Politecnico di Torino

Master Degree Dissertation
Master Degree in Mechatronic Engineering

Study and Development of RISC-V Mitigation Approach Based on Redundancy

By

Lucio Milanesi

******

Supervisor:
Prof. Sterpone Luca, Supervisor
Prof. Corrado De Sio, Co-Supervisor

Politecnico di Torino
2023

A mio padre che, nonostante io avessi solo 4 anni, già mi parlava di onde elettromagnetiche

Declaration

I hereby declare that the contents and organization of this dissertation constitute my own original work and do not compromise in any way the rights of third parties, including those relating to the security of personal data.

Lucio Milanesi
2023

Abstract

Field Programmable Gate Arrays (FPGAs) are programmable devices that can be adapted to different, specific applications through thorough hardware programming. Since their first appearance in 1985, they revolutionized the world of technology development, especially regarding electronics.
Their memory and configuration-RAM, however, are not immune to Single Event Upsets (SEUs), radiation-induced bit-flips which can corrupt stored data and/or device behaviour. Since their early discovery in 1954, SEUs have been considered responsible for many different failures.
In order to mitigate the radiations effects, whose consequences include but are not limited to SEUs, hardening techniques have to be implemented. For this project, the redundancy technique was chosen given its satisfying cost-benefit ratio. The Triple Modular Redundancy (TMR) method makes use of three identical modules, whose outputs are sent to a majority voter to identify the one obtained 2 out of 3 times (most likely correct).
The main goal of this thesis, in collaboration with the European Space Agency (ESA), was to compare the SEU-induced processor error probability before and after TMR implementation, possibly improving it. Given the space application of this work, a flexible, secure and long-lasting hardware was mandatory: that is why the GitHub open-source, RISC-V ISA based, VHDL-described NEORV32 processor was chosen.
By default it was composed of a single-core CPU, IMEM and DMEM, caches and different optional modules with a standard clock frequency of 100 MHz. To obtain an error rate profile, an intensive fault analysis has been performed, corrupting bitstreams (sequence of bits needed to "shape" the FPGA and obtain the desired hardware and behaviour) with an increasing number N of SEUs, simulated through simple xor operations targeting random bits in the sensitive parts of the bitstream.

Through a Python script, 10000 tests have been executed for each amount of simulated SEUs, starting from 0 and increasing by 10 up until 200. The single test consisted of a simple C two 3x3 integer matrix multiplication program, hard-coded in the bitstream and thus automatically executed after programming.
The result would then be compared to the stored, correct one and, in case of errors of any kind, an error count would be incremented. After 10000 tests the error probability for that amount of simulated SEUs would be calculated and the whole process would be repeated with the new amount, until eventually obtaining the processor error rate profile.
The default, single-core NEORV32 processor showed little resistance towards SEUs, reaching over 50% error probability with just 30 SEUs. It also revealed a saturating trend towards 95% after 100 SEUs, making it unserviceable after such threshold.
After triplicating the core and adding a TMR module, the results improved significantly. Even if the low memory resources of the Xilinx PYNQ Z2 (the board used for this project) did not allow the creation of separate IMEMs and DMEMs for each core, the achieved performance was surprising.
Along with an up to 76% reduction of the error probability, the triple-core version of the NEORV32 processor showed a marked improvement when it came to SEU resistance. In the considered interval (i.e. the single-core version confidence interval, e.g. 0-100 SEUs) the average error probability reduction was slightly over 50%. In other words, the processor reliability was more than doubled, leading to a way more than doubled working area and a saturation trend which tended to about 85% (10% less).
The results were fully satisfactory and proved how TMR can be a useful hardening technique, causing a negligible increase of FPGA area consumption but granting much better performances. In the future, implementing NEORV32 processor on a more powerful FPGA could be considered: it would allow allocating separate memory resources (IMEM and DMEM) for each core, preventing the risk of common errors due to shared data corruption.

Acronyms

ALU Arithmetic-Logic Unit.

ARM Advanced RISC Machine.

ASRAM Asymmetric Static Random Access Memory.

CISC Complex Instruction Set Computer.

CLB Configurable Logic Block.

CPLD Complex Programmable Logic Device.

CPU Central Processing Unit.

CRAM Configuration Random Access Memory.

CRC Cyclic Redundancy Check.

CU Control Unit.

DDR SDRAM Double Data Rate Synchronous Dynamic Random Access Memory.

DMEM Data Memory.

DRAM Dynamic Random Access Memory.

ECC Error Correction Code.

ECC DRAM Error Correcting Code Dynamic Random Access Memory.

ECR Error Correction with Remap.

EDR Error Detection with Remap.

EEPROM Electrically Erasable Programmable Read Only Memory.

EPROM Erasable Programmable Read Only Memory.

ESA European Space Agency.

FPGA Field Programmable Gate Array.

GPU Graphic Processing Unit.

IMEM Instruction Memory.

ISA Instruction Set Architecture.

LUT Look-Up Table.

MMU Memory Management Unit.

MPU Memory Protection Unit.

MROM Masked Read Only Memory.

PAR Place and Route.

PIP Programmable Interconnection Point.

PR Partial Reconfiguration.

PROM Programmable Read Only Memory.

RAM Random Access Memory.

RISC Reduced Instruction Set Computer.

ROM Read-Only Memory.

RTL Register Transfer Level.

SDRAM Synchronous Dynamic Random Access Memory.

SEE Single Event Effect.

SET Single Event Transient.

SEU Single Event Upset.

SOC System on Chip.

SPLD Simple Programmable Logic Device.

SRAM Static Random Access Memory.

TMR Triple Modular Redundancy.

UART Universal Asynchronous Receiver-Transmitter.

VHDL VHSIC Hardware Description Language.

VHSIC Very High Speed Integrated Circuit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep within the soul of any man in the world, lies the fear of the Unknown. Analysis and tests, rules and procedures, even technologies have been created to simulate a logic path, a precise consequence that derives from precise actions, and penalties for when the established paths are bypassed. Especially in electronics, where the hierarchical pyramid is clear and everything behaves according to predefined stats and requirements, the presence of the Unknown must be accounted for.
It acts in the form of radiations (mostly solar) that hit device cells programmed by an apposite binary code, causing a bit-flip. Their effect can be approximated to a switch button, turning on (1) cells which were supposed to stay inactive (0) and vice-versa. They may, through the accumulation of charge, even change transistors internal voltages leading to potentially corruptive or destructive behaviour.
Since their first discovery in 1954, Single Event Upsets (SEUs) have been responsible for many failures ranging from Aerospace to Electoral sectors. Proof of their existence has been found in Implantable Cardioverter Defibrillators (eg pacemakers) as well [2], and thus it is of the utmost importance to minimize their effects.
The fields which are most subject to them are those where radiations grow in number and intensity, leading to the Spatial Exploration being the main candidate. This project applications, though, are not limited to this sector as vehicles and standard electronic devices are all subject to radiations and solar rays.
In collaboration with the European Space Agency, this work aimed to reduce the Single Event Effects (SEEs) caused by SEUs. Such goal was meant to be achieved

through the implementation of a Triple Modular Redundancy check, and address the question of whether it was a worthy hardening technique or not.

Chapter 2

Background

2.1 Field Programmable Gate Arrays

Field Programmable Gate Array are semiconductor devices that are built around a matrix of customizable logic blocks (CLBs) coupled via programmable interconnects. After being manufactured, FPGAs can be reprogrammed to meet specific application or feature needs. Although one-time programmable (OTP) FPGAs are available, the most common models are SRAM-based and may be reprogrammed as the design changes. This capability sets FPGAs apart from Central Processing Units (CPUs), Graphics Processing Units (GPUs) and Application Specific Integrated Circuits (ASICs): each one represents a different type of computer processors, and every one has its most suitable applications.

CPUs, despite being extremely adaptable, feature an immutable underlying hardware. Once a CPU's circuitry has been manufactured, it cannot be altered. It relies on software instructions regarding which specific operation (arithmetic function) to execute on which memory data. The hardware must be able to perform all conceivable operations, which are summoned by software commands and are typically executed one at a time. FPGAs, on the other hand, can simultaneously process enormous quantities of data. The benefit of adaptive hardware over CPUs varies by application, largely depending on the nature of the computation and its ability to run in parallel. With suitable applications, a 20X performance

improvement (when compared to CPUs standards) can be easily obtained.

GPUs rectify a significant flaw of CPUs – the inability to process a large quantity of data in parallel – and are capable of operating on very large data sets. GPUs and CPUs similarities lie in their fixed hardware and reliance on software commands to execute operations. A single instruction is able to process over a thousand blocks of data, making them suitable for domains such as graphical acceleration, high performance computation, video processing, and various types of machine learning, among others. However, a GPU's fundamental architecture and data transmission are fixed prior to production.

Application-Specific Integrated Circuits (ASICs) are designed with the goal of optimizing the execution of a predetermined task, providing them a performance and speed advantage over general processors (for that specific task). This characteristic makes them an appealing option for operations requiring extreme computing power, such as intensive data mining or cryptocurrency mining. However, given their design strictly oriented towards specific functions, their ability to perform other tasks outside the intended one is highly limited. In contrast to a general-purpose processor, which can be repurposed for a variety of activities, an ASIC cannot be reconfigured because its circuitry was designed to perform only one type of job.

When compared to all the various solutions listed above, FPGAs have - in terms of flexibility - a distinct advantage over every single one of them. FPGAs can be reprogrammed to perform multiple duties, allowing the same hardware to be utilized across multiple applications. This reduces expenses for businesses that must develop multiple solutions with comparable underlying requirements, and makes them optimal for applications where performance and configurability are crucial.
Today they are chosen for different sectors, including aerospace engineering, defense, artificial intelligence (AI), industrial IoT (Internet of Things), wired and wireless networking, and automotive development. In essence, FPGA devices are frequently found in environments where consumers require real-time data. FPGAs also aid in the acceleration of functions that would ordinarily be performed

by software. This makes them a useful instrument for outsourcing performance-intensive tasks, such as artificial intelligence deep neural network (DNN) inference.
FPGA programming is based on Hardware Description Languages (HDLs) to shape circuits and actually build the desired hardware. Hardware programming differs from software one - among the other things - since it is not executed sequentially but everything happens simultaneously. The output is a binary file (i.e. a bitstream, a sequence of bits) that, once loaded onto an FPGA device, assembles lower-level elements, such as logic gates and memory units, according to the specific application. Memory and power usage constraints can be specified - in the programming tool - to enhance customization and satisfy requirements.
A more detailed explanation of the FPGA development flow is presented in section 2.4.

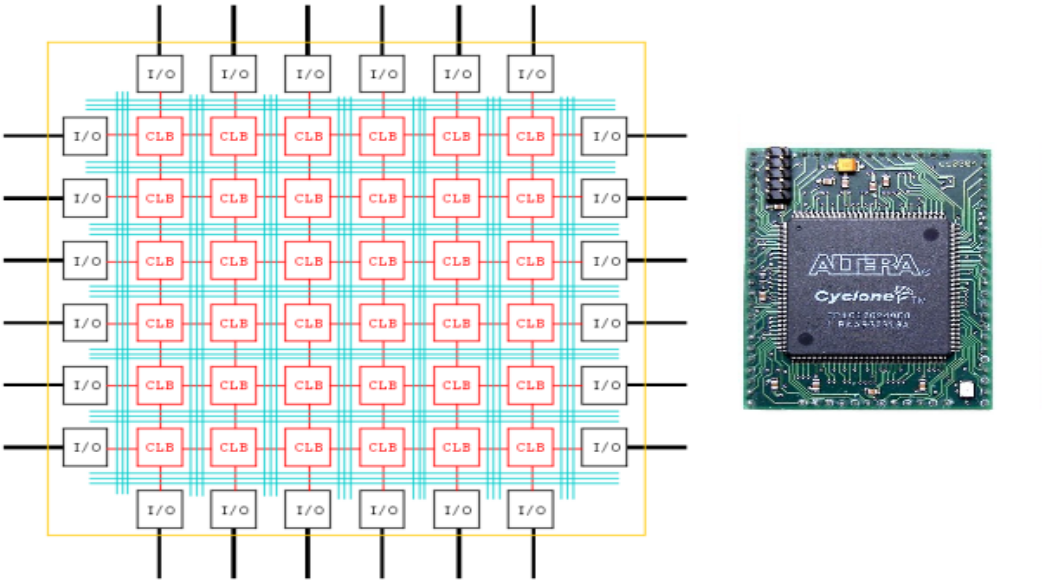

Fig. 2.1 FPGA schematics

2.2 History of FPGAs

The first Field Programmable Gate Arrays were released in the early 1980s, but it wasn't until the late 1990s that they really took off. The first businesses to invest

in this emerging technology were Altera and Xilinx, with the first commercially accessible FPGAs being the EP300 and XC2064, respectively. The world began to consider them as an alternative to ASICs, which had been employed to create systems that were too complicated for regular CPUs or GPUs, a little more than ten years after they were first developed.
Field Programmable Gate Arrays still constitute a wise choice even by today's standards since they're cheaper and consume less power than its competitor technologies. Their applications include the aerospace and medical industries, as well as image processing, automobiles, and communications.

