# POLITECNICO DI TORINO

**Master's Degree in Electronic Engineering**


Politecnico
di Torino

**Master's Degree Thesis**

# Study and testing of a processor for Embedded Systems in the new space era

**Supervisors**

Prof. Luca STERPONE

Prof. Sarah AZIMI

Dr. Eleonora VACCA

**Candidate**

Giommaria PILO

**Academic Year 2022/2023**

# Summary

Space exploration has always fascinated the human race. Since the beginning space has been an object of speculation and study, with human often attributing to it spiritual and magical significance. During the XX century, space exploration was one of the research fields that attracted more funds and resonated best with the general public. Nowadays the world is entering a new era of space exploration. Launch costs have reduced significantly and the number of actors launching satellites into orbit will continue to increase. Major powers have set up space forces and are militarizing and weaponizing space in support of terrestrial warfighting capabilities. Lower costs of space travel are opening the door to the establishment of extraterrestrial footholds. Experts have identified 17,000 asteroids which can be exploited for resources' extraction, hinting at incredible profits for those who first will be able to claim them.

The aerospace industry is constantly on the search for more powerful, more efficient, and safer electronic components, while maintaining a high degree of testability, and a low cost. However, all electronic components in space have to withstand a harsh environment with heavy doses of ionizing radiation. This causes on the designs faults such as SETs, SEUs, and SEMUs. The occurrence of these events must not interfere with the ability of an electronic system to carry out its mission. The high failure rate of components in the space environment means that the aerospace industry has a great need for electronic components that satisfy the mission requirements for performance as well as presenting a high reliability. To fulfill these demands the processor Rempro was introduced, a soft-core for FPGAs designed by an Italian company to push forwards the limits of survivability for a computing system. The main goal of the project is to obtain a system capable of surviving as long as possible in the space environment without any need for human

intervention.

Starting from the implementation of the Rempro processor, this thesis work starts with an investigation of the processor architecture and development process. To increase reliability, two approaches are possible, Radiation Hardening by Process which involves using components that are more resistant to radiation due to intrinsical properties of the node technology used; and Radiation Hardening by Design, which increases tolerance using particular design techniques. To take advantage of the RHBP, Rempro targets a Flash FPGA for its implementation. This makes it immune to Single Event Upsets (SEUs), which are bit flips that affect sequential elements such as Flip-Flops, memories and so on. These faults are particularly dangerous for FPGAs because if they happen inside the configuration memory of the device they may modify the function that the Programmable Logic Blocks of the FPGA perform, drastically changing the behavior of the circuit. While Flash FPGAs are not affected by these problems, as the tensions necessary to flip bits are higher than what can be caused by ionized particles, they are still affected by the total dose of radiation, which damages the transistors over time, and the Single Event Transients, which are voltage spikes in the device logic, which can cause wrong values to be sampled in the sequential components of the circuit. To address radiation-induced faults, one of the most common RHBD techniques is used Triple Module Redundancy. This approach involves the triplication of critical modules of a design, using voters to preserve the correct behavior of the system in case of malfunctions to one of the modules. The processor goes to one step further implementing not only TMR on the computational cores but also N-MR on the device. The system is imagined in its entirety as a number of boards each mounting an FPGA with a TMR version of the processor in it. Each single system is equipped with a self-checking mechanism able to detect the occurrence of an unrecoverable fault. When the system cannot recover from the malfunction a substitution routine activates a new board that is able to autoconfigure itself to carry out the program first started in the defective board. In this way it is possible to obtain a system capable of remaining online as long as possible, executing critical parts of the mission, with a high degree of reliability, without the need for any kind of servicing.

To this extent if a triplet detects an unrecoverable malfunction in one of its

processors, it has to be capable to transfer the processor state to a triplet in another board in order to carry on execution. For this reason a bus was realized to communicate between two boards, together with a possible communication protocol and memory transfer routine to transfer the program state to the triplet in the new board. In order to simplify the testing phase, an UART module was also developed for the processor.

After the design phase, the new features were tested to validate their behavior. First test involved verifying the UART module worked correctly, in order to make sure all the following test could be carried out with no problems. The second one involved measuring the latency of the wires composing the bus to confirm that it would not cause problems during communication, and allow for the timely transfer of data between processors. The last test was to validate the bus and transfer routine themselves. Confirming the last one was an important step to establish the feasibility of the project.

In order to place the processor in the design space of other aerospace-oriented soft cores, it is compared against a RISC-V processor, the NEORV32. After studying its features and architecture, three benchmark programs have been realized, each with a different workload and a different stimulus for the processors modules. The benchmarks, written in assembly, constitute some common computational use cases for processors.

In order to evaluate and compare the reliability and fault tolerance of both processors, these programs were used as workload to execute while being subjected to a fault injection campaign. The designs were evaluated post-implementation to match as closely as possible the real case. A Python framework was developed to manage the campaign, generate the Tcl scripts for the ModelSim simulation and elaborate the results. The transient injections were performed at random times during the simulation, in random nodes inside the design, 100 times per node to gather meaningful statistics for the entirety of the design. Using a master Tcl script, it was possible to automate the injection process and the gathering of data on the signals and the results of the injection itself. Analyzing the results its possible to identify the criticalities of the design in order to increase efficiency when adding fault mitigation.

The general structure of the thesis is as follows. Chapter 1 gives an introduction

to the problems deriving from operating electronic circuits in space. Chapter 2 details the programs and hardware used in this thesis work. Chapter 3 describes the architecture of the Rempro system, how it originated and the architectural considerations that were made during its development. Chapter 4 illustrates the implementations tests performed to validate the new features added during the Rempro development. Chapter 5 introduces the RISC-V ISA and the NEORV32 processor, chosen for a comparison with Rempro. Chapter 6 explains how the fault injection campaign was planned and carried out. Chapter 7 presents the results of the fault injection and analyzes them with the aid of charts and graphs. Chapter 8 summarizes the conclusions and describes prospects for future research work.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**3PIC** Third Party Intellectual Property Cores 2

**AES** Advanced Encryption Standard 13

**ALU** Arithmetic Logic Unit 21, 25, 55, 66, 71

**ASCII** American Standard Code for Information Interchange 38, 42, 45, 48

**ASIC** Application Specific Integrated Circuit 12, 26

**BJT** Bipolar Junction Transistor 7

**CAD** Computer Aided Design 2

**CISC** Complex Instruction Set Computer 52

**CLI** Command Line Interface 16

**CMOS** Complementary Metal Oxide Semiconductor 13

**COTS** Commercial-Off-the-Shelf 2

**CPU** Central Processing Unit 52–57, 71

**CRCW** Concurrent Read Concurrent Write 20

**CSR** Control and Status Register 57

**DDR** Double Data Rate 15

**DRC** Design Rule Check 15

**DUT** Device Under Test 30

**FF** Flip-Flop 6, 59–66, 70–73, 75

**FIFO** First In First Out 13, 54, 56

**FPGA** Field Programmable Gate Array 2, 10, 12–15, 19, 20, 22, 23, 25–30, 32, 42–44, 52, 54, 55, 59, 60, 77

**FSM** Finite State Machine 31–33, 35, 57

**GND** Ground 32, 45

**GPIO** General Purpose Input/Output 32–35, 42, 43, 45–47

**GUI** Graphical User Interface 16

**HDL** Hardware Description Language 10, 15, 42

**IC** Integrated Circuit 12, 26

**IDE** Integrated Design Environment 16, 17

**IP** Intellectual Property 2, 15, 51

**IPB** Instruction Prefetch Buffer 54, 56

**IQR** Interquartile Range 72

**IR** Instruction Register 21

**ISA** Instruction Set Architecture 2, 19, 38, 45, 51, 52, 55–57, 78

**ISP** In System Programming 13

**JTAG** Joint Test Action Group 13

**LCD** Liquid Crystal Display 15

**LED** Light Emitting Diode 15, 35, 42

**LVCMOS** Low Voltage Complementary Metal Oxide Semiconductor 13

**LVDS** Low-Voltage Differential Signaling 15

**MBU** Multiple Bit Upset 5

**MUX** Multiplexer 31, 35, 38, 39, 45, 46

**NVM** Non Volatile Memory 13

**PC** Program Counter 21, 23, 24, 32

**PLL** Phase-Locked Loop 22

**RAM** Random Access Memory 13, 15, 20, 22–25, 27, 28, 32, 42, 55

**RHBD** Radiation Hardening By Design 6–8

**RHBP** Radiation Hardening By Process 7, 8, 19

**RISC** Reduced Instruction Set Computer 52

**RISC-V** Reduced Instruction Set Computer V 2, 51, 52, 57, 71, 73, 78

**RTL** Register Transfer Level 2, 15, 30, 47, 59

**SBU** Single Bit Upset 5

**SDC** Synopsys Design Constraints 17

**SDF** Standard Delay Format 59–61, 63

**SET** Single Event Transient 6, 7, 59, 62, 66

**SEU** Single Event Upset 5, 7

**SMP** Symmetric MultiProcessor 19

**SoC** System-on-a-Chip 15, 17, 57

**SRAM** Static Random Access Memory 6, 12–14

**TMR** Triple Modular Redundancy 8, 19, 22, 26, 32, 44, 72, 77, 78

**UART** Universal Asynchronous Receiver-Transmitter 30–35, 37–39, 41–43, 45–48, 77

**UMA** Uniform Memory Access 19

**USB** Universal Serial Bus 32, 35, 45, 48

# Part I

# Introduction

# Chapter 1

# Electronics in space

The increase in satellite launches, resulting from improvements in aerospace technology and miniaturization, has made numerous low-cost Commercial-Off-the-Shelf (COTS) components available in various fields such as instrumentation, data handling and acquisition, and telecommunications, among others. Furthermore, the cost of launches has been reduced, and Cubesat designs can now be financed and built even by academic institutions.

The aerospace industry has witnessed an increasing utilization of COTS components, Third Party Intellectual Property Cores (3PIC), and Computer Aided Design (CAD). Due to the lower production volume of dedicated electronic subsystems and components for space applications, it is more cost-effective to repurpose legacy components (i.e., not specifically designed for space) in order to strike a balance between development costs and computational performance.

The rise of the Reduced Instruction Set Computer V (RISC-V) Instruction Set Architecture (ISA) (Figure 1.1), along with other open-source Intellectual Property (IP), enables a better understanding of their vulnerabilities compared to the black-box characterization typically encountered when dealing with proprietary COTS processors. The availability of Register Transfer Level (RTL) models is a major advantage in these processors, enabling a more comprehensive evaluation of the effects of upsets in the microarchitecture. These IPs are often implemented in programmable logic, in particular in Field Programmable Gate Arrays (FPGAs) [1].

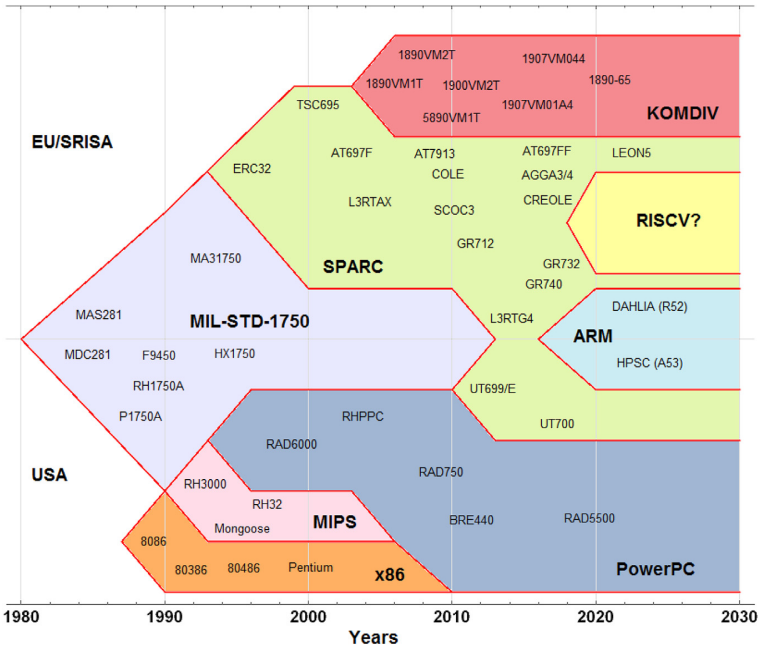



**Figure 1.1:** History and perspectives of ISA used in space programs [2].

After analyzing the vulnerability of a system to faults, it can be mitigated through redundancy at various levels of the design [2]. However, this approach comes with significant costs in terms of area, power, and performance, necessitating a compromise. The balance between vulnerability mitigation, increased area and power consumption, and performance loss depends on the dependability requirements (availability, reliability, safety, confidentiality, integrity, and maintainability) and performance objectives, as well as the target sector.

In space applications, significant attention is devoted to the availability and reliability of electronic systems. Reliability is defined as the probability, denoted by $R(t)$, that a system will continue to function correctly until the end of the time period $t_0 - t$, given that it was working correctly at the instant $t_0$. On the other hand, availability is defined as the probability, denoted by $A(t)$, that a system will be functioning correctly at a specific time t. Unlike reliability, which considers a time interval, availability refers to a single moment [3].

Space missions can endure for up to 15 years, and the entire space system is expected to provide service for at least 99.9% of the time. Additionally, since servicing the system is typically impossible, it is crucial that the system has the

capability to recover on its own if a fault occurs. Consequently, availability and reliability stand as primary requirements for any electronic system employed in a space mission [4].

## 1.1 Threats to dependability in space applications

According to [3] threats are phenomena that can impact a system, resulting in a decrease in its dependability, and they can be classified as faults, errors, and failures. A failure occurs when the delivered service deviates from the correct service. There are various failure modes that describe how a service can deviate from the correct behavior, but a failure signifies that at least one of the system's external states deviates from the correct state. This deviation is referred to as an error, which is a part of the overall system state that may lead to subsequent failure in providing the correct service. The cause of an error is known as a fault. Faults can be internal or external, and the presence of a vulnerability, i.e., an internal fault that allows an external fault to affect the system, is necessary for the external fault to cause an error and potentially lead to a failure (Figure 1.2).

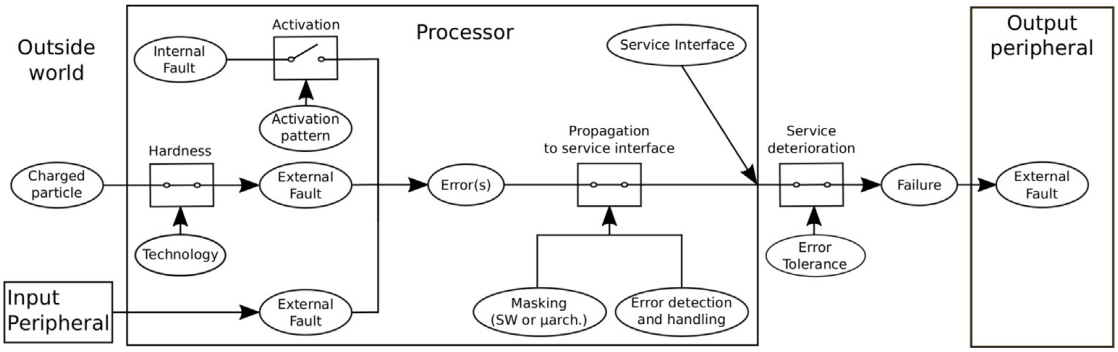



**Figure 1.2:** Typical interactions of threats with a processor providing a service to an output peripheral [4].

The relationship between faults, errors, and failures can be described as follows:

When a fault is present in a system but has not yet manifested any effect, it is considered dormant. Once the fault is activated, it leads to a system state that deviates from the expected one, resulting in an error. The error can propagate

within the system until it reaches the service interface, disrupting the delivery of the correct service and causing a misbehavior or failure. A service failure in a system can trigger external faults in downstream systems that receive service from it.

In space processors, the typical faults are external ones caused by changes in the charge stored in nodes due to particle strikes [5]. These faults are the cause of soft errors because they can be corrected by simply overwriting them with the correct value [6]. They are distinct from hard errors, which necessitate identifying the specific fault causing them in order to accurately replace the faulty unit [7].

### 1.1.1 Upsets

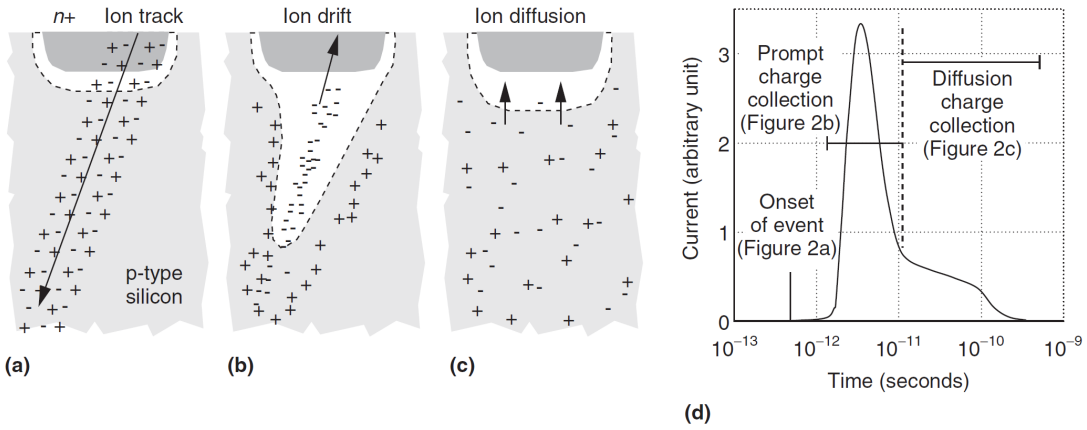



**Figure 1.3:** Charge generation and collection in a reverse-biased junction: formation of a cylindrical track of electron-hole pairs (a), funnel shape extending high field depletion region deeper into substrate (b), diffusion beginning to dominate collection process (c), and the resultant current pulse caused by the passage of a high-energy ion (d) [6].

Ionizing particles striking a sequential circuit (Figure 1.3) can alter the value stored in one or more sequential elements [8], known as Single Event Upset (SEU) or Single Bit Upset (SBU) when a single element is affected, and Multiple Bit Upset (MBU) when multiple elements are affected. The upset rate, $\lambda_{ev}$, depends on factors such as the level of radiation and shielding in the environment, the technology employed, and the specific component choices within the technology.

The environment experienced by the system also depends on the shielding, which is beyond the control of the designer. An electronic enclosure can greatly mitigate the occurrence of upsets in an electronic system. However, there is a limit to the effectiveness of shielding. It has been demonstrated that Galactic Cosmic Rays are not affected by shielding depths, leading to a plateau where increasing the shielding further does not result in a significant reduction in the upset rate [9].

In the same technology, different types of sequential elements, such as Flip-Flops (FFs) and Static Random Access Memory (SRAM) arrays, can exhibit varying upset rates. For instance, in the case of the OpenSPARC T2 in 65 nm technology, a range of sequential elements is employed, including SRAM arrays optimized for density, less dense yet higher-performance SRAM arrays, and FFs. Comparatively, the less dense SRAM arrays demonstrate half the susceptibility to upsets when compared to the density-optimized ones, while the FFs are one third as susceptible as the density-optimized SRAM arrays [10]. However, it is important to note that the situation may be reversed in different technologies [11].

