is instanced we should associate it to one replica inside a NMR structure; in our case we have a TMR and so we will use three Breakage Monitor, one for each replica of the Compressed Decoder. Each time the associate block have an error, err_detected_i

should be asserted. This error increases reg_count_q by an INCREMENT value. Instead, if err_detected_i is equal to zero reg_count_q is decreased by a DECREASE value; both INCREMENT and DECREASE are SV parameters that can be changed. When reg_count_q is higher than the BREAKING_THRESHOLD, is_broken_o is asserted, this causes the activation of the clock gating which freezes the block maintaining is_broken_o to one. The block can also be set broken using the set_broken_i input signal.

This mechanism of increment and decrease allow to discard transient errors because their frequency is really low, so for one increment there are thousands of decreases. Instead, when a permanent error exists, it typically appears more frequently and so the increase rate is higher, this leads to an increase of reg_count_q and, as a consequence, to the detection of a permanent error. Anyway you can steer INCREMENT, DECREASE and BREAKING_THRESHOLD parameters in order to fit the error rate of your application. The SV module allows the change of the register size: INC_DEC_BIT is the number of bit for registers that contain INCREMENT and DECREASE variables, instead COUNT_BIT is the number of bit of reg_count register. Naturally we fit the number of bit with the parameter's values.

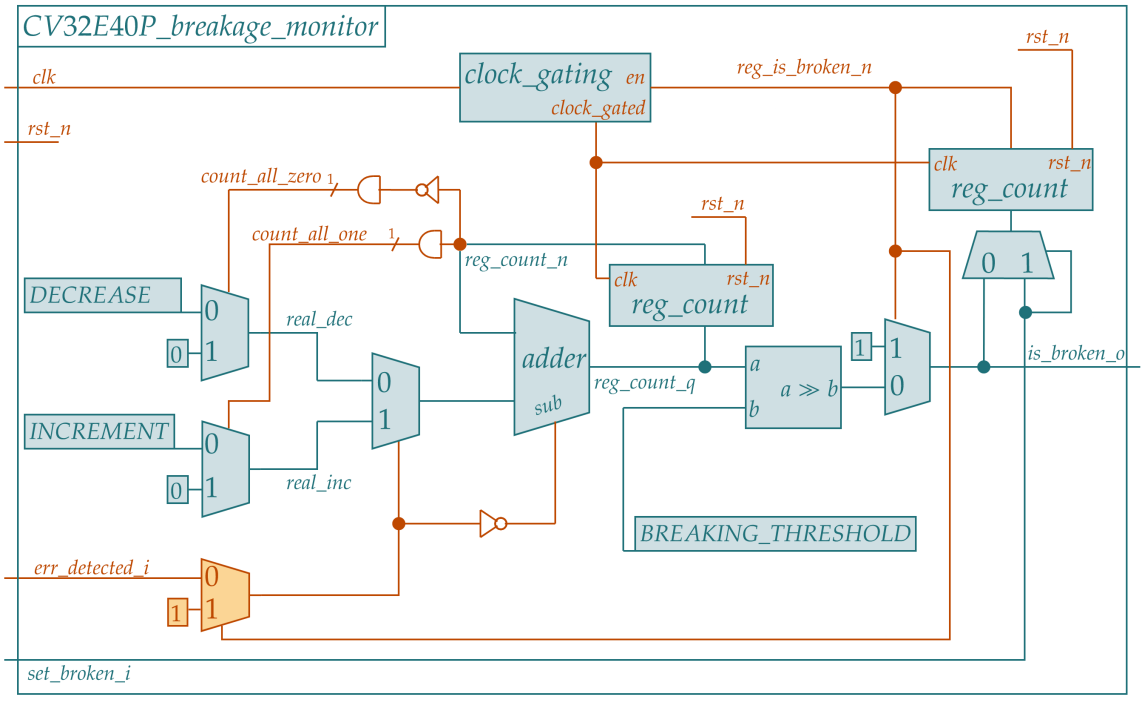

Figure 4.4: Breakage Monitor to detect permanent errors.

4.1.4 Configurable Compressed Decoder

Finally we create the **cv32e40p_compressed_decoder_ft** in **Figure 4.5**, using all previous blocks to create a new FT configurable Compressed Decoder (CD). On the right there are three replicas of the CD, the nine outputs are grouped by name so all instr_o outputs are connected to instr_o_to_vote, is_compressed_o outputs are connected to is_compressed_o_to_vote and finally illegal_instr_o outputs are connected to illegal_instr_o_to_vote.
These *_to_vote signals are connected to the relative conf_voter, as you can see in **Figure 4.5** the first conf_voter have three voter inside due to TOUT=1, this is an example of setting.

Each conf_voter have the block_err_o outputs, the first bit of these signals is referred to the errors of the first replica, the second bit to the second replica and the third to the third replica. The first block_err_o signal is referred to errors on instr_o, the second block_err_o to is_compressed_o and the third to illegal_instr_o.
We apply OR operation to each element of block_err_o in order to find possible errors in the corresponding replica, e.g., if the first replica have an error on is_compressed_o the second conf_voter will have block_err_o=[1,0,0], while the other two conf_voter will have block_err_o=[0,0,0], so we execute the OR between all block_err_o[0] and we find that is occured an error in the first replica.

The outputs of the OR are connected to the inputs of the Breakage Monitor, in this way each Breakage Monitor controls the corresponding replica. When a replica is considered broken by the Breakage Monitor the corresponding bit of the is_broken_o vector is set to 1 and all conf_voter will no longer consider the corresponding block. In this case the compressed_decoder_ft became a duplex system and if one of the two replicas failed the operation should be redone.

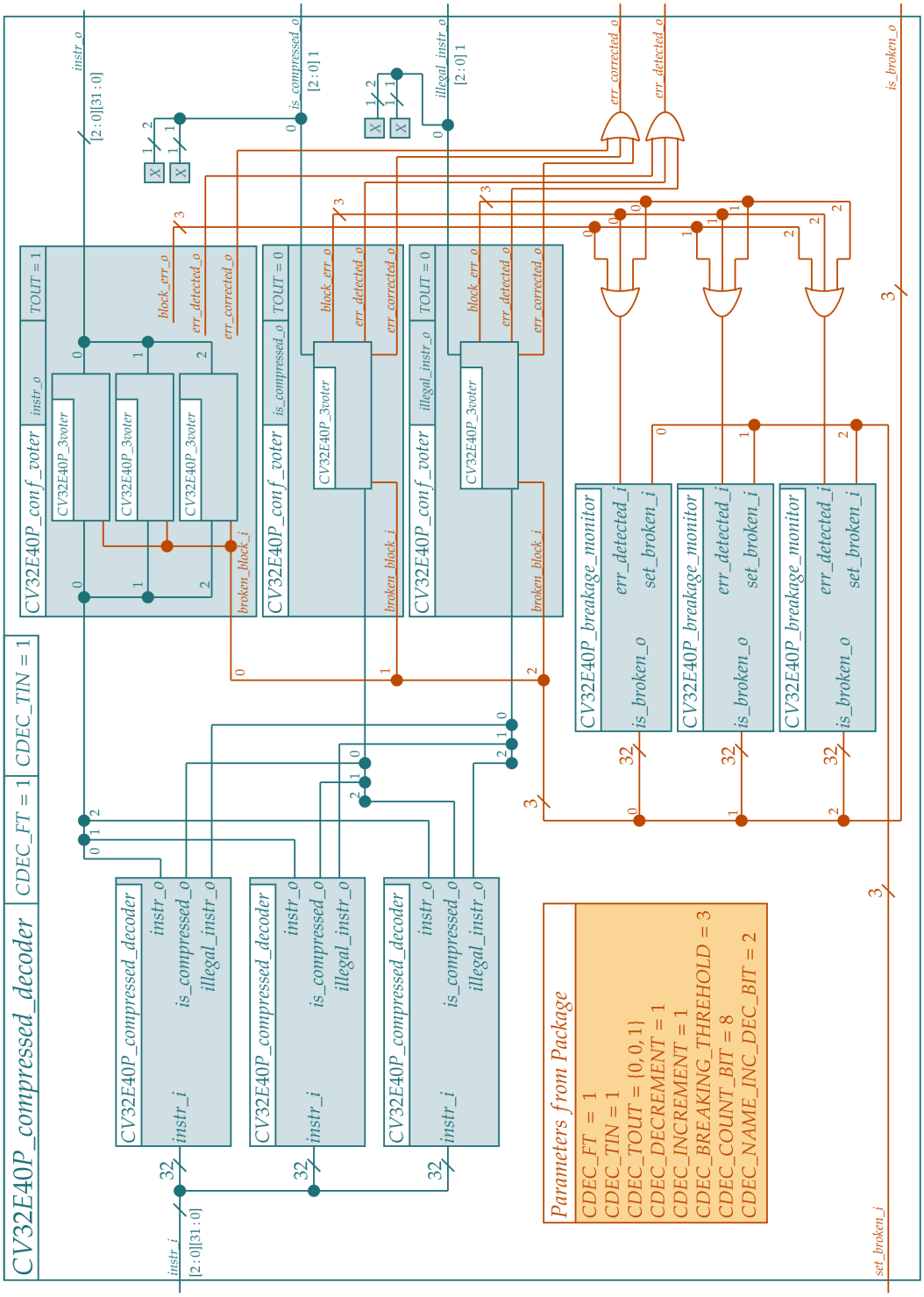

Figure 4.5: A possible configuration of the Configurable Compressed Decoder.

set_broken_i is a three bit signal used to set broken a replica from outside the block, at the same time exist the is_broken_o signal used to communicate if there are permanent broken block to the outside of the block.
The last signal can be used to save internal replicas status in some non-volatile memory (e.g., CSR registers) and use these information to reload replicas status inside the FT compressed decoder if the system is rebooted with set_broken_i.

Finally we have err_corrected_o and err_detected_o that communicate when an error occurs and if it has been corrected. These two signals can be used at high level to check architecture integrity and eventually indicate if the system should be replaced due to high error rate.

In the Parameters From Package table there are all SV parameters that can be changed to configure the block. So the features of the block are:

- **Fault Tolerance:** If CDEC_FT is equal to one is used TMR technique as you see in the picture, otherwise is used only one Compressed Decoder and the cv32e40p_compressed_decoder_ft became the basic cv32e40p_compressed_decoder.
- **Triple Input:** When you set CDEC_TIN is equal to one should be provided three instr_i inputs, each input is connected to the corresponding compressed decoder, otherwise is used only one input equal for each replica. This feature is useful if the previous block implement triple voter configuration.
- **Triple Output:** Each bit of CDEC_TOUT corresponds to a conf_voter and so to an output, the MSB is the last bit and it corresponds to the first output. The order of the output corresponds to the appearance order in the port declaration od the cv32e40p_compressed_decoder. When a bit of the CDEC_TOUT is 1 the corresponding conf_voter implement a triple voter, this means that the corresponding output will have three valid signals from triple voter.
- **Permanent Error Settings:** The remaining parameters are used to set the Breakage Monitor parameters, all Breakage monitor will have the same configuration.

As we have seen this block is highly configurable. Due to triple input - output option both input and output signals are triplicated in the port declaration. Anyway in some configuration not all input and output signals are connected, this means that care must be taken in the block connection and parameters configuration.

In the next chapter we will use the cv32e40p_compressed_decoder to create the Travulog template in order to automatically convert each sub-block of the IF stage.

4.2 Travulog

To create a configurable FT architecture it is necessary to apply fault tolerant techniques to all the blocks of the IF Stage and then allow to apply or not techniques in each block by means of parameters. It was therefore decided to create a metalanguage within the SystemVerilog (SV) that allows the creation of the new architecture starting from a template and a base module, in this way we can define a template and apply it to each block of the IF stage. The automatized application of Fault Fault tolerant methods already exists in some commercial application but it work at netlist level [40], our solution is at RTL level and it is open source.
As you can see in **Figure 4.6** the transformation of the main module is done by a Python object linked to the template, when you give the base module to the object a new module is created. The new FT architecture is created on the basis of the Travulog template and the base module, it is therefore an interface layer that makes the old module Fault Tolerant. To allow the conversion through the Travulog object it is necessary to have an object that contains all the data of the associated SystemVerilog base module, so we have created a new Python class that parses a SV file containing a module, this class has been called moddata. As shown in the **Figure 4.6** the file containing the basic module must be transformed into the corresponding moddata object and then passed to the Travulog object which generates the new architecture.