2.3 AMD Xilinx PYNQ Z2: the FPGA used for this work

Xilinx, Inc. was one among the first American technology and semiconductor firms, specialized in the production of programmable logic devices. The company is renowned for developing the first fabless manufacturing approach and the first commercially successful field-programmable gate array (FPGA). Ross Freeman, Bernard Vonderschmitt, and James V. Barnett II co-founded Xilinx in Silicon Valley in 1984; its headquarters are in San Jose, California. Today it produces FPGAs, CPLDs, SPLDs, design tools and reference designs and its customers make over 51% of the whole programmable logic market.
The Zynq-7000 family, announced in March 2011, stands out among the many product lines that Xilinx has to offer. It incorporates a full ARM Cortex-A9 MPCore processor-based system atop a 28 nm FPGA for system architects and embedded software developers. The Zynq-7000 series of SoCs targets high-end embedded-system applications such video surveillance, automotive driver assistance, next-generation wireless, and industrial automation.
Belonging to this family is the FPGA used for this work, the Xilinx PYNQ-Z2 (figure 2.2). A few of the Zynq-7000 SoC Features [3]:

- Dual ARM® Cortex™-A9 MPCore™ with CoreSight™
- 32 KB Instruction, 32 KB Data per processor L1 Cache
- 512 KB unified L2 Cache
- 256 KB On-Chip Memory

- 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO
- 85K logic cells (13300 logic slices, each with four 6-input LUTs and 8 flip-flops)
- 630 KB of fast block RAM
- Internal clock speeds exceeding 450MHz

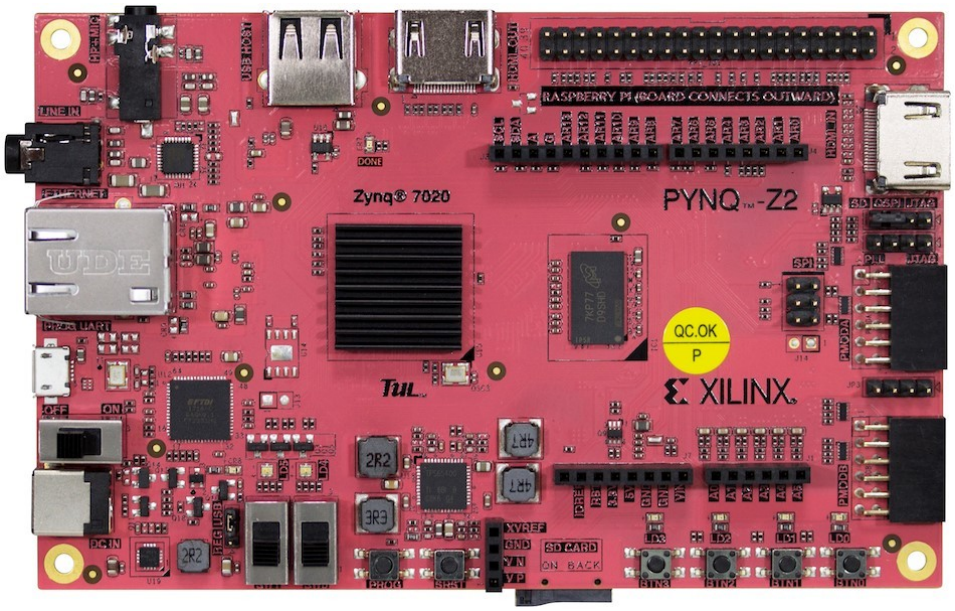

Fig. 2.2 PYNQ Z2 board

2.4 Bitstream generation

As described earlier, a bitstream is a sequence of binary values used to program CLBs inside an FPGA target. It is the result of a chain of complex operations performed by a specific software - Vivado in this case - developed by AMD Xilinx for its FPGA devices. The main steps are:

Elaboration: sometimes included in the synthesis phase, it consists of reading and understanding the different VHDL-written files, looking for elements

descriptions in the code. Its output is a technology-independent netlist, i.e. a file format that specifies the components, connectivity, and optionally the placement and routing of an electronic circuit's components.

Synthesis: starting from the independent netlist, the goal of the synthesis tool is to obtain its technology-dependent version, known as unrouted netlist. It is achieved by applying timing constraints and high & low level optimizations to the elaborated design, adapting the abstract hardware configuration to the specific FPGA device capabilities. It accurately describes FPGA-related primitives (the smallest configurable logic elements) and how they are connected, but it lacks placement information.

Implementation: also known as Place and Route (PAR), this last step aims to physically map primitives and wires to the actual target device. The result is enclosed inside the routed netlist file, ready to be translated into a .bit file.

Bitstream Generation: last step of the series, it provides a .bit file which is a sequence of logic bits needed to shape the FPGA configuration and program its behaviour.

Testing: although not a strict component of this process, it is often performed after FPGA programming to ensure that everything behaves as expected. It makes use of testbenches, separate VHDL modules whose purpose is to stimulate the achieved design and verify its behaviour and outputs.
Testbench files usually contain only local signals and inputs/outputs for the Device Under Test (DUT), without signals entering or exiting it. It may also include a copy of the expected results, in order to facilitate the comparison between the correct outputs and the produced ones.

2.5 Impact of Single Event Upsets on FPGAs

A single ionizing particle (ions, electrons, photons, etc.) impacting a sensitive node in a live micro-electronic device results in a single-event upset, or SEU (see fig 2.3). The free charge produced by ionization in or near a key node of a logic element (a memory *bit*) is what causes the state change. Single Event Effect (SEE), which refers to the fault in a device's output or operation brought on by a strike, is

temporary because reprogramming the device returns it to its original, expected behavior. It might appear unimportant and improbable to happen at first, but it was sufficient to provide a candidate in Belgium during the 2003 elections 4096 more votes, for instance.
To have a clearer view on how SEUs can influence the FPGA behaviour, it should be noted that FPGA devices use both memory in user logic (i.e. registers) and in Configuration RAM (CRAM). The latter is a memory loaded with the user's design, defining all logic and routing in the device and shaping what is known as *netlist*, namely a map of circuital connections. In SRAM-based Xilinx devices, signal routing is performed through the usage of interconnection matrices named Programmable Interconnection Points (PIPs). As a consequence, SEUs in the configuration bits of interconnection bridges may modify one PIP, potentially interfering with the signal propagation between CLBs and, on a larger scale, circuit modules [4].
Different fault effect scenarios can be identified [5]:

Open: an open error occurs when a PIP configuration (related to a specific input IN - output OUT connection) is set to an open state by a SEU strike. This prevents the IN pin to drive the OUT one, leading elements constituting that link to become dangling. What's more, the uncontrolled output may be latched to another - hopefully unused - input.

Bridge: starting from the same situation characterizing the open error, if the dangling output is connected to an input already employed in other configurations, a bridge error takes place. It has a much bigger impact, since the output is not null anymore, but instead is driven by a random, unknown input and the resulting behaviour is unpredictable.

Input Antenna: an input antenna error arises when an output, already in use and connected to its input, latches on to another unused one.

Output Antenna: opposite to the previous one, an output antenna error presents itself when an input, already connected to a predefined output, drives another - unused one - too.

Conflict: last but not least, an undesired input-output connection (where both pins are already in use) results in a conflict error, with the consequent propagation of unknown values.

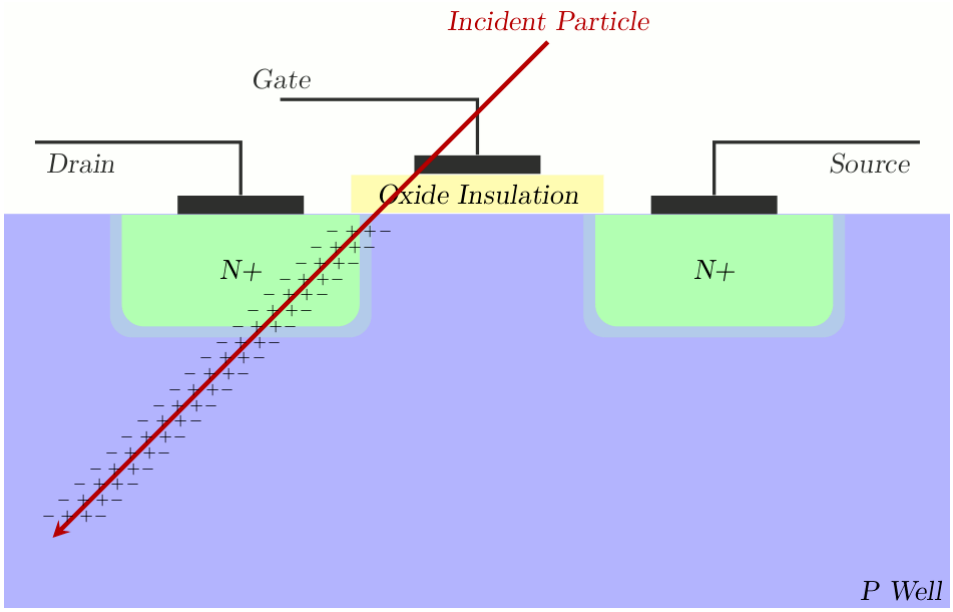

Fig. 2.3 Simple representation of a SEU

SEUs have been proven to be responsible for a number of errors since they were originally discovered in 1954 during nuclear tests, and many more are still under investigation. Devices are highly susceptible to SEU-induced failures, particularly in Aerospace and Space applications where radiations (with solar being their prime example) become stronger. The flight of an A330 Airbus on July 10, 2008, which almost resulted in a crash ostensibly because of an Event Upset, was one of the most famous.

2.6 System on Chip technology

A System-on-Chip (SoC) is an integrated circuit that integrates the majority or all of the parts of a computer or other electronic system. On a single substrate or microchip, these components frequently include an on-chip central processing unit (CPU), memory interfaces, input/output devices, input/output interfaces and secondary storage interfaces, along with other elements like radio modems and a graphics processing unit (GPU).
The typical traditional motherboard-based PC architecture, which divides components based on function and connects them via a central interface circuit board, contrasts with newline SoCs. SoCs combine all of these parts into a single inte-

grated circuit, as opposed to motherboards, which hold and connect removable or swappable components. Instead of connecting these modules as independent parts or expansion cards, a motherboard often integrates a CPU, graphics and memory interfaces, secondary storage, USB connectivity, and I/O interfaces on a single chip.
A SoC performs better and uses less power than a multi-chip architecture while also having a smaller semiconductor die size than a multi-chip design with the same functionality. This comes at the cost of reduced replaceability of components.

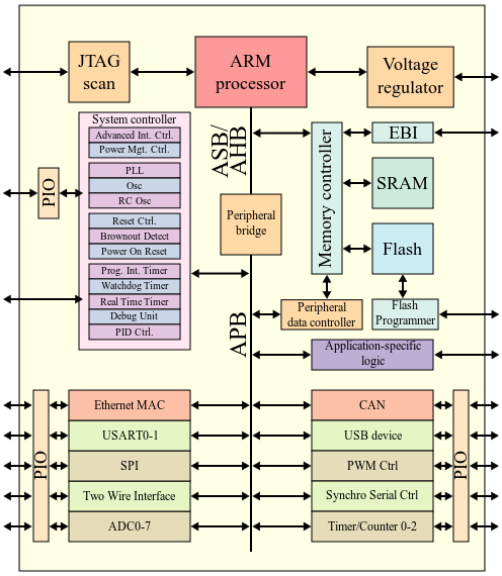

Fig. 2.4 System on a Chip schematic

2.7 NEORV32 Processor

The NEORV32 Processor is a platform-independent system on chip (SoC) that resembles a microcontroller and is designed around the NEORV32 RISC-V CPU, created by Stephan Nolting and provided for free on GitHub platform [6]. The processor is designed to serve as an auxiliary controller in bigger SoC designs or as a fully operational standalone bespoke microcontroller that can even fit into low-power & low-density FPGAs.
In order to deliver defined and predictable behavior at any time, execution safety

is given special attention. The CPU makes sure that all memory accesses are correctly acknowledged and that any erroneous or flawed instructions are consistently identified as such. The application program is alerted whenever an unexpected circumstance arises via precise and resumable hardware exceptions.