This differentiation between SRAM and FFs can be attributed to the presence of temporal masking in FFs, which is absent in SRAM. In the case of an upstream sequential element connected to a downstream combinational logic path, an upset occurring in the sequential element between $t = t_{samp} - T_{prop}$ and $t = t_{samp}$ (where $t_{samp}$ is the sampling instant and $T_{prop}$ is the propagation time required for a signal to be sampled after propagating through the combinational path) will not propagate to the next sequential element. As the frequency increases, more faults are masked due to this temporal masking effect [12].

### 1.1.2 Single Event Transients

As Radiation Hardening By Design (RHBD) methodologies offer solutions to decrease the sensitivity of sequential elements to upsets, it is expected that effects on combinational logic will dominate in future technologies [13]. This effects, are known as Single Event Transients (SETs). When a particle strikes a combinational node, it can generate a voltage transient. This transient pulse may be sampled by downstream sequential elements, leading to single or multiple errors in those elements. While users may not always be able to distinguish between SETs and upsets, it is important to note that SETs are generated differently and require

distinct mitigation techniques compared to SEUs. Additionally, SETs are also influenced by electrical and logical masking and exhibit a different type of temporal masking mechanism. If the pulse reaches the sequential element outside of the sampling window, it is not sampled, and an error does not occur. As the frequency increases, the sampling windows occupy a larger proportion of the total period, resulting in SETs occurring more frequently at higher frequencies.

In larger technology nodes (greater than 90 nm), SETs are not predominant due to the electrical masking provided by the large capacitance and the temporal masking facilitated by lower frequencies [14]. However, in more recent technology nodes characterized by smaller capacitances and higher frequencies, there is an increased possibility that a spike will be latched. A study presented in [15] demonstrates that higher frequency does not necessarily translate to a higher error rate. However, it does alter the relative vulnerability of sequential and combinational logic in the circuit, necessitating different mitigation and redundancy solutions at different frequencies.

## 1.2 Fault mitigation techniques

The necessity for fault tolerance arises from the inevitability of faults. Despite designers' efforts to eliminate hardware defects and software bugs, it is practically impossible to eradicate them entirely. Additionally, if a system consists of 100 components, each with 99.99% reliability, the overall reliability of the system would only be 99.01%. As the number of non-redundant components increases, the overall reliability of the system decreases [16]. Moreover, considering the increased probability of component failures due to aging and environmental stress factors such as radiation, it becomes evident that a system must be capable of functioning correctly even in the presence of faults.

There are two primary methods for achieving fault tolerance: Radiation Hardening By Process (RHBP) and RHBD. In RHBP, a custom technology process is employed that is more effective in preventing faults compared to commercial processes. This can involve using insulating substrates for building gates, employing logic gates based on Bipolar Junction Transistors (BJTs), utilizing wide band-gap substrates, and implementing gate shielding. While this approach significantly

enhances system reliability, it comes at the expense of performance, power, and area, as these technologies are based on older production processes that result in larger node sizes. Furthermore, due to the more complex fabrication processes required, the development costs are higher, and the production volumes tend to be lower, making these technologies more expensive to deploy.

In cases where the cost of RHBP techniques is prohibitively high, RHBD is often employed as an alternative to achieve a similar level of reliability while leveraging the lower costs and higher performance of commercial components. This is typically accomplished through the use of space redundancy (that is redundancy either in hardware, software or information), or by employing time redundancy.

- *Hardware redundancy* is performed replicating N times the modules of a system. It can be passive or active. In passive hardware redundancy a majority voter is used to receive the output of the replicated modules and select one. In case of Triple Modular Redundancy (TMR), for example, if one of the modules presents an erroneous output, the voter will mask it and still produce a correct result. However, if two modules are defective, the wrong output will be selected. Moreover, if the gate driving the voter input, or the voter itself manifests a fault, it can cause the generation of errors even if the modules connected to it are properly working. In active hardware redundancy the voter is a comparator driving a latch. When the voter detects an output mismatch, the output is not sampled and a system recovery routine is enabled, to locate, and remove the fault from the system.

- In the same way hardware redundancy replicates hardware modules in order to achieve higher reliability, *software redundancy* replicates software features in order to increase reliability [17]. Software redundancy can be either single version or multi version. In single version redundancy, mechanisms for fault detection, containment and recovery are added to a software component. Multi version uses redundant software components developed following design diversity rules, such as using different development teams, languages, algorithms to increase the diversity of the redundant components.

- *Information redundancy* involves the addition of redundant data during the storage of information in memory. The purpose is to enable error detection

and recovery in case the data becomes corrupted. This redundancy plays a crucial role in maintaining system functionality, as seen in the continued operation of CD-ROMs and DVDs even when they present physical defects. To achieve this, check bits are commonly used such as pairing codes, Hamming codes, or other sophisticated algorithms.

- When the disadvantages in terms of cost, size, weight, area, and power consumption associated with hardware redundancy are not feasible, an alternative approach known as *time redundancy* is often preferred. Time redundancy involves the repetition of tasks performed by a system multiple times, with the aim of increasing fault tolerance. In time redundancy, the results of these repeated tasks are compared against each other to ensure their correctness and identify potential errors. Similar to hardware redundancy, voter techniques can be employed to determine the correct outcome by selecting the most common or consistent result among the repeated tasks.

### 1.2.1 Fault tolerance evaluation

After applying one or more mitigation schemes to the design and conducting tests to verify that the requirements regarding area, performance, and power have been met, it becomes necessary to demonstrate that the mitigated design achieves the desired failure rate when exposed to a challenging environment. To assess its fault tolerance, specific tools and platforms are used for testing. A common practice is fault injection, which involves using a setup capable of generating faults within a design to emulate the fault models of interest (Figure 1.4). This allows for the evaluation of the system's dependability.

Over the years, numerous techniques for injecting faults in system prototypes and models have been developed, categorizing them into five main categories [19]:

- *Hardware-based fault injection*: This method induces faults at the physical level by introducing stressing parameters into the environment. Examples include subjecting the hardware to heavy ion radiation or electromagnetic interference, injecting voltage spikes on the power rails of the device, or using lasers to modify the values of circuit pins.

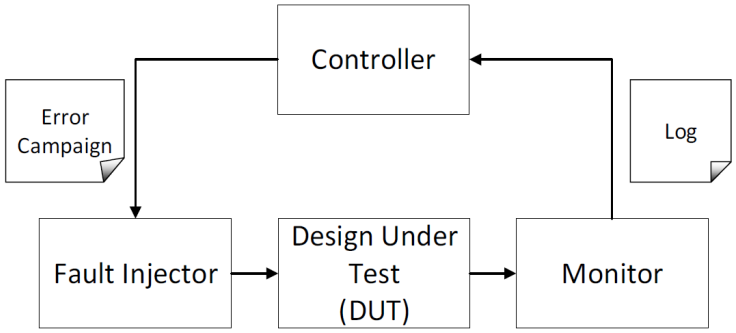



**Figure 1.4:** Basic components of a fault injection environment [18]

- *Software-based fault injection*: These techniques aim to emulate the effects of hardware faults using software. They simulate the errors that occur at the hardware level.
- *Simulation-based fault injection*: This approach involves injecting faults into a Hardware Description Language (HDL) model of the target system, often implemented in VHDL or Verilog. It allows for early fault tolerance evaluation when only a model is available and enables continuous reevaluation of results at each abstraction level.
- *Emulation-based fault injection*: This technique utilizes FPGA-based emulation to simulate the target circuit, assuming the HDL description is fully synthesizable. It enables evaluation of circuit behavior within its environment and facilitates real-time fault injections.
- *Hybrid fault injection*: This approach combines techniques from different categories, such as combining software and hardware fault injections.

Fault injection techniques enable the assessment of whether the system's response aligns with its specifications when subjected to a specific range of faults. Typically, faults are injected at selected system nodes and during specific system states, identified through preliminary system analysis. Fault injection serves two purposes: fault removal and fault forecasting. Fault removal involves identifying and rectifying deficiencies in the fault mitigation solutions implemented in the system. This allows for the correction of any weaknesses in the system's ability to handle faults. On

the other hand, fault forecasting involves evaluating the existing faults in the system, anticipating their potential future occurrences, and assessing the associated consequences. By employing fault injection, engineers can fine-tune fault mitigation strategies, enhance system robustness, and gain valuable insights into fault behavior and its impact on the system.

# Chapter 2

# Technology overview

The work of this thesis is focused of soft cores, implemented in an FPGA. Due to their ease of use, the Microchip's ProASIC3 FPGAs were chosen. An FPGA, which stands for Field Programmable Gate Array, is a type of Integrated Circuit (IC) that offers programmable logic functionality along with configurable interconnects. Unlike traditional Application Specific Integrated Circuits (ASICs) that are designed for specific tasks, FPGAs are highly flexible and can be reprogrammed or reconfigured to perform various digital logic functions. This reconfiguration process is similar to loading a program into a microcontroller, but instead of passing machine-code instructions, the bit stream controls the configuration of hardware blocks that implement different logic functions. All kinds of combinational and sequential circuits can be implemented, making FPGAs an ideal tool for prototyping. While performance is lower than what is obtainable with ASICs, modern FPGAs have frequencies of more than 600 MHz.

## 2.1 The ProASIC3 family of Flash-based FPGAs

The ProASIC3 is a Flash-based FPGA, composed of an array of programmable logic elements, called VersaTiles, and a configurable interconnection network called VersaNe (Figure 2.1). Each VersaTile is capable of being configured as a three input, logic function, a D-flip-flop or a latch, both with an optional enable. This allows a really high core utilization rate compared to SRAM-based cells. In addition,

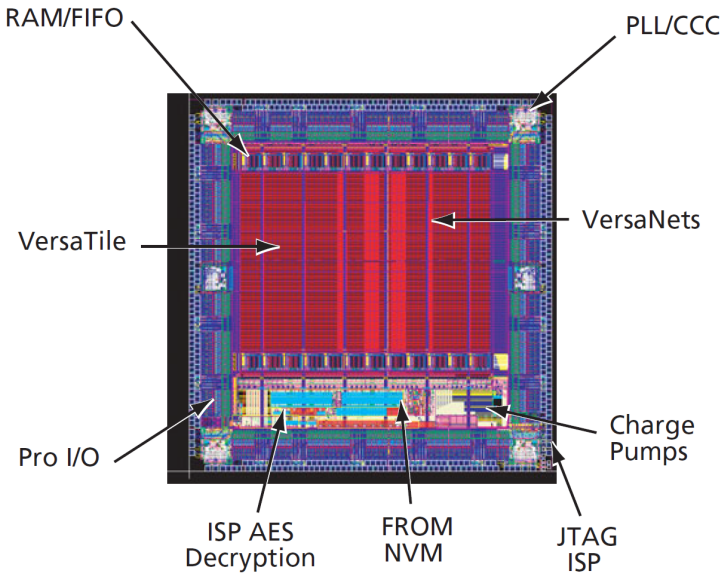



**Figure 2.1:** ProASIC3/E Architecture

VersaTile inputs can all be inverted and outputs can both be connected to ultra-fast local and efficient very-long line routing resources, simplifying technology mapping and netlist optimization. In addition to that, ProASIC3 also feature a series of SRAM blocks, which can either be configured as Random Access Memory (RAM) or as First In First Out (FIFO), a block of Flash Non Volatile Memory (NVM), an In System Programming (ISP) Advanced Encryption Standard (AES) Decryption block and a Joint Test Action Group (JTAG) ISP.

The ProASIC3 uses a Flash-based Low Voltage Complementary Metal Oxide Semiconductor (LVCMOS) process with seven metal layers. Using standard Complementary Metal Oxide Semiconductor (CMOS) techniques, it features a combination of fine granularity and flexible routing, allowing for high logic utilization without compromising device routability and performance. The Flash programming element the FPGA is based on uses only two transistors. Thanks to the lower area occupied, it allows for a higher density compared to the traditional six transistors in SRAM based cells. The Flash switches are distributed throughout the device, giving non volatile configurable interconnection programming[20].

Another advantage to Flash compared to SRAM, is the immunity to hard errors, due to radiation [21][22]. This is fundamental in space applications as it guarantees

that the configuration memory of the FPGA won't be affected, changing the logic configuration of the circuit, and possibily disrupting the device mission. SRAM-based FPGAs are vulnerable to radiation that may result in hard errors. If the I/O in an SRAM-based FPGA suffers a firm error, its output could be turned on, causing significant board damage, and its input could be turned off, resulting in critical signal loss (Figure 6 and Figure 7). By eliminating this danger, using Flash memory means that no other fault mitigation technique has to be implemented to protect the configuration memory.

## 2.2 The ProASIC3/E Evaluation Board

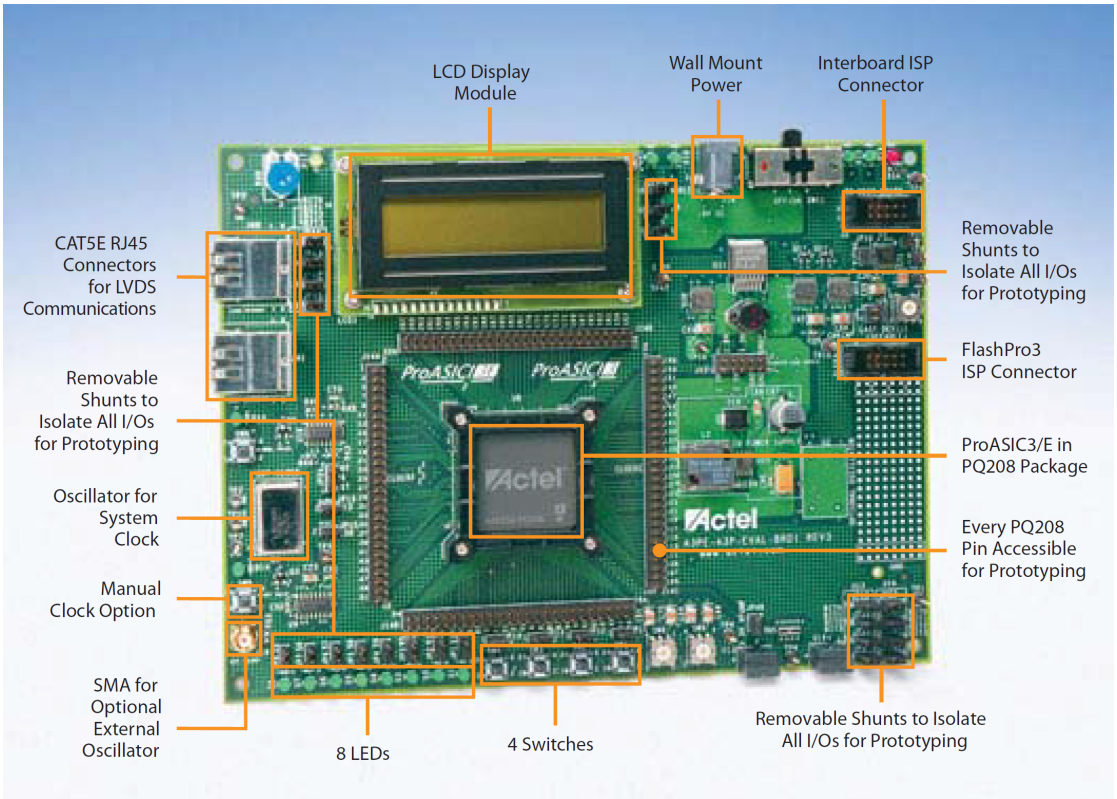



**Figure 2.2:** The ProASIC3/E Proto Kit Evaluation Board

The ProASIC3 Evaluation Board (Figure 2.2) is a board mounting the ProASIC3 A3P250PQ208 chip. The ProASIC3 A3P250 is one of the smaller ProASIC3 FPGAs, having less features compared to its bigger brothers. It contains 6144 VersaTiles,

8 4608-bit RAM blocks, for a total of 36 Kbits of RAM, and 1 Kbit of Flash ROM[23]. It is equipped with four I/O banks, for a total of 157 I/O available to the user. The board allows to implement a large number of I/O standards, in a range of different voltages (1.5 V, 1.8 V, 2.5 V, and 3.3 V). In addition, the registers available in the I/O tile can be used to support high-performance register inputs and outputs. The registers can also be used to support the JESD-79C Double Data Rate (DDR) standard within the I/O structure. The I/Os provide programmable slew rates, drive strengths, weak pull-up and weak pull-down circuits, and many devices are capable of hot insertion and multi-voltage input and output tolerance. These features simplify communication with the FPGA device, and in the process, reduce the number of system components, saves board space, and minimizes power consumption. A 40 MHz oscillator for system clock and a manual clock option is also provided. The board finally comes with a variety of Light Emitting Diodes (LEDs), buttons, switches, and jumpers as it is expected in a prototyping board, a large Liquid Crystal Display (LCD) alphanumeric display, two RJ45 CAT5E connectors for Low-Voltage Differential Signaling (LVDS) communication and 10-pin programming connector for the FlashPro4 programmer[24].

## 2.3 Microsemi VHDL ProASIC3 design flow

The ProASIC3 Evaluation Board is meant to be used toghether with Libero System-on-a-Chip (SoC) Design Suite, a software suite created to manage projects on the Microsemi FPGA families. The suite offers a collection of programs to manage HDL designs, simulate and analyze designs at every stage of the design flow (Figure 2.3). The project can include both traditional RTL designs and IP cores that are generated by the program, based on libraries provided by Microsemi.