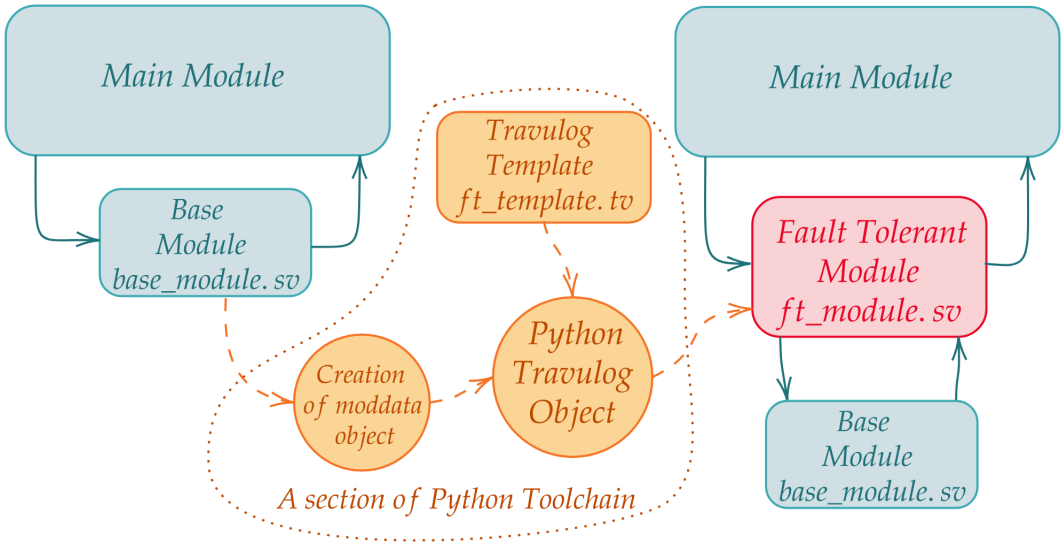

Figure 4.6: Flow diagram of architecture transformation using Travulog template

The conversion of a SVerilog module using a Travulog template implies the use of Python code, in fact you have to create a moddata object related to the Base module, the Travulog

object related to the TV template and then give moddata obj to TV obj to obtain the new SV code.
For these reason, in order simplify the use of Travulog template we create the Hidden Travulog (HTV) code or HTravulog. This code is hidden inside a synthesizable SV module using four slash and a space: "//// " .
This code should be written in the main SV module and it indicates how to apply Travulog template. Using HTV code you can also create new module from a SV block inside the current module and the apply your template to this new block. The complete functionality are described in HTravulog section.

To create the Travulog language we started from the fault tolerant compress_decoder_ft and we created a series of commands that allow us to create the FT compress decoder from the basic one. We will now analyze the various Travulog commands through pieces of the Travulog template and its conversion into SVerilog.

4.2.1 Declaration of ports

Ports declaration is the first element in a System Verilog module, in this part of a new FT block probably you will see many of the IO signals of the basic architecture, for this reason we automatized the creation of this part allowing to use basic block signals name.

In the listing 4.1 there is a part of the Travulog template, this piece of code allows to generate the System Verilog of the listing 4.2 which is the IO declaration of the FT Compressed Decoder. The Travulog commands used are the following :

PARAMETER_DECLARATION: The command PARAMETER_DECLARATION copies the parameter declaration from the BLOCK module in the new System Verilog module. BLOCK is an identifier used in the Travulog object that should be linked to a moddata object. Note that you can have multiple ID since they are managed as a diction:ary, e.g., if you set {"BLOCK":moddata_obj1, "BLOCK2":moddata_obj2} in the Travulog object you can use both BLOCK and BLOCK2 identifiers in the Travulog code.

DECLARATION_FOREACH: This command cycles on the given signals and it substitutes: INOUT with "input" or "output", BITINIT with the bits definition and SIGNAME with the name of the signal. The first argument is the module id, the second one is the type of signal: IN for input port of the module, OUT for output, IN_OUT for both input and output and INTERN for internal signals of the module. You can also indicate some signals to exclude from the list using "NOT sig1 sig2 ... " as you can see at line 7. e.g., clk and rst_n signals are excluded since they should not

be triplicated. DECLARATION_FOREACH can also be used for the declaration of internal signals and for assign statement as we will see later.

MODULE_NAME: This is a parameter which is substituted with the name given to the Travulog object, it is the name of the new module.

```
module MODULE_NAME

  PARAMETER_DECLARATION BLOCK

(
  // compressed decoder input output
  DECLARATION_FOREACH BLOCK IN_OUT NOT clk rst_n
    INOUT logic [2:0]BITINIT SIGNAME,
  END_DECLARATION_FOREACH


  input logic clk,
  input logic rst_n,

  // fault tolerant state
  input logic [2:0] set_broken_i,
  output logic [2:0] is_broken_o,
  output logic err_detected_o,
  output logic err_corrected_o
);

```

Listing 4.1: Declaration Travulog Code

```
module cv32e40p_compressed_decoder_ft
#(
  parameter FPU = 0
)
(
  // compressed decoder input output
  input  logic [2:0] [31:0] instr_i,
  output logic [2:0] [31:0] instr_o,
  output logic [2:0] is_compressed_o,
  output logic [2:0] illegal_instr_o,

  input logic clk,
  input logic rst_n,

  // fault tolerant state
  input logic [2:0] set_broken_i,
  output logic [2:0] is_broken_o,
  output logic err_detected_o,
  output logic err_corrected_o
);
```

Listing 4.2: Declaration SVerilog code derived

4.2.2 Internal signals and assign

In the following listings there is the continuation of the previous ports definition. The first "declaration_foreach" create the signals used to connect the three block outputs to the voter while the second one creates block error signals. In the last two lines there is the compound parameter "SIG_NUM-BLOCK-OUT" inside the square bracket, this parameter is substituted in the right listing with the number (SIG_NUM) of output ports (OUT) of the module (BLOCK) minus one, anyway instead of OUT you can use IN, PARAM, INTERN or IN_OUT in order to have the correct signals number.

```
// Signals out to each compressed
// decoder block to be voted
DECLARATION_FOREACH BLOCK OUT
logic [2:0]BITINIT SIGNAME_to_vote ;
END_DECLARATION_FOREACH

// Error signals
```

```
// Signals out to each compressed
// decoder block to be voted
logic [2:0] [31:0] instr_o_to_vote ;
logic [2:0] is_compressed_o_to_vote ;
logic [2:0] illegal_instr_o_to_vote ;

// Error signals
```

```
8  DECLARATION_FOREACH BLOCK OUT
9  logic [2:0] SIGNAME_block_err ;
10 END_DECLARATION_FOREACH
11
12 // Signals that use error signal to
13 // find if there is one error on each
14 // block, it is the or of previous signals
15 logic [2:0] block_err_detected;
16 logic [SIG_NUM-BLOCK-OUT:0] err_detected;
17 logic [SIG_NUM-BLOCK-OUT:0] err_corrected;
18
```

Listing 4.3: Travulog Code

```
logic [2:0] instr_o_block_err ;
logic [2:0] is_compressed_o_block_err ;
logic [2:0] illegal_instr_o_block_err ;

// Signals that use error signal to
// find if there is one error on each
// block, it is the or of previous signals
logic [2:0] block_err_detected;
logic [2:0] err_detected;
logic [2:0] err_corrected;
```

Listing 4.4: SVerilog code derived

4.2.3 Instance

In order to create a new instance from an existing block, the INSTANCE command shown in the listing 4.5 was create. After the INSTANCE keyword, there are the block id and the name of the new instance. As shown in listing 4.5 the BLOCK_MODNAME parameter is used to create the name of the new instance, this parameter is replaced in the preprocessing phase with the name of the BLOCK module.

The connection commands are indicated inside the INSTANCE command. The command at line two indicates to connect the parameters to the same name, in fact we have .FPU (FPU). At line 3 instead it is indicated to connect clk and rst_n signals to the same name, the next line connects the inputs to their same name but it adds the suffix "[0]", however, at the signals already connected in the previous if are not appended the suffix. In the conversion to SV on the right there are no clocks and no reset so the behavior just described is not evident.

As already mentioned, the connections inside the INSTANCE command are sensitive to the order in which they are made, for example the IF on the inputs should be before assigning the inputs generically with "IN = IN", otherwise the ifs are not considered. Same consideration for outputs and parameters.

```
1  INSTANCE BLOCK BLOCK_MODNAME_no_ft
2  PARAM=PARAM
3  IF clk rst_n IN=IN
4  IN= IN[0]
5  OUT = OUT[0]
6  END_INSTANCE
7
8
9
10
11
12
13
14
```

```
cv32e40p_compressed_decoder
#(
    .FPU (FPU)
)
compressed_decoder_no_ft
(
    // Input ports of
    compressed_decoder_no_ft
    .instr_i                ( instr_i[0]),

    // Output ports of
    compressed_decoder_no_ft
    .instr_o        ( instr_o[0]),
    .is_compressed_o( is_compressed_o[0] ),
```

```
15
16
17
```

Listing 4.5: Instance Travulog Code

```
    .illegal_instr_o(  illegal_instr_o[0] )
);
```

Listing 4.6: Instance SVerilog code derived

In the cv32e40p_compressed_decoder_ft module there are as many conf_voter as the output of the cv32e40p_compressed_decoder (CD), in fact the CD is triplicated and we need a conf_voter for each output. In order to automatize the creation and the connection of the conf_voter we create the command INSTANCE_FOREACH. After this keyword there is the id of the block and the list of signals on which cycle on. It can be used IN, OUT, IN_OUT and INTERN as abbreviation for the list, in this way the code is compressed and clearer.

As just mentioned INSTANCE_FOREACH cycles on the signals list and it substitutes in the verilog code: BITNUMBER with the number of bits of the signal, INDEX with the current cycle number and SIGNAME with the name of the signal.