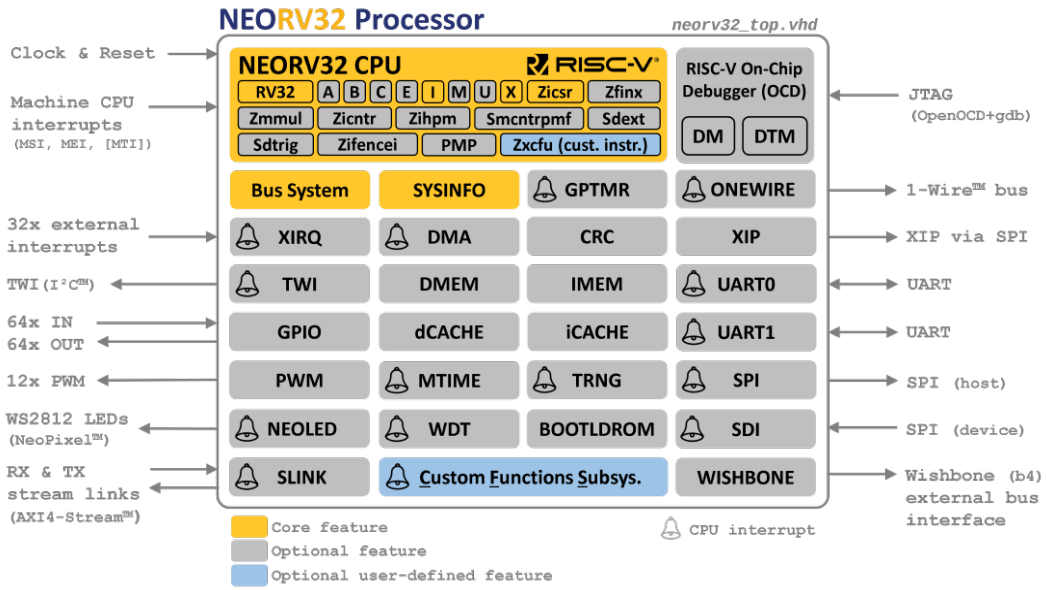

Fig. 2.5 NEORV32 Processor official schematic (source GitHub project)

Chapter 3

State of Art

3.1 RISC-V vs ARM: same goals, different strategies

RISC-V is an open standard instruction set architecture (ISA) whose foundations are laid by the reduced instruction set computer (RISC) framework. Since it is offered under royalty-free open-source licenses, it is now supported by a number of well-known software toolchains, in contrast to the majority of other ISA designs (with ARM serving as their leading example). It's important to step back and consider what a processor's properties are in order to comprehend why RISC-V was chosen (for this application) above its rivals.

3.1.1 Processor ISA Differences

The Instruction Set Architecture (ISA), a design outlining the set of instructions a processor can comprehend and implement, is fundamental to the operation of every processor. It influences the capabilities and performance of a processor by serving as a vital connection between hardware and software. The selected ISA affects software development and has a long-term impact on the productivity, interoperability, and adaptability of a CPU.
ISAs can be broadly categorized as either Open or Closed. Closed ISAs, such as ARM, are proprietary and strictly governed by specific companies (in this case, Arm Holdings), providing known stability and compatibility but limiting flexibility.

On the other hand, open ISAs, like RISC-V, are community-driven and offer more customization options, encouraging innovation and adaption to particular needs.

3.1.2 Architectural Characteristics

RISC-V structure

The RISC-V architecture is based on the RISC blueprint, which places an emphasis on a compact, straightforward, and effective instruction set (as opposed to CISC, Complex Instruction etc.). A load-store architecture, a fixed-length 32-bit instruction format, and a limited amount of general-purpose registers are some of RISC-V fundamental architectural characteristics.
Different integer instruction set extensions that specify the fundamental instruction set for various address space sizes are supported by RISC-V, including RV32I (32 bits), RV64I (64 bits), and RV128I (128 bits). Little-endian byte ordering, which places the smallest significant byte of multi-byte data at the lowest memory address, is used by RISC-V in the memory system.
Listed below are several distinguishing characteristics of RISC-V architecture:

Modularity & Extensibility: The modularity and extensibility of the RISC-V architecture is one of its defining characteristics. The ISA is designed to be easily extended with special instructions and coprocessors, enabling customized implementations that adhere to particular application needs.
This adaptability is made possible by the modular architecture of the system, which enables the base ISA to be combined with optional standard extensions like the M extension for integer multiplication and division, the A extension for atomic operations, and the F and D extensions for single- and double-precision floating-point arithmetic.

Compressed instruction set: In contrast to ARM's Thumb instruction set, RISC-V additionally supports RV32C (or RV64C for 64-bit), a compressed instruction set extension that offers 16-bit compressed instructions that can be combined with the usual 32-bit instructions.
This feature makes RISC-V especially suitable for embedded systems and low-power applications by reducing code size and increasing energy efficiency.

Privilege levels & Virtual Memory: The support for virtual memory and privilege levels in RISC-V's architecture is a crucial component. Machine mode (M-mode), Supervisor mode (S-mode), and User mode (U-mode) are the three privilege levels that are specified in the RISC-V Privileged Architecture Specification.
These privilege levels offer a way to isolate the user programs, hypervisors, and operating system kernel, assuring system security and stability. A virtual memory system based on a multi-level page table structure is also supported by RISC-V, allowing for effective memory management and security.

ARM structure

The RISC foundation serves as the foundation for ARM architecture, which similarly emphasizes simplicity and power economy. The load-store architecture, a combination of fixed-length 32-bit and variable-length Thumb instructions, and a significant amount of general-purpose registers are among ARM's core architectural features. Bi-endian byte-ordering is used in the memory system, allowing an ARM processor or device to handle and transfer data in both endian forms without any issues at the hardware level.
Each family of ARM processors is designed to meet a particular set of performance and power criteria. The most common ARM processor families are the Cortex-A, Cortex-R, and Cortex-M series. High-performance applications like those found in servers, tablets, and smartphones are catered to by the Cortex-A family. These processors support cutting-edge functions including hardware virtualization, superscalar pipelines, and out-of-order execution. The Cortex-R family provides quick interrupt response times and deterministic behavior, making it ideal for real-time systems. These processors are frequently utilized in applications that are safety-critical, industrial, and automotive. The Cortex-M family is designed specifically for microcontrollers and low-power gadgets, with an emphasis on usability and energy efficiency.
The ARM architecture has the following special characteristics:

Thumb Instruction Set: The Thumb instruction set, which offers 16-bit compressed instructions for better code density and energy economy, is commonly implemented by ARM processors. As an optional 16-bit addition to

the standard 32-bit ARM instructions, ARM developed the Thumb instruction set.
The ability to reduce code size while retaining acceptable performance makes this feature appropriate for memory-constrained devices like embedded systems.

Memory Management & Protection: Memory management and protection are supported at several levels by ARM processors, including a Memory Protection Unit (MPU) for straightforward systems and a Memory Management Unit (MMU) for more intricate systems with support for virtual memory.
The 64-bit address space support and the AArch64 execution state, which offers a new 64-bit instruction set in addition to the current 32-bit ARM and Thumb instruction sets, were introduced with the ARMv8-A architecture, which was released in 2011.

Optional Enhancements: The NEON SIMD (Single Instruction, Multiple Data) extension for multimedia and signal processing workloads and the Cryptography extension for hardware-accelerated encryption and decryption are two optional extensions that ARM processors may contain in addition to the base ISA.
With the help of these additions, ARM processors can effectively handle a variety of workloads while still consuming little power and taking up little silicon space.

3.1.3 Licensing policies

This last comparison is focused on the licensing and business model of RISC-V and ARM, illuminating how these strategies affect the creation, acceptance, and customization of processors.

RISC-V :

Open-Source Licensing: Permissive open-source and royalty-free licenses, like the Apache License 2.0, are used to operate RISC-V. Developers can access, study, alter, and distribute the architecture without restriction, which promotes openness, cooperation, and innovation.

Flexibility: Organizations are able to modify the processor design to suit their own requirements because of RISC-V's open-source nature. Extensions and configurations allow for customization, making it possible to create processors that are optimized for a variety of applications.

Reduced Costs: The absence of licensing fees is one of the main benefits of RISC-V. This can greatly reduce the costs related to RISC-V processor adoption and product development.

Ownership Control: RISC-V users have complete control over their processor designs, which lessens reliance on a single vendor. This ownership control may be especially helpful to businesses looking to safeguard their intellectual property.

ARM :

Licensing Tiers: Depending on the licensing level, ARM offers a number of licensing tiers that grant access to different instruction sets and architectures. Companies can select the level of access that best suits their needs using this tiered concept.

Proprietary Elements: While ARM offers openness through its architecture, some cutting-edge features or technologies may be proprietary and call for licensing agreements. This combination of openness and proprietary components enables ARM to balance customisation and the protection of important innovations.

Licensing Fees: ARM's licensing approach frequently entails licensing costs depending on the degree of usage. These fees affect the overall cost structure for businesses using ARM processors and contribute to ARM's revenue strategy.

Vendor Relationship: ARM or its licensees are frequently partners in the adoption of ARM CPUs. Businesses may work closely with ARM to gain access to deluxe features, support, and customizations.

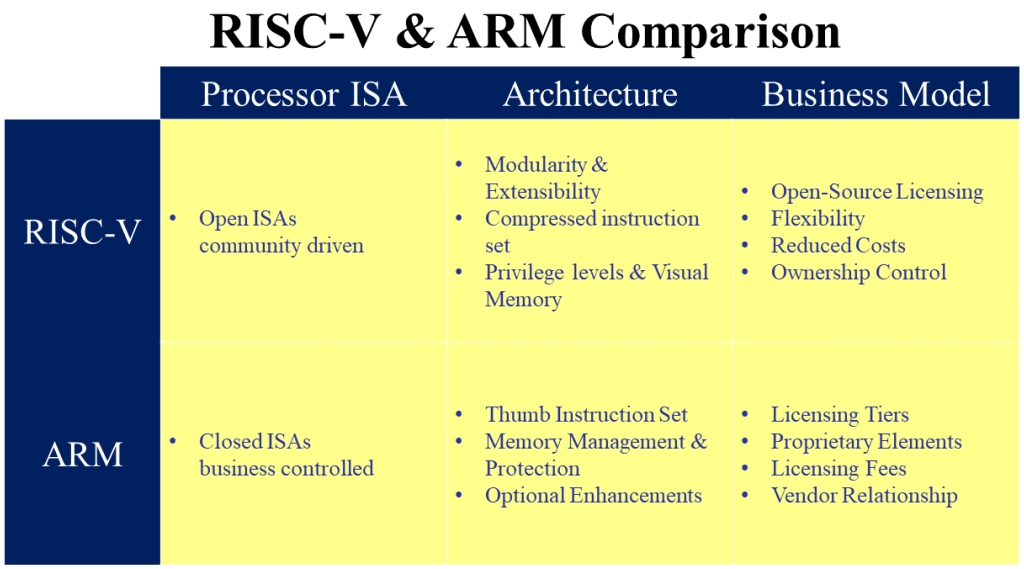

Fig. 3.1 A quick look at RISC-V and ARM differences

3.1.4 Advantages of adopting RISC-V solutions

As introduced, this work aimed to lower the SEU-induced error rate of a processor implemented on FPGA. With ESA being the engine company, the main application of such processor relies in the Aerospace sector. Why they would choose RISC-V over ARM and other competitors is understandable considering the following advantages:

- Trusted components are necessary for aerospace applications, such as the one taken into account for this work. The open ISA of RISC-V may be more advantageous for verification than closed architectures like ARM. Due to RISC-V's openness, businesses are free to choose whether to make their RISC-V IP cores' whole register-transfer level (RTL) source code available to designers without worrying about being subject to ISA licensing and protected IP restrictions. The RTL can then be thoroughly examined by designers, confirming there was no malevolent intent and building trust.
- Platforms used in aerospace and military frequently have service lives measured in decades rather than in months or years. Due to the underlying RISC-V specs and instruction set being frozen, RISC-V offers an architec-

ture that is well suited for extended service life. Designers may expect the platform they are basing their work on to be stable for a very long time. Future RISC-V chips will continue to support software developed for current RISC-V technology. This implies that code can be tested only once and then utilized on RISC-V processor generations to come.
This long-term stability is generally absent from closed, commercial ISAs. Since they were not created with extensibility in mind, software modifications that add new features or functions frequently need to be rebuilt from the ground up in order to stay current.