At each step of the design flow the program allows the user to:

- Perform a logical simulation of the design
- Define constraints, I/O and clock planning,
- Perform Design Rule Checks (DRCs)
- Modify implementation results

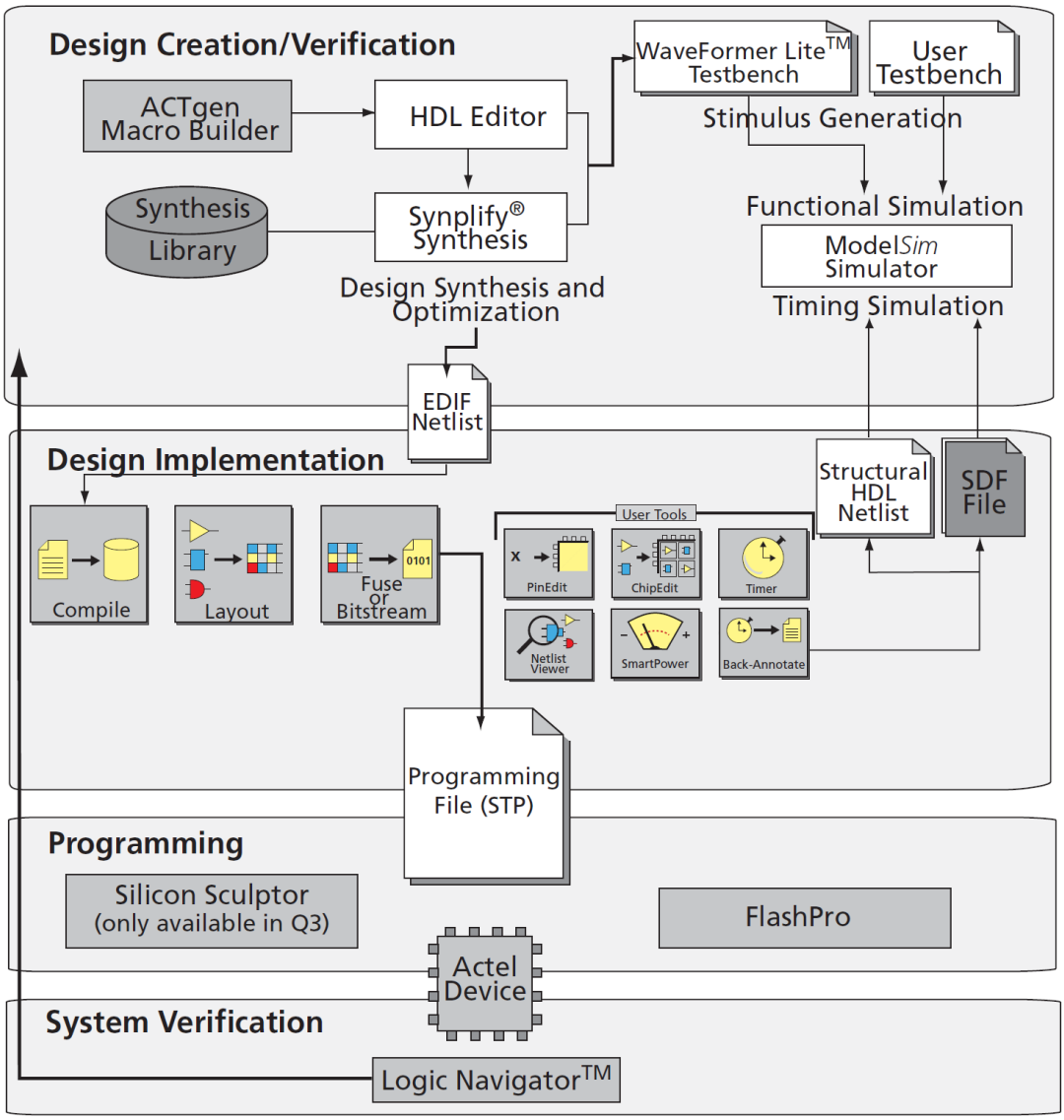



**Figure 2.3:** VHDL-Based Libero SoC 11.9 Design Flow

The project can be managed directly using the Graphical User Interface (GUI) in the Libero Integrated Design Environment (IDE), as well as by means of Tcl scripts lauched from the Command Line Interface (CLI). Each approach has its own advantages, with the GUI allowing to easily view the design through it evolution during the design steps, and the CLI allowing to automate most of the design steps for large batch processing. The Tcl script can be run either when launching Libero

from the console, using the command `libero SCRIPT:<path to the script>` or in the IDE. The Libero SoC Design Suite has support for:

- Tcl
- Synopsys Design Constraints (SDC)
- Verilog, VHDL, VHDL-2008, SystemVerilog
- SystemC, C, C++

# Part II

# Development of the Rempro system

# Chapter 3

# Rempro, a new challenger

Despite the fact that designing a processor for space applications is a daunting challenge, there is no shortage of competition in the field. The potential rewards for those who succeed are undeniably enticing, offering the opportunity to be at the forefront of an expanding industry, which is invaluable for both large and small companies. One such company is IES S.r.l., an Italian company based in Anzio (RM) that specializes in civil and military telecommunication systems.

Building upon one of their existing products, the HElabor, they are working to adapt it to meet the rigorous requirements of the aerospace industry. The new system, named Rempro, will be implemented in a RHBP FPGA and incorporate TMR to ensure reliability for long-term space missions. In this chapter, a brief description of the HElabor and the Rempro systems ISA and internal architecture will be provided, along with an explanation of the design choices made to incorporate fault mitigation techniques in the core. Additionally, implementation tests will be presented to verify the functionality of certain aspects.

## 3.1 HElabor

HElabor is a homogeneous Symmetric MultiProcessor (SMP) with private and shared memory in the same address space. It is designed for Uniform Memory Access (UMA) for each processor, with each processor accessing shared memory in the same way, and in parallel with each other. The memory access scheme is

Concurrent Read Concurrent Write (CRCW), that is both writing and reading can happen simultaneously.

The system is designed to be implemented on an FPGA and can be easily scaled up or down according to the application requirements. The number of cores, dimensions of memory blocks, addition of specific functions, the number of I/O ports, and even data and address dimensions can be controlled prior to implementation, allowing for a high degree of customization.

HElabor is made with parallel computation in mind. The interconnected system structure is intrinsically parallel making it easier to develop parallel applications. When implementing a system, the blocks composing the system can be each implemented in a different processor, as load balancing is not required and it's possible to have processors that only execute a few instructions when activated while others work constantly. Moreover, the shared memory and dedicated deterministic channels between processor make extremely straightforward the definitions of synchronization signals and checks.

The internal architecture of HElabor is unconventional, to say the least. It deviates from current electronic design paradigms by employing a small pipeline consisting only of fetch and execute cycles. The datapath is also simplified, lacking a register file. Instead, memory locations are utilized in place of registers. Each processor has its own directly accessible RAM block. Additionally, the highest memory addresses are allocated for special locations that possess additional properties. These properties include the ability to be incremented during another instruction, function as an accumulator register, support indirect addressing, and more.

HElabor consists in a variable number of 16 bit processors, with 16 bit addresses for a maximum of 65.536 locations both for data and instructions. Each processor has a two stages pipeline, meaning that all the instructions are fetched, decoded, and executed in two clock cycles. There are no stalls and conditional branches are executed in a single clock cycle, independently from the fact that the branch is taken or not. This allows developers to know for certain how many clock cycles each section of a program will take, helping considerably when synchronizing two processors.

The first stage of the processor deals with fetching instructions and incrementing

the Program Counter (PC) by one each clock cycle. In case of procedure calls, the PC can be saved to the stack. A multiplexer select whether to access the program memory using the PC, the value in the stack or a value received from the execution stage in case of jumps. At each rising edge of the clock, the stage accesses memory and saves the instruction in the Instruction Register (IR). The instruction has a width of 39 bits, with a 7 bit opcode field and two 16 bit operator fields. The operator bits can contain memory addresses, immediates, or jump addresses with branch conditions.

The execution stage is tasked with decoding instructions, reading from memory, performing arithmetic operations, evaluating branches, and writing black results. While the memory is the only component where data can be stored, the processor still has dedicated register to perform special operations. They are still placed in memory but have their own dedicated output ports. Instructions have only two operator fields, one for sources and one for destinations. that means that in case of two operand instructions, if an immediate is used, the other operand will be either the destination register, or the accumulator register `A`. If no immediate is used the source register will be the first operand and `A` the second as only a single location in memory can be read per instruction, beside `A`. There is only one addressing mode, and can be performed only using the three `INDEX` registers to store the effective memory address we need to access. `INDEX0` in particular has the ability to be increased by one after a memory access, allowing faster access to data arrays. The Arithmetic Logic Unit (ALU) is capable of performing 16 bit additions, subtractions and shifts; an 8 bit multiplier is also present to complete the functionality.

## 3.2 From HElabor to Rempro

The HElabor multiprocessor is not ready for space applications yet. A proper reliability evaluation needs to be conducted, and proper fault mitigation techniques need to be implemented, before even starting to suggest to use it in a space mission. The purpose of this thesis is to start these preliminary studies so that they can be used as a basis for further investigations on its space capabilities.

Since the HElabor is a multiprocessor, it seems reasonable to use a single

processor as a building block to realize a hardened architecture. The main idea is to implement TMR, with cold standby sparing. In this way if any of the three processors manifests a permanent fault, it can be switched with a spare one. The program in the meantime would be paused, the new processor would recover the program state copying the memory from one of the working processors, and then the program would continue. Using a number of different boards, each with an FPGA containing multiple copies of the processors, would greatly increase the lifespan of the device, along with its reliability. Each of the boards would have its clock signal synchronized with the others, using Phase-Locked Loops (PLLs).

This new architecture, called Rempro, has still to face some challenges before it is ready. How to detect a fault? How would this recovery process be executed? How to transfer memory? How to guarantee synchronization between processors? Two main strategies emerged in the study: one that uses a watchdog, and one without.

### 3.2.1 Active monitoring with a watchdog

The first solution proposed to use a monitoring system to detect anomalies such as frequency degradation, data and memory addresses out of the expected ranges, or timeout expiration. Each watchdog would act as a Master for the processor RAM, and would be able to access it together with the program counter value. While the voters would mask temporary faults, the watchdog would be detecting permanent ones, allowing the system to be available in both cases.

When the watchdog detects a fault in its processor, it should turn it off and turn on a replacement. It would also send messages to the watchdogs in the other processors to start the memory transfer for the new processor. The RAM content, the Stack and the program counter would need to be transferred from a working processor, and the new processor would need to be connected to the voter. Each watchdog would supervise the system synchronization and would be able to tell if a recovery process is being executed. When the recovery is completed, the watchdog for the new processor would send a synchronization signal in order to continue the execution of the program.

While this idea is appealing, as it allows to concentrate, fault detection, recovery and synchronization in a single device, the supervisor itself is susceptible to faults.

That means that any fault happening inside it needs to be detected and handled by the supervisor, starting the recovery process. Another issue to take into account is the fact that since the supervisor is able to access the RAM, the stack and the PC, connecting the supervisor to the processor would need a great number of signals, increasing the possible failure points. Moreover, the supervisors need to communicate with a bus, meaning that a communication protocol has to be defined, taking into consideration the possibility that a supervisor could fail.

The added complexity needed to solve this issues is incompatible with the project objective of simplicity, low power consumption and low area use so a simpler solution is needed.

### 3.2.2 Resistive voter

In order to forego the use of a watchdog, a possible solution would involve using only voters and make them take part in the sostitution and recovery process. One of the architectures considered places a single processor in each FPGA, with its data and program memory, and more FPGAs placed in the same board. Two voters would be necessary: one to compare the processors output lines for a total of 96 connections in input to the voter (since each processor has a total of 2×16 bit outputs), the other to compare the data read line between processor and memory, requiring 48 connections. To simplify the voter logic, an analog voter based on resistive voltage dividers could be used. As an added benefit it would be impervious to faults due to radiation. The voter, acting as a voltage divider, outputs a voltage that is lower than half that of the voltage supply in case at least two of the processor outputs have a low value, or a voltage higher than half the voltage supply in case at least two of the outputs have a high value. The inactive processors would still be connected to the voter but their outputs need to be in high impedance to avoid affecting the result. The output of the voter would then need to be converted back to a digital value using a comparator such as the ones available as the FPGA inputs.

The voter output is compared inside each FPGA to the output of each processor using a series of XOR gates and feeding the result of the comparison to a series of ORs. This signal is shared with the other processors, both active and inactive, and saved in RAM as an N bit signal called `FAIL`, where N is the number of processor.

Each processor is represented with a bit and each processor can only set its own bit. A second register, called `PROACT`, is used to control the replacement of faulty processors. The `PROACT` register contains a N bit signal similar to that of `FAIL`, where each bit represents a processor and setting it will turn on the corresponding processor. The output of `PROACT` does not turn on the board directly however. The value coming from the three different processors is fed to a voter in order to mask discrepancies due to faults.

During the recovery process, the RAM, the stack, and the PC need to be tranferred from a working processor. to simplify the procedure, the PC and the stack are moved to special locations in RAM so that a complete transfer of memory would include this data as well. The preliminary idea is to implement in software a periodic check to see whether the `FAIL` register has registered a failure. If this is the case, the bit corresponding to a new processor in the `PROACT` register is set and the one corresponding to the failing processor is cleared to turn it off. When the new processor is active, the memory transfer routine is started and a working processor transfer its memory content to the new one, along with PC and stack. The `FAIL` bit is then cleared and the program is resumed. The power controls for the boards are daisy-chained together so each board can turn on the following board and turn off the previous one. This scheme simplifies the power connections but forces each board to be controlled by the previous one. If after a full round where each board failed and was replaced once, the first board to fail did not recover from the fault yet, the whole system is not able to carry on and it is unable to start the following processor.

### 3.2.3 Internal voter

In order to simplify even more the realization a possibility would be to use internal logic to compare signals from different processors and to turn on and off boards. Different solutions are proposed, differing on the number of processors in each FPGA and the control logic used to check the outputs and perform fault recovery.

### Three processors in a FPGA

Each FPGA would feature three processors, with a voter before their output and one between memory and processor, on the read line. A single FPGA is placed in a board, and the boards are stacked on top of each other, economizing on space. Only one is active at any time, so in case a fault causes a failure inside the FPGA, one of the voters will signal the problem, and the two working processors will turn on the new board and start a memory transfer. Once that is completed, the defective board is turned off and the program resumes. To perform the memory transfer the three memory outputs are passed to the new processor passing through a voter to ensure the data coming from the defective board is not corrupted. Since a single board is active most of the time, the others do not draw power making this design extremely power efficient.

### Two processors in a FPGA

In each FPGA two processors are implemented. The second one is used to compare the outputs and with a series of XORs and ORs, a mismatch can be detected either in the RAM read line to the processor or in the output. The signals. coming from each processor couple, are grouped together and fed back to each FPGA, so that every processor is aware that a board is having issues and can take measures to replace it. THe voter for the outputs and the RAM are again the analog resistive voltage dividers, but the RAM voter is only used during the memory transfer routine to ensure data integrity. This architecture has a problem: since the comparison results are fed back to each FPGA if the output or the input pins of the FPGA are faulty, it might be difficult to notice and would create issues in the replacement procedure.

### One processor in a FPGA

The last solution was using a single processor in each FPGA. In order to check for malfunctions, a check-word is computed to represent the system state. Sending this check word to the other FPGAs the processors can check whether each processor is in the same program state and start a substitution routine in case of a mismatch. The check word can be computed XORing together either the RAM or the ALU

output. While its possible that the same words could code for different states, the following clock cycles would most certainly result in different check words, so ambiguity has a very low probability of masking problems.

## 3.3 Final modifications to the architecture

While each solutions has its pros and cons, the one with the most promise is the first one, using three processors in a single FPGA. The Rempro system will then be composed of multiple HEprot processors, that is TMR versions of the HEpro single processor. Having a lower power consumption and having to take into account a single board at a time make it an ideal candidate for further development. The possibility to include an entire TMR system would also help with development, since the entire system can be tested without the need to monitor multiple boards. It is also worth noting that a future evolution of this architecture could involve the realization of ASICs containing the system, allowing the composition of multiple TMR systems connecting toghether the IC,

Another issue to take into account is the hardening of the power control circuit for each board. If the circuit fails, the board cannot activate and the system is not able to replace it with the next board, meaning that the entire system comes to a halt. To protect it from faults, electromechanic devices such as relays can be used to connect the board to the power supply. The circuit needs to be turned on by the previous board in the daisy-chain and turned off by the following one. For this either a normal relay or a bistable relay. Using 2 PNPs and a normal relay, a circuit presenting a bistable behaviour can be realized. A low pulse on the SET signal would activate the relay, which would then stay on. A second negative pulse on RESET would deactivate the relay. A different circuit could be realized using instead 2 NPNs and a relay. The relay, however, when activated in both examples would consume power. This does not happen using a bistable relay. Using two PNPs and two NPNs, it is possible to invert the verse of the current in the spool. Another possibility is using a dual coil relay, in a circuit that would only need two transistors. It is important to further make sure that the transistors employed are big enough to be insensitive to radiation.

In order for the recovery process to work, signals coming from all three memory

modules in a processor, need to exit from each FPGA and connect to every other one. Each single FPGA would need 48 inputs and 48 outputs to connect to it, one or two 16 bit output ports and one or two 16 bit input ports for a total of 192 pins necessary. While the maximum number of pins in FPGAs such as the ProASIC3 A3P1000 by Microchip, allows for up to 300 I/O user pins, the number is certainly high. In order to reduce the number of lines a 48 bit bus could be used to connect each FPGA. In each RAM module, a special location is added to receive and send data on this bus. One processor will write on the location sending the data on the bus, while all the others keep the line in high impedance to avoid conflicts. Attention must be paid while writing because no arbitration is present in the bus, so two processors writing at the same time will result in data being corrupted.

The final architecture used for the implementation tests includes:

- The 48 bit bus, divided in 3 16 bit sections, each connected to a different processor by means of a special RAM location, `BUSC`. When a triplet of processors writes to `BUSC` the content of the location is immediatly transfered to the bus and, on the following rising edge of the clock, all the `BUSC` locations in the other triplets assume the value written by the first processor. Every processor can then read that location as another register. The line is not arbitrated so to avoid conflicts only one triplet can write on the bus at all times, the other triplets must keep the value in the register to high impedence (Z). One clock cycle after the triplet has written to the location, if there are no other write commands, the line is brought back to high impedance so that other triplets can use the bus.

- Five voters are present in each FPGA, 3 for RAM locations and one for each of the two outputs. A location warning is placed in each processor's RAM and each voter controls 3 bits in it, for every processor. Each voter has three inputs and one output. When it detects a mismatch between one of its inputs and its output, it set the corresponding bit in every warning location, signaling that the processor connected to that input is misbehaving.

- Two signals, `PWCON` and `PWCOFF` to turn on the following board and turn off the previous one. These are connected to each processor through the location `PWC` and through voters, connected to the two output signals. The bit 0 of the

location `PWC` is connected to the signal `PWCOFF`, while bit 1 is connected to the signal `PWCON`

- the Stack has been moved to the RAM, it is made of a `STACKPOINTER` location followed by 8 locations for storing return addresses of procedure calls.
- A location `CKCYCLE` is added to the RAM in place of the location `LAST`, allowing to use it for comparisons and speeding up the execution of loops. The location is a 16 bit counter with a starts counting up from the moment the processor is reset, incrementing once for every clock rising edge, and allowing to count the number of clock cycles passed from the last reset.

`PWCON` and `PWCOFF` act on the power supply of each board by means of a relay. In order to simulate successfully this behavior in the development board, the two signals control a SR latch that acts on the reset signal of the entire board, stopping the execution when the `PWCOFF` signal is sent by activating the system `RESET` signal and restarting the board when the `PWCON` signal is sent, deactivating the `RESET` signal.

The substitution and recovery process is described as follows:

- One of the voters of the triplet of processors detects a mismatch between its inputs and therefore sets the corresponding bit in the location `WARNING` inside the RAM of each processor.
- When the program in each processor hits a checkpoint determined by the program, it checks whether there is any failure bit set in the location `WARNING`.
- If a failure has been detected, the program calls the subroutine that substitutes the FPGA and transfers the memory content to the new one.
- The subroutine sets the bit one of the location `PWC` to 1 for one clock cycle to turn on the next board.
- The triplet then tries to gain control of the bus writing a code to it for a specific number of clock cycles to wait for the second FPGA to start.
- As soon as the second triplet awakes, it reads the bus and, since data is present, it calls the memory transfer procedure. It then waits for the first triplet to

finish transmitting, afterwards, it sends an acknowledgment code in the bus for the first triplet to receive.

- The old triplet starts then transmitting the entire memory content to the new one, copying each location in series to the register `BUSC`.
- When the transfer is complete the triplet waits to be turned off by the new one.
- The second triplet sets the first bit of `PWC` to 1 to turn off the old board.

The recovery process relies heavily on the fact that both the processors have the same clock so it is of the utmost importance that measures are taken to make sure the clock generator is able to send the same clock, with the same phase, duty cycle, and frequency, to both FPGAs.

# Chapter 4

# Implementation tests for the Rempro system

In order to realize the multiprocessor system, a few preliminary investigations on the capabilities of the ProASIC3 FPGA and the Rempro need to be conducted. The ability of two processors to communicate through a bus needs to be explored further and the delay introduced by the wires used to realize the bus needs to be measured and taken into account. Moreover, the processor lacks the features needed to communicate serially with a PC which is a fundamental function in order to conduct any kind of test. In this chapter a Universal Asynchronous Receiver-Transmitter (UART) is introduced to the system and tested. This module is then used to communicate with an host PC while measuring the wire delay and verifying the processor ability to communicate with another instance of itself using an external bus.