In the listing below you can see an example of how the INSTANCE_FOREACH command works.

```
1  INSTANCE_FOREACH BLOCK OUT
2      // Voter for TOVOTE signal, triple voter
       if
3      // PARAM_NAME_TOUT[INDEX] == 1
4      cv32e40p_conf_voter
5      #(
6          .L1(BITNUMBER),
7          .TOUT(PARAM_NAME_TOUT[INDEX])
8      ) voter_SIGNAME_INDEX
9      (
10         .to_vote_i( SIGNAME_to_vote ),
11         .voted_o( SIGNAME),
12         .block_err_o( SIGNAME_block_err),
13         .broken_block_i(is_broken_o),
14         .err_detected_o(err_detected[INDEX]),
15        .err_corrected_o(err_corrected[INDEX])
16     );
17 END_INSTANCE_FOREACH
18
19
20
21
22
23
24
25
26
27
28
29
30
31
```

```
// Voter for TOVOTE signal, triple voter if
 // CODE_TOUT[0] == 1
cv32e40p_conf_voter
#(
    .L1(32),
    .TOUT(CODE_TOUT[0])
) voter_instr_o_0
(
    .to_vote_i( instr_o_to_vote ),
    .voted_o( instr_o),
    .block_err_o( instr_o_block_err),
    .broken_block_i(is_broken_o),
    .err_detected_o(err_detected[0]),
    .err_corrected_o(err_corrected[0])
);
// Voter for TOVOTE signal, triple voter if
// CODE_TOUT[1] == 1
cv32e40p_conf_voter
#(
    .L1(1),
    .TOUT(CODE_TOUT[1])
) voter_is_compressed_o_1
(
    .to_vote_i( is_compressed_o_to_vote ),
    .voted_o( is_compressed_o),
    .block_err_o(
    is_compressed_o_block_err),
    .broken_block_i(is_broken_o),
    .err_detected_o(err_detected[1]),
    .err_corrected_o(err_corrected[1])
);
// Voter for TOVOTE signal, triple voter if
```

```
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

Listing 4.7: Instance foreach Travulog Code

```
// CODE_TOUT[2] == 1
cv32e40p_conf_voter
#(
    .L1(1),
    .TOUT(CODE_TOUT[2])
) voter_illegal_instr_o_2
(
    .to_vote_i( illegal_instr_o_to_vote ),
    .voted_o( illegal_instr_o),
    .block_err_o(
    illegal_instr_o_block_err),
    .broken_block_i(is_broken_o),
    .err_detected_o(err_detected[2]),
    .err_corrected_o(err_corrected[2])
);
```

Listing 4.8: Instance foreach SVerilog code

4.2.4 Multiple Operation

An instance of compressed_decoder have an error if there is at least one output signal wrong, for these reason error_detected signals is found by applying OR operation between *_block_err signals out of the conf_voters as explained in Section 4.1.4. The number of *_block_err signals is equal to the number of output of the module used, in this case the three output of the compressed_decoder, for this reason we create the Travulog command OP_FOREACH that apply a logic operation to a set of signals.
After the keyword OP_FOREACH there should be the module id and then the signals identifier (IN, OUT, IN_OUT, INTERN, PARAM), these two element identify the list of signals to cycle on. Then there should be the operation to apply (|, & etc) and the pattern to use for each signal (SIGNAME will be replaced with the name of the signal).
In the listing 4.9 is applied a suffix to the signame in order to apply OR to correct block_err signals, instead in the right listing there is the conversion of Travulog code, each TV line becomes three lines of SV.

```
1  assign block_err_detected[0] = OP_FOREACH
       BLOCK OUT | SIGNAME_block_err[0] ;
2
3
4  assign block_err_detected[1] = OP_FOREACH
       BLOCK OUT | SIGNAME_block_err[1] ;
5
6
7  assign block_err_detected[2] = OP_FOREACH
       BLOCK OUT | SIGNAME_block_err[2] ;
8
9
10
```

```
assign block_err_detected[0] =
          instr_o_block_err[0]
        | is_compressed_o_block_err[0]
        | illegal_instr_o_block_err[0];
assign block_err_detected[1] =
          instr_o_block_err[1]
        | is_compressed_o_block_err[1]
        | illegal_instr_o_block_err[1];
assign block_err_detected[2] =
          instr_o_block_err[2]
        | is_compressed_o_block_err[2]
        | illegal_instr_o_block_err[2];
```

Listing 4.9: Instance foreach Travulog Code

Listing 4.10: Instance foreach SVerilog code

4.2.5 Converted SV and parameters template

At the end of the Travulog conversion you will obtain the compressed_decoder_ft.sv file (listing 4.11) containing an architecture layer that makes the compressed_decoder fault tolerant.

```
import cv32e40p_pkg2::*;
import cv32e40p_pkg::*;

module cv32e40p_compressed_decoder_ft
#(
    parameter FPU         = 0
)
(

    // compressed decoder input output
    input logic [2:0]      [31:0]   instr_i,
    output logic [2:0]      [31:0]   instr_o,
    output logic [2:0]           is_compressed_o,
    output logic [2:0]           illegal_instr_o,

    input logic clk,
    input logic rst_n,

    // fault tolerant state
    input logic [2:0] set_broken_i,
    output logic [2:0] is_broken_o,
    output logic err_detected_o,
    output logic err_corrected_o
);
    // Signals out to each compressed decoder block to be voted
    logic [2:0]      [31:0]   instr_o_to_vote ;
    logic [2:0]           is_compressed_o_to_vote ;
    logic [2:0]           illegal_instr_o_to_vote ;

    // Error signals
    logic [2:0] instr_o_block_err ;
    logic [2:0] is_compressed_o_block_err ;
    logic [2:0] illegal_instr_o_block_err ;

    // Signals that use error signal to find if there is one error on
    // each block, it is the or of previous signals
    logic [2:0] block_err_detected;
    logic [2:0] err_detected;
    logic [2:0] err_corrected;

    // variable for generate cycle
    generate
        case (CODE_FT)
            0 : begin
```

```
45                  cv32e40p_compressed_decoder
46                  #(
47                       .FPU          (  FPU          )
48                  )
49                   compressed_decoder_no_ft
50                  (
51                      // Input ports of compressed_decoder_no_ft
52                      .instr_i        (  instr_i[0]             ),
53
54                      // Output ports of compressed_decoder_no_ft
55                      .instr_o        (  instr_o[0]             ),
56                      .is_compressed_o    (  is_compressed_o[0]         ),
57                      .illegal_instr_o    (  illegal_instr_o[0]         )
58                  );
59                  // Since we don't use FT can't be detected an
60                  // error
61                  assign block_err_detected = {1'b0,1'b0,1'b0};
62              end
63              default : begin
64                  // Input case
65                  case (CODE_TIN)
66                      0 : begin // Single input
67                          genvar i;
68                          for (i=0; i<3; i=i+1)  begin
69                              cv32e40p_compressed_decoder
70                              #(
71                                   .FPU          (  FPU          )
72                              )
73                               compressed_decoder_single_input
74                              (
75                                  // Input ports of compressed_decoder_single_input
76                                  .instr_i        (  instr_i[0]             ),
77
78                                  // Output ports of compressed_decoder_single_input
79                                  .instr_o        (  instr_o_to_vote[i]         ),
80                                  .is_compressed_o    (  is_compressed_o_to_vote[i]     ),
81                                  .illegal_instr_o    (  illegal_instr_o_to_vote[i]     )
82                              );
83                          end
84                      end
85                      default : begin // Triplicated input
86                          genvar i;
87                          for (i=0; i<3; i=i+1)  begin
88                              cv32e40p_compressed_decoder
89                              #(
90                                   .FPU          (  FPU          )
91                              )
92                               compressed_decoder_tiple_input
93                              (
94                                  // Input ports of compressed_decoder_tiple_input
95                                  .instr_i        (  instr_i[i]             ),
96
97                                  // Output ports of compressed_decoder_tiple_input
98                                  .instr_o        (  instr_o_to_vote[i]         ),
99                                  .is_compressed_o    (  is_compressed_o_to_vote[i]     ),
100                                 .illegal_instr_o    (  illegal_instr_o_to_vote[i]     )
101                             );
```

```
102                         end
103                     end
104                 endcase
105
106                 // Voter for TOVOTE signal, triple voter if
107                 // CODE_TOUT[0] == 1
108                 cv32e40p_conf_voter
109                 #(
110                     .L1(32),
111                     .TOUT(CODE_TOUT[0])
112                 ) voter_instr_o_0
113                 (
114                     .to_vote_i( instr_o_to_vote ),
115                     .voted_o( instr_o),
116                     .block_err_o( instr_o_block_err),
117                     .broken_block_i(is_broken_o),
118                     .err_detected_o(err_detected[0]),
119                     .err_corrected_o(err_corrected[0])
120                 );
121
122                 // Voter for TOVOTE signal, triple voter if
123                 // CODE_TOUT[1] == 1
124                 cv32e40p_conf_voter
125                 #(
126                     .L1(1),
127                     .TOUT(CODE_TOUT[1])
128                 ) voter_is_compressed_o_1
129                 (
130                     .to_vote_i( is_compressed_o_to_vote ),
131                     .voted_o( is_compressed_o),
132                     .block_err_o( is_compressed_o_block_err),
133                     .broken_block_i(is_broken_o),
134                     .err_detected_o(err_detected[1]),
135                     .err_corrected_o(err_corrected[1])
136                 );
137
138                 // Voter for TOVOTE signal, triple voter if
139                 // CODE_TOUT[2] == 1
140                 cv32e40p_conf_voter
141                 #(
142                     .L1(1),
143                     .TOUT(CODE_TOUT[2])
144                 ) voter_illegal_instr_o_2
145                 (
146                     .to_vote_i( illegal_instr_o_to_vote ),
147                     .voted_o( illegal_instr_o),
148                     .block_err_o( illegal_instr_o_block_err),
149                     .broken_block_i(is_broken_o),
150                     .err_detected_o(err_detected[2]),
151                     .err_corrected_o(err_corrected[2])
152                 );
153
154
155                 assign err_detected_o =  err_detected[0]
156                             | err_detected[1]
157                             | err_detected[2];
158                 assign err_corrected_o =  err_corrected[0]
```

```
159                             | err_corrected[1]
160                             | err_corrected[2];
161
162                 assign block_err_detected[0] =  instr_o_block_err[0]
163                                 | is_compressed_o_block_err[0]
164                                 | illegal_instr_o_block_err[0];
165                 assign block_err_detected[1] =  instr_o_block_err[1]
166                                 | is_compressed_o_block_err[1]
167                                 | illegal_instr_o_block_err[1];
168                 assign block_err_detected[2] =  instr_o_block_err[2]
169                                 | is_compressed_o_block_err[2]
170                                 | illegal_instr_o_block_err[2];
171
172                 genvar m;
173                 for (m=0;  m<3 ; m=m+1) begin
174                     // This block is a counter that is incremented each
175                     // time there is an error and decremented when it
176                     // there is not. The value returned is is_broken_o
177                     // , if it is one the block is broken and should't be
178                     // used
179                     cv32e40p_breakage_monitor
180                     #(
181                         .DECREMENT(CODE_DECREMENT),
182                         .INCREMENT(CODE_INCREMENT),
183                         .BREAKING_THRESHOLD(CODE_BREAKING_THRESHOLD),
184                         .COUNT_BIT(CODE_COUNT_BIT),
185                         .INC_DEC_BIT(CODE_INC_DEC_BIT)
186                     ) breakage_monitor
187                     (
188                         .rst_n(rst_n),
189                         .clk(clk),
190                         .err_detected_i(block_err_detected[m]),
191                         .set_broken_i(set_broken_i[m]),
192                         .is_broken_o(is_broken_o[m])
193                     );
194                     // We find is the block have an error.
195                 end
196
197             end
198         endcase
199
200     endgenerate
201
202 endmodule
```

Listing 4.11: Fault Tolerant compressed decoder layer

This SVerilog architecture needs some parameters: CODE_FT, CODE_TIN, CODE_TOUT, CODE_DECREMENT, CODE_INCREMENT, CODE_BREAKING_THRESHOLD, CODE_COUNT_BIT, CODE_INC_DEC_BIT. These parameters are contained in the cv32e40p_pkg2 package imported at line one of listing 4.11, the creation of this package can be automatized using a TV parameters template, for example our fault tolerant TV template ft_template.sv needs its TV parameters template ft_template_parameters.sv, this template is shown in listing 4.12.

```
1  ////////////////////////////////////////////////////////////////////
2  // MODULE_NAME_ft
3  ////////////////////////////////////////////////////////////////////
4  parameter int PARAM_NAME_FT = 1;
5  parameter int PARAM_NAME_TIN = 0;
6        // TOUT is referred to output signal in order of definition
7        //
8        // OP_FOREACH BLOCK OUT // TOUT[INDEX]-refers-to->>SIGNAME
9  parameter int PARAM_NAME_TOUT [SIG_NUM-BLOCK-OUT:0] = { OP_FOREACH BLOCK OUT , 0  };
10
11 // Parameter for breakage monitors
12 parameter PARAM_NAME_DECREMENT = 1;
13 parameter PARAM_NAME_INCREMENT = 1;
14 parameter PARAM_NAME_BREAKING_THRESHOLD = 3;
15 parameter PARAM_NAME_COUNT_BIT = 8;
16 parameter PARAM_NAME_INC_DEC_BIT = 2;
17
18 parameter MAIN_MOD_ID_CURRENT_MOD_ID = MODULE_ORDER;
19
20
```

Listing 4.12: Parameters template for Fault tolerant module

This Template should be converted in System Verilog using our Toolchain.

4.2.6 Apply the template in Python

In order to apply the Travulog template to a module you can use HTravulog code where the module is instanced or you can directly use Python code. In this section we follow the second way and so we explain how Python code is structured and how to use it.
The whole code of the toolchain is in the https://github.com/Elia1996/Travulog repository, so you can clone it and use it.
In order to use Travulog template you need the files *moddata.py* and *travulog.py*, the first file contains many functions and the moddata class that enable the parsing an manipulation of a SVerilog module, instead the second file contains the travulog class.
If you want to apply a Travulog template, the faster way is to look at test_travulog.py file reported below.