- Given the high cost of constructing new platforms from scratch, aerospace platforms frequently have lengthy service lifetimes. The capabilities of current platforms are frequently upgraded by incorporating technological advancements in subsequent models, even when major redesigns are expensive. For these kinds of situations, RISC-V provides designers with a base upon which to develop.
  Along with stability, RISC-V is also built to be extensible. The initial instruction set is fixed, but additions that add new features can be made without impairing the operation of software that is already in place. Applications can be catered for by creating unique accelerators that combine the standard instruction set with particular extensions.
  Developers are permitted to alter chip designs however they see fit thanks to the open RISC-V ISA license. For greater efficiency, processor microarchitectures can be modified and changed. Hardware can be created to achieve particular objectives like high performance, secure data processing, or minimal power usage.

- Intellectual property laws protect commercial ISAs. Any developer interested in creating a new chip based on the ARM architecture would normally be required to pay both upfront costs and per-chip royalties in order to do so. Contrarily, the RISC-V ISA is open source, making it available for usage as the foundational architecture for processor designs by anyone.
  However, RISC-V-based hardware designs are not required to be open source. Developed designs have the option to be kept private as exclusive intellectual property, and eventually be sold by their authors or generate profit for them in anyway they feel to be appropriate.

3.2 Bitstreams and hardening techniques

In order to program the PYNQ board, bitstreams generated by the AMD Xilinx Vivado Suite have been used (see section 2.4). A bistream is a sequence of bits (binary values) which is sent to a target FPGA device to configure its CLBs according to desired applications, leading to a custom implemented hardware.
One or more bits can be flipped as a consequence of SEUs, resulting in unpredictable and/or undesired behaviour. Such dramatic epilogue is not unavoidable however, as many different techniques have been developed to prevent SEUs from turning into SEEs. This procedure is known as *Hardening*, and can be performed both on hardware and software components. Along with preventing errors from occurring, different methodologies have also been presented in order to detect whether SEUs took place and, if they did, restore the system state to the expected one.
These techniques vary in complexity and reliability, meaning each method has its limitations and drawbacks, and could only protect the device or reveal faults up to an extent. The current state of art allows to define different approaches, ranging from simple operation-checks up to core duplication:

3.2.1 Partial Reconfiguration (PR) with Error Correction Code (ECC)

Readback, which is the process of reading from post-configuration memory, is a method for detecting static bitstream disruptions [7]. It offers a non-intrusive way to read the status of each flip-flop and configuration memory cell inside the FPGA. In order to ensure the integrity of the bitstream during configuration, the majority of contemporary FPGAs have a cyclic redundancy check (CRC) register that uses a conventional 32-bit CRC checksum technique. The data blocks or frames are given a CRC checksum value, which is then placed into the bitstream. In order to update the CRC, a CRC checker reads back the frames or blocks. The predicted checksum for the current configuration can be compared to this CRC checksum; if they do not match, a SEU may have happened. [1, 8, 9]
The mitigation method suggested in [10] uses a small auxiliary FPGA for error checking and recovery together with reserving a few rows of FPGA for checksum storage. The frames are repeatedly read back from the primary FPGA by the auxil-

iary FPGA, and the CRC checksums of the frames are recalculated and compared to the primary checksum as part of the error checking process. Since the system state and ECC are both kept in the auxiliary memory, the system must be returned to its most recent correct state if any single bit error is discovered by the ECC. This recovery process is known as rollback recovery.
The faults can be found by readback and CRC checksum; these errors can be fixed by a procedure called PR, which is a post configuration write to the configuration memory [11]. The literature has proposed methods for partially reconfiguring a corrupted module of a triple modular redundant (TMR) implementation, which tackles the speed penalty associated with such methods and offers a generalized method for reducing it [12]. Mission-specific flexibility on demand is made possible by dynamic PR, which gives the FPGA device the ability to be reconfigured while other operations are still running on the remainder of the device.

3.2.2 Scrubbing

Scrubbing is a method for restoring the initial state via a periodic post-configuration configuration memory write [13, 14]. It is an efficient technique for preventing errors in the configuration memory of SRAM-based FPGAs. Scrubbing is primarily employed to prevent the accumulation of configuration memory errors, and therefore a suitable scrub interval is chosen to ensure that the probability of multiple disturbances accumulating is virtually zero [15].
The component that conducts this function is known as the scrubber. In certain circumstances, the complete bitstream is injected into the configuration layer, and the application is briefly suspended and then reinitialized. The scrubber's detection phase is optional, which makes it more complicated but provides a robust mitigation.
A read back, on the other hand, is performed in the background and does not interfere with the performance.

3.2.3 Mitigation in Routing Resources

An FPGA programmable routing consists of routing within each logic block and routing between logic blocks. Switch blocks define the routing structure between logic blocks. The interconnect matrix defines the routing structure within the

logic blocks. The interconnections are made using programmable switches, which are comprised of a pass transistor controlled by a static RAM cell [16]. A detailed view is presented in figure 3.2:

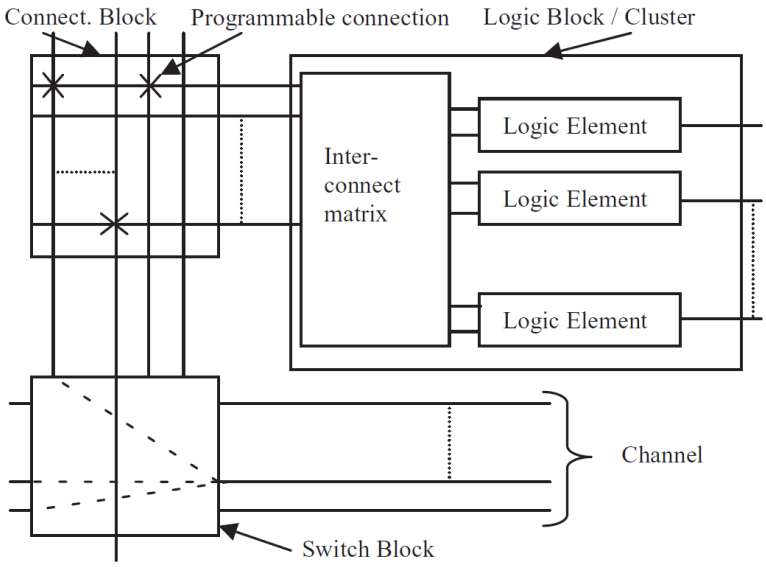

Fig. 3.2 FPGA routing architecture [1]

The majority of configuration bits are devoted to routing resources: nearly 90% of them. A reliability-oriented place and route algorithm (RoRA) is implemented, which first performs a reliability-oriented placement of each logic block in the design, and then routes the signals between the logic blocks ensuring that multiple errors affecting two different connections are not possible.

There are three kinds of errors which occur in switch blocks due to SEUs: open error, bridge error and short error. An open error occurs when a SEU causes an ON switch to become OFF; a bridging error occurs when a SEU causes two distinct nets to be coupled together; and a short error occurs when a net is connected to an unused routing resource [17, 18].
Using the unused programmable switches in the switch module, a new programming method for SRAM has been proposed for SEU-caused open errors: some of the switches are programmed to be ON or OFF depending on the type of connection. If a SEU disconnects a connection, other programmed switches can

reconnect the terminals and prevent the network from opening.
Another method based on programmable and hardwired switch module structure to mitigate SEU-caused short errors involves replacing some programmable switches with hard-wired nets, removing some programmable switches, and programming the remaining switches as before [19].
Less than 40% of the configuration bits are critical, and the vast majority of these bits are zero. A technique for soft error tolerance in configuration memory [20] proposes the use of an asymmetric SRAM (ASRAM) cell optimized for zero storage. This decreases failure risks by 25% when compared to the original design.
An optimization algorithm to enhance the number of zeros in the bitstream while maintaining functionality has also been developed [20]. However, this technique is only beneficial for reducing bridging errors and is not applicable to routing open errors.

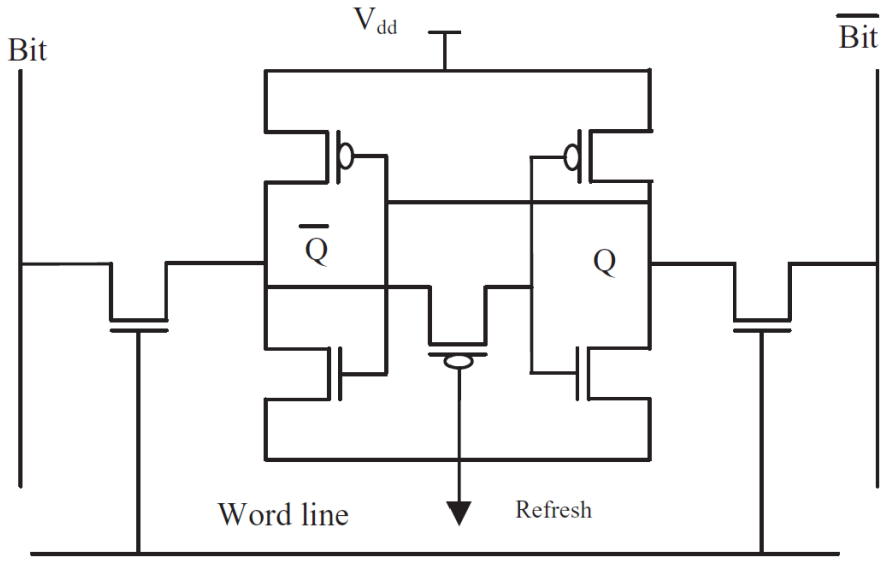

Fig. 3.3 Asymmetric SRAM cell [1]

The flexibility of the switch box is the utmost number of switches that can be used: in conventional switch boxes, this number is three. The unused switches increase the likelihood of bridge and short errors, so another hardening technique employs a flexibility of two, thereby decreasing the likelihood of errors. Each track entering and exiting one side of the switch block is connected to two other tracks

on the remaining three sides [21].
Moreover, a decoder-based switch box architecture [17] could reduce the amount of SRAM bits necessary for configuring the switch box. While causing no effect on the switch box's routing capability, it would reduce the likelihood of SEU errors in the switch module.

3.2.4 Mitigation in Logic Resources

A Look-Up Table (LUT) with k inputs can implement any function with k inputs 3.4: critical bits are stored in SRAM cells. A logic error may cause one of the LUTs' entries to be inverted, thereby altering the functionality of the mapped logical function.

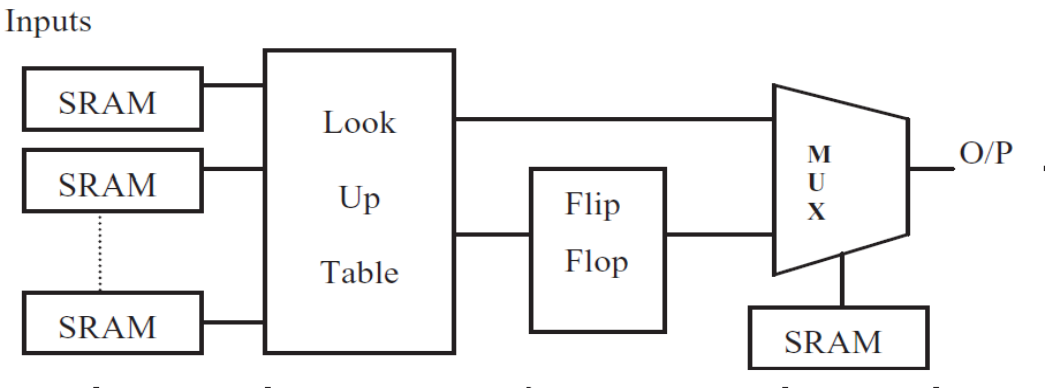

Fig. 3.4 K-input LUT [1]

Specific CLB architectures can be implemented to decompose 4-input LUTs into 3-input functions to minimize self-maps [22]: these are called Error Detection with Remap (EDR) and Error Correction with Remap (ECR). They split the logic function in CLB into two sub-functions and combines them through a carry chain, which makes the circuit more resistant to SEUs.

3.3 NEORV32 Processor Main Components

The chosen hardware hardening technique for this work was, however, the TMR implementation at core level, which meant that the single-core version of the NEORV32 processor had to be turned into a triple-core one (see section 4.2). In order to understand how this was made possible, a closer look at the processor basic components is required.

3.3.1 Core vs CPU

Before delving into the processor architecture, especially the one constituting the NEORV32 one, an important difference has to be marked: how a core differs from a CPU.

What is a core

A core is a processing unit of the CPU. It is responsible for executing programs and numerous other operations on a computer. Memory, control unit, and arithmetic-logic unit are the three primary components of a core. Each component of the core is tasked with specific responsibilities:

Control Unit (CU): this unit allows the core of a computer system to communicate with other components. It deals with signals and data in order to handle the other two modules behaviours.

Arithmetic-Logic Unit (ALU): it comprises of electronic circuits that execute arithmetic and logical operations. The ALU typically performs four arithmetic operations: addition, subtraction, multiplication, and division. Additionally, it commonly executes three logical operations: equal-to, less-than, and greater-than.