## 4.1 The UART module for the Rempro system

One of the fundamental parts of running a test, is being able to communicate with the Device Under Test (DUT). To allow the Rempro system to send data to a host PC a UART interface has been designed, starting from an open-source RTL module.

UART is a widely used serial communication protocol. It allows two electronic

devices to communicate bidirectionally in an easy way, as its implementation is cheap, and simple. Its main characteristics are:

- **Serial:** the UART protocol uses a single signal to transmit all the bits composing some piece of data, one after the other.
- **Asynchronous:** UART is an asynchronous protocol, which means it does not rely on a clock signal to synchronize the data transmission. Instead, it uses start and stop bits to frame each data byte.
- **Data Format:** UART transmits data in the form of individual bytes (usually 8 bits) along with optional parity and stop bits. The start bit is a low value that lasts a clock cycle indicates the beginning of a data byte, followed by the data bits, optional parity bit (for error checking), and stop bit(s) that are one or two high values in the line to mark the end of the byte.

The UART module is a Finite State Machine (FSM), activated by a start bit. After receiving the signal the module reads the byte that is to be transmitted and then sets the signals necessary to implement the protocol. The baud rate, that is the number of symbols transmitted per second, is a user constraint, set up by changing a GENERIC value in the instancing of the unit. This value is equal to the number of clock cycles the FSM needs to count before transmitting another symbol.

While the module allows both for transmission and reception of data, only the transmission function was implemented for the purpose of the tests. The UART is capable of transmitting up to 9 bits of data per single transmission. Since memory locations in Rempro contain 16 bits each, it is not possible to transmit them all at once. The solution is to send the data contained in the registers in two bouts of 8 bits each. A Multiplexer (MUX) with two 8 bit inputs is then used to split each value into two byte sized blocks.

The UART is controlled by a few different signals that need to be set in order to work properly. In particular, the signal `TX_DV` is used to start the UART, `TX_BYTE` is an 8 bit signal used to present to the module the data that is to be sent. There are three output signals, `TX_DONE` is used to signal to the processor, or any controller module, that data has been sent, `TX_SERIAL` is used to actually transmit the data using the UART protocol, and `TX_ACTIVE` is used to signal the controller that the UART module has been activated.

**Figure 4.1:** The Silicon Labs CP210x USB to UART Bridge used to connect the output pins of the UART transmitter to the host PC.

A Universal Serial Bus (USB) to UART bridge (Silicon Labs CP210x USB to UART Bridge, in figure 4.1) is used to connect to the host PC. Three wires are used to connect the bridge to the board, two for the connection to 3.3 V and Ground (GND), and one to connect the transmission output of the UART to the receiver input of the bridge. A General Purpose Input/Output (GPIO) is used to present the output of the module to one of the FPGA external pins.

## 4.2 Measurement of the latency of a wire used to connect two GPIO pins of the A3P250 FPGA

The Rempro system requires that multiple boards, mounting ProASIC3 A3P1000 FPGAs are connected toghether. Each FPGA will implement a TMR triplet of processors. Only one triplet will be active at all time, and when it detects a failure on one of its processors, it will substitute the FPGA with a new one and turn itself off. The substitution process requires the entire program state, including, PC, Stack and RAM content. This means that a considerable amount of data needs to be transferred from one board to another, and that the reading and writing times for the GPIO pins need to be respected. A preliminary test is therefore required in order to measure the latency of GPIO connections.

To test the latency, we realized a setup on a single board that uses two GPIO pins to transmit and receive a single bit. A FSM, implemented in hardware, manages the transmission and reception of the signal, and the latency is measured by a 16 bit counter implemented in hardware as well. The start and stop signals of the

counter are managed by the FSM.

Multiple measures were taken with different lengths of wire connecting the two pins. The reception of the pulse from one GPIO to the other was monitored both by the serial transmission of the counter value after it was stopped and using a digital oscilloscope.

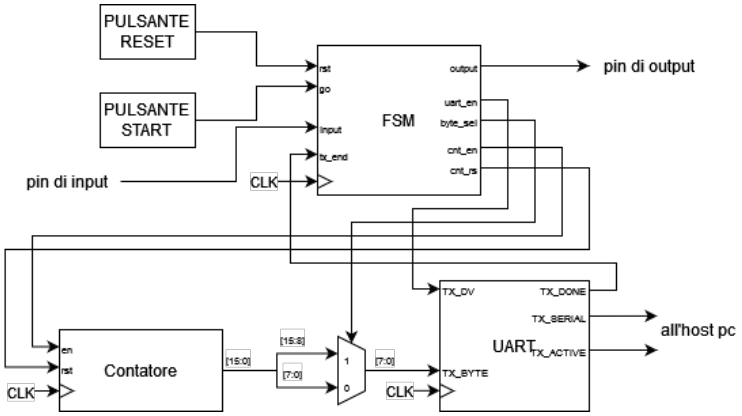



**Figure 4.2:** Block diagram showing the signals and ports involved in FSM communication with the UART interface.

### 4.2.1 The FSM

An hardware-implemented FSM was used to control the generation of the pulse and to check the actual reception of the signal. The FSM is composed of 7 states. The first one is the **reset**, where the system waits for the start signal to be input by the user. The signal called `go` is controlled by the user and assigned to a button on the board in the implementation phase. When the user pushes the button, the signal `go` is set to one, and it triggers the transition to the next state **hold**.

In hold the `output` signal is set to 1 and the counter is enabled to start the count. The system maintains this state until the signal emitted is received at the `input` port.

When the value at the `input` port is high, the system transitions to the state **send_one**, where the counter is stopped and the UART module is enabled. At the same time the first byte of the counter value is transmitted to the host PC.

The system transitions to the next state only when it receives an end transmission

signal from the UART. When this happens, it transitions to the state **send_two** and the final byte of the counter value is transmitted to the host PC. When that is completed the system goes to the **stop** state, where it waits for the reset signal to start again.

The value measured by the counter represents the latency of the transmission between two GPIOs as it registers the cycles that pass between the sending of a signal on one GPIO and its reception on another connected with an external wire.

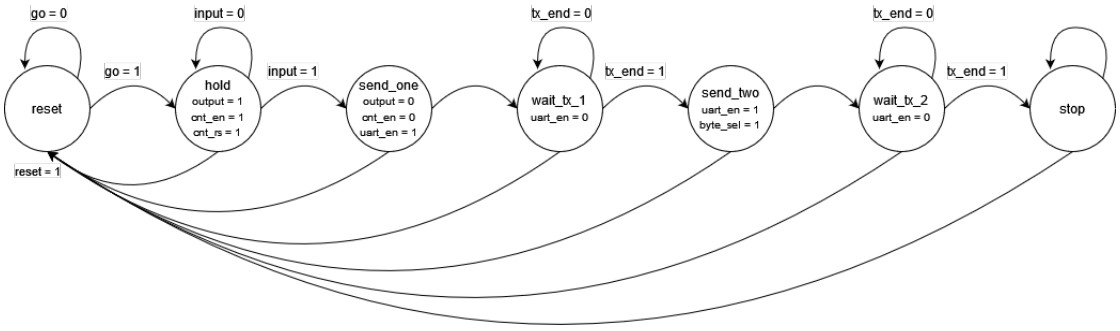



**Figure 4.3:** State diagram of the FSM

### 4.2.2 Simulation

In order to highlight the transmission process in simulation, the amount of clock cycles waited by the UART to transmit each symbol was changed from 4167, representing a Baud rate of 9600, to 1, so that the UART signal transitions every clock cycle. In figure 4.4 the simulated waveforms of different signals can be seen. The data transmission is shown in blue, the enable signal for the state machine is in yellow. The output pulse coming from the measurement system is in orange, while the return signal is shown in pink.

### 4.2.3 Implementation

After verifying that the system worked properly in simulation, the whole setup was implemented through the Microsemi design flow. The target device is the ProASIC3 A3P250.

Using the Libero SoC 11.9 software suite, the input and output ports were assigned to the GPIO pins of the board as shown in figure 4.5. The 40 MHz

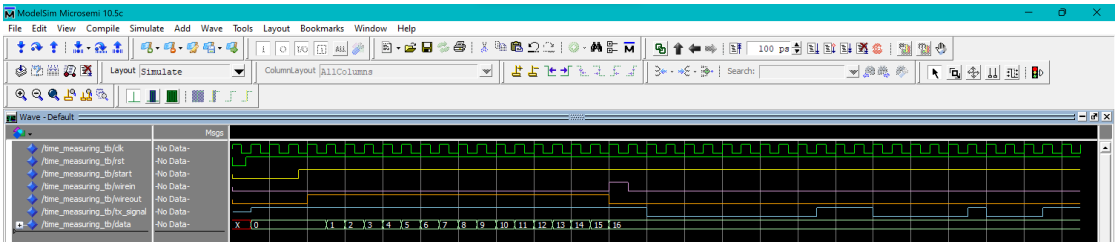



**Figure 4.4:** RTL simulation of the measure setup functionality, including the FSM and UART module

oscillator on the board was used to generate the clock signal, connected to pin 26 (in red). One of the buttons on the board was assigned to the *reset* signal, while the next one was assigned to the FSM *go* signal, shown in green in the image.

The output of the UART was connected to a GPIO (in yellow), configured in pull-up and connected with a jumper to the input of the USB to UART bridge as in figure 4.6. The input and output ports for the measuring pulse were assigned to pins 134 and 45, so that the signal comes out and is received by two different I/O banks. The pins were configured in pull down and shown in blue in the diagram. Moreover, to visually confirm the successful transmission, the counter value was split in two equal parts, given in input to a MUX, and its output was connected to the eight LEDs in the board. To show the two different halves, one of the buttons of the board was connected to the selection signal of the MUX.

A python script, was used to read inputs coming in from the COM port associated to the UART bridge, waiting to receive data from the system. When data is received, the script prints the value to the terminal (figure 4.7).

The received value means that between the emission and reception of the signal, less than a clock cycle passes. This is strongly dependent on the frequency of the logic circuit, that can only be set at 40 MHz and doesn't allow to measure the time interval with better precision.

### 4.2.4 Oscilloscope measurements

Another measurement was conducted using a digital oscilloscope that allows a higher resolution compared to the counter based measure. Since the FSM holds to 1 the *output* signal for a single clock cycle, the time required to reach the *input* pin

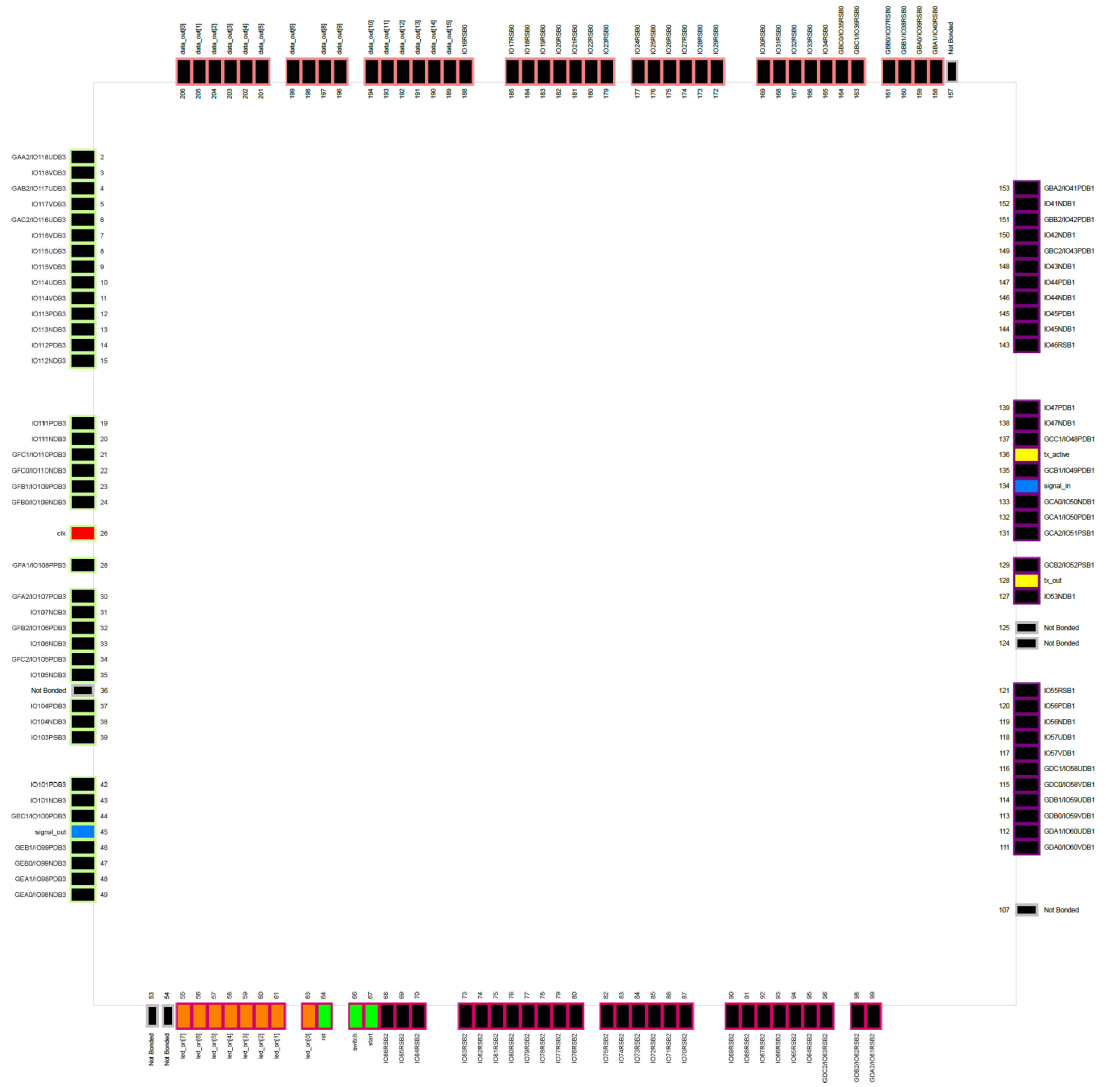


**Figure 4.5:** Map of the GPIO pin assignments of the design

is clearly lower than that. Using the oscilloscope to observe the shape of the signal on the monitor (figure 4.8), it's possible to estimate the interval length that passes between the rising and falling edges of the pulse, using the cursors to measure it. Subtracting from it the clock period of 25 ns, latency can be obtained. Placing the cursors at 50% of the rising and falling edge of the pulse, it is possible to measure a width of 31.15 ns. Subtracting from this value, the clock period, we obtain 6.15 ns.

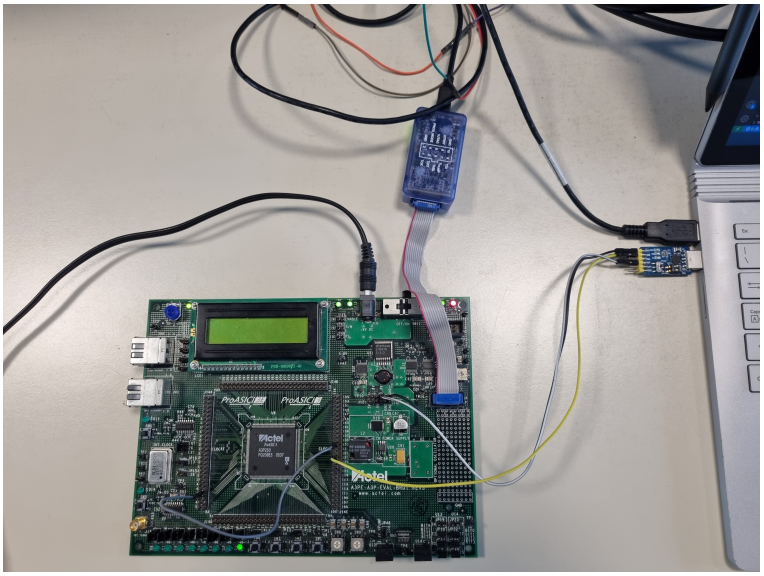


**Figure 4.6:** Photo of the testing setup showing the external connections to the board

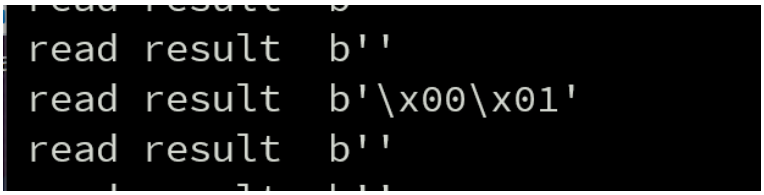



**Figure 4.7:** Correct execution of the measure procedure received from the host PC through the serial interface

### 4.2.5 Conclusions

The wire latency was measured to be 6.15 ns, considering a working frequency of 15 MHz it is sufficiently small not to cause synchronization problems between two processors in different boards and can therefore be neglected.

## 4.3 Test on the UART communication capabilities of the single Hepro processor

The next thing to be verified was that the correct working of a single processor and its ability to communicate effectively with a host PC using the UART module.

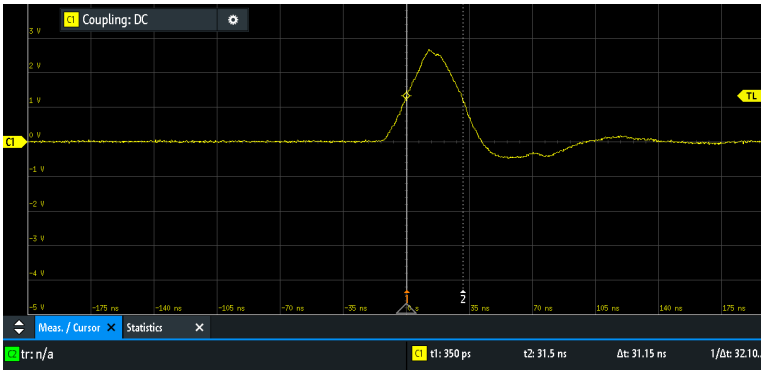



**Figure 4.8:** Measure of the width of the pulse emitted by the system by means of an oscilloscope

The UART interface is implemented in programmable logic and allows serial transmission of data towards the host PC.

In the host PC a serial interface, developed in python, waits to receive data. As a benchmark, a simple program was used to print as output of the processor the character codes for "Hello World".

The processor connects to the UART module using the output ports `OUTPUTA`, `OUTPUTB`, and `INPUTB`. The memory saves values on 16 bits while the UART protocols allows to trasmit only up to 9 bits at a time. To get around this limitation, a MUX, controlled by the processor, is used to split data into two halves of 1 byte each. Particularly, `OUTPUTA` transmits the 16 bit value, `OUTPUTB` is used to send the UART activation signal and to control the MUX that selects which half of the data to send, and `INPUTB` receives from the UART the signal for the transmission end. All the signals are controlled by the processor by means of writing and reading memory locations.