```
1  #!/usr/bin/python3
2  from travulog import *
3
4  template_fname = "templates/ft_template.sv"
5  template_params_fname = "templates/ft_template_parameters.sv"
6  module_fname_dict = {"BLOCK" : "./test/arch/cv32e40p_compressed_decoder.sv"}
7  module_prefix = "cv32e40p_"
8
9  tr = travulog(template_fname, template_params_fname, module_fname_dict, module_prefix)
10
11 print(tr.GetElaboratedTemplate("New_module_name","PARAM_NAME"))
12 print(tr.GetElaboratedTemplateParams("New_module_name","PARAM_NAME"))
13
```

Listing 4.13: Basic Python code to use Travulog

As you can see after the import of the travulog module are defined some parameters:

- **template_fname:** this is the fault tolerant template written in Travulog, the extension is irrelevant.
- **template_params_fname:** this is the TV template of parameters, it is used at line twelve where it is converted according to the current module name that in our case is the compressed_decoder,
- **module_fname_dict:** This is a dictionary which connects modules ids with corresponding filenames of SVerilog modules. In this case BLOCK is connected to the compressed_decoder filename and for this reason the Travulog object create the fault tolerant compressed decoder. Anyway you can have multiple module ids inside the TV template, you only need to set it in this dictionary.
- **module_prefix:** This should be the prefix of each filename and module name, it should always end with a "_" character and it is used to create correct parameter base name.

Once created the travulog object using these parameters, you can use GetElaboratedTemplate function to create the new cv32e40p_compressed_decoder_ft.sv file text and you can use GetElaboratedTemplateParams function to create the parameters initialization for the package. For directly use the new architecture you should create a new package file containing the parameters initialization and import this package in the new ft module.
The way in which has been structured the Python code enable the creation of new powerful Travulog commands, so the TV commands analyzed is only a small example of what can be done using this tool since TV code can be extended according to your architectural needs.

4.3 Hidden Travulog

As already mentioned, Hidden Travulog or HTV is a particular comment code inside a SVerilog file, the difference respect to a comment is the "//// " beginning key (HTKEY) instead of "//". The HTV parser only analyze the code after the HTKEY so be careful to correcty write "//// " before the HTV code.
The result of a hidden code as HTV is the higher architecture maintenance, indeed the SVerilog architecture can be simulated, synthesised and modified with HTV code in it. After that a change in the base arch is done you can proceed with the HTravulog conversion, at the end of the process you obtain a new architecture that contains the base architecture changes. This maintenance process is shown in figure 4.7.

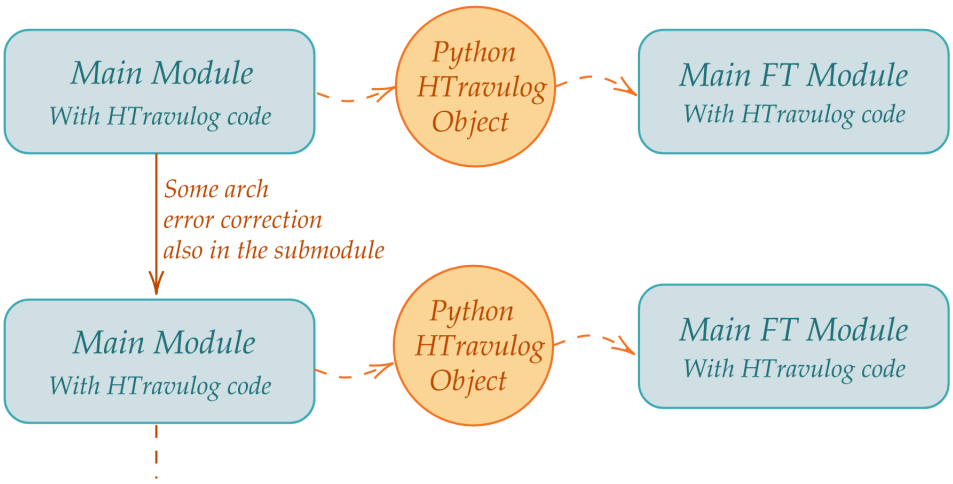

Figure 4.7: HTravulog used during architecture life cycle

For the purpose of this Master Thesis are created only few HTV commands that can be extended for other uses. Anyway at this implementation step, HTV code can be used inside a whatever SVerilog module for these purpose:

- **Add some line:** The command ADD_LINE allows to add an arbitrary line to the converted architecture.
- **Change internal signals:** The command FOREACH allows to cycle on input, output, internal signals and parameters in order to use their name for some connections or declarations.
- **Create new module:** When you want to apply a Travulog template to a piece of your *architecture called A* but it is written as a part of the module, you can use

CREATE_MODULE command. With this command you can transform the piece of *arch A* in a new *module B* written in a new SV file, additionally the command automatically creates the instance of *module B* in the converted *architecture A*.

- **Apply a Travulog template to a module:** In the main module of an architecture you normally have many instances of modules, for each instance you can use ADD_MODULE_LAYER. This commmand transforms the instanced module using a Travulog template and it create the new correct instance.

These are the main transformation you can apply with HTravulog code.
Now in order to explain how HTV parser works we divide SVerilog module in these four parts:

- **Introduction:** It is the SV code from the beginning of the file up to the module declaration, here you usually import package and you write the architecture license. In this part you can use these HTV commands: IMPORT, ADD_LINE, NEW_MODULE_NAME and NEW_MODULE_FILE. These command are explained in 4.3.1 section.
- **Port declarations:** In this part are defined the input and output of the module, at the moment this part can't be modified in order to maintain the same interface, anyway the code is organized in order to simplify the extension of HTV code also in this part.
- **Intern signals and assign:** This part starts at the end of port declaration and ends with the HTV command "END_DECLARATIONS", In this part you should define all intern signals and all assignment that you want to change during conversion. Just before the begin of the architecture structure definition you should place the "END_DECLARATIONS" command in order to notify the tool that the third part ends. This part is analyzed in Section 4.3.2.

  In this section you can use only the command FOREACH.
- **Architecture definition:** This part starts after the "END_DECLARATIONS" command and ends with the module. Here you can use CREATE_MODULE and ADD_MODULE_LAYER commands, these are two powerful commands that are described in section 4.3.3 and 4.3.4.

In the following section we analyze each part and each command in detail.

4.3.1 Introduction Part

This is the part of code before the declaration of the module, in listing 4.14 there is an example of SV mixed with HTV code used in cv32e40p_if_stage.sv file.

```
1 //// IMPORT htv_pkg.tv
2
3 import cv32e40p_pkg::*;
4 //// ADD_LINE import cv32e40p_pkg2::*;
5
6 //// NEW_MODULE_NAME cv32e40p_if_stage
7 //// NEW_MODULE_FILE OUT_DIR/cv32e40p_if_stage_ft.sv
8
```

Listing 4.14: Introduction of cv32e40p if stage

IMPORT command: At line one there is the IMPORT command used to import a parameters file, this file is parsed by the tool in order to import some parameters. In listing 4.15 there is an example of *htv_pkg.tv* file, all parameter are mandatory. BASEDIR is an abbreviations of the complete base path, we use BASEDIR for space reasons but the real file contains the complete path.

```
1 IN_DIR BASEDIR/test/arch
2 OUT_DIR BASEDIR/out
3 TEMPLATE ft_template
4         FILE BASEDIR/templates/ft_template.sv
5         PARAM_FILE BASEDIR/templates/ft_template_parameters.sv
6 END_TEMPLATE
7 PKG_FILE BASEDIR/templates/cv32e40p_pkg2.sv
8 PKG_OUT_FILE BASEDIR/out/cv32e40p_pkg2.sv
9 MODULE_PREFIX cv32e40p_
10
```

Listing 4.15: Introduction of cv32e40p if stage

The first two line set IN_DIR and OUT_DIR, these two parameters can be used in the main file as abbreviations of the corresponding directory set. Usually you set IN_DIR as the directory where there is all base architecture and OUT_DIR as the directory where you want to save the transformed architecture.
The TEMPLATE statement is a command that you should use to define Travulog templates used in ADD_MODULE_LAYER command as you can see in listing 4.16. Right after TEMPLATE there is the *template id* that you will use in HTV code, in the next line is defined the Travulog file of the template with the FILE key and at line five is defined the parameters template file with the PARAM_FILE key. The TEMPLATE statement terminate with the keyword END_TEMPLATE at line six, anyway in this file there is the possibility to define multiple template id using the structure defined before.

```
1 //// IMPORT htv_pkg.tv
2     .
3     .
4     .
5 //// ADD_MODULE_LAYER
6 //// TEMPLATE ft_template
```

```
7  //// INFILE IN_DIR/cv32e40p_prefetch_buffer.sv
8  //// OUTFILE OUT_DIR/cv32e40p_prefetch_buffer_ft.sv
9      .
10     verilog instance
11     .
12 //// END_ADD_MODULE_LAYER
13
```

Listing 4.16: Introduction of cv32e40p if stage

Line seven and eight set the package template file (see listing 4.13) and the output package file that will be created by the Toolchain. The last variable is the module prefix, this prefix is used to create automatic short name to substitute at MODNAME variable inside Travulog/HTravulog.
This parameters file has be introduced later in the Toolchain to simplify Python interface, for these reason all parameters are mandatory, anyway the code can be changed e.g., avoiding compulsory of MODULE_PREFIX that in some architecture probably doesn't exist.
The IMPORT of the same parameter file can be used in any HTravulog/SystemVerilog file you want.

ADD_LINE command: In some cases there is the needs to add a particular line in the converted SystemVerilog for this reason we create the ADD_LINE HTV command. It simply copies the string after " ADD_LINE " and write it in the converted file.
In the listing 4.15 the command is used at line four and it adds a new package in the converted file. This package will be created by the Toolchain from the package template PKG_FILE indicated in the IMPORT file.

NEW_MODULE_NAME & NEW_MODULE_FILE commands:
NEW_MODULE_NAME has a clear name, it set the name of the converted module. Instead NEW_MODULE_FILE set the file name of the new converted module. As you can see we use the OUT_DIR parameter in the definition of the name of the file, this parameter has been set previously in the IMPORT file.

4.3.2 Internal signals

In the current Toolchain state the definitions of all internal signals should be placed immediately after the IO declaration and must end with "END_DECLARATIONS" keyword (line 31 of listing 4.17).
In this part of SV you can insert the HTV code that will creates the new internal signals definition and assignment . Indeed all SV declaration will be deleted, in the converted file there will be only the elaboration of HTV code.

In listing 4.17 you can see all HTV code used in the internal signals part of the IF Stage, there is only one HTV command used called FOREACH, it allow to cycle on a certain list of signals substituting the name of each signal in SIGNAME parameter and the bit number definition in BITINIT.
FOREACH can be used both for declaration or for assignment, it also support the NOT statement as at line 5,9 and 17 of listing 4.17, on the right you can see the resulting SV code