Memory: registers and cache constitute the memory constructed within the core. Registers are used to store data, whereas caches are high-speed RAM useful to quickly provide information.

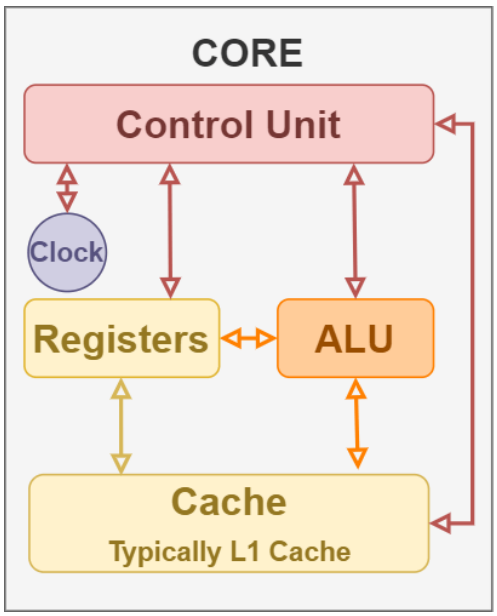

Fig. 3.5 Core architecture

A more detailed analysis can be found in the following sections.

What is a CPU

A computer's central processing unit (CPU) is the component responsible for coordinating the processors and executing tasks. As a consequence, a computer with a single CPU is able to simultaneously execute n duties, where n is the number of cores.
Along with hosting and coordinating the processing cores, the CPU handles communication between the other modules of a computer system and the processing cores (via their control unit). Typically, the CPU incorporates an additional cache level shared by all cores (usually a L2-L3 cache) and may include additional components, which benefit from being close to the cores.

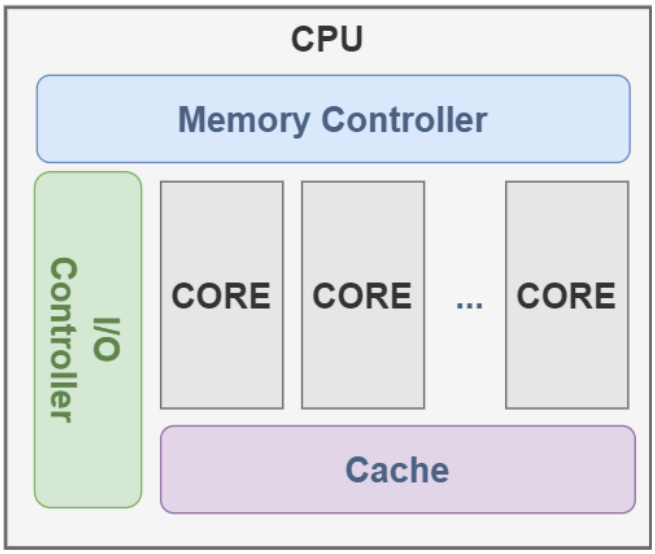

Fig. 3.6 CPU architecture

In conclusion, the main differences are those listed in table 3.1

Table 3.1 Core vs CPU: main differences

Processing Core	**Central Processing Unit (CPU)**
Processing element of the CPU	Processing component of the system
One (Single-core) Multiple (Multi-core)	One (Uniprocessor) Multiple (Multiprocessor)
Control Unit; Arithmetic-Logic Unit; Memory (Cache & registers)	Controllers; caches; Processing cores

The default NEORV32 project features a uniprocessor, single-core system, and the goal of this thesis was to triplicate its core turning it into a uniprocessor, triple-core version. Along with the core itself, other sections had to be instantiated multiple times: to understand which and why, a brief description of the main NEORV32 components follows.

3.3.2 Instruction Memory

IMEM, which stands for Instruction Memory, is NEORV32 processor ROM memory. Read Only Memory (ROM) is a type of non-volatile memory used to store permanent, fixed data (firmware) - even if updates allow to modify such data. As explained in section 4.1, it proved essential to reduce tests execution times and error probability.
There are 4 types of ROM technologies, namely:

Masked ROM (MROM): affordable hard-wired devices with preprogrammed data and instructions; those were the first to be developed.

Programmable ROM (PROM): one-time programmable memory, with no possibility to change or erase content of any kind afterwards.

Erasable PROM (EPROM): reprogrammable memory which exploits exposure to ultraviolet light to destroy stored charges and thus erase data, returning to a programming-ready state.

Electrically EPROM (EEPROM): electrically-erasable memory, which employs a significantly more convenient deleting process thanks to a less invasive bombing technique, along with the possibility to select precise locations and data amounts to wipe out.

3.3.3 Data Memory

DMEM, which stands for Data Memory, is instead NEORV32 processor RAM memory. Random Access Memory is a kind of volatile memory whose values can be read and changed freely, useful to store operations results and addresses. It is divided in two types, both of which are necessary in a standard computer system.

Dynamic RAM (DRAM)

DRAM is a compact technology and among the first memories to be commercialized. A single DRAM cell contains one transistor and one capacitor, and has higher storage capability when compared to an SRAM cell. It is therefore employed in the main memory architecture, and several types can be realized:

Synchronous DRAM (SDRAM): it increases performance through its pins, which improve data synchronization between main memory and the processor.

Double Data Rate SDRAM (DDR SDRAM): a doubled-speed version of SDRAM, which nowadays features 6 versions (up to DDR6).

Error Correcting Code DRAM (ECC DRAM): DRAM memory with the ability to detect errors and potentially fix them.

Static RAM (SRAM)

SRAM is a quick access RAM that can store data as long as power is available, updating saved values at the same time as they change. It features CMOS technology and, due to the presence of 4-6 transistors for each cell, a periodic refresh cycle is not necessary.
Higher costs, more energy and heat consumptions and less storage availability (when compared to DRAMs) constitute the price to pay to employ such fast and effective technology. That is why it is mainly used in cache memory (see section 3.3.4), which are meant to be small but efficient.

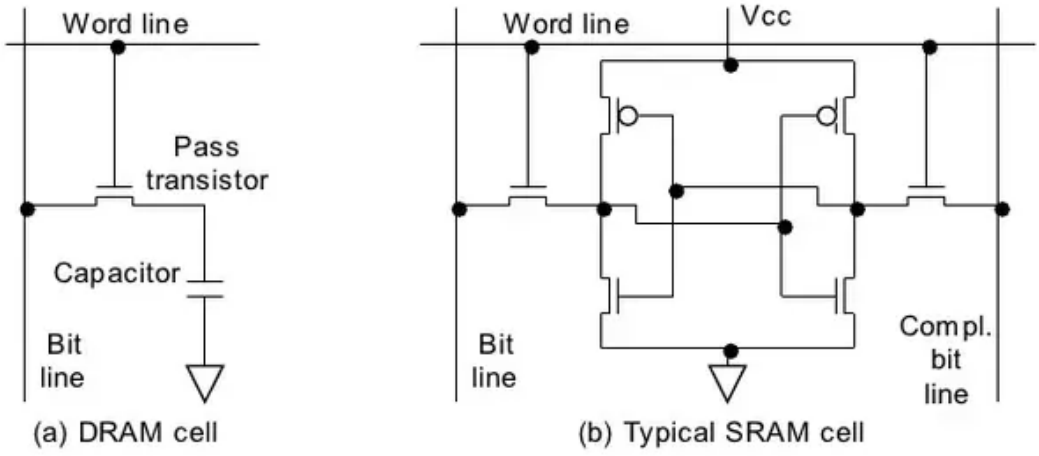

Fig. 3.7 DRAM cell (left) and SRAM cell (right)

3.3.4 Caches

A cache is a hardware or software component that stores data in order to expedite future requests: data stored in a cache may be the result of an earlier computation or a copy of data stored elsewhere. A cache strike occurs when the requested data can be found in a cache, while a cache error occurs when it cannot. Cache hits are served by reading data from the cache, which is speedier than recomputing a result or reading from main memory; therefore, the greater the number of requests that can be served from the cache, the quicker the system performs.
For caches to be cost-effective and to facilitate the efficient use of data, they must be relatively tiny. Despite this, caches have proven their worth in many areas of computation, as typical computer applications access data according to locality principles. These patterns are temporal locality, where requested data has been recently requested, and spatial locality, where it is physically near to previously requested data.
The NEORV32 processor makes use of a split cache architecture (opposite to the unified cache one) that consists of two physically distinct portions, one of which is dedicated to storing instructions (i-cache) and the other to storing data (d-cache). Since both are hardware-managed and target the same physical address space at the same hierarchy level (see fig 3.8), they are logically deemed to be a single, divided cache. Instruction retrieve requests are exclusively serviced by the i-cache, while memory operand read and write requests are exclusively serviced by the d-cache.
Cache memory is structured into levels that are used to describe the proximity and speed of access to primary or CPU memory. Typically a three level hierarchy is used (namely L1, L2, and L3, fig 3.8) although a fourth level L4 is sometimes implemented.

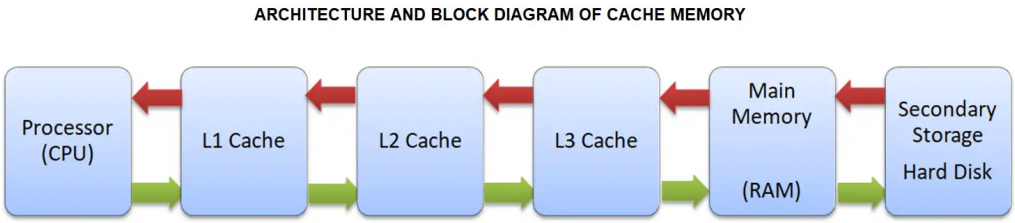

Fig. 3.8 Cache Hierarchy

L1 Cache

Level 1 cache, also referred to as registers, is a type of memory that is implanted on the processor chip as the CPU chip. It is a primary memory which does not shine in size but is significantly faster than other forms of memory. This level 1 cache serves the purpose of storing data by first receiving it from the central processing unit (CPU), after which it saves the information as quickly as possible. The data stored in such cache level is data that is of the utmost importance for the CPU to complete certain tasks.
This L1 cache is further subdivided into components, known as the information cache, which provides details about the operations that the CPU should perform, and the data cache, which stores data regarding the specifics of the data on which these operations should be performed. In average, for nowadays technologies, an L1 cache is designed with a size of 1-2 MB.

L2 Cache

Level 2 cache is default cache memory, slower than level 1 cache because data stored in this memory is only temporarily stored. However, the level 2 cache has a larger storage capacity when compared to level 1 cache. The level 2 cache has a typical size interval, ranging from 256KB to 8MB, although this size is likely to increase in the future.

L3 Cache

Level 3 cache, also considered the primary memory, is a highly specialized kind of memory that is designed to perform better than level 1 cache and level 2 cache. Level 3 cache is larger than both level 1 and level 2 ones, but it is the slowest of the three. In multi-core processors, each core has its own level 1 and may partially share level 2 cache, but all cores share a level 3 cache with twice the performance of RAM memory.

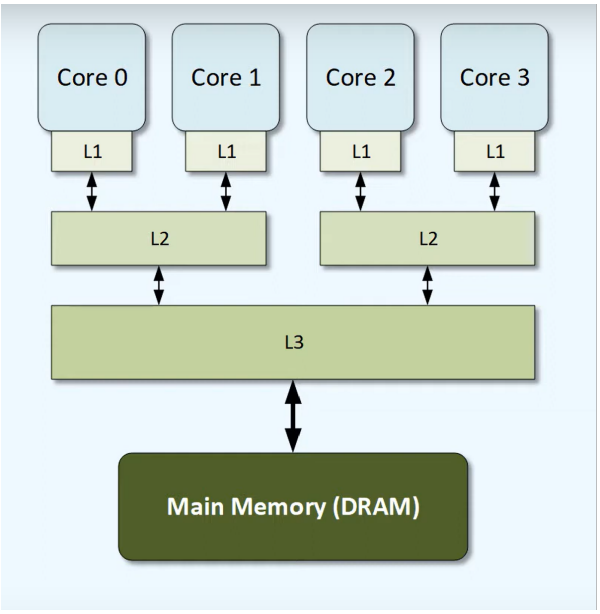

Fig. 3.9 Cache Architecture in Multi-Core CPU

L4 Cache

Level 4 cache, also considered the secondary memory, is external and is slower than level 3 cache or main memory. However, unlike level 3 cache, level 4 cache data remain stored even when the power is off. This type of cache is typically implemented with DRAM instead of SRAM.