### 4.3.1 Firmware

The firmware was written using the specific assembly language for the Hepro processor. There are no high level functions so to create a program capable of sending to the host PC the American Standard Code for Information Interchange (ASCII) characters for "HELLO WORLD", the resources and ISA available needs to be used. The processor has two global input and output ports, `INPUTA`, `INPUTB`,

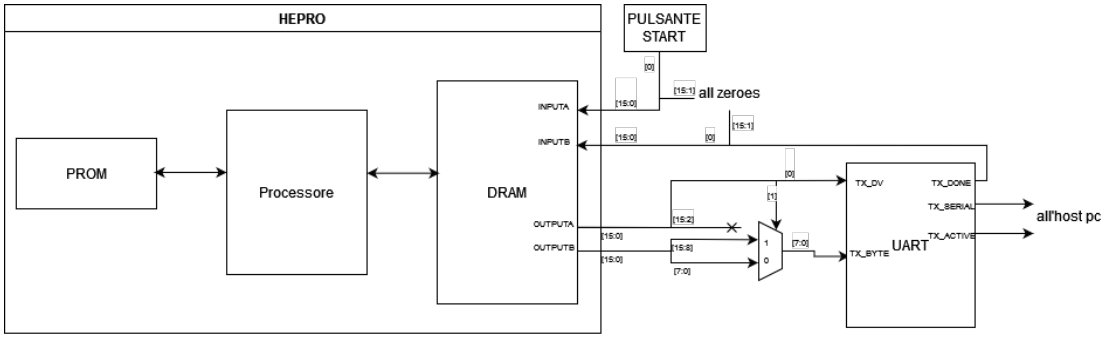



**Figure 4.9:** Block diagram showing the signals and ports involved in the communication between the processor and the UART interface.

`OUTPUTA`, `OUTPUTB`, associated to special memory addresses. These are the only communication ports we have access to exchange information with the processor. In this example bit 0 of `INPUTA` is connected to an input of the test setup that will be assigned in the implementation phase to a pin connected to a button in the board. This allows the user to start the program when the button is pressed. The first bit of the second port, `INPUTB`, receives the end transmission signal from the UART module. The port `OUTPUTA` passes the value to be transmitter to the MUX selecting which half is to be sent to the UART module. Bit 0 of `OUTPUTB` is used to control `TX_DV` and start the UART module, while bit 1 is connected to the MUX selection port.

The first block of instructions loads in consecutive memory locations the hexadecimal values of the character codes for the sting "HELLO WORLD " (with a whitespace at the end: since the program is set up to perform two 8 bit transmissions at a time, the number of characters will be always even). The processor then reads a location at a time writing the value to the special location corresponding to port `OUTPUTA`. It then writes to the location corresponding to `OUTPUTB` to select the byte to transmit and start the UART module. The only used bits are the two least significant ones: bit 0 start the transmission when its value is 1, and bit 1 selects which of the bytes is transmitted. To avoid conflicts in transmission, the `TX_DV` signal must be mantained high only for a clock cycle. The program therefore writes values 1 and 3 to `OUTPUTB` to start transmission for the first and second byte respectively and 0 and 2 after a clock cycle to disable `TX_DV` without changing the MUX selection. The processor then waits for the `TX_DONE` signal before sending

the next byte.

```
#DEF       STOR     100                      ; posizione in memoria
                                             ; dei valori salvati

;----------------------CARICAMENTO IN MEMORIA----------------

           MVR      STOR     INDEX0
           MVR      4845H    INDIR0I
           MVR      4C4CH    INDIR0I
           MVR      4F20H    INDIR0I
           MVR      574FH    INDIR0I
           MVR      524CH    INDIR0I
           MVR      4420H    INDIR0

;----------------------TRASMISSIONE--------------------------

           CLR      COMP
START      JMP      START    INPUTA EQU 1     ; il programma attende
                                             ; la pressione di un
                                             ; pulsante sulla board,
                                             ; collegato all
                                             ; INPUTA del processore
           CLR      A
           CLR      OUTPUTB
           MVR      STOR     INDEX0
LOOP       CLR      COMP
           MRR      INDIR0I  OUTPUTA
           MVR      1        OUTPUTB         ; invio del primo byte
                                             ; del dato impostando
                                             ; a 1 il primo bit di
                                             ; OUTPUTB

           CLR      OUTPUTB                  ; il primo bit di
                                             ; OUTPUTB viene
                                             ; riportato a 0

WAIT1      JMP      WAIT1    INPUTB EQU 1    ; viene attesa la fine
                                             ; della trasmissione

           MVR      3        OUTPUTB         ; viene inviato il
                                             ; secondo byte del
                                             ; dato impostando
                                             ; a 1 il primo e
                                             ; il secondo bit di
                                             ; OUTPUTB

           MVR      2        OUTPUTB         ; il primo bit di
                                             ; OUTPUTB viene
```

```
                                                    ; riportato a 0

WAIT2       JMP       WAIT2     INPUTB EQU 1        ; viene attesa la fine
                                                    ; della trasmissione

            CLR       OUTPUTB
            MVR       5         COMP                ; si inserisce in comp
                                                    ; il numero di dati N-1

            JINC      LOOP      A INF 1             ; l operazione viene
                                                    ; ripetuta per il
                                                    ; numero di dati
                                                    ; da trasmettere
STOP        JMP       STOP
```

### 4.3.2 Simulation

To highlight data transmission during simulation, the number of clock cycles that the UART module counts to respect the Baud rate was changed from 4167 to 1. In this way a transition every clock cycle can be observed. In figure 4.10 the waveform window of the simulation can be seen. In it the data transmission is shown in blue, the `TX_ACTIVE` signal is shown in yellow, in orange the activation signal `TX_DV` and in pink the end transmission signal `TX_DONE`.

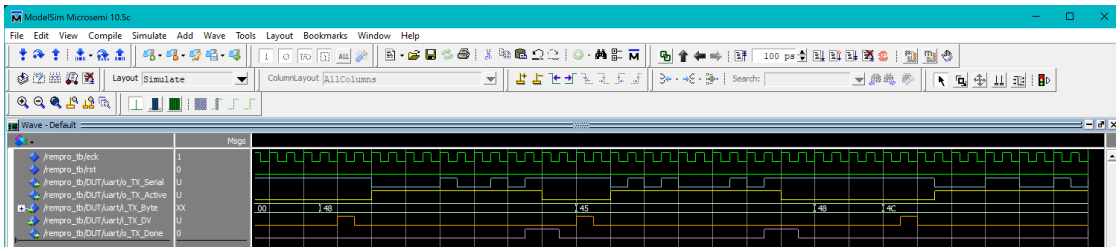



**Figure 4.10:** RTL simulation of the hardware setup functionality, including the processor and UART module

### 4.3.3 Implementation

After verifying that the system worked properly in simulation, the whole setup was implemented through the Microsemi design flow. The target device is the ProASIC3 A3P250.

Using the Libero SoC 11.9 software suite, the input and output ports were assigned to the GPIO pins of the board. The 40 MHz oscillator on the board was used to generate the clock signal, connected to pin 26 (in red). One of the buttons on the board corresponding to pin 67 was assigned to the start signal for the program. The `reset` signal was assigned to pin 129 and connected to a jumper. To perform a system reset, the other connector of the jumper was plugged to a 3.3 V supply pin on the board.

The signal at the output of the UART module was assigned to pin 133 and connected to the `RX` input of the USB to UART bridge with a wire. The UART activity signal was assigned to pin 132 but left unconnected as it was not necessary for our purposes. Moreover, to visually confirm the successful transmission, the byte sent to the UART module was sent as output of the FPGA and connected to the eight LEDs in the board.

In table 4.1, the resources used by the processor alone and by the entire system are shown. It can be seen that the overhead introduced by the UART module is negligible at the area level and does not contribute to performance degradation.

| | Cores (Combinational) | Cores (Sequential) | RAM Blocks | Frequency |
|---|---|---|---|---|
| Processor | 1264 | 373 | 2 | 16.035 MHz |
| Processor and UART | 1295 | 406 | 2 | 16.865 MHz |

**Table 4.1:** Resource usage and implementation frequency comparison between processor only implementation and processor and UART implementation.

A python script, was used to read inputs coming in from the COM port associated to the UART bridge, waiting to receive data from the system. When data is received, the script prints the characters to the terminal using ASCII encoding (figure 4.11).

### 4.3.4 Conclusions

The correct behaviour of the processor was validated in an hardware implementation. The results align with the HDL simulation executed until now. The processor executes reading and writing instuction to and from memory, and it is capable

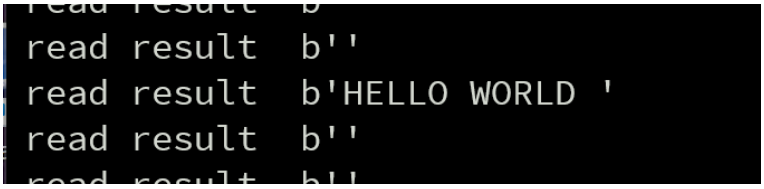



**Figure 4.11:** Correct execution of the benchmark program received by the host PC through the serial interface

of external transmission of data, allowing its implementation in more complex systems.

## 4.4 Tests on the implementation of a bidirectional bus for memory transfer

After tests have verified the correct behavior of the system when communicating with an host PC and after measuring the latency due to the connection between two pins in the board, it is now necessary to test the ability of two processors to communicate with each other.

The requirements of the Rempro system include a routine to substitute a board in case a failure is detected. This substitution routine has to allow the system to resume execution where the old triplet was interrupted. To achieve this, the routine has to transfer the processor state, including the memory locations, the Program Counter, and the stack, from the defective triplet to the new one. This data is relayed through a bidirectional parallel bus.

To verify this behavior, a test system was realized, including two Hepro processors connected externally through GPIO pins connected by 16 jumpers, an UART interface implemented in the same FPGA, and an host PC. The two processors execute the same firmware, which includes a test program, that uses the UART to send to the host PC the character codes for the string "HELLO WORLD". In the host PC a serial interface developed in python waits to receive data.

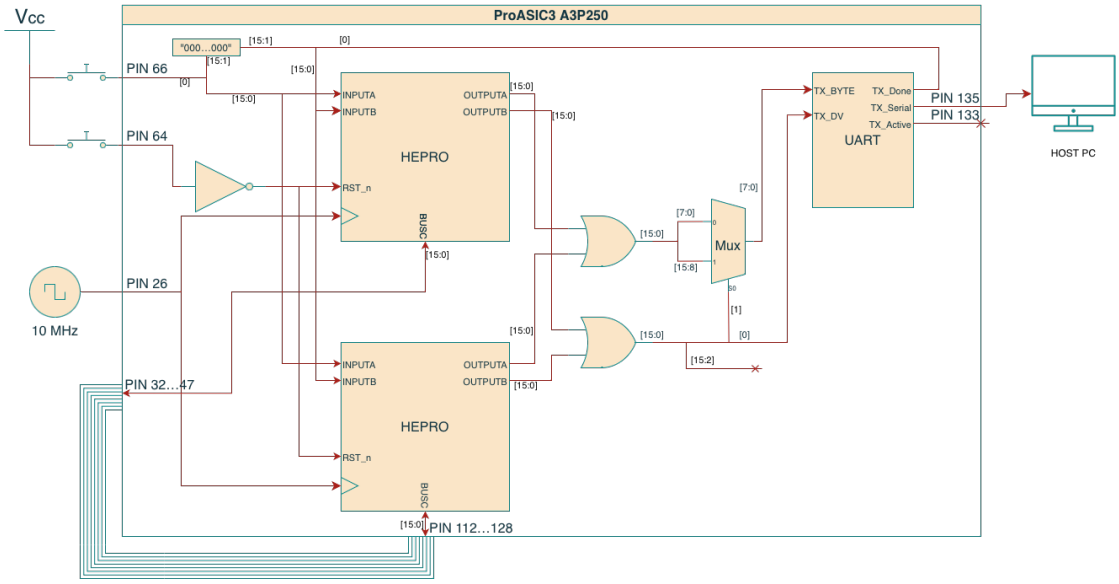



**Figure 4.12:** Diagram showing the internal and external connections of the test system.

### 4.4.1 The test setup

While the final version of the Rempro system is composed by multiple triplets in TMR configuration, to test bus communication and the recovery routine, a simpler architecture was adopted, with two Hepro processors in a single FPGA, comunicating via an external bus. With this solution, implementation is simplified preserving however the conditions to observe. Specifically, in Rempro, every processor in a triplet, communicates only with a single processor of the other triplets. In order to reproduce as closely as possible the real conditions in our test, two independent I/O banks were used to connect the two processors with the external bus. In this way the objects of the study, such as the communication between two processors and the recovery routine are isolated from the external conditions, removing complexity from the test structure.

Both processors should have the ability to turn off and on the other. For this reason, the `PWC` memory location in each processor, is used to control the reset signal of the next processor, simulating an inactive condition. Moreover the system can also be reset by pressing a button on the board, while a second button allows to start the execution of the program.

To enable communication with the host PC, a UART module was implemented alongside the processors couple and connected to them through the ports `OUTPUTA`, `OUTPUTB`, and `INPUTA` as shown in figure 4.12. Due to the fact that during transmission, only one processor is active at all times, excluding during substitution, no conflicts are expected to occur in the control of the UART interface, so the outputs are connected together to the UART inputs using an OR gate. Data locations are 16 bits in length each while the protocol allows to transmit 9 bits at a time at most. To overcome such a limitation, a MUX controlled by the processors, is used to split the 16 bit words into two 8 bit halves. Specifically, `OUTPUTA` transmits the 16 bit word to be sent, `OUTPUTB` is used to drive the UART activation signal and to select which of the two byte to send., while `INPUTB` is used to receive the end trasmission signal from the UART. Each of the signals is controlled by the firmware by means of memory reads and writes.

To connect to the host pc a USB to UART bridge (Silicon Labs CP210x USB to UART Bridge shown in figure 4.1) was used. The board was connected to the bridge using three jumpers: two connecting the GND and 3.3 V and one to connect the `RX` pin of the bridge to the output of the UART module. To implement this output, a GPIO pin on the board is assigned during the implementation design flow.

### 4.4.2 The substitution routine

The firmware was written in the assembly language specific for the processor. There are no available high level functions so the resources and the ISA must be used to their full extent to write a program according to the requirements. The firmware must pause execution to substitute the active processor, then resume and trasmit to the host PC the ASCII character for the string "HELLO WORLD". The processors have two global input ports and two global output ports,`INPUTA`, `INPUTB`, `OUTPUTA`, and `OUTPUTB`, associated to special memory addresses. The same input ports in both processors receive the same signals while the same output ports are connected toghether to the same inputs using OR gates. In addition to this, the special locations `BUSC` and `PWC` are used respectively to exchange data and to turn on and off the processors. These are the only communication ports accessible to pass data to and from the processors.

In this setup, bit 0 of `INPUTA` is connected to one input of the test system, connected to the implementation phase to the GPIO pin associated to a button on the board. This will let the user start the program when the button is pushed. The first bit of `INPUTB` receives the end transmission signal from the UART module. Port `OUTPUTA` is connected to the MUX tasked with selecting which of the two bytes, composing each word in memory, is sent to the UART. Bit 0 of the `OUTPUTB` is connected to UART port `TX_DV` to control the start of the transmission, while bit 2 is connected to the selection port of the MUX.

The firmware for both processors is the same. When the program starts the processor reads the memory location associated with the bus, `BUSC`, and if it doesn't detect data on the bus, it starts the normal execution of the program. If data on the bus is detected, it calls the procedure to receive the memory state from the other processor.

During execution the program reaches a checkpoint and calls the procedure to substitute the processor, writes to the memory location `PWC` to activate the second processor, then it writes some data to the bus to keep control over it. The second processor, detects that the bus is busy and calls the procedure to recover the memory state. It then waits for the first to stop writing on the bus and sends a signal to confirm synchronization. From that moment, the first sends down the bus the data contained in every memory location while the second reads the bus and saves the data in the corresponding location. After the transfer is over, the first processor waits to be turned off and the second writes on `PWC` to turn off the first. Since calling a procedure results in the PC being saved to the stack, and the stack is transferred alongside the rest of the memory during the process, returning from the procedure will result in the second processor resuming the execution of the program where the first was interrupted.

The program executed before and after the substitution routine loads in consecutive memory locations the hexadecimal values of the character codes for the string "HELLO WORLD " (the blankspace is used to transmit an even number of characters). Once the writing procedure is completed, the substitution routine is called. When the new processor resumes execution, it reads one location at a time writing its value to the special memory address associated to the global output port `OUTPUTA`. Writing in the memory location connected to port `OUTPUTB`, the

processor selects which byte is sent and starts transmission. The only two used bits are the two least significant ones, with bit 0 starting the transmission when its value is 1, and bit 1 selecting which byte is transmitted. To avoid misbehaviour during transmission the start signal must be kept high only for a single clock cycle. In `OUTPUTB` the values 1 and 3 are written to start the transmission of the first and second byte, respectively, and values 0 and 2 after a single clock cycle to disable the start signal without changing the selected byte. The processor then waits for the `TX_End` signal from the UART to arrive at `INPUTB` to signal the end of the transmission before sending the next byte.

### 4.4.3 Simulation

An RTL simulation of the design was performed to verify the behaviour of the system. In figure 4.13, the waveform window of the simulation can be seen. In it the two ends of the bus can be seen, transmitting the memory content, from the first to the second.

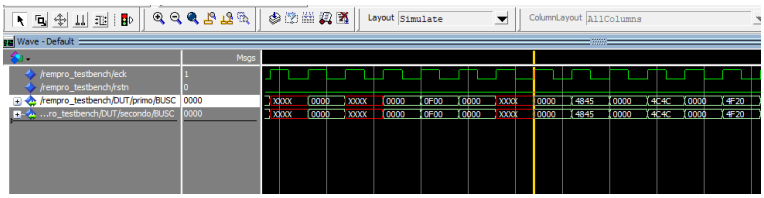
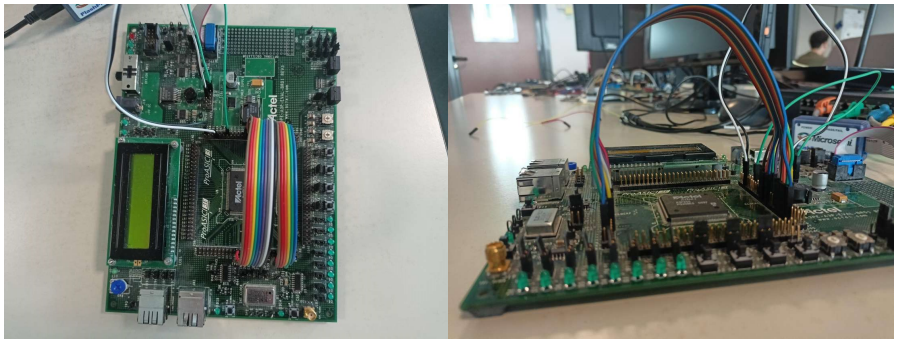



**Figure 4.13:** RTL simulation of the memory transfer functionality, including the two processors and the bus.

### 4.4.4 Implementation

After the simulations showed positive results, the entire hardware setup comprising the two processors and UART module was implemented through the Microsemi Design Flow. The target device is the ProASIC3 A3P250. During the design flow the inputs and outputs of the design were assigned to the GPIO pins of the board. An external signal generator was used to obtain a clock signal with a frequency of 10 MHz. The clock signal was assigned to pin 26. One of the buttons in the board, corresponding to pin 6, was assigned to the program start signal. The reset signal

was assigned to another one of the buttons, to pin 64. The signal at the output of the UART module, was connected to pin 135 and connected to the `RX` input of the UART to USB bridge by means of a wire. The activity signal for the UART was assigned to pin 133 but in was left disconnected, as not used in our system. To realize the bus, 16 jumpers were used to connect together 16 pin from the East I/O bank to the West I/O bank (Figure 4.14).