```
1  //// FOREACH MAIN_MOD_INTERN
2  ////    logic [2:0]BITINIT SIGNAME_tr;
3  //// END_FOREACH
4
5  //// FOREACH MAIN_MOD_OUT NOT if_busy_o
6  ////  logic [2:0]BITINIT SIGNAME_tr;
7  //// END_FOREACH
8
9   //// FOREACH MAIN_MOD_OUT NOT if_busy_o
10 ////  assign SIGNAME = SIGNAME_tr[0];
11 //// END_FOREACH
12
13 //// FOREACH NEW_OUT
14 ////  logic [5:0]BITINIT SIGNAME_ft;
15 //// END_FOREACH
16
17 //// FOREACH NEW_IN NOT clk rst_n
18 ////  logic [5:0]BITINIT SIGNAME_ft;
19 ////  assign SIGNAME_ft =
20    {3'b0, 3'b0,3'b0, 3'b0, 3'b0, 3'b0};
21 //// END_FOREACH
22
23 //// FOREACH prefetch_busy
24 ////   assign if_busy_o=SIGNAME_tr[0];
25 //// END_FOREACH
26
27 //// FOREACH fetch_failed
28 ////  assign SIGNAME_tr =
29         {1'b0, 1'b0, 1'b0};
30 //// END_FOREACH
31
32 ////////////////////////////////////////
33 //// END_DECLARATIONS
34 ////////////////////////////////////////
35
36
37
38
39
40
41
42
43
44
45
46
```

```
logic [2:0]          if_valid_tr;
logic [2:0]          if_ready_tr;
logic [2:0]          prefetch_busy_tr;
logic [2:0]          branch_req_tr;
logic [2:0] [31:0]   branch_addr_n_tr;
logic [2:0]          fetch_valid_tr;
logic [2:0]          fetch_ready_tr;
logic [2:0] [31:0]   fetch_rdata_tr;
logic [2:0] [31:0]   exc_pc_tr;
logic [2:0] [23:0]   trap_base_addr_tr;
logic [2:0]  [4:0]   exc_vec_pc_mux_tr;
logic [2:0]          fetch_failed_tr;
logic [2:0]          aligner_ready_tr;
logic [2:0]          instr_valid_tr;
logic [2:0]          illegal_c_insn_tr;
logic [2:0] [31:0]   instr_aligned_tr;
logic [2:0] [31:0]   instr_decompressed_tr;
logic [2:0]          instr_compressed_int_tr;
logic [2:0]          instr_req_o_tr;
logic [2:0] [31:0]   instr_addr_o_tr;
logic [2:0]          instr_valid_id_o_tr;
logic [2:0] [31:0]   instr_rdata_id_o_tr;
logic [2:0]          is_compressed_id_o_tr;
logic [2:0]          illegal_c_insn_id_o_tr;
logic [2:0] [31:0]   pc_if_o_tr;
logic [2:0] [31:0]   pc_id_o_tr;
logic [2:0]          is_fetch_failed_o_tr;
logic [2:0]          csr_mtvec_init_o_tr;
logic [2:0]          perf_imiss_o_tr;

assign instr_req_o = instr_req_o_tr[0];
assign instr_addr_o = instr_addr_o_tr[0];
assign instr_valid_id_o =
    instr_valid_id_o_tr[0];
assign instr_rdata_id_o =
    instr_rdata_id_o_tr[0];
assign is_compressed_id_o =
    is_compressed_id_o_tr[0];
assign illegal_c_insn_id_o =
    illegal_c_insn_id_o_tr[0];
assign pc_if_o = pc_if_o_tr[0];
assign pc_id_o = pc_id_o_tr[0];
assign is_fetch_failed_o =
    is_fetch_failed_o_tr[0];
assign csr_mtvec_init_o =
    csr_mtvec_init_o_tr[0];
```

```

```

Listing 4.17: HTV code for new signals

```
assign perf_imiss_o = perf_imiss_o_tr[0];

logic [5:0] [2:0]   is_broken_o_ft;
logic [5:0]         err_detected_o_ft;
logic [5:0]         err_corrected_o_ft;
logic [5:0] [2:0]   set_broken_i_ft;
assign set_broken_i_ft = {3'b0, 3'b0, 3'b0,
    3'b0, 3'b0, 3'b0};

assign if_busy_o = prefetch_busy_tr[0];

assign fetch_failed_tr = {1'b0, 1'b0, 1'b0};
```

Listing 4.18: Converted SV code

4.3.3 Create a new module

In order to apply our FT template we should have a file containing the SV module to transform. For these reason we should divide the IF Stage in six block as depicted in **Figure 3.5**, anyway only the Prefetch Buffer, the Aligner and the Compressed decoder are already in separate file, for this reason we create a new command to create a module from a piece of SV that describe a separate architecture.

The command is called CREATE_MODULE as you see in listing 4.19, it takes as the first argument the string on the same line cv32e40p_if_stage_fsm_logic, this should be the name of the module to create. In the following line is set the OUTFILE, it is the file that will be used to save the new module SystemVerilog. In this line is used the OUT_DIR parameter set in the IMPORT file.
After these two setting are defined inputs and outputs of the new module, these IO signals could be both internal signals or IO of the main module (the IF Stage in our case).

```
//// ADD_MODULE_LAYER
//// TEMPLATE ft_template
//// INFILE OUT_DIR/cv32e40p_if_stage_fsm_logic.sv
//// OUTFILE OUT_DIR/cv32e40p_if_stage_fsm_logic_ft.sv
////
//// CONNECT  IF clk rst_n IN = IN
////         IF MAIN_MOD_IN IN = {IN , IN , IN }
////           IF NEW_IN IN = IN_ft[MAIN_MOD_ID_CURRENT_MOD_ID]
////         IN = IN_tr
////         IF NEW_OUT OUT = OUT_ft[MAIN_MOD_ID_CURRENT_MOD_ID]
////         OUT = OUT_tr
//// END_CONNECT

////   CREATE_MODULE cv32e40p_if_stage_fsm_logic
////   OUTFILE OUT_DIR/cv32e40p_if_stage_fsm_logic.sv
////
```

```
17 ////   IN pc_set_i
18 ////      fetch_valid
19 ////      req_i
20 ////      if_valid
21 ////      aligner_ready
22 ////   END_IN
23 ////   OUT branch_req
24 ////       fetch_ready
25 ////       perf_imiss_o
26 ////   END_OUT
27     .
28 // FSM state transition logic SV code
29     .
30     .
31 ////   END_CREATE_MODULE
32 //// END_ADD_MODULE_LAYER
33
```

Listing 4.19: HTV code to create a new block and transform it using TV template

Using the settings explained before the command execute these steps:

- **Check:** The directory of the oufile is checked, the input and output signals are checked to verified that are used inside the SV code of the new module.
- **Port declaration:** The IO signals of the new module are searched in the main module to find the number of bit, this information is used to create the new declaration statement.
- **Internal signals:** All signals used inside the SV code of the new module that aren't in the IO declaration are considered internal signals and they will be init in the new block.
- **Creation of New Module:** Using all previous information the new module is created and saved in the outfile.
- **Instance:** Finally in the new main module is written the instance of the new module, the connection is automatic. The creation of the new main module is done using a python string and it is written in the file at the end of the process. Thanks to this can be used nested HTV command as you see in listing 4.19 where the CREATE_BLOCK command is nested inside a ADD_NEW_LAYER command.

4.3.4 Use Travulog template

The final objective of the HTV code in this Thesis is the application of the Travulog template automatically. Indeed the HTV command ADD_NEW_LAYER is designed to simplify the transformation of an instanced module using a Travulog template. The

name of the command is referred to the type of template, in our case inside the converted Travulog module is instanced the basic module and so we need to include this module during compilation. If you imagine the hierarchical order of the final structure the top module is the if_stage then we have the cv32e40p_if_stage_fsm_logic_ft (it is instanced inside the if_stage) and finally the cv32e40p_if_stage_fsm_logic module (it is instanced inside the cv32e40p_if_stage_fsm_logic_ft). Therefore the command is called ADD_NEW_LAYER because it add an architectural layer between the main module and one of its submodules.

We use this specific name because a Travulog template can be used to create also a standalone architecture, probably for other applications then Fault Tolerance. In this way if you need to apply such template you can create your command and use it.

```
1  cv32e40p_if_stage_fsm_logic_ft if_stage_fsm_logic_ft
2  (
3      // Input ports of if_stage_fsm_logic_ft
4      .pc_set_i               (  { pc_set_i  , pc_set_i  , pc_set_i  } ),
5      .fetch_valid            (  fetch_valid_tr                       ),
6      .req_i                  (  { req_i  , req_i  , req_i  }         ),
7      .if_valid               (  if_valid_tr                          ),
8      .aligner_ready          (  aligner_ready_tr                     ),
9
10     // Input diff ports of if_stage_fsm_logic_ft
11     .clk                    (  clk                                  ),
12     .rst_n                  (  rst_n                                ),
13     .set_broken_i           (  set_broken_i_ft[CVIFST_IFSTFSLOFT] ),
14
15     // Output ports of if_stage_fsm_logic_ft
16     .branch_req             (  branch_req_tr                        ),
17     .fetch_ready            (  fetch_ready_tr                       ),
18     .perf_imiss_o           (  perf_imiss_o_tr                      ),
19
20     // Output diff ports of if_stage_fsm_logic_ft
21     .is_broken_o            (  is_broken_o_ft[CVIFST_IFSTFSLOFT] ),
22     .err_detected_o         (  err_detected_o_ft[CVIFST_IFSTFSLOFT] ),
23     .err_corrected_o        (  err_corrected_o_ft[CVIFST_IFSTFSLOFT] )
24 );
25
```

Listing 4.20: SV result code after the conversion from HTV code of if stage fsm logic module created using CREATE MODULE command

In listing 4.19 you can see the ADD_MODULE_LAYER command. The command arguments must start at the second line, and their order can be whatever, the complete command start with "//// ADD_MODULE_LAYER" and it ends with "//// END_ADD_MODULE_LAYER", the SystemVerilog code inside the command should be an instance of a module, in listing 4.19 this instance is created by the command CREATE_MODULE. The arguments that must be set are:

- **TEMPLATE id:** This argument indicates what is the Travulog template to use, the id used must be defined in the IMPORT file as described in paragraph 4.3.1. This Travulog Template will be used to add a layer that apply FT techniques.

- **INFILE filename:** This is the full path of the file containing the module to convert that should be the same instanced in the SV below (line 27 - 30 of listing 4.19). The extension is irrelevant and in the path can be used parameters defined in the import file, in listing 4.19 line 3 the directory of the INFILE is OUT_DIR, this is why the CREATE_MODULE save the new module file in the OUT_DIR as set at line 15. So CREATE_MODULE create the new module in the OUT_DIR and replace the CREATE_MODULE command with the instance of the new module, then the ADD_MODULE_LAYER use the new module in OUT_DIR directory and the instanced SV to create the FT layer.

- **OUTFILE filename:** This parameter sets the name of the output filename in which will be saved the Travulog template converted.

- **CONNECT .. END_CONNECT:** This command allow the designer to decide the final connection of the Converted template instance. In order to understand this command notation you can look at the final instance in listing 4.20, the first IF on the clk and rst_n orders to connect these signals with the same name at line 11 and 12 of listing 4.20.

  The second IF order to triplicate the input signals if they are input of the main module, indeed MAIN_MOD_IN will be replaced with the list of the if_stage input, so at line 4 there is the triplication of pc_set_i signal. Also the clk and rst_n are input signals of the if_stage, anyway they are already connected in the previous IF and so they are not considered in this IF.

  In the third IF there are three new parameters: NEW_IN that is the list of new signals respect to the basic module (in this case: set_broken_i, clk and rst_n), MAIN_MOD_ID that is an automatic ID created b the Toolchain that is composed of the concatenation of the first two letters of each word in the name of the main module, so CV32e40p_IF_STage become CVIFST, finally CURRENT_MOD_ID is another automatic id created in the same way as MAIN_MOD_ID but referred to the name of the new module which is extrapolated by the OUTFILE filename. Knowing the means of this parameters, the third IF orders to connect the new input how it is done at line 13 of listing 4.20, indeed clk and rst_n are already connected by previous IF.

  The fourth IF connect the remaining input simply adding a suffix.

The fifth IF use the parameter NEW_OUT that is like NEW_IN but for outputs, so this IF connects new output adding a suffix how it is done at line 21,22 and 23 of listing 4.20.

The last IF connects the remaining output adding a suffix.

These parameters enable the creation of a custom instance automatically. In listing 4.20 all connection is done using the same name, this is why the previous CREATE_MODULE command creates a basic instance in which each IO is connected with itself. Anyway ADD_MODULE_LAYER command is powerful since it look at previous connection and use it for the new instance with some changes according to CONNECT command and new signals from template. To clarify ideas we include listing 4.21 where there is the instance created by a ADD_MODULE_LAYER command applied to the instance of the Compressed Decoder, the old instance was hand written and so there are custom connection as you can see at line 8, 16, 17 and 18. This custom connection are elaborated by the ADD_MODULE_LAYER according to CONNECT command and the new connection is created.