3.3.5 Bus Switch

A bus switch is a module whose aim is to facilitate data communication between multiple digital system devices or components. Its primary purpose is to multiplex and arbitrate various transit requests and responses, and allows multiple devices or components to share a single bus (or communication channel). Since multiple devices may request access to such shared bus at the same time, the bus switch includes arbitration logic to determine, in a fair and efficient manner, which device is granted access.
The bus switch is also useful when multiple devices require access to a shared resource (such as memory, peripheral, or communication interface). It ensures that only one device at a time can access the resource, preventing data corruption

and contention issues. It also processes read and write requests from connected interfaces.
Lastly, the bus switch is also responsible for handling error conditions. If an error occurs during a bus transaction (such as data corruption or an expiration), the requesting port may receive an error signal. Moreover, the NEORV32 bus switch enables read-only configuration of the connected ports. This allows to specify if a specific port is permitted to conduct write operations on the shared resource.
In the NEORV32 processor, the bus switch is explicitly used to connect the core with its relative i-cache and d-cache memories.

3.3.6 Core Triplication

Now that all of the main processor components have been introduced, it is finally possible to comprehend how the core triplication has been possible. First of all it was mandatory to instantiate three times the processor core, each with its own signals and inner modules: this meant triplicating ALUs, CUs, register files and buses.
Secondarily, its external connected entities had to be replicated: since not everything was needed multiple times, this was a very delicate step. In fact, each new instantiation would take up space in both the FPGA area and the bitstream file, leading to higher consumptions and SEU sensitivity.
The essential items that had to be repeated were those previously described, namely instruction-caches and data-caches, bus-switches and, ideally, IMEMs and DMEMs. As explained in section 4.2, the latter two could not be triplicated due to PYNQ Z2 low memory availability, but in the same section the eventual solution is presented.
It is important to note that just like the bus switch handles accesses to i-cache and d-cache, the NEORV32 gateway module controls communication with IMEM and DMEM. Had those been instantiated multiple times, so would have the gateway entity - one for each core: however, in this case, it was totally unnecessary. Luckily, when it came to the other components, no problem arose in their triplication.
Proof of the hardened NEORV32 processor is shown in figure 3.10, where:

Yellow: shows the several bus switch entities.

Blue: marks the three cores. Note that, even though the denomination features *neorv32_cpu*, it actually refers to the core level (differences are explained in section 3.3.1). Its default name was only kept in order to avoid jeopardizing modules communication.

Green: highlights the different data caches.

Red: indicates the multiple instruction caches.

Purple: testifies the presence of the TMR module.

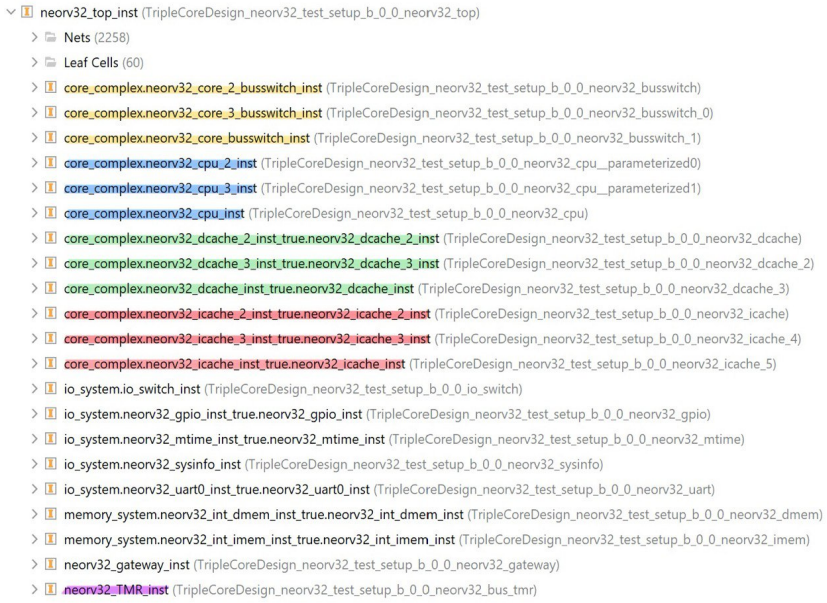

Fig. 3.10 Proof of Core Triplication in Vivado Implemented Design (see section 2.4)

Chapter 4

Methodology

Reliability is a key concept in technology fields. It defines the possibility to trust a component or a device, with little-to-zero questioning about its functioning and outputs. It also means, in some applications like the one considered for this work, entrusting life itself. That is why it is of the utmost importance to guarantee the best possible performances under any situation, and minimize any possible issue. If the previous works implemented Triple Modular Redundancy at the Arithmetic Logic Unit (ALU) level [23], the goal of this project was to triplicate the Processing Core and only then applying TMR.

4.1 Execution of a Fault Analysis

As previously mentioned, FPGAs device can be programmed through bitstream injections. SEUs, on the other hand, corrupt these sequences potentially leading to unpredictable and/or unwanted implemented hardware and corresponding behaviours. For this project, SEUs have been emulated through bitstream bit-flips, i.e. changes applied to the *golden* bitstream file obtained by toggling N selected bits (from 0 to 1 and viceversa) [24].
In order to simulate bitstream bit-flips, GitHub-private python library PyXEL [25] has been used. As first thing, the .bit file generated by Vivado after synthesis and implementation was converted to a .pmb one. PMB stands for Polar Uplink Tool Bitmap, i.e. a 2D visual representation of the correspondence between the single bit and the relative configured resource in the device.

Such operation was needed to identify the *Bitstream Active Parts*, which are the actual bitstream sections that are interested by the current configuration. A thorough analysis revealed few and distinct sections (fig 4.1), yet quite large, indicating that specific areas of the FPGA can be more SEU-susceptible than others.

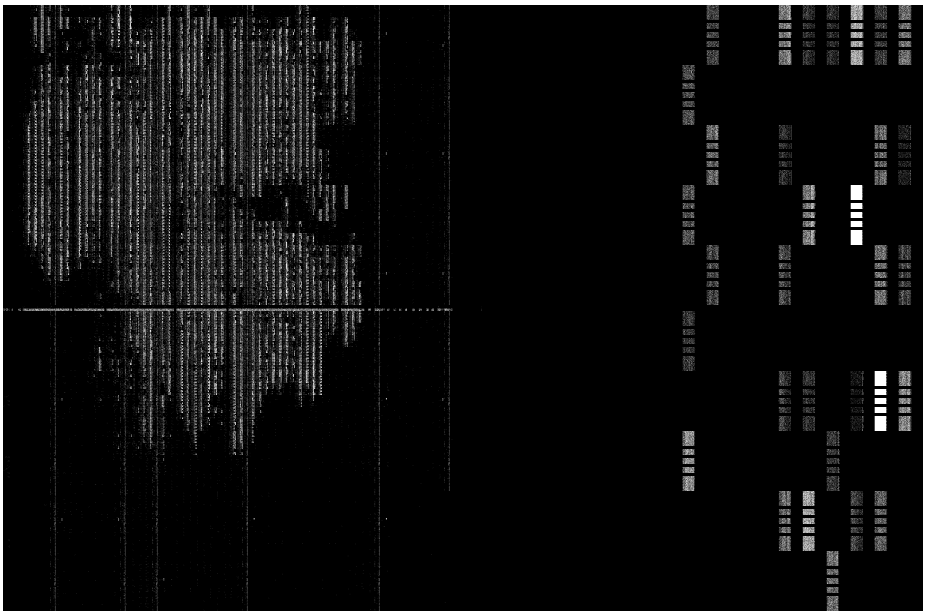

Fig. 4.1 Partial visualization of the bitstream active parts with PyXEL

Bit-flips simulation has been performed through a simple XOR operation which toggled the valued of a cartesian-coordinate defined bit. Toggling only bits belonging to the up-cited active parts can have a few advantages:

- It covers the worst case scenario, since almost every simulated SEU strikes meaningful sections of the bitstream.
- The responsibility of success shifts from luck in SEUs avoiding sensitive portions to real hardware robustness.

The whole fault injection process featured a series of well-established steps:

1. Acquisition of the default Vivado-generated bitstream and subsequent corruption with N simulated SEUs.
2. Injection of the faulty bitstream into the FPGA.

3. Verification of the obtained results, if obtained (i.e. if the device was able to send comprehensible data within the timeout).

4. Recording (and optional categorization) of the error, if occurred.

5. Presentation of the results, after 10000 tests have been performed.

The goal of this work was to determine and compare the SEU-induced error rate profiles, before and after TMR implementation. A 0-to-200 range has been considered regarding SEUs occurrences, whereas the output was an error probability (in percentage) corresponding to the processor failure possibility. For each number of simulated SEUs, 10000 tests have been performed in order to evaluate a reliable error rate (i.e. for 30 SEUs 10k tests, for 40 too, and so on).
A detailed workflow scheme of the whole process, starting from bitstream generation up to the results analysis, is presented in the following map:

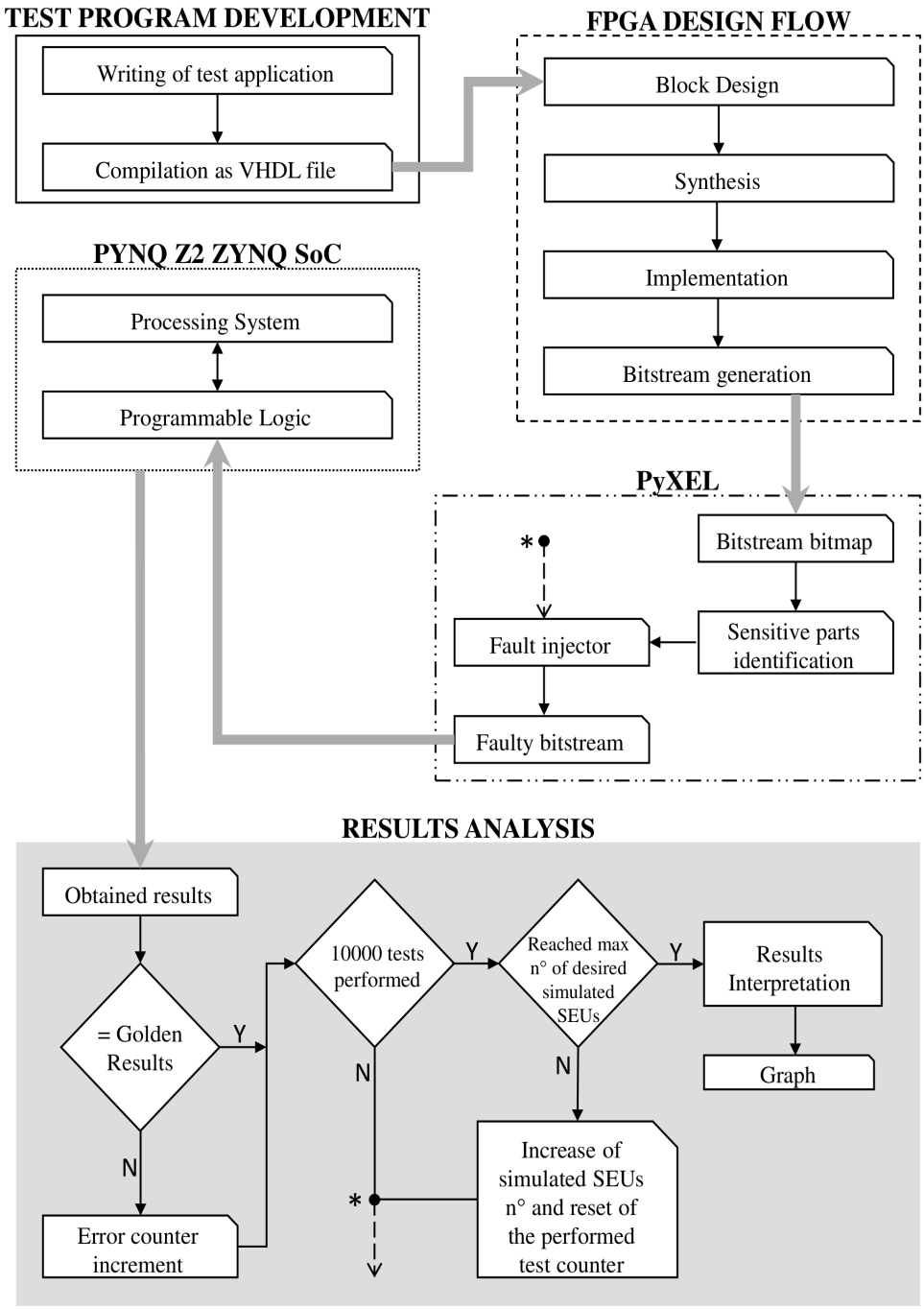

Fig. 4.2 Methodology Workflow Map

Responsible for the fault analysis execution was an apposite Python script, whose duties included initialising the PYNQ FPGA and storing temporary results. This *backup maneuver* proved necessary since the whole process could take over a week of uninterrupted execution, and a few times the host computer would experience unexpected reboots leading to results losses. Storing temporary outcomes allowed, in these cases, to restart from checkpoints and shorten the time needed to complete the analysis.
In order to test the correct functioning of the implemented hardware, the NEORV32 processor, a test program had to be sent to the board. As a first implementation, the NEORV32 Bootloader had been configured, so that the application could be sent in binary mode through UART communication. This method, despite finely working, required much more time when compared to the one eventually used. In fact, each execution used to take little less than a minute, causing the whole testing process to last for the inadmissible period of 1.7 years (approximately).
In addition to that, such procedure would have been way more susceptible to SEUs effects. This was due to the fact that many steps preceded the start of the test program, as the bootloader required precise instructions. It meant both that the bitstream active parts grew in number and size, leading to more sensitive areas, and that failure had more chances to occur.