**(a)** Top view **(b)** Side view

**Figure 4.14:** Photo of the test setup showing the external board connections and the physical bus implementation

A python script, connected to the COM port associated to the USB to UART bridge, read continuously from the port, waiting to receive data from the processor. When data is received, it is printed to the terminal using ASCII encoding (Figure 4.15.

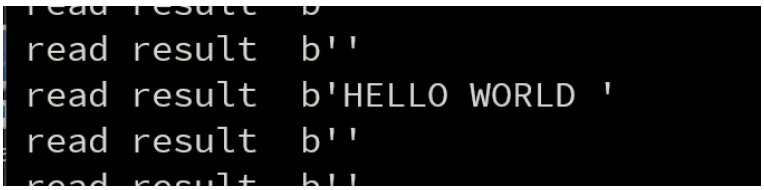



**Figure 4.15:** Correct execution of the benchmark program received from the host PC through the serial interface.

### 4.4.5 Conclusions

The results of this test allowed to validate two important components of the Rempro system. Both the bus transmission and the recovery procedure behaved as expected. The processors couple was able to pause execution, substitute the active processor and transfer the program state to the new processor while deactivating the first one. The second processor then reprised the execution where the first one had stopped, and completed the task correctly. The processors remained synchronized and no data was lost. This demonstrates the validity of the initial idea and encourages further development of the system.

# Part III

# Fault tolerance evaluation

# Chapter 5

# The RISC-V processor NEORV32

Designing the electronic system of a spacecraft is an exercise in balancing the different constraints of size, weight, power and cost that are imposed on each module while still hitting the performance target required to carry out the mission adequately. Spacecraft designers traditionally rely on legacy processors such as the RAD750 or the GR712RC. While these are well established and mature architectures, innovation has stagnated for them as the industry move towards the new trend of open source IP cores such as the RISC-V ISA. Cores implementing this architecture offer greater performance and a wide range of hardware supported instructions enabling more efficient software. Due to the open-source nature of the RISC-V project, cores and development tools are widely and cheaply available for any kind of endeavor [25].

## 5.1 The RISC-V ISA

Development on the RISC-V ISA started in May 2010 [26], with the aim of realizing an open source computer system that would be of aid to both academic and industrial users. While it is not the first open source ISA to ever be published (the DLX, for example, was introduced in the 1990s), it is certainly the more popular at the present time [27].

The term Reduced Instruction Set Computer (RISC) refers to a classification of ISAs into RISCs and Complex Instruction Set Computers (CISCs). The CISC approach to design proposed extensively large ISAs, providing instructions for any possible use case of the machine, RISC ISAs were very limited, containing only a limited set of instructions, requiring therefore more instructions to perform the same tasks [28].

The RISC-V design team set to create a RISC ISA that would be able to support the widest variety of pratical use cases in order to build a large community of users, researchers and companies around the project.

RISC-V International is the non-profit business association based in Switzerland that owns, mantains, and spearheads the development of the RISC-V ISA. Its members have access to and participate in the development of the RISC-V ISA specification and extensions as well as related hardware and software.

## 5.2 Overview of the NEORV32 processor

The NEORV32 is one such cores implementing the RISC-V ISA. The project is intended as a ready-to-go auxiliary processor or stand-alone microcontroller [29]. The purpose is to provide an easy to use platform including the core and development tools that allow an effortless implementation with a high level of modularity and flexibility. The project is intended to be implemented as a soft-core in FPGA. This allows the user to configure the core with the *exact* hardware features required to run the programs.

The project documentation details every phase of the implementation process guiding users to all necessary steps to go from the netlist to the fully implemented core on the FPGA

NEORV32 is not based on another core, it is built from the ground up following the RISC-V ISA specs. The goal is to provide another option in the RISC-V / soft-core space embracing concepts such as documentation, platform-independence and portability, RISC-V compatibility, extensibility, customization, and ease of use. Special attention is furthermore payed to execution safety using Full Virtualization. The Central Processing Unit (CPU) aims to provide fall-backs for everything that could go wrong. This includes malformed instruction words, privilege escalations

and even memory accesses that are checked for address space holes and deterministic response times of memory-mapped devices. Precise exceptions allow a defined and fully-synchronized state of the CPU at every time an in every situation.

## 5.3 The NEORV32 CPU

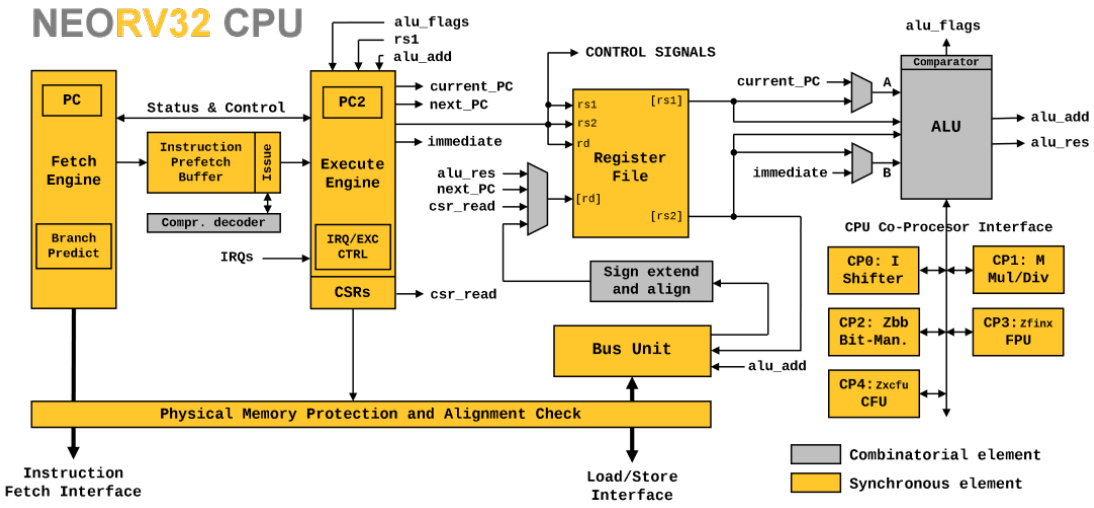



**Figure 5.1:** Diagram of the NEORV32 CPU

### 5.3.1 Multicycle architecture

While most mainstream CPUs feature a pipelined architecture, NEORV32 is based on a multicycle architecture where every instruction is executed in multiple clock cycles. Compared with pipelined architectures, this allows for a simpler, more compact design, as pipeline hazard detection is not necessary, maintaining however an higher throughput and frequency with respect to single cycle architectures.

Compared to single cycle and fully pipelined architectures, multi cycle architectures have higher throughput and clock speed than single cycle ones and less hardware complexity, area, and power consumption with respect to fully pipelined designs. In particular the main advantages are:

- The lower area requirements come from the lack of pipeline hazard detection

and resolution logic, as well as the ability to use core modules to perform multiple tasks.

- As opposed to single cycle architectures, it does not require a memory that can be read asynchronously, which would not be feasible to implement in FPGAs. Furthermore, such design usually have a very long critical path tremendously reducing maximal operating frequency.

- Compared to pipelined design, it executes only one instruction at a time. When multiple instructions are in a pipeline, if an exception is raised at the end of the pipeline, all instructions in the earlier stages have to be invalidated. That requires additional logic which is not necessary in multicycle architectures.

- In addition to the last point, executing a single instruction at a time, besides reducing costs, it simplifies simulation, verification and debugging, state preservation and restoration during exceptions.

To reduce the loss of performance that characterizes a pure multi-cycle architecture, the NEORV32 CPU uses a hybrid approach: instruction fetch and execution are decoupled and operate independently. Data is exchanged through a queue composing a simple 2 stage pipeline. Each stage is implemented as a multicycle architecture.

### 5.3.2 CPU architecture

The CPU is designed as a pipelined multicycle architecture in which each instruction is executed as multiple micro operations. To improve performance the front-end, that is the instruction fetch is separated from the back-end, the execution stage, by means of a FIFO queue, called Instruction Prefetch Buffer (IPB). In this way the fetch stage can fetch new instructions even if the execution stage is still processing the previous ones.

The architecture is halfway between a traditional pipelined architecture, in which each stage takes one cycle to complete when not stalled, and a multicycle one, in which every instruction is executed in a series of micro operations including the fetch. Combining toghether the two paradigms allows for a better throughput

compared to multicycle architectures and a lower hardware cost compared to pipelined architectures.

The CPU is designed as a Von Neumann machine, providing two independent interfaces for instruction fetch and data access. The two interfaces however, are merged into a single internal bus via a prioritizing bus switch where data accesses have higher priority. In this way all memory addresses are mapped to a single 32-bit address space.

### Register File

The register file contains the architecture registers. For `rv32i` architecture there are 32 32-bit registers, while in `rv32e` architecture there are 16 32-bit registers. Register `zero` or `x0`, always reads as 0 and cannot be written to.

The register file is implemented as a synchronous memory with synchronous read and write. Register `zero` is also mapped to a physical location in order to avoid adding logic that returns zero when reading the register and shortening the critical path.

The register file uses two access ports. One is read-only and is used to access the second source register in instructions, the other is a read/write port for accessing the first source register or for writing data to the destination register in instructions. A single dual port RAM can therefore be used and the register file can be fully mapped to the FPGA block RAM. Moreover, since reading and writing never occur on the same clock cycle, there is no need for bypass logic.

### ALU

The ALU processes data from the register files but is also used in memory and branch address computation. All the `I` ISA Extension [30] operations are implemented and take a single clock cycle to complete. More sophisticated operations are performed by the ALU co-processors.

The co-processors are iterative units that take several cycles to process data. Besides the base ISA's shift operation all processing based ISA extensions are implemented as co-processors.

All co-processors are required to complete processing within a time bound,

defined by the user as a constant in the VHDL. If the unit takes more time to complete, the operations is terminated and an illegal instruction exception is raised.

### Bus Unit

The bus unit handles data memory access via load and store operations. It is able to perform data adjustment when accessing sub-words and performs sign extension in signed load operations. The `PMP` ISA extension [30] is also included to perform permission checks on all data and memory accesses. All bus interface signals are driven by registers so even complex bus networks will not affect the maximum frequency.

The CPU does not support hardware based handling of unaligned memory so any unaligned access will result in a bus load/store unaligned access exception. The exception handler can however be used to emulate unaligned memory access.

### Control Unit

The control unit generates all the signals required by the different CPU modules. It is composed of a front-end and a back-end.

- The **front-end** controls the fetching of instructions in chunks of 32 bits as either a single 32-bits instruction, two 16-bits instructions, or some mixture of those. The instructions, along with control and exception information is stored in a FIFO, the IPB. The depth of the FIFO can be configured by the user.

  Using the FIFO the front-end can do speculative instruction fetches, fetching the following consecutive instructions continuously at all times. In this way front-end and back-end can be decoupled so that each operates in parallel, improving performance. All potential hazards caused by the speculative instruction fetch are already handled by the CPU front-end ensuring a defined execution stage while preventing security side attacks. The front-end implements a very simple branch prediction that stops fetching further instruction while a branch/jump/call operation is in progress.

- In the **back-end** instruction data from the IPB is decompressed (if the `C` ISA

extension [30] is enabled) and sent to the CPU back-end for actual execution. Execution is conducted by a FSM that controls all of the CPU modules. The back-end also includes the Control and Status Registers (CSRs) as well as the trap controller.

### 5.3.3 Other features

#### Sleep Mode

The NEORV32 features a sleep mode that can be entered to power down the core and reduce dynamic power consumption. In sleep mode all the internal CPU operations are suspended. The peripherals and the I/O ports, like timers and interfaces.

#### Full Virtualization

The NEORV32 shares with the RISC-V ISA the goal of providing maximum virtualization capabilities at the CPU and SoC level to enforce a high standard of execution safety. All the traps detailed in the RISC-V specifications are supported providing defined hardware fall-backs for any expected and unexpected situations that may arise during operation. For each trap the core maintains a defined and fully synchronized state in the whole system, that is there are no out-of-order operations that may need to be rolled back. In this way the execution behavior is always well defined and predictable, improving the execution safety.

# Chapter 6

# Analysis of the reliability of the Rempro and NEORV32 processors via fault injection

After finalizing a working prototipe of the Rempro processor, the next step is evaluating its reliability and comparing it to the NEORV32 running the same program. The reliability is the probability, $R(t)$, that a component is working correctly in a time interval $t - t_0$, provided it was working correctly at time $t_0$.

## 6.1 Fault injection

The Rempro system must tollerate the harsh space environment and has to be able to maintain its operation under the exposure to heavy doses of ionizing radiation. In order to simulate the voltage pulses induced by radiation into the system, a fault injection campaign was carried out on both processors and its results were compared for both processors in order to evaluate their ability to tolerate faults. Another objective of the fault injection is also to identify the most vulnerable modules of the design in order to focus the efforts on those when implementing fault mitigation techniques on the design.

Fault injection can be carried out in hardware, software, emulation, and simulation. In this initial evaluation phase, a simulation approach was used to test our devices using the simulation software ModelSim. The technique allows tto simulate faults and malfunctions inside the processor, analysing the response of the system and identifying possible vulnerabilities. During the campaign, voltage pulses of variable length are introduced in specific points of the design in order to evaluate the ability of the processor to tolerate the faults and keep working correctly. The results of this campaign will provide us with important indications on how to improve the reliability of the system and implement better fault mitigation measures in a more effective way.

The simulation in ModelSim allows to carry out the campaign in a controlled and repeatable way, allowing the detailed analysis of the results. Using custom scripts, we can automate the fault injection process and the data gathering, insuring an efficient evaluation of the processor's fault tolerance.

## 6.2 The post implementation netlist

The purpose of the fault injection campaign is to simulate the voltage pulses on the circuit nodes, testing that is, the vulnerability to SETs. This technique allows to evaluate the reliability of the system when subjected tu sudden perturbations and transients due to environmental ionizing radiation.

In order to replicate as closely as possible the real conditions, the target for the simulation campains will be the post-layout netlist. This netlist, which also includes information on delays in Standard Delay Format (SDF), offers a more accurate representation of the final circuit with respect to the RTL description. During the layout process, information on the module hierarchy is not preserved, and only the cells are represented, that is the logic functions and FFs implemented using the configurable blocks of the FPGA described in the specific library for the ProASIC chip.

To identify all the instances of the cells constituting the system, the ModelSim command `find -r ./DUT/* -file <file path>` was used [31]. This command exports a list of these cells to a file located in the specified path. Each line of the file represents an instance of a specific cell and contains information on its path,

its name and the type of cell. Subsequently, some preliminary simulations were conducted, using such information to better understand the characteristics of the post-implementation netlist.

During these preliminary simulations some interesting characteristics emerged. In particular, some significant differences were observed between the amplitude of the pulse at the input port of the FF and the amplitude inside it. In some cases such difference was more than ten times the amplitude itself. Moreover, a delay in the order of the nanoseconds was measured at the inputs of some FFs.

## 6.3 Post-implementation model of FF delay

The post-layout netlist describes the design as implemented using the programmable blocks of the FPGA, positioned in the FPGA itself. The delays, included the internal ones of the programmable block, and those derived from the placing, and routing of the signals, are computed and saved in the SDF file to be used during simulations.

The FFs present in the design are modeled including the delays computed in the place and route process. In the SDF file, for each FF instance, this data is listed:

```
(CELL
  (CELLTYPE "DFN1E1C0")
  (INSTANCE \\dmem\/registro_19\/counter\[8\]\\)
  (DELAY
    (ABSOLUTE
      (PORT D (15.33:19.00:20.75) (14.28:17.71:19.33))
      (PORT CLK (5.79:7.18:7.84) (5.83:7.23:7.90))
      (IOPATH CLK Q (4.09:5.14:5.81) (5.20:6.53:7.37))
      (PORT CLR (5.68:7.04:7.69) (5.73:7.10:7.76))
      (IOPATH CLR Q () (3.76:4.73:5.34))
      (PORT E (5.95:7.38:8.06) (5.34:6.63:7.24))
    )
  )
  (TIMINGCHECK
    (SETUP (posedge D) (posedge CLK) (3.80:4.77:5.39))
    (SETUP (negedge D) (posedge CLK) (3.56:4.47:5.04))
    (HOLD (posedge D) (posedge CLK) (0.00:0.00:0.00))
    (HOLD (negedge D) (posedge CLK) (0.00:0.00:0.00))
    (WIDTH (posedge CLK) (4.06:4.77:4.77))
    (WIDTH (negedge CLK) (3.67:4.31:4.31))
    (RECOVERY (posedge CLR) (posedge CLK) (2.10:2.63:2.97))
    (HOLD (posedge CLR) (posedge CLK) (0.00:0.00:0.00))
```

```
    (WIDTH (negedge CLR) (2.52:2.96:2.96))
    (SETUP (posedge E) (posedge CLK) (3.07:3.85:4.35))
    (SETUP (negedge E) (posedge CLK) (4.29:5.39:6.08))
    (HOLD (posedge E) (posedge CLK) (0.00:0.00:0.00))
    (HOLD (negedge E) (posedge CLK) (0.00:0.00:0.00))
  )
)
```

From the cell description it is possible to deduce the absolute delays, between the arrival of the signal at the input port and the arrival inside the FF, as well as the setup and hold times for every signal. In the simulation every FF will present a input signal and a corresponding internal signal, that includes the delays listed in the SDF file. For every signal, there are two delay groups, one for the rising edge and one for the falling edge, while in every group there are three types of delay, for the best, worst and typical case. In our fault injection campaigns the typical delays are used.

## 6.4 SET pulse injection methodology and observed effects

In order to inject a pulse in the design, three pieces of data are required: the node, that is the location in the circuit that is to be disturbed, the injection instant, and the pulse duration.

- The node can be the input or output of a component, a junction between more connections, or an internal signal of a component. The node selection is described in Section 6.5.
- The injection instant is the moment in which the pulse is introduced in the node. This is fundamental to determine the effects of an injection, as depending on the chosen instant, the FF could already have at its input an high value, or could even be inactive.
- The pulse duration determines for how long the signal associated with a given node is kept high. The literature suggests a common series of values for the pulses caused by ionizing radiation. To select the amplitude the sample and

hold times of the logic and the working frequency of the system must be taken into account.

The effects of an SET on the logic of the system may vary depending on these three parameters. Observing what happens when this pulses reach a sequential element (such as a FF, as shown in figures 6.1-6.5) the most common events are:

- *FF enabled but incorrect timing:* the signal arrives on an enabled FF, with an high enable signal, but the pulse stays outside the FF sampling window so the glitch does not affect the output (figure 6.2).
- *Correct timing but inactive FF:* the signal arrives to the FF during the sampling interval but the FF is inactive. There are not any observed effects on the output signal (figure 6.3).
- *Sampled signal:* the pulse arrives during the sampling window and is observed by the enabled FF. The output of the FF changes following the pulse and the error is propagated (figure 6.4).
- *Unperturbed node:* the signal at the input of the FF is already high in the instant the pulse is injected. The glitch will not have any effect on the FF (figure 6.5).

## 6.5 Generation of the fault injection scripts

A Python script was developed in order to simplify the fault injection process. The script makes use of the instances list file obtained previously, and is capable to generate an arbitrary number of ModelSim Tcl scripts. Using these scripts, it is possible to perform fault injection on different nodes, in different time instants on the the circuit. This approach offers the necessary flexibility to test the system in a variety of realistic scenarios, and evaluaate it's tolerance to perturbations.