```
1  cv32e40p_compressed_decoder_ft
2  #(
3          .FPU                 (  FPU                 )
4  )
5   compressed_decoder_ft
6  (
7          // Input ports of compressed_decoder_ft
8          .instr_i             (  instr_aligned_tr              ),
9
10         // Input diff ports of compressed_decoder_ft
11         .clk                 (  clk                           ),
12         .rst_n               (  rst_n                         ),
13         .set_broken_i        (  set_broken_i_ft[CVIFST_CODEFT]  ),
14
15         // Output ports of compressed_decoder_ft
16         .instr_o             (  instr_decompressed_tr         ),
17         .is_compressed_o     (  instr_compressed_int_tr       ),
18         .illegal_instr_o     (  illegal_c_insn_tr             ),
19
20         // Output diff ports of compressed_decoder_ft
21         .is_broken_o         (  is_broken_o_ft[CVIFST_CODEFT]    ),
22         .err_detected_o      (  err_detected_o_ft[CVIFST_CODEFT] ),
23         .err_corrected_o     (  err_corrected_o_ft[CVIFST_CODEFT] )
24 );
25
```

Listing 4.21: SV result code after the conversion from HTV code of Compressed Decoder module

4.4 Test of the Toolchain

In order to simplify the use of the Toolchain we show a typical example of directory tree that can be used:

```
├── htravulog.py
├── moddata.py
├── travulog.py
├── test_htravulog.sh
├── out
│   ├── cv32e40p_aligner_ft.sv
│   ├── cv32e40p_aligner.sv
│   ├── cv32e40p_compressed_decoder_ft.sv
│   ├── cv32e40p_compressed_decoder.sv
│   ├── cv32e40p_if_pipeline_ft.sv
│   ├── cv32e40p_if_pipeline.sv
│   ├── cv32e40p_if_stage_fsm_logic_ft.sv
│   ├── cv32e40p_if_stage_fsm_logic.sv
│   ├── cv32e40p_if_stage_ft.sv
│   ├── cv32e40p_pkg2.sv
│   ├── cv32e40p_prefetch_buffer_ft.sv
│   ├── cv32e40p_prefetch_buffer.sv
│   ├── cv32e40p_program_counter_definition_ft.sv
│   └── cv32e40p_program_counter_definition.sv
├── templates
│   ├── cv32e40p_pkg2.sv
│   ├── ft_template_parameters.sv
│   └── ft_template.sv
└── test
    └── arch
        ├── cv32e40p_aligner.sv
        ├── cv32e40p_compressed_decoder.sv
        ├── cv32e40p_if_stage.sv
        ├── cv32e40p_prefetch_buffer.sv
        ├── htv_pkg.tv
        └── include
            └── cv32e40p_pkg.sv
```

This is a part of the directory organization in the Toolchain repository https://github.com/Elia1996/Travulog, the first three file are the whole Toolchain written in Python, the fourth file is a bash script that run the analysis of testarchcv32e40p_if_stage.sv file in which there is the HTravulog code.
In the templates directory there is the template parameters file (PARAM_FILE in htv_pkg.tv) ft_template_parameters.sv, the package structure (PKG_OUT_FILE) file cv32e40p_pkg2.sv and the Travulog FT template (TEMPLATE FILE) ft_template.sv.
In the directory testarch there is a part of the core architecture that we need, here there is the IMPORT file htv_pkg.tv where there are parameters useful for correct Toolchain conversion and the if_stage with HTV code inside.
Finally in out directory there is the SV code generated by the Toolchain, here you can see the new modules: program_counter_definition, if_stage_fsm_logic and if_pipeline. Each basic module has the correlated FT layer which is instanced inside the if_stage.

Chapter 5

Verification

The verification of our architecture was done starting from the existing verification repository of the cv32e40p. It is maintained by the Open HW Group https://github.com/openhwgroup/core-v-verif and all its feature are described in detail here https://core-v-docs-verif-strat.readthedocs.io/en/latest/.
This part of the thesis is done together with Marcello Neri and Luca Fiore, we fork the core-v-verif repository and we start to create our tools in https://github.com/RISKVFT/core-v-verif, here we create some scripts which automatize verification through:

- **Benchmarks compilation:** benchmarks are a set of programs written in C language which should be cross compiled in order to run on the cv32e4op core. The toolchain for the compilation already exists, we simpify the interface to speed up experimentation.
- **Core Verification:** we need a way to verify that our new stage inside the core is working properly, so we create a comparison structure in which the reference core is simulated saving inputs and outputs of the reference stage. Then our new stage inside the core is simulated using reference inputs as stimulus and finally the outputa of our stage are compared to the reference one to find any differences. This comparison can be done for whatever software of the benchmark.
- **Fault Injection:** Once the new Instruction Fetch is verified we should check the behaviour of the stage respect to faults, to do this we start from the verification structure: as first step we save reference inputs and output of the stage as before, then we simulate the core with the new stage but we inject fault during simulation, finally the outputs are compared to find how many errors arrive to the output of the stage. The final FT level is the ratio between the output errors and injected errors.

  In this process the most challenging part is the fault injection, indeed we spent many time to understand how inject faults, anyway now our script is able to inject transient

and permanent faults, the injection is uniformly distributed along simulation time and on all bits of the signals, finally the signals in which inject faults can be changed.

We choose Statistical Fault Injection since it is the most common used methods, it exists also analytical estimation of FT level [41] and ground radiation test.

- **FT level evaluation:** During all simulation our script save many data about faults and errors, in this way at the end of the process it automatically calculates the number of faults and the FT level (which is $1 - \frac{N_{out\_errors}}{N_{injected\_faults}}$).In order to have a better coverage of core functionality during Fault Injection we simulate all benchmark and we create a script that elaborate all FI data of these simulations, giving a global FT level percentage to speed up FI verification.

The script which manage all the previous functionalities is called *comp_sim.sh* and it is located in core-v-verif/cv32/sim/core/. In order to simplify the use of the script we create a man page *comp_sim_man* in the same directory which you can visit using man command : "man ./comp_sim_man". The man page can be visualized also with the command "./comp_sim.sh -h".
In the next sections we analyze the simulation flow we use in the comp_sim script and then we discuss the main results.

5.1 Simulation Flow

Now we start to describe in detail the flow we have followed to verify and simulate fault injection in the if stage, during the description we also show directories organization, scripts behaviour and command arguments of comp_sim.sh. We finally underline some design and simulation choice. This is a brief summary of the next paragraph:

- **Initial Setup:** It describe the basic operations to perform in order to begin to simulate the new stage.
- **Benchmark Compilation:** It describe where to place benchmark programs and how to cross-compile it using comp_sim.sh.
- **Core Functional Verification:** It describe all steps performed to compare reference and new architecture together, it also explain how to execute this comparison using comp_sim.sh script.
- **Fault Injection Process:** Here is described the structure of the Fault Injection simulation using vsim_stage_compar.tcl script, it is also described how to execute a FI simulation using comp_sim.sh.
- **Number of simulations and Accuracy:** It describe the Statistical Fault Injection method and the formula used to find the number of simulation to perform for a certain accuracy.

5.1.1 Initial Setup

When you clone for the first time the core-v-verif repo from `https://github.com/RISKVFT/core-v-verif.git` you should checkout the FT_verif_Elia branch and run a little script which set current base directory in all scripts, its name is *set_core_v_verif.sh* and it is located in core-v-verif/ directory. Now all scripts know what is the path before core-v-verif dir and they could build absolute paths correctly.

The complete cv32e40p core with our new if stage is in `https://github.com/RISKVFT/cv32e40p/tree/FT_Elia` repository, so as first step set it as the ft architecture, this can be done using **-a** option of comp_sim.sh, in particular we use "-a ftr https://github.com/RISKVFT/cv32e40p" to set the repository url and "-a ftb FT_Elia" to set our branch. We also set the reference architecture with "-a refr https://github.com/RISKVFT/cv32e40p" and "-a refb master", this will be the reference architecture that we will use to do the functional verification. At the end of settings we can check them using "-a i".

Each time that the script is run it checks the repositories and the branches of reference and ft repositories which are located at core-v-verif/core-v-cores, if the repository or the branch is wrong the script clones and changes branch automatically.

5.1.2 Benchmark Compilation

In order to compile all software of the benchmark we create the directory cv32/tests/programs/custom_FT where we saved all c file. Then we create the build_all.py script in this directory which allows to cross compile all c file using make files, the output hex files are saved in cv32/tests/programs/out dir.
In order to easly copile c files we should set the compiler script and the out directory into comp_sim using the command:
"-b d cv32/tests/programs/build_all.py cv32/tests/programs/out", the path should be relative to the core-v-verif directory. Now after any change to the c file we can easly recompile it with "-b c" command. At the end of the process we have all .hex file in the "out" directory and they are ready to be used in core simulation.

5.1.3 Core Functional Verification

The functional verification of the core is the first important step to check the correct behaviour of the new core in normal condition.
To do this we should compare the output of the reference core with the core containing the new stage, anyway this approach is slow since we needs to simulate the whole core using many resources and time.
In order to reduce computational load we decide to implement a way to compare both the complete core or only a certain stage of the core. To do this we follow these steps:

- **Save Reference Input:** Both for core or for stage comparison the first step is to run the reference architecture and save the input data. So we create a tcl script in cv32/sim/questa directory called vsim_save_data_in.tcl which can be run with -b asvb ref hello_world save_data_in if_stage" , in this command asvb argument is a legend for the means and order of the next arguments: "a" is related to "ref" that is the reference architecture, "s" indicates the software to use that in this case is "hello_world" (the software should be a .hex file in the out dir of the benchmark which we compile before), "v" indicates the suffix of the tcl script, in this case "save_data_in", and finally "b" is related to the name of the stage.

  The order of the letter in the first arguments give the order of the next arguments, e.g.,the previous command is equivalent to "-b avbs ref save_data_in if_stage hello_world".

The command described simulate the reference architecture with hello_world software using vsim_save_data_in.tcl as QuestaSim script for saving the input data of the if_stage in a .vcd file located in /cv32/sim/core/sim_FT/dataset/, this vcd file will be automatically named "gold-ref-if_stage-hello_world-in.vcd".

- **Save Reference Output:** In this case the procedure is very close to the previous, indeed the command to run is "-b asvb ref hello_world save_data_out if_stage", it will creates the file "gold-ref-if_stage-hello_world-out.wlf . This is why to enable the output comparison we needs a wlf file.

- **Compare Architectures:**

  To compare the two architecture we should use vsim_stage_compare.tcl script (always located in questa directory). This script is the most complex because it contains all code to do comparison, fault injection and data saving. The behaviour is changed using some environmental variables which are set by the comp_sim script. For the current step we use this script to simulate the new stage using the Reference Inputs as stage stimulus and to compare the simulation outputs with the Reference wlf file, all these operations can be done with the command: "-b atvsb ref ft stage_compare hello_world if_stage". Here comp_sim use the previous vcd and wlf file created, so if they don't exist we will have some errors.

- **Show results:** At the end of the simulation all data file are saved in sim_FT/sim_out directory, here there will be a file called cnt_error-if_stage-hello_world-1-0.txt , where "1" is referred to the number of simulation while "0" means that we don't use fault injection. The second file is the info-if_stage-hello_world-1-0.txt, it contain the simulation time duration and the number of signals in which can be applied fault injection. Finally there is a file called signals_fault_injection-if_stage-hello_world-1-0.txt which we will use in FI. So at the end of the simulation the cnt_error file should contain "0" as error number, in this way we have checked that the new stage give the same output of the reference.

- **GUI:** When some errors occurs we could run again the comparison command using the -g option to open the GUI while QuestaSim does comparison, in this way we could see all signals and the source of error in the architecture.

The previous steps should be repeated for each c firmware in the benchmark in order to increase as much as possible the coverage of the stage verification.
After a high number of test we decide to condense the flow above in a single command: "-sfiupi atvsb ref ft stage_compare hello_world if_stage" . This command is powerful because it checks that the existence of wlf and vcd files, if they don't exist the script begins the process to create them automatically, then it runs the simulation to verify the

stage. As before if we need to use the gui we should only add "-g" as last argument and during simulation the gui will open.
Our new if_stage is highly configurable through SV parameters, therefore a complete verification means the check of each configuration, anyway there are approximately 96 possible state for each sub-block and so about 500 possible configuration for the overall stage. This is a really high number and so we can't verify all possibility. Probably this can be done with a script able to change the SV parameters and then simulate the core in each configuration, anyway this can be considered as a future work.
Knowing this issue we decide some configuration useful to show the ability of the new architecture and we verify only these, the analysis is done in the Results section 5.2.

5.1.4 Fault Injection Process

The fault injection process is managed by the vsim_stage_compare.tcl script that we already mentioned before. These are the main steps that the script follows:

- **Save Variable:** comp_sim sets many variable according to options and arguments used, these environmental variables are set using export bash command and saved in tcl variables inside the stage_compare script.
- **Set signals for FI:** Inside stage_compare there is a list called sim_fi_sig that should contain the name of signals in which inject faults. To create this variable we use "find nets" command with regular expression to select signals inside the if_stage adapt to fault injection. We can decide to inject faults only in sequential parts or also in combinatorial parts, we also can decide to exclude reset and clock signals because they are usually protected at layout level.
- **Fault distribution:** We should apply a uniform distribution of faults on all bits in order to simulate real behaviour of particle strike, so we copy each signal N times as the number of the bits of that signal, in this way when we use random selection we have a uniform distribution along all bits, e.g., if we have the signals sig1 with 4 bits and sig2 with 2 bits the final sim_fi_sig list will be [sig1 sig1 sig1 sig1 sig2 sig2].
- **Manage previous simulations:** Sometimes we happened to stop a simulation before the end, for these cases we create a recovery part that automatically load old interrupted simulation. This load needs to avoid the repetition of fault injection on the same signals at the same time.
- **FI of stage:** Suppose to run "-sfiupi atfcsb ref ft 1 100 hello_world if_stage" the new option here are: "f" which set or not fault injection (1 -> FI, 0 ->not FI) and "c" which refers to the number of simulation to do, in this case 100. When we run this

command, comp_sim run the Makefile in sim_FT directory which set the correct vcd as input stimulus and in turn run the stage_compare script inside QuestaSim. Stage_compare looks at the env variables "FI" and "CYCLE" to know whether inject faults and how many simulation it has to do. Then are set signals in which inject faults, are checked previous simulation and finally the fault injection on the stage begins.

In this part the first operation is to load the reference simulation or the wlf file saved previously, then is created the clock because vcdstim don't set it, finally is selected a random simulation time fi_instant and a random signal sig_fi, inside the selected signal is extracted a bit bit_number. The triplet fi_instant, sig_fi and bit_number creates a unique simulation id which is saved inside the signals_fault_injections files in the sim_FT/sim_out directory. Now the simulation starts and continue up to fi_instant, here the selected bit_number of the sig_fi signal is flipped and the output comparison begins.

This cycle corresponds to one Fault Injection Simulation and it is repeated CYCLE times as we set in comp_sim command.

As we already mentioned the fault is injected flipping the bit, anyway this could be done both using "force -deposit" command or the "force -freeze" command, the first create a transient error that can be overwritten, the second instead flip the bit permanently up to the end of simulation.

We observe that usually the faults lead to an error in few clock cycle, so we simulate some clock cycle and compare the signals, if there are errors we stop, otherwise we continue simulating a tenth of the remaining time and so on. This methods decrease significantly the simulation time. At the end of each simulation we write the simulation results on the signals_fault_injections file.

Using this methods the Fault injection is fast and efficient, it also inject fault uniformly in bits and time.

5.1.5 Number of simulations and Accuracy

A single FI simulation can needs from three seconds up to thirty seconds related to the workload of the software used, for this reason the choice of the number of simulations is important to reach good accuracy with the lowest simulation time. In order to achieve the best trade-off we use the Statistical Fault Injection (SFI) technique expressed in the

formula [42]:

$$n_{trade-off} = \frac{N_{max}}{1 - e^2 \frac{N_{max}-1}{t^2 \cdot p \cdot (1-p)}} \tag{5.1}$$

In this formula we have these parameters:

- N_{max}**:** Max number of injectables faults, it is equal to the total number of injectables bits multiplied by the simulation clock cycle number.
- **e:** It is the margin error of the final probability obtained during the simulations, this parameter have strong impact on $n_{trade-off}$, the more "e" decrease, the more $n_{trade-off}$ increase, a typical value for e is 0.05. In this way at the end of the FI process we will have a FT% between FT%-5% and FT%+5%.