As a consequence, it felt mandatory to opt for the other configuration, which featured a disabled bootloader but an enabled IMEM. The test application was now hardcoded into the bitstream, and automatically executed upon bitstream injection. This was made possible through the *make clean_all install* command for a compatible RISC-V toolchain, an array of programming tools used for complex software development tasks.
It generated a NEORV32_application_image.vhd file that could be added to the Vivado bitstream generation flow (see section 2.4). Doing so reduced error likeliness and, with its ~3sec duration, dramatically shortened the analysis execution time to little more than one week.
Given the limited resourced of the PYNQ Z2 FPGA, the test program had to be simple but effective. A 2, 3 by 3, integer matrix multiplication application felt appropriate, with the correct result being fixed and known to the Python script. Whenever the NEORV32 output could not be delivered or read correctly, or did

not match with the stored one, the error count was updated by a +1 increase.
After 10000 tests the error count would become definitive, and would be considered the error rate corresponding to that precise number of simulated SEUs (N). Last but not least, such N value would be incremented by a 10-default step value, potentially tuned accordingly to the obtained error rate. Had it been too close to the previous one, then the step would be increased, otherwise decreased: eventually another 10000 executions would start.
The core goal of this thesis was to delineate and compare the processor error rate before and after hardening, namely TMR implementation in this case. Before analyzing the results, a detailed introduction of the Triple Modular Redundancy technique is now presented.

4.2 Triple Modular Redundancy: explanation and implementation

Triple Modular Redundancy (TMR) is a fault-tolerant variant of N-modular redundancy in which three systems perform a process and the result is processed by a majority-voting system to generate a single output. If one of the three systems fails, the other two can correct and conceal the error: that is why it is frequently adopted as an hardening-by-design method [26–29]. In the general case of N-modular redundancy, any positive number of replications of the same action are utilized. In order for error correction by majority vote to be possible, the number is typically assumed to be at least three, and it is also assumed to be an odd number so that ties cannot occur.
It can also be coupled with the Isolation Design Flow (IDF), a design technique used to guarantee the non-interference of functions within the same processor by physically isolating the resources, thereby preventing fault propagation between modules. During the placement phase of design, a fence must be used to separate each module from the others, where fences are rows/columns of unused resources that separate two isolated regions. The on-chip communication must utilize reliable channels, namely routes (nets connecting isolated modules) that connect only one source and one destination (point-to-point connection) and traverse

only fence tiles separating the two isolated regions the route is connecting [30].

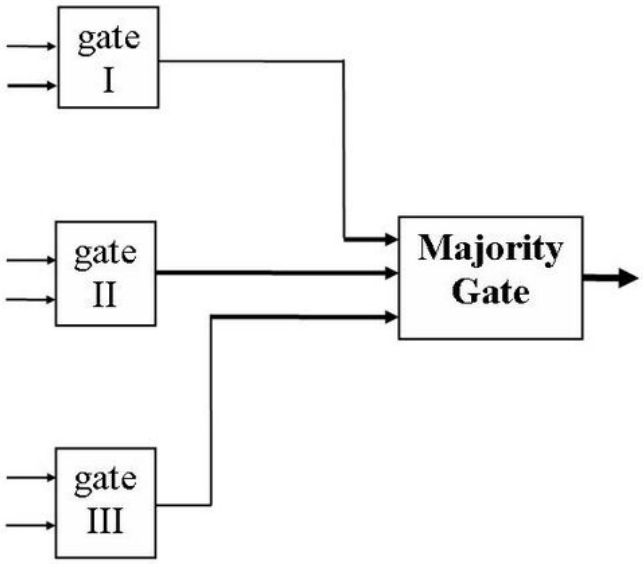

Fig. 4.3 TMR logic scheme

TMR check can be placed at many levels, but this project aimed to place it on the very top of the hardware hierarchy: between multiple cores. This meant that the NEORV32 processor, single core by default, had to be turned into a three core processor. Triplicating the core has a few drawbacks, like occupying more FPGA area and thus enhancing the probability that SEUs strike used parts of the bitstream (fig 4.4 and fig 4.5).

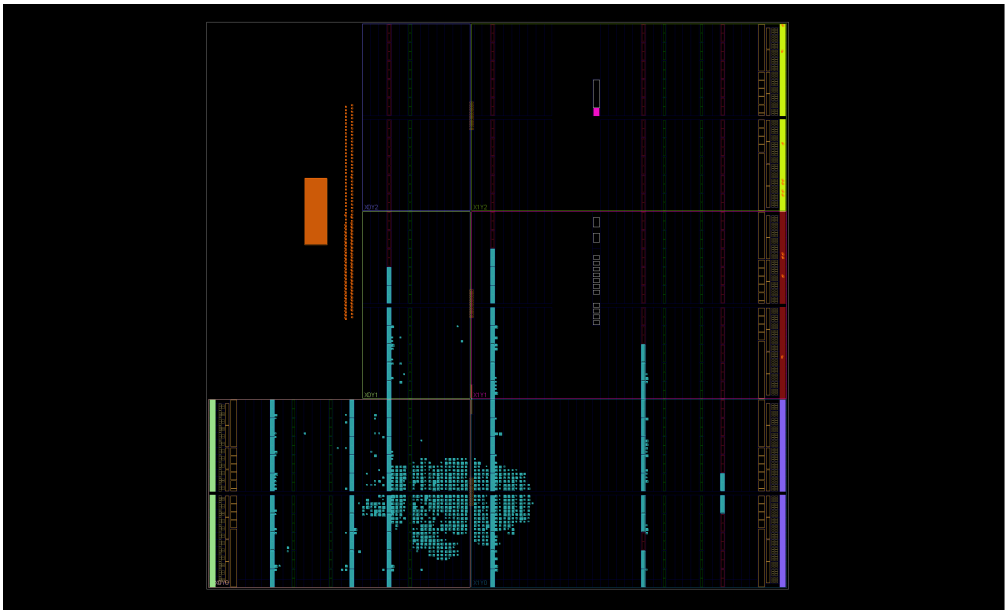

Fig. 4.4 Area occupied by Single-core NEORV32 on PYNQ Z2 in Vivado Design

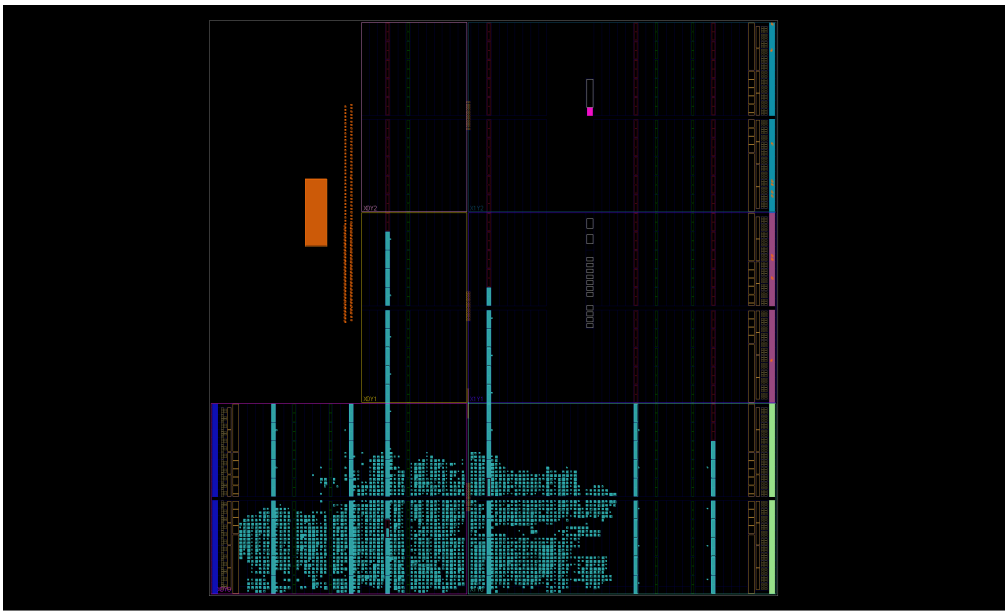

Fig. 4.5 Area occupied by Triple-core NEORV32 on PYNQ Z2 in Vivado Design

However, it also has many advantages: the first and foremost is the enabling of parallel computing, since different cores mean different, separate and autonomous calculating machines. Errors on one core processing unit are not reflected on the others, and can instead be detected and corrected through TMR

implementation.
Ideally, each core should have its own IMEM and DMEM, allowing full operational autonomy. Such desirable configuration was not possible on the PYNQ Z2 however, due to its limited blockRAM resource (~630 kB). Even if the simple matrix test application (described above) was just 219 kB in size, it would have required at least 3x256kB (min size of suitable IMEM) and it was not feasible.
That's why the TMR module has been instantiated according to the figure 4.6:

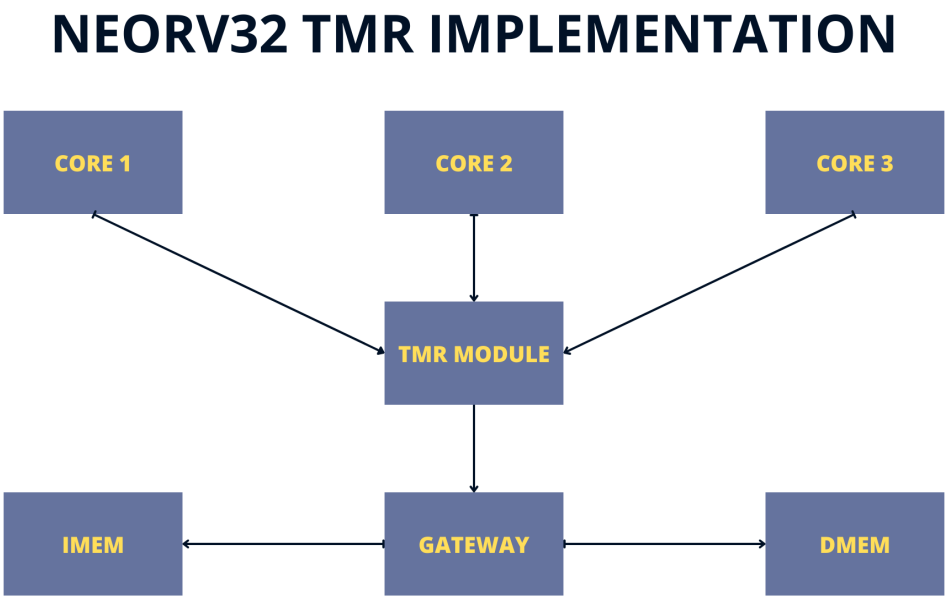

Fig. 4.6 TMR implementation scheme

The gateway module regulates accesses to IMEM and DMEM and is polled by the value exiting from the TMR block. By collocating the latter here, storing corrupted data into memory is less likely to happen since, should only one core be faulty anytime during an execution, the majority voter would ignore it.
With particular reference to the actual implemented TMR module in the NEORV32 processor, it is important to highlight that it does not only check for data equality but it compares all fields of the NEORV32 data bus type:

bus.addr: 32-bit logic vector containing the access address.

bus.data: 32-bit logic vector containing data to be written.

bus.ben: 4-bit logic vector corresponding to the byte enable signal.

bus.we: 1-bit single shot write request.

bus.re: 1-bit single shot read request.

bus.src: 1-bit access source marker (1 = instruction, 0 = data).

bus.priv: 1-bit signal set to indicate privileged access (in Machine mode).

bus.rvso: 1-bit signal set to indicate atomic operation.

This choice was made to realize the best possible TMR check, preventing the possibility to be fooled by equal .data fields potentially hiding different addresses, with one (or more) of those wrong.

Chapter 5

Results

Based on the results obtained in the testing phase, the following graphs, trends and tables have been drawn up.