To set up the Python script, it is necessary to specify the start and end times of the program, the name of the file containing all of the cell instances with which the system is physically implemented, the number and type of instances to select, the duration of the impulse, and the number of injections to perform for each

node. The nodes are the positions in which the pulses will be injected; for this fault injection campaign the chosen nodes were the inputs of the standard cells constituting the physical netlist of the design.

Once started, the script reads the file containing the instances list and selects from it a random subset with a number of elements chosen by the user. It is also possible to make a selection based on the type of cells or the name. Since cells of the same type present the same name for input and output signals, the script is capable of detecting the input and outputs for every instance. Using this information. the script creates a number of Tcl scripts for ModelSim that inject the pulse in a random time instant inside the simulation interval. Such interval is dependent on the benchmark application used as a stimulus during the fault injection campaigns.

Using ModelSim, we can observe the effects that faults cause, monitoring the two output ports of Rempro. The script reads then the values that arrive on the two output ports, `OUTPUTA` and `OUTPUTB`, saving them in a list. In order to verify whether the faults have caused a deviation from the correct behavior of the program, this list of values is compared against a golden reference, that is a list of values acquired in a similar way from a simulation where no injection was performed.

Inside the scripts some commands are included to measure some important data. For example:

- **Pulse measure:** from the analysis of the SDF file and the preliminary verifications, it is possible to determine that the signal arriving on the input port experiences a delay entering the cell. This has two components, the interconnection delau and the port delay. The interconnection delay represents the time taken by the signals to flow through the interconnections between the system components. The port delay is due to the glue logic and the characteristics of the component: when a signal is applied to a port, the component takes some time to switch and affect the corresponding output signal. These delays are summed up and associated to the inputs of the logic components. For this reason the scripts compute the effective duration of the pulse inside the cell, measuring the actual starting and ending instants.

- **Clock measure:** the delay of the clock signal at the input port of the 'FF is

measured. This delay is due to the routing path of the clock.

- **Injection statistics:** since the basis of the campaign is the injection of pulses at random instants inside the entire simulation interval, it is possible that, due to the nature of the sequential circuits, there are some conditions for which the fault is masked or does not cause any observable effect on the target node. In the first case, this is because the signal driving the node is already in a high logical state at the instant of application of the transitory pulse. In the second case the pulse, while arriving during the target FF sampling window, it is not sampled due to the FF being in an inactive state (i.e. enable signal low).

Finally a master Tcl script was developed to monitor the entire experiment, taking care of everything from the injection of the faults, to the analysis and comparison of the results against the golden reference. Moreover, the master script is also tasked with gathering all the data and measures explained previously (pulse length variation, clock delays, pulse masking) and registering statistics for each node. The results are saved in two different files: one to gather the results of every single script and one for the total results for each instance.

During the execution of the injection scripts, different statistics are gathered to evaluate the effectiveness of the mitigation techniques implemented. For example, it is possible to compute the percentage of injections that caused failures over the total, or the number of times in which the injection caused or not a failure, and other metrics of interest. All this information is gathered and saved in the apposed files for a detailed analysis at a later time.

The combined use of the Python script and the Tcl master script, allows to automate the fault injection process and analyze a large amount of scenarios. The injection parameters can be altered, changing the number of injections per node, or the duration of the pulse, in order to evaluate the behavior of the system under stress conditions. Moreover, the random approach in the selection of the instances allows to cover a large portion of the circuit and identify the most critical areas.

The analysis of the results obtained from the fault injection simulations gives an important knowledge base to address the system vulnerabilities and improve the fault tolerance of the system. The weak points of the project can be identified

and the effectiveness of the adopted mitigation techniques can be evaluated. This data is fundamental to optimize the design and guarantee an high reliability and safety of the electronic components employed in space missions.

## 6.6 Planning the fault injection campaign

The fault injection campaign was performed focusing on the input signals of the FFs. The transient caused by the radiation was therefore modeled as a glitch on the input signal on `D` port. Such choice was motivated by the fact that performing the simulation in post-layout, the input ports are already associated with the propagation delay of the combinatory path. That means that even injecting in proximity of the FF, it is possible to take into account the broadening effect. That is to say that depending on the associated delay, toghether with the injection instant, and the sampling time, the pulse can be filtered (that is, disappear and not affect the logic at all) or widen.

The total number of FFs in Rempro was 371, while the NEORV32, being a larger design, contained NUMBER FFs. Due to the high number, only 500 FFs were selected from the set of NEORV32 instances, while the entire set of Rempro instances was used as a target for the campaign. The working frequency for Rempro is 12.5 MHz, while the setup time, different for each type of FF is between 400 ps and 550 ps. On the other end the NEORV32 runs at a faster frequency of 40 MHz. For this reason, the pulse duration was set to 600 ps, sufficiently big to be sampled but still realistic enough as a model of the effects induced by ionizing particles.

In order to increase statistics gathering and having a more exhaustive set of results, for each node 100 injections were performed in different time instants, selected randomly from the simulation interval. This allows to evaluate the component sensitivity to faults in specific moments, and to gather a wide range of data for future analysis.

The entire campaign entails that both processors be tested during the execution of three different benchmark programs. In this way it is possible to stimulate different datapath resources in a different way, observing how the same transients, applied on the same nodes, have consequences varying in severity depending on the application workload. The three benchmark were written in the assembly language

of the two processors, since Rempro lacks a compilator for high level languages. In particular:

- The first program, **Arithmetic test**, executes a series of arithmetic operations, jumps and branches in order to represent the execution of a generic program, stimulating a condition of balanced workload for the processor.
- The second, **Matrix Multiplication**, is a function to compute the product of two matrices. Having a large number of computation instructions, it focuses the workload on the ALU.
- The third, **Quicksort**, is an implementation of the Quick Sort sorting algorithm. It was chosen as it contains a large number of loops and jumps, recursive calls and reads and writes to memory, allowing to stress the conditional jumps module, memory, and the stack.

Using these three programs this fault injection campaign provides information on a good amount of use cases and working conditions of the processor, allowing to highlight which workloads make the system more susceptible to faults. Moreover, it will allow to observe the response of both systems to SETs and identify the most sensitive FFs. These results will be fundamental to implement effectively the fault mitigation techniques on the project. Moreover they will constitute a solid basis for future analysis of the reliability and for the optimization of the fault tolerant strategies in space missions.

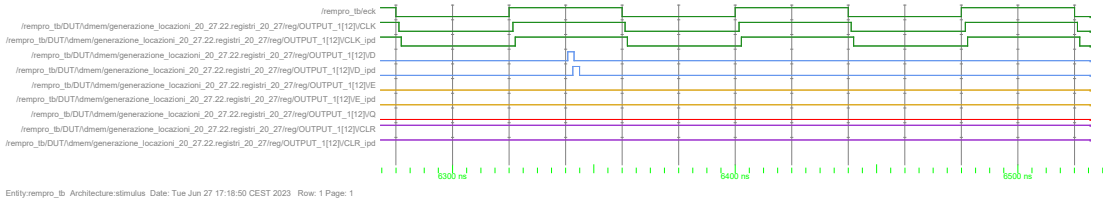



**Figure 6.1:** Signals of a FF before a SET injection.

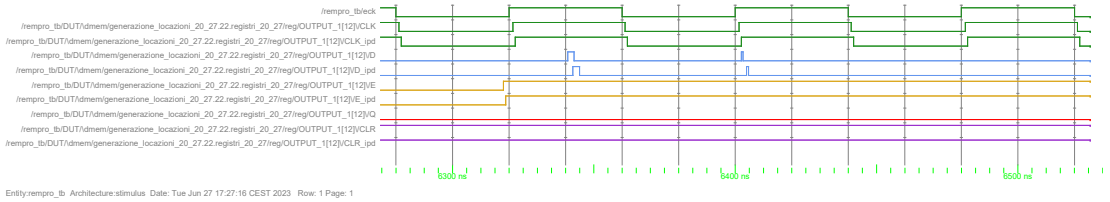



**Figure 6.2:** Example of a SET injection outside the sampling window of the FF.

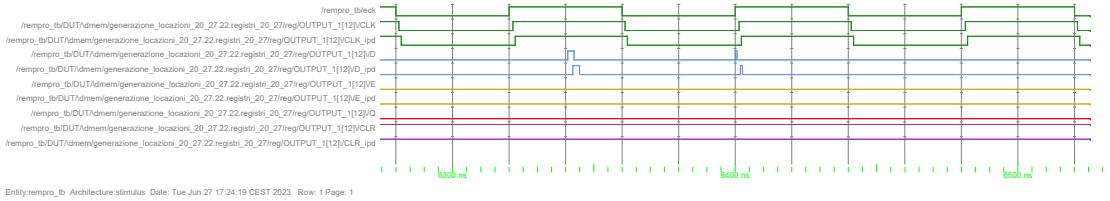



**Figure 6.3:** Example of SET injection in a instant when the FF is inactive (low value of `enable`).

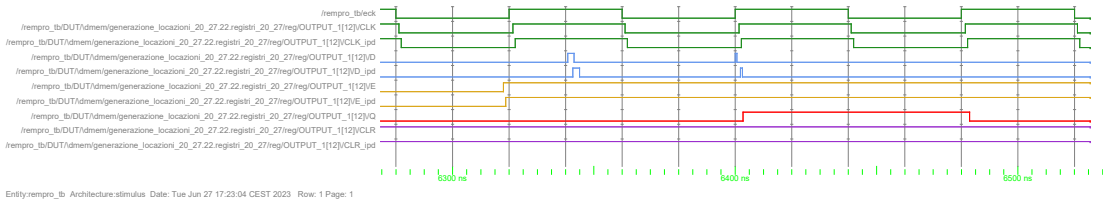



**Figure 6.4:** Example of injected pulse that was correctly sampled by the FF.

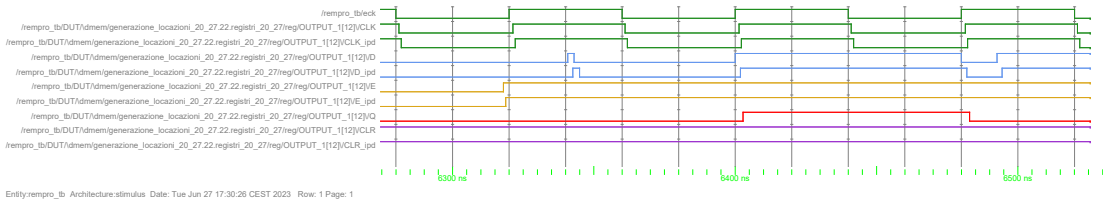



**Figure 6.5:** Example of injection in which the perturbed signal was already high.

# Part IV

# Results and conclusions

# Chapter 7

# Analysis of the results of the fault injection

## 7.1 Total results of the fault injection campaigns

| Processor | Work load | Fail | Not Inj. | Masked | Wrong Timing | Enable Inact. | Total |
|---|---|---|---|---|---|---|---|
| Rempro | Arith | 107 | 6594 | 2137 | 25606 | 2751 | 37195 |
| | MatMul | 120 | 5326 | 2137 | 23612 | 5614 | 36809 |
| | Qsort | 40 | 11286 | 2979 | 23698 | 1325 | 39328 |
| NEORV32 | Arith | 75 | 6073 | 2627 | 40467 | 788 | 50030 |
| | MatMul | 86 | 5153 | 4942 | 39243 | 643 | 50067 |
| | Qsort | 120 | 5489 | 3594 | 40132 | 715 | 50050 |

**Table 7.1:** Table summarizing the results of the fault injection campaigns.

After carrying out the fault injection campaign to completion, the results are summed up in table 7.1. The same data can also be consulted in the form of a pie chart in figure 7.1. From the graph and the table it is clear that Rempro has still some way to go before it can match and surpass the base reliability of NEORV32. It must be however noted, that no fault mitigation measures are yet in place in the project. Due to the lower complexity of the processor, adding such feature would

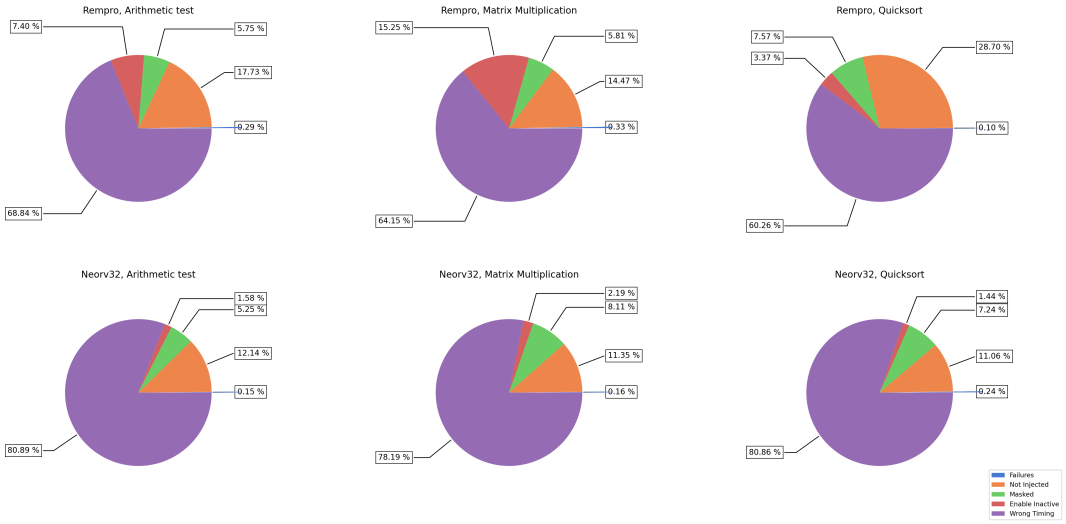



**Figure 7.1:** Comparison between the fault injection campaigns on the NEORV32 and Rempro processors.

be easier compared to the bigger NEORV32, and it will involve a lower overhead in terms of area, power consumption, etc. The results of the fault injection were divided into five categories:

- **Failures:** When the pulse is sampled by the sequential logic, and it causes a wrong computation in the processor, it is registered as a failure.
- **Not injected:** When the positive pulse is injected on a node already presenting a high value, the pulse will not cause a glitch in the line and the circuit will behave as expected. This is registered as "Not injected".
- **Masked:** When the pulse is sampled but the output of the computations remains the same, that means that the error was not propagated and it is logged in the results as "Masked".
- **Wrong timing:** When the pulse is inject in an instant which is outside the sampling window of the FF, it will not be sampled and will not cause a misbehavior. It is logged as "Wrong timing",
- **Enable inactive:** When the pulse is correctly injected but the sequential

element it feeds into is inactive, the pulse will not be sampled. This is registered as "Enable inactive".

As it is shown in results, the percentage of failures in Rempro is double that of NEORV32 in almost all cases, except when running the Quicksort program. Since very little computation instructions are executed in it, this may signify that the ALU module is more sensitive to faults in this design. The opposite is true for the NEORV32 architecture, which is more sensitive when executing branch-heavy tasks. Moreover, most of the injections were not sampled by the FFs because they were not injected during the sampling window. This is expected since sequential circuits only sample signals in a small timeframe compared to the clock frequency. Comparing the results from Rempro and NEORV32, it appears that injections performed at the wrong instant were more frequent in the RISC-V architecture. This is probably due to the higher working frequency achieved by the NEORV32 processor. In Rempro it is also shown that a significant percentage of injections were performed on disabled sequential elements, while this was less frequent on the NEORV32. This is due to a clear difference in the architecture and management of the resources (table 7.3), and it is also mirrored by a significant difference in the switching power found between the two (table 7.2), as explained in section 7.2. Since in Rempro the switching power is noticeably lower than in NEORV32 it is reasonable to assume that a larger percentage of sequential circuits are disabled when a certain module is not in use, resulting in an higher number of pulses injected in inactive parts of the logic. The percentage of "Masked" injection, is more or less similar, while Rempro present a larger number of "Not injected" instructions.

## 7.2 Power and area analysis

The results of the implementation process for both processors are summed up in the tables 7.2 and 7.3. The Rempro processor is significantly less demanding both in terms of power and resource usage. The design presents a drastically lower switching activity compared to the RISC-V processor. The power consumption values are all close to each other for every type of load, so there is not a strong dependency on the stimulated modules of the CPU. The resource usage for Rempro is lower, requiring less than half the resources of the NEORV32. The lower power

| | NEORV32 | | | Rempro | | |
|---|---|---|---|---|---|---|
| | Arithmetic test | | | | | |
| | Best | Typ. | Worst | Best | Typ. | Worst |
| Total | 28.38 mW | 31.29 mW | 57.78 mW | 6.98 mW | 7.42 mW | 31.55 mW |
| Stat. | 4.91 mW | 5.22 mW | 28.97 mW | 4.91 mW | 5.22 mW | 28.97 mW |
| Dyn. | 23.48 mW | 26.07 mW | 28.81 mW | 2.08 mW | 2.32 mW | 2.58 mW |
| | Matrix Multiplication | | | | | |
| | Best | Typ. | Worst | Best | Typ. | Worst |
| Total | 28.41 mW | 31.33 mW | 57.82 mW | 7.23 mW | 7.82 mW | 31.85 mW |
| Stat. | 4.91 mW | 5.22 mW | 28.97 mW | 4.91 mW | 5.22 mW | 28.97 mW |
| Dyn. | 23.51 mW | 26.11 mW | 28.85 mW | 2.33 mW | 2.60 mW | 2.88 mW |
| | Quicksort | | | | | |
| | Best | Typ. | Worst | Best | Typ. | Worst |
| Total | 28.29 mW | 31.19 mW | 57.66 mW | 7.15 mW | 7.72 mW | 31.75 mW |
| Stat. | 4.91 mW | 5.22 mW | 28.97 mW | 4.91 mW | 5.22 mW | 28.97 mW |
| Dyn. | 23.38 mW | 25.97 mW | 28.69 mW | 2.24 mW | 2.50 mW | 2.78 mW |

**Table 7.2:** Comparison between the two processors in terms of power consumption when executing the benchmark programs

consumption and resource usage are clear advantages of a simpler architecture. This also means that once TMR is implemented, there will be a lower overhead in both power and area usage.

## 7.3 Timing analysis

Among the data gathered during the fault injection campaign, there was also information on the delays of the FFs on which the pulses were injected. Such data is presented as boxplots in figure 7.2. In box plots, the colored box is delimited by the first and third quartile of the distribution, while the internal line represent the median. The whiskers are drawn from the first and third quartile to the smallest and largest values that fall within 1.5 times the Interquartile Range (IQR) value. The graph compares the distributions of the following values measured in both

| | NEORV32 | Rempro | Total |
|---|---|---|---|
| CORE | 5873 (95.59%) | 2637 (42.92%) | 6144 |
| IO | 34 (22.52%) | 35 (23.18%) | 151 |
| Diff. IO | 0 (0%) | 0 (0%) | 34 |
| GLOBAL (Chip+Quadrant) | 6 (33.33%) | 4 (22.22%) | 18 |
| PLL | 0 (0%) | 0 (0%) | 1 |
| RAM/FIFO | 8 (100%) | 2 (25%) | 8 |
| Low Static ICC | 0 (0%) | 0 (0%) | 1 |
| FlashROM | 0 (0%) | 0 (0%) | 1 |
| User JTAG | 0 (0%) | 0 (0%) | 1 |

**Table 7.3:** Comparison between the two processors in terms of resources utilization after the placement.

processors while running the three different workloads:

- *Real Width:* While all the injected pulses have the same width of 600 ns, inside a sequential circuit, due to the different rise and fall delay of the signal, it can be broadened or narrowed. This can influence whether the pulse will be sampled or not, as a broadened pulse is more likely to fall within the sampling window. From the graph it can be seen that in Arithmetic test and Matrix Multiplication the pulse width distribution is skewed towards values higher than 600 ns meaning that the pulse is usually broadened when it is inside a sequential circuit.