- **t:** It is the cut-off point from which depends the confidence level, for t equal to 1.96 , 2.57 and 3.09 we have confidence level of 90%, 95% and 99%.
- **p:** It is the estimated probability of fault, even if it is unknown we decide to use the p value that maximize the product $p \cdot (1-p)$ in order to have maximum n_trade-off, this value is 0.5.
- $n_{trade-off}$**:** It is the estimated number of simulation in order to have a certain statistical confidence level of the final fault tolerance level, according to the previous parameters.

Since N_{max} depends on the simulation clock cycle, each software will have the corresponding $n_{trade-off}$. In order to evaluate this number of simulation we create the tcl script vsim_cycle_to_certain_coverage.tcl which can be executed with the command "-b asvb ref hello_world cov if_stage" where cov is a script abbreviation for cycle_to_certain_coverage. Inside cycle_to_certain_coverage you can set the signals you what to use in fault injection, the error margin and the cut off point obtaining the number of simulations you should run.

In this section we have described the software architecture we create over the core-v-verif repository in order to do functional and FI simulation in automatic way. In the next section we describe the main results we reach for our stage using the techniques explained above.

5.2 Results

Initially we should analyze the fault tolerance level of the original architecture in order to have a comparison metrics for the FT arch. In table 5.1 there are all benchmark software with the relative FT percentage:

Software	**FT Level [%]**	**N sim**	**N err**
coremark_1	89.2	987	107
counters	89.0	1028	113
csr_instructions	90.0	999	100
cv32e40p_csr_access_test	88.2	941	111
dhrystone	89.7	970	100
fibonacci	87.2	993	127
generic_exception_test	87.9	1046	127
hello_world	86.4	963	131
illegal	90.7	951	88
interrupt_bootstrap	87.7	972	120
interrupt_test	89.8	983	100
misalign	89.8	976	100
modeled_csr_por	89.4	970	103
perf_counters_instructions	88.9	967	107
requested_csr_por	88.5	1023	118
riscv_arithmetic_basic_test_0	78.4	1044	225
riscv_arithmetic_basic_test_1	79.2	1023	213
riscv_ebreak_test_0	87.0	1000	130

Table 5.1: Results of FT Level for each software for the reference IF stage architecture, transient errors, confidence level of 99% and error of 4%

The global FT level is **87%**. The fault injection is done both on sequential and combinatorial signals, in this way we have a good approximation of real FT level. Now we are ready to test the new architecture in two different configuration.

Full FT configuration The first architecture we verify is full FT so each sub-block has: FT variable set to 1, we don't use triple voter so all TOUT are 0 and also all TIN are 0, for permanent fault configuration we use increment and decrement equal to 1 and the Breaking Threshold equal to 3 anyway this part isn't tested. The configuration setted can be considered a fine-grain TMR plus permanent error detection through alpha counter architecture. The results of trasient Fault Injection is shown in Table 5.2

Software	FT Level [%]	N sim	N err
coremark_1	99.4	534	3
counters	99.6	546	2
csr_instructions	99.0	617	6
cv32e40p_csr_access_test	98.6	589	8
dhrystone	99.5	568	3
fibonacci	99.7	608	2
generic_exception_test	99.0	578	6
hello_world	99.1	580	5
illegal	98.9	561	6
interrupt_bootstrap	98.8	588	7
interrupt_test	99.0	599	6
misalign	99.6	554	2
modeled_csr_por	99.0	578	6
perf_counters_instructions	99.3	584	4
requested_csr_por	99.8	537	1
riscv_arithmetic_basic_test_0	98.8	575	7
riscv_arithmetic_basic_test_1	99.3	577	4
riscv_ebreak_test_0	99.7	597	2

Table 5.2: Results of FT Level for each software with all blocks protected in the new architecture, transient fault, confidence level of 99% and error of 5%

The global fault tolerance of the stage compleatly protected is **99.2%**, it isn't 100% because we inject fault also in the signals which connect blocks each other. The errors introduced by these signals can be eliminated using triple voter configuration. Anyway this solution isn't effective because there is a complete triplication of each sub blocks which can be optimized using TMR on the overall stage, the TMR on the complete IF stage reduces the number of voters and avoid the sub blocks interconnection problem. We test this solution to have the maximum FT level reachable with our architecture, anyway in real application will be used trade-off configuration like the one shown in the next paragraph.

FT trade-off configuration We want to set a configuration in which the lower number of TMR sub-block reach a considerable Fault Level, to do this the first step is to understand what is the critical part of the whole stage. For this reason we use the reference architecture results, in Table 5.3 there is the result of this simulation grouped in blocks.

subblocks	**N err**	**N sim**	**FT level [%]**
aligner_i	1952	11902	83.6
prefetch_buffer_i	2142	19484	89.0

Table 5.3: Results of FT Level for the two main blocks of the reference IF stage

As you can see the prefetch_buffer is the block with the higher number of errors and so the most impactful in terms of Fault Tolerant . We should look at the total number of errors since the FT level of each block doesn't consider the number of bits. Looking at the total number of error we automatically consider the number of bits and so the probability that a particle reaches the block. These statements are true because we inject faults using uniform distribution along signals and bits and so a higher number of error in a stage means a higher contribution to FT level.
We also exclude the other 4 blocks because they are little and so have lower impact on FT level.

Starting from these considerations we protect only the prefetch buffer, the protection of this block is easy to perform since we should only set PRBU_FT = 1 in the cv32e40p_pkg2.sv package. After this configuration we run fault injection and we find the results in table 5.4.

Software	FT Level [%]	N sim	N err
coremark_1	96.8	468	15
counters	96.5	519	18
csr_instructions	96.0	524	21
cv32e40p_csr_access_test	96.4	498	18
dhrystone	95.4	482	22
fibonacci	95.4	525	24
generic_exception_test	96.8	537	17
hello_world	97.7	483	11
illegal	96.6	502	17
interrupt_bootstrap	98.5	475	7
interrupt_test	96.9	518	16
misalign	96.7	512	17
modeled_csr_por	97.9	487	10
perf_counters_instructions	96.0	529	21
requested_csr_por	96.2	497	19
riscv_arithmetic_basic_test_0	96.7	522	17
riscv_arithmetic_basic_test_1	94.2	504	29
riscv_ebreak_test_0	98.6	502	7

Table 5.4: Results of FT Level for each software with only Prefetch Buffer protected in the new architecture ,transient fault, confidence level of 99% and error of 5%

The global FT level is **96%**, this is a good number if we want increase the FT level with very low overhead in area. This result is near to the FT level of the full FT configuration and for this reason is is a good trade off.

Chapter 6

Conclusions

In this thesis we have designed an Instruction Fetch stage for the cv32e40p core with these properties:

- **Automatic Creation:** We create a Travulog/HTravulog Toolchain able to transform the original Instruction Fetch in a Fault Tolerant stage. This means that we automatize the creation of the stage starting from: some Travulog template, some HTravulog code in the original IF stage and the Toolchain.
- **Configurable FT level:** In the design of the templates we have focused our attention to the configurability of the final Fault Tolerant IF stage, indeed we use many parameters which can change the FT level and properties of each block. In this way the new IF stage will be configurable according to the application.
- **Maintainability:** The automatic creation of the Fault Tolerant IF stage allows to the designer a higher level of maintainability because a change in the original architecture can be applied to the converted IF stage easly running the Toolchain.

Another important result of the thesis is the creation of the Travulog/HTravulog Toolchain that make possible the automatic creation of the FT IF stage.

The Toolchain is open source and can be used for many application, indeed it enable the transformation of an architecture using a Travulog Template. During this thesis we only show Fault Tolerant Template, anyway you can design whatever type of template you needs and verify the Toolchain support, if new Travulog commands is needed you can add it to the Python Toolchain.

In this way we create a tool to transform architectures reducing design time and encouraging the template reuse. For example the Fault Tolerance template designed during this

thesis can be used for other architectures.

In this thesis we also show that both Full and Trade-off configurations of the new Fault Tolerant IF stage created with the Toolchain has good results in terms of FT level (respectively 99% and 96%).

Future Work Surely the main work that could be done is test the remaining configurations of the new IF stage to see how it works. This is a long job because of the simulation time and it cannot be done completely, so you should choose the most important configurations and test those. The response to the permanent errors when varying the FT in the blocks and the use of the triple voter should certainly be tested.
The thesis opens new developments towards automating the design of architectures, so an important work is the improve of the toolchain. For example to facilitate the use we could add others Travulog/HTravulog commands and create a complete manual of the language. Many work can also be done on the creation of Template Library for Fault Tolerant transformation, in this way complex FT architectures can be reused reducing design time and during the creation of the template can be exploited all new FT techniques (e.g., genetic algorithm [34], resilient structure [43], byzantine fault tolerance [44]).

Bibliography

[1] *Smartphone statistics*. URL: https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/.

[2] *Personal Computer statistics*. URL: https://www.statista.com/statistics/263393/global-pc-shipments-since-1st-quarter-2009-by-vendor/.

[3] *Car statistics*. URL: https://www.best-selling-cars.com/international/2019-full-year-international-worldwide-car-sales/.

[4] *Total number of worldwide smartphone used*. URL: https://financesonline.com/number-of-smartphone-users-worldwide/.

[5] *Total number of cars*. URL: https://www.live-counter.com/number-of-cars/.

[6] *Total number of satellites*. URL: https://www.pixalytics.com/satellites-orbiting-earth-2020/?format=pdf.

[7] G. G. Maxwell. "Pacemaker reliability: design to explant". In: *Annual Reliability and Maintainability Symposium 1995 Proceedings*. 1995, pp. 460–464. DOI: 10.1109/RAMS.1995.513285.

[8] *Marcello Neri's Thesis*. URL: https://webthesis.biblio.polito.it/17871/1/tesi.pdf.

[9] *Luca Fiore's Thesis*. URL: https://webthesis.biblio.polito.it/17869/1/tesi.pdf.

[10] Elena Dubrova. "Fault Tolerant Design : An Introduction". In: *Ece.Nus.Edu.Sg* X.X (2013).

[11] Shubu Mukherjee. *Architecture Design for Soft Errors*. 2008. DOI: 10.1016/B978-0-12-369529-1.X5001-0.

[12] Hairong Sun, J.J. Han, and H. Levendel. "Availability requirement for a fault-management server in high-availability communication systems". In: *IEEE Transactions on Reliability* 52.2 (2003), pp. 238–244. DOI: 10.1109/TR.2003.812624.

[13] L. Valcarenghi, M. Kantor, P. Cholda, et al. “Guaranteeing High Availability to Client-Server Communications”. In: *2008 10th Anniversary International Conference on Transparent Optical Networks*. Vol. 3. 2008, pp. 34–37. DOI: 10.1109/ICTON.2008.4598649.

[14] ECSS. “Space Product Assurance - Techniques for radiation effects mitigation in ASICs and FPGAs handbook”. In: *Structure* April (2016).

[15] Paul S. Ho and Thomas Kwok. “Electromigration in metals”. In: *Reports on Progress in Physics* 52.3 (1989). ISSN: 00344885. DOI: 10.1088/0034-4885/52/3/002.

[16] *Fault-Tolerant Systems*. 2021. DOI: 10.1016/c2018-0-02160-x.

[17] Z. Zhang, R. Wang, Y. Wang, et al. “Impacts of Channel Doping on NBTI Reliability and Variability in Nanoscale FinFETs”. In: *2019 IEEE 26th International Symposium on Physical and Failure Analysis of Integrated Circuits (IPFA)*. 2019, pp. 1–4. DOI: 10.1109/IPFA47161.2019.8984834.

[18] S. Das, T. P. Dash, S. Dey, et al. “NBTI Degradation and Recovery in Nanowire FETs”. In: *2019 Devices for Integrated Circuit (DevIC)*. 2019, pp. 70–74. DOI: 10.1109/DEVIC.2019.8783566.

[19] K. Ota, R. Ichihara, M. Suzuki, et al. “Random Telegraph Noise after Hot Carrier Injection in Tri-gate Nanowire Transistor”. In: *2019 Electron Devices Technology and Manufacturing Conference (EDTM)*. 2019, pp. 169–171.

[20] W. Lin, W. Tsai, C. C. Cheng, et al. “Hot-Carrier Injection-Induced Disturb and Improvement Methods in 3D NAND Flash Memory”. In: *2019 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*. 2019, pp. 1–2. DOI: 10.1109/VLSI-TSA.2019.8804652.

[21] H. Kim, M. Jin, H. Sagong, et al. “A systematic study of gate dielectric TDDB in FinFET technology”. In: *2018 IEEE International Reliability Physics Symposium (IRPS)*. 2018. DOI: 10.1109/IRPS.2018.8353577.

[22] K. Joshi, S. W. Chang, D. S. Huang, et al. “Study of dynamic TDDB in scaled FinFET technologies”. In: *2018 IEEE International Reliability Physics Symposium (IRPS)*. 2018. DOI: 10.1109/IRPS.2018.8353665.

[23] Kirby Kruckmeyer Robert Baumann. *Radiation Handbook for Electronics*. 2020.

[24] “COSMIC-RAY PICTURE OF THE HELIOSPHERE.” In: *Johns Hopkins APL Technical Digest (Applied Physics Laboratory)* 6.1 (1985). ISSN: 02705214.

[25] P. Hazucha and C. Svensson. “Impact of CMOS technology scaling on the atmospheric neutron soft error rate”. In: *IEEE Transactions on Nuclear Science* 47.6 (2000), pp. 2586–2594. DOI: 10.1109/23.903813.

[26] Mengfei Yang, Gengxin Hua, Yanjun Feng, et al. *Fault-Tolerance Techniques for Spacecraft Control Computers*. 2017. DOI: 10.1002/9781119107392.

[27] P. V. Nekrasov, A. B. Karakozov, D. V. Bobrovskyi, et al. "Investigation of Single Event Functional Interrupts in Microcontoller with PIC17 Architecture". In: *2015 15th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. 2015, pp. 1–4. DOI: 10.1109/RADECS.2015.7365625.

[28] N. Seifert and N. Tam. "Timing vulnerability factors of sequentials". In: *IEEE Transactions on Device and Materials Reliability* 4.3 (2004), pp. 516–522. DOI: 10.1109/TDMR.2004.831993.

[29] Avizienis and Kelly. "Fault Tolerance by Design Diversity: Concepts and Experiments". In: *Computer* 17.8 (1984), pp. 67–80. DOI: 10.1109/MC.1984.1659219.

[30] L. A. Tambara, F. L. Kastensmidt, J. R. Azambuja, et al. "Evaluating the effectiveness of a diversity TMR scheme under neutrons". In: *2013 14th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. 2013, pp. 1–5. DOI: 10.1109/RADECS.2013.6937382.

[31] M. Masadeh, A. Aoun, O. Hasan, et al. "Highly-Reliable Approximate Quadruple Modular Redundancy with Approximation-Aware Voting". In: *2020 32nd International Conference on Microelectronics (ICM)*. 2020, pp. 1–4. DOI: 10.1109/ICM50269.2020.9331771.

[32] K. Siozios and D. Soudris. "A Methodology for Alleviating the Performance Degradation of TMR Solutions". In: *IEEE Embedded Systems Letters* 2.4 (2010), pp. 111–114. DOI: 10.1109/LES.2010.2083632.

[33] H. T. Vierhaus. "Combining fault tolerance and self repair in a virtual TMR scheme". In: *2013 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*. 2013, pp. 12–18.

[34] F. S. Khodadad and M. Jahed. "Optimization of a Cascading TMR system configuration using Genetic Algorithm". In: *IEEE 10th International Conference on Industrial Informatics*. 2012, pp. 470–474. DOI: 10.1109/INDIN.2012.6300853.

[35] K Asanovic and DA Patterson. "Instruction sets should be free: The case for RISC-V". In: *EECS Department, University* (2014).

[36] *RISC-V history*. URL: https://riscv.org/about/history/.

[37] *RISCV unprivileged isa*. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[38] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, 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* 25.10 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506.

[39] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, 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)*. 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976.

[40] L. Sterpone and M. Violante. "A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs". In: *IEEE Transactions on Nuclear Science* 52.6 (2005), pp. 2217–2223. DOI: 10.1109/TNS.2005.860745.

[41] G. Asadi and M. B. Tahoori. "An analytical approach for soft error rate estimation in digital circuits". In: *2005 IEEE International Symposium on Circuits and Systems*. 2005, 2991–2994 Vol. 3. DOI: 10.1109/ISCAS.2005.1465256.

[42] R. Leveugle, A. Calvez, P. Maistri, et al. "Statistical fault injection: Quantified error and confidence". In: *Proceedings -Design, Automation and Test in Europe, DATE*. 2009. DOI: 10.1109/date.2009.5090716.

[43] M. S. Farias, N. Nedjah, and P. V. R. de Carvalho. "Resilient Hardware Design for Critical Systems". In: *2019 IEEE 10th Latin American Symposium on Circuits Systems (LASCAS)*. 2019, pp. 237–240. DOI: 10.1109/LASCAS.2019.8667549.

[44] Leander Jehl and Hein Meling. "Towards Byzantine fault tolerant publish/subscribe: A state machine approach". In: *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems, HotDep 2013*. Association for Computing Machinery, 2013. ISBN: 9781450324571. DOI: 10.1145/2524224.2524232.

[45] L. A. C. Benites and F. L. Kastensmidt. "Automated design flow for applying Triple Modular Redundancy (TMR) in complex digital circuits". In: *2018 IEEE 19th Latin-American Test Symposium (LATS)*. 2018, pp. 1–4. DOI: 10.1109/LATW.2018.8349668.

[46] H. Pham, S. Pillement, and S. J. Piestrak. "Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor". In: *IEEE Transactions on Computers* 62.6 (2013), pp. 1179–1192. DOI: 10.1109/TC.2012.55.

[47] *RISCV privileged isa*. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf.

[48] *PULP-Platform Simulation Verification*. URL: https://core-v-docs-verif-strat.readthedocs.io/en/latest/pulp_verif.html.

[49] Davide Rossi, Antonio Pullini, Igor Loi, et al. "193 MOPS/mW at 162 MOPS, 0.32V to 1.15V voltage range multi-core accelerator for energy efficient parallel and sequential digital processing". In: *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*. 2016, pp. 1–3. DOI: 10.1109/CoolChips.2016.7503670.