5.1 SEU-Induced Error Rate for Single-Core NEORV32 Processor

The GitHub opensource project NEORV32, in its default version, when tested led to this error rate profile:

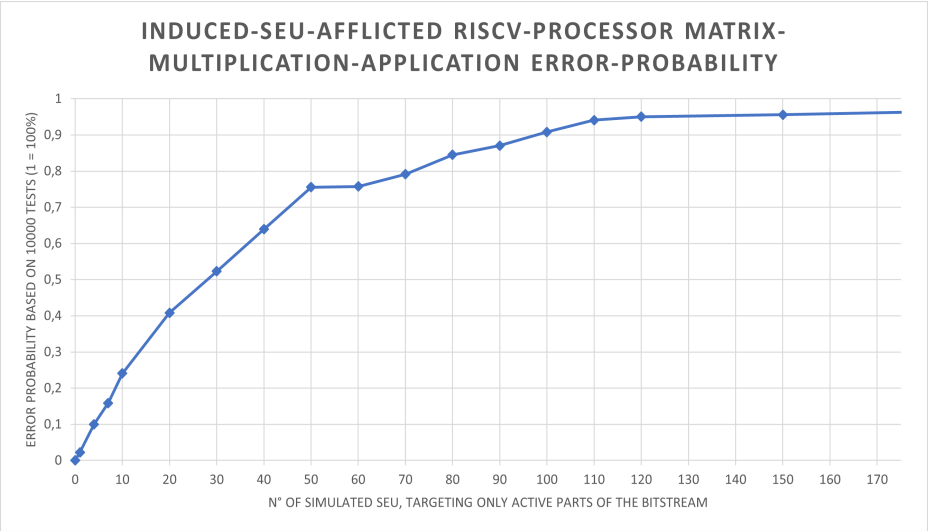

Fig. 5.1 Single-Core NEORV32 processor error probability trend

Table 5.1 Single-Core NEORV32 processor error probability values

N° of simulated SEUs	Corresponding Error Probability
0	0%
1	2.25%
4	10.01%
7	15.86%
10	24.07%
20	40.82%
30	52.3%
40	63.95%
50	75.54%
60	75.75%
70	79.14%
80	84.53%
90	87.03%
100	90.81%
110	94.09%
120	95.03%
150	95.57%
200	96.92%

As shown in graph 5.1 and in table 5.1, the default single-core version of the NEORV32 processor is quite vulnerable to SEEs. It only takes about 30 SEUs to produce more incorrect than correct results (P(30) = 52.3%), and even with just 10 SEUs there is still a 1 out of 4 possibility that errors occur (P(10) = 24.07%).
Defining an operational confidence interval for the NEORV32 processor can be useful to understand its SEU resistance. Setting an upper bound threshold allows to consider the processor serviceable as long as the error probability is contained in the range [0,threshold]. It is essential to determine the processor failure point, and enables fair comparisons between processors.
For this work, a 90% tolerance interval has been chosen: this meant classifying the single-core NEORV32 as working up until 100 SEUs (where P(100) = 90.81%). Further testing revealed a flattening trend, with an error rate almost saturating at

97% when 200 SEUs strike the device. It will eventually converge towards a 100% error probability, but it may require hundreds of impacts.

5.2 SEU-Induced Error Rate for Triple-Core NEORV32 Processor

The GitHub opensource project NEORV32, in its TMR-improved version, when tested led to this error rate profile:

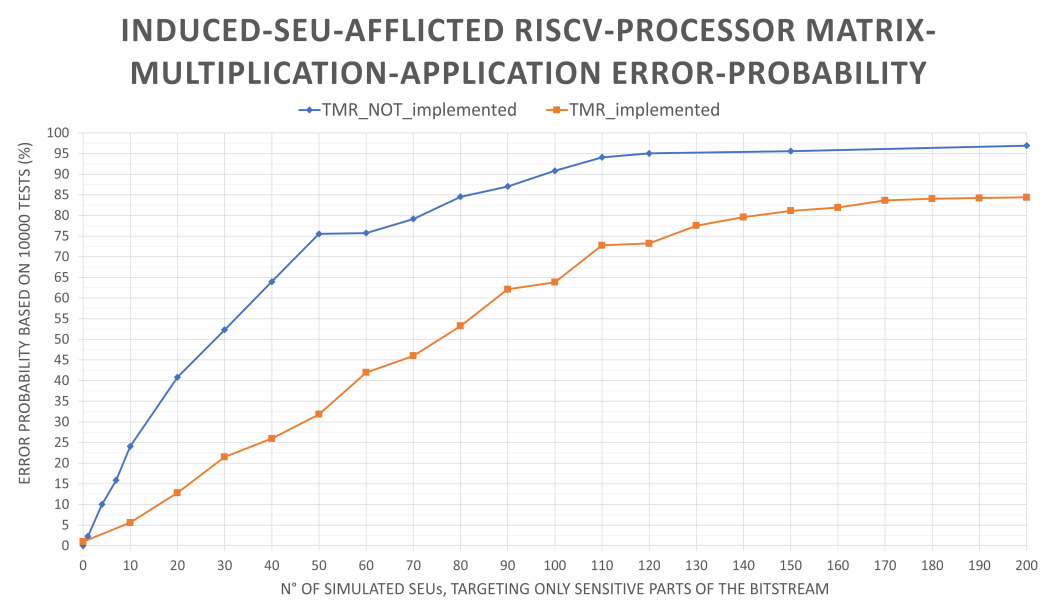

Fig. 5.2 Triple-Core NEORV32 processor error probability trend

Table 5.2 Triple-Core NEORV32 processor error probability values

N° of simulated SEUs	Corresponding Error Probability
0	0%
10	5.57%
20	12.83%
30	21.51%
40	25.94%
50	31.83%
60	41.96%
70	45.99%
80	53.24%
90	62.11%
100	63.82%
110	72.77%
120	73.22%
130	77.52%
140	79.58%
150	81.11%
160	81.89%
170	83.65%
180	84.04%
190	84.20%
200	84.39%

The triple-core NEORV32 with TMR implementation proved to be much more robust against SEEs. As shown in graph 5.2 and in table 5.2, when compared to the previous results there are three main differences:

Single Values Gain : injecting the same number of simulated SEUs, a significant improvement stands out for each N value. Its prime example is the error probability for N = 10, which is now about $\frac{1}{5}$ of the one obtained with a single core processor (5.57% against 24.07%). This indicates a 76.86% error rate reduction, meaning that (for 10 SEUs) a 76.86% gain in processor robustness and reliability has been obtained.

A more detailed comparison between single values is presented in section 5.3.

Confidence interval : should the same threshold as before be chosen, i.e. 90%, the processor operational interval would exceed 200 SEUs. Basically, the acceptable working area would be way more than doubled. Such result would not change much even in the case of an upper bound of 80%, since the single-core version would become unserviceable after 70 SEUs whereas the triple-core would resist until 140.

Saturation : last but not least, the single-core NEORV32 showed a saturating trend towards 97%, whereas the same SEU values now lead to about 85%. Even this version will, eventually, converge at 100% error probability but, if hundreds of SEUs were required to so for the single-core processor, thousands may instead now be needed.

5.3 Comparison between the two profiles

In the following table, the two error rates are compared (the interval considered is the tolerable working range for the single-core processor):

Table 5.3 Comparison of error probabilities before and after TMR implementation

N° of simulated SEUs	Error Probability Reduction (w.r.t. default value)
10	76.86%
20	68.57%
30	58.87%
40	59.44%
50	57.86%
60	44.61%
70	41.89%
80	37.02%
90	28.63%
100	29.72%
Average error reduction	50.35%

As clearly depicted in table 5.3, the average error probability in the considered interval, namely the single-core version confidence one, was slightly more than halved by the core triplication and the TMR implementation. As a consequence, the NEORV32 processor robustness and reliability has been slightly more than doubled.

Chapter 6

Conclusions & Future Works

Space missions are dangerous by nature, but the human desire for knowledge and exploration provides the courage to face challenges. In an environment already full of risks though, it is crucial to have the best available technology. Along with fast, reliable devices, this means having adequate protection against possible menaces such as, in this case, cosmic rays.
These radiations can cause SEUs inside FPGA devices, leading to potential SEEs and subsequent errors and failures. That is why it is of the utmost importance to employ an hardware whose robustness against them is the brought to its maximum. Such result can be obtained through hardening implementation and, among the several different applicable techniques, the Triple Modular Redundancy one was chosen for this work.
The tested processor was the GitHub open source NEORV32 processor, based on RISC-V ISA and described in VHDL. The testing process featured a Python script programming the Xilinx PYNQ Z2 FPGA 10000 times, each time with a different corrupted bitstream with a predefined amount of corrupted bits (simulated SEUs). The program would then read the obtained result and would check it with the correct, stored one, increasing the error counter in case of mismatches or impossibility to communicate.
After 10000 tests, the number of simulated SEUs would increase by 10 and the whole process would be repeated: this constituted the performed fault analysis. Its result was an error rate profile, which showed the error probability related to the amount of bitstream bit-flips.
The goal of this thesis was to analyze the NEORV32 processor error probability

profile before and after TMR implementation, which took place at core level (i.e. the processing core was chosen to be triplicated). Despite technical limitations, such as not enough memory resources to allocate separate IMEMs and DMEMs for each core, the project proved successful.
Such hardened version, in the working area of the default NEORV32 CPU, showed an average >50% error rate reduction. As a consequence, the triple-core NEORV32 processor more than doubled its confidence interval, leading to minor error probabilities and with higher numbers of SEUs needed to incur them.

This thesis lays the foundation for future, more in-depth analyses, which could be performed according to a few suggestions based on the limitations of this work. First of all, an FPGA with higher memory availability could be used to allocate separate IMEMs and DMEMs for each core, in order to avoid the risk of common errors due to shared resources corruption.
In addition to that, employing more cores for the modular redundancy control could result in interesting improvements. However, it would come at the cost of increased FPGA area and power consumptions, thus it should be carefully evaluated whether or not to pursue this idea.
A promising technique could be implementing .tcl scripts, possibly written in Python, to guarantee Isolation Flow constraints so that SEUs striking FPGAs may not cause multiple SEEs. These are constituted by specific hardware paths that must be followed to guarantee physical distance between different FPGA allocated resources. It is a way of reducing the risk of multiple errors due to a single particle, as a single radiation could not damage several sections thanks to actual module distances.
Last but not least, a more detailed error rate profile representation could be presented: different types of programs could be tested, in order to stress several processor sections, and errors could be categorized according to their nature (communication, value, timing etc..).

References

[1] T. S. Nidhin, Anindya Bhattacharyya, R. P. Behera & T. Jayanthi. A review on SEU mitigation techniques for FPGA configuration memory. 2017.

[2] E. Normand P.D. Bradley. Single event upsets in implantable cardioverter defibrillators. 1994.

[3] AMD Xilinx. PYNQ Z2 - python productivity for zynq. https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html.

[4] S. Azimi, C. De Sio, A. Portaluri, D. Rizzieri and L. Sterpone. A comparative radiation analysis of reconfigurable memory technologies: FinFET versus bulk CMOS. 2022.

[5] P. Bernardi, M. Sonza Reorda and D. Bortolato. Evaluating the effects of seus affecting the configuration memory of an SRAMbased FPGA. 2004.

[6] Stephan Nolting. NEORV32 processor - GitHub. https://github.com/stnolting/neorv32.

[7] AMD Xilinx. Correcting single- event upset through virtex partial reconfiguration, 2000.

[8] Xilinx. PR user guide. 2012.

[9] P. Brinkley, Avnet, and C. Carmichael. SEU mitigation design techniques for the XQR4000. 2000.

[10] G.-H. Asadi and M. B. Tahoori. Soft error mitigation for SRAM-based FPGAs. 2005.

[11] C. Carmichael. Correcting single-event upsets through virtex partial configuration. 2000.

[12] E. Cetin, O. Diessel, and L. Gong. Improving fmax of FPGA circuits employing DPR to recover from configuration memory upsets. 2015.

[13] I. Herrera-Alzu and M. L opez Vallejo. Design techniques for xilinx virtex FPGA configuration memory scrubbers. 2013.

[14] L.Sterpone and M. Violante. Hardening FPGA-based systems against SEUs: A new design methodology. 2006.

[15] A. Evans, R. Wong, S. Wen and G. Chen. New insights into the impact of SEUs in FPGA CRAMs. 2015.

[16] M. I. Masud. FPGA routing structures: A novel switch block and depopulated interconnect matrix architectures. 1999.

[17] H. Ebrahimi, M. S. Zamani, and H. R. Zarandi. Mitigating soft errors in SRAM-based FPGAs by decoding configuration bits in switch boxes. 2011.

[18] E. S. S. Reddy, V. Chandrasekhar, M. Sashikanth