- *D Rise Delay:* The delay associated to the input signal D of the FF when it transitions from a low to a high value.

- *D Fall Delay:* The delay associated to the input signal D of the FF when it transitions from a high to a low value.

- *CK Internal Delay:* Internal delay of the rising edge of the CK signal.

It can be seen that the values measured from the RISC-V processor present a significantly lower variance. The different values measured in Rempro, are distributed in a larger span. The longer whiskers in the boxplots for the Rempro mean also that the values that fall outside the first and third quartile are much

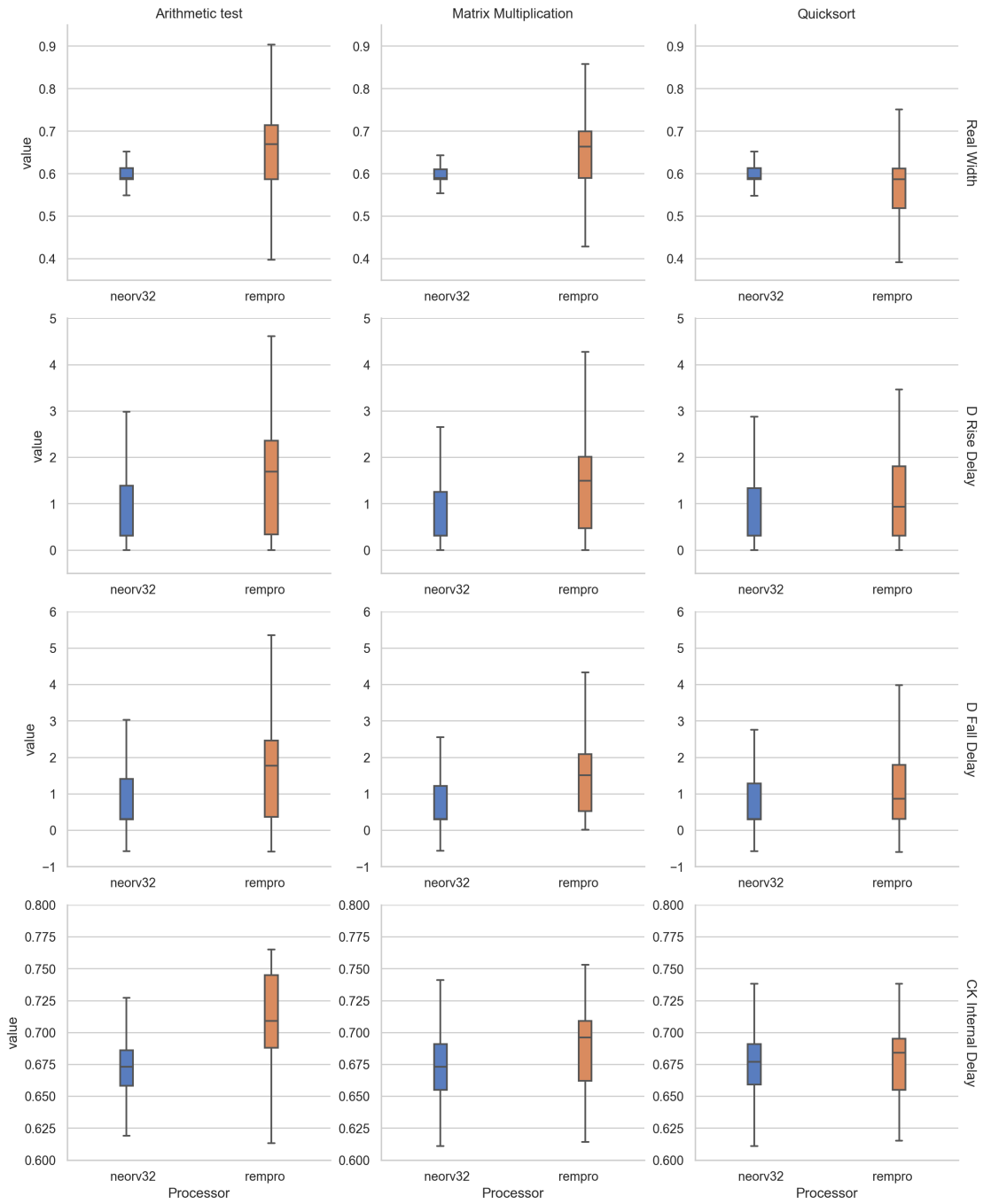



**Figure 7.2:** Comparison between the internal flip-flop delays between the NEORV32 and Rempro processors.

more sparsely distributed compared to NEORV32 where even the values that fall outside of the box remain generally closer to its edges.

## 7.4 Injection campaign with negative pulses

The first campaign was carried out using only positive pulses, injected forcing a high value on a node for 600 ns. In order to verify that the polarity of the pulse does not affect the results, a small campaign of injections using negative pulses was also conducted.

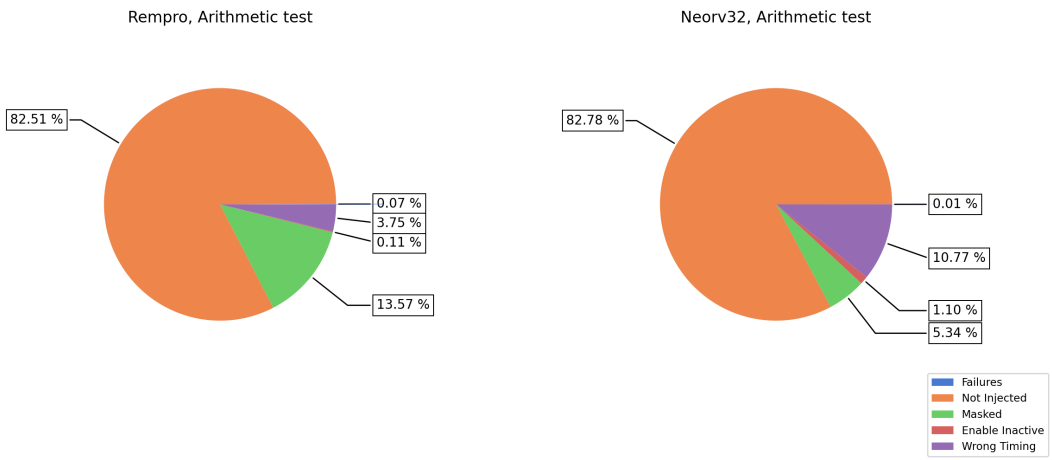



**Figure 7.3:** Comparison between the fault injection campaigns on the NEORV32 and Rempro processors.

As it can be seen from figure 7.3, the results are very different from those obtained using positive pulses. This is due to the difference in the internal fall and rise delays for input signals inside FFs. Due to the fall delay being longer than the rise delay, negative pulses, composed, that is, by a fall transition followed by a rise transition, can easily disappear inside the FF as the rise happens before the fall. For this reason most of the pulses were not injected, and the number of failures was significantly lower as fewer pulses were able to make it inside the sequential circuit and even less were able to be sampled.

Looking at the timing analysis on the other end, it can be seen that both the

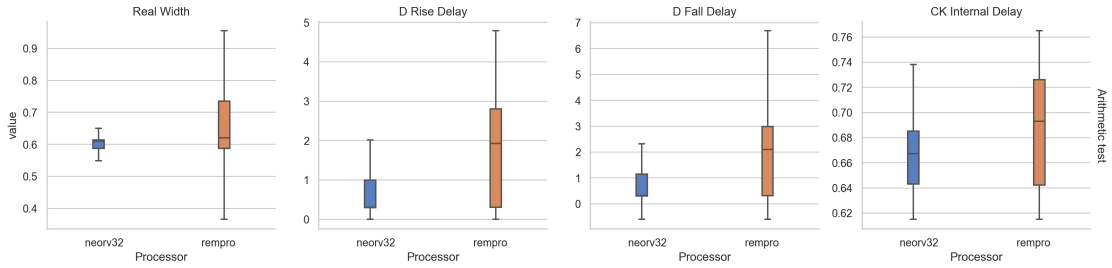



**Figure 7.4:** Comparison between the internal flip-flop delays between the NEORV32 and Rempro processors.

processors show an increased spread of their datapoints, with Rempro being more evident. This is certainly due to the fact that a large number of injections resulted in no pulse being injected, meaning that no data could be gathered on the timing of rise and fall, resulting in fewer datapoints representing only a subset of injection cases where the rise and fall delay allowed the pulse to be propagated inside the sequential logic.

While these results differ from the one with positive pulses, it is clear that due to the characteristics of the technology employed, both designs are much more resistant to negative pulses, and therefore focusing only on the positive pulses did not introduce a bias in the study, since the effect of the negative pulses was negligible.

# Chapter 8

# Conclusions

In the presented research work a new soft-core to be implemented on an FPGA for space use was presented and tested. Its architecture was studied and an implementation of a fault mitigation technique was proposed to make the design more resistant to radiation. This technique, known as TMR, was implemented both at the core and device level.

A new architecture was proposed in which the system is imagined in its entirety as a number of boards each mounting an FPGA with a TMR version of the processor in it. Each single system is equipped with a self-checking mechanism able to detect the occurrence of an unrecoverable fault. When the system cannot recover from the malfunction a substitution routine activates a new board that is able to autoconfigure itself to carry out the program first started in the defective board. In this way it is possible to obtain a system capable of remaining online as long as possible, executing critical parts of the mission, with a high degree of reliability, without the need for any kind of servicing.

A physical bus and a memory recovery routine was developed to allow the processors in different boards to communicate and transfer the program state. The features were tested and the results confirm the feasibility of the design. A UART module was realized to communicate with a host PC for the test.

Once it was confirmed that the system would behave correctly, a fault injection campaign was planned and executed to obtain a baseline for the fault tolerance of the base processor before the introduction of the fault mitigation improvements.

The same stimuli were applied to an equivalent RISC-V processor, the NEORV32 in order to have a comparison with a system using the popular ISA. The results of these campaigns were analyzed and presented showing that while the Rempro processor has a lower power consumption and area utilization compared to the NEORV32, its fault tolerance is slightly worse in its base version. However, the lower complexity means that the overhead added when implementing the fault mitigation techniques will be lower and more efficient.

## 8.1 Future Work

Realizing a processor for use in space is a complex task that requires many different steps to be conducted. The development of the Rempro processor is still in embryonal phase. A lot of work still has to be performed to prepare Rempro to the challenges of the space environment and to adhere the standard set by the space agencies. So far the processor was subjected to a preliminary evaluation in terms of fault tolerance and compared to an equivalent RISC-V processor. Fault mitigation in the form of TMR needs to be implemented and tested in a board and be subjected to fault injection campaigns to evaluate the improvements in the fault coverage and radiation tolerance. Future work should focus on studying and implementing such changes, analyzing the benefits and drawbacks and verifying it meets the requirements set by the aerospace industry for adoption in space missions.

# Bibliography


[1] Corrado De Sio, Sarah Azimi, Andrea Portaluri, and Luca Sterpone. «SEU Evaluation of Hardened-by-Replication Software in RISC- V Soft Processor». In: *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. Oct. 2021, pp. 1–6. DOI: 10.1109/DFT52944.2021.9568342 (cit. on p. 2).

[2] M.S. Gorbunov. «Design of fault-tolerant microprocessors for space applications». In: *Acta Astronautica* 163 (2019). Space Flight Safety-2018, pp. 252–258. ISSN: 0094-5765. DOI: https://doi.org/10.1016/j.actaastro.2019.04.029. URL: https://www.sciencedirect.com/science/article/pii/S0094576518321799 (cit. on p. 3).

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. «Basic concepts and taxonomy of dependable and secure computing». In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1941-0018. DOI: 10.1109/TDSC.2004.2 (cit. on pp. 3, 4).

[4] Stefano Di Mascio, Alessandra Menicucci, Eberhard Gill, Gianluca Furano, and Claudio Monteleone. «Open-source IP cores for space: A processor-level perspective on soft errors in the RISC-V era». In: *Computer Science Review* 39 (2021), p. 100349. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2020.100349. URL: https://www.sciencedirect.com/science/article/pii/S1574013720304494 (cit. on p. 4).

[5] Luca Sterpone and Sarah Azimi. «Effective Characterization of Radiation-induced SET on Flash-based FPGAs». In: *2017 17th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. Oct. 2017, pp. 1–4. DOI: 10.1109/RADECS.2017.8696255 (cit. on p. 5).

[6] R. Baumann. «Soft errors in advanced computer systems». In: *IEEE Design & Test of Computers* 22.3 (May 2005), pp. 258–266. ISSN: 1558-1918. DOI: 10.1109/MDT.2005.69 (cit. on p. 5).

[7] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. «Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design». In: *SIGPLAN Not.* 43.3 (Mar. 2008), pp. 265–276. ISSN: 0362-1340. DOI: 10.1145/1353536.1346315. URL: https://doi.org/10.1145/1353536.1346315 (cit. on p. 5).

[8] L. Sterpone and M. Violante. «A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs». In: *IEEE Transactions on Nuclear Science* 54.4 (Aug. 2007), pp. 965–970. ISSN: 1558-1578. DOI: 10.1109/TNS.2007.904080 (cit. on p. 5).

[9] Jonathan A. Pellish et al. «Impact of Spacecraft Shielding on Direct Ionization Soft Error Rates for Sub-130 nm Technologies». In: *IEEE Transactions on Nuclear Science* 57.6 (Dec. 2010), pp. 3183–3189. ISSN: 1558-1578. DOI: 10.1109/TNS.2010.2084595 (cit. on p. 6).

[10] David L Weaver. *OpenSPARC internals: OpenSPARC T1/T2 CMT throughput computing*. Sun Microsystems, 2008 (cit. on p. 6).

[11] Anand Dixit and Alan Wood. «The impact of new technology on soft error rates». In: *2011 International Reliability Physics Symposium*. Apr. 2011, 5B.4.1–5B.4.7. DOI: 10.1109/IRPS.2011.5784522 (cit. on p. 6).

[12] Mojtaba Ebrahimi, Adrian Evans, Mehdi B. Tahoori, Enrico Costenaro, Dan Alexandrescu, Vikas Chandra, and Razi Seyyedi. «Comprehensive Analysis of Sequential and Combinational Soft Errors in an Embedded Processor». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (Oct. 2015), pp. 1586–1599. ISSN: 1937-4151. DOI: 10.1109/TCAD.2015.2422845 (cit. on p. 6).

[13] Luca Sterpone and Sarah Azimi. «Radiation-induced SET on Flash-based FPGAs: Analysis and Filtering Methods». In: *ARCS 2017; 30th International Conference on Architecture of Computing Systems*. Apr. 2017, pp. 1–6 (cit. on p. 6).

[14] R.C. Baumann. «Radiation-induced soft errors in advanced semiconductor technologies». In: *IEEE Transactions on Device and Materials Reliability* 5.3 (Sept. 2005), pp. 305–316. ISSN: 1558-2574. DOI: 10.1109/TDMR.2005.853449 (cit. on p. 7).

[15] N. N. Mahatme, S. Jagannathan, T. D. Loveless, L. W. Massengill, B. L. Bhuva, S.-J. Wen, and R. Wong. «Comparison of Combinational and Sequential Error Rates for a Deep Submicron Process». In: *IEEE Transactions on Nuclear Science* 58.6 (Dec. 2011), pp. 2719–2725. ISSN: 1558-1578. DOI: 10.1109/TNS.2011.2171993 (cit. on p. 7).

[16] Elena Dubrova. *Fault-tolerant design*. New York: Springer New York, 2013. ISBN: 9781461421122 (cit. on p. 7).

[17] Karl Janson, Carl Johann Treudler, Thomas Hollstein, Jaan Raik, Maksim Jenihhin, and Goerschwin Fey. «Software-Level TMR Approach for On-Board Data Processing in Space Applications». In: *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. Apr. 2018, pp. 147–152. DOI: 10.1109/DDECS.2018.00033 (cit. on p. 8).

[18] Óscar Ruano, Francisco García-Herrero, Luis Alberto Aranda, Alfonso Sánchez-Macián, Laura Rodriguez, and Juan Antonio Maestro. «Fault Injection Emulation for Systems in FPGAs: Tools, Techniques and Methodology, a Tutorial». In: *Sensors* 21.4 (2021). ISSN: 1424-8220. DOI: 10.3390/s21041392. URL: https://www.mdpi.com/1424-8220/21/4/1392 (cit. on p. 10).

[19] Haissam Ziade, Rafic Ayoubi, and R. Velazco. «A Survey on Fault Injection Techniques». In: *Int. Arab J. Inf. Technol.* 1.2 (Jan. 2004), pp. 171–186 (cit. on p. 9).

[20] *ProASIC3/E Production FPGAs*. White Paper. Microchip (Actel), Jan. 2005. URL: https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/SupportingCollateral/pa3_e_tech_wp.pdf (cit. on p. 13).

[21] H.R. Schwartz, D.K. Nichols, and A.H. Johnston. «Single-event upset in flash memories». In: *Nuclear Science, IEEE Transactions on* 44 (Jan. 1998), pp. 2315–2324. DOI: 10.1109/23.659053 (cit. on p. 13).

[22] Cong Peng, Wei Chen, Yinhong Luo, Fengqi Zhang, Xiaobin Tang, Zibo Wang, Lili Ding, and Xiaoqiang Guo. «Low-energy proton-induced single event effect in NAND flash memories». In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 969 (2020), p. 164064. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2020.164064. URL: https://www.sciencedirect.com/science/article/pii/S0168900220304952 (cit. on p. 13).

[23] *ProASIC3 Flash Family FPGAs*. Datasheet. Microchip (Actel), Mar. 2016. URL: https://www.microsemi.com/document-portal/doc_view/130704-ds0097-proasic3-flash-family-fpgas-datasheet (cit. on p. 15).

[24] *ProASIC3/E Proto Kit User's Guide and Tutorial*. Microchip (Actel). 2012. URL: https://www.microsemi.com/document-portal/doc_view/130776-proasic3-e-proto-kit-user-s-guide-and-tutorial (cit. on p. 15).

[25] Steven Malone, Patrick Saenz, and Patrick Phelan. «RISC-V Processors for Spaceflight Embedded Platforms». In: *2023 IEEE Aerospace Conference*. Mar. 2023, pp. 1–11. DOI: 10.1109/AERO55745.2023.10115850 (cit. on p. 51).

[26] RISC-V International. *History of RISC-V*. 2023. URL: https://riscv.org/about/history/ (visited on 09/07/2023) (cit. on p. 51).

[27] Krste Asanović and David A Patterson. «Instruction sets should be free: The case for risc-v». In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014) (cit. on p. 51).

[28] Tony Chen and David A Patterson. «Risc-v geneology». In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-6* (2016) (cit. on p. 52).

[29] Stephan Nolting. *The NEORV32 RISC-V Processor: Datasheet*. 2023. DOI: 10.5281/zenodo.8087446. URL: https://stnolting.github.io/neorv32/ (visited on 07/15/2023) (cit. on p. 52).

[30] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California,

Berkeley, May 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html (cit. on pp. 55–57).

[31] *ModelSim User's Manual*. Mentor Graphics Corporation. 2012. URL: https://www.microsemi.com/document-portal/doc_view/131619-modelsim-user (cit. on p. 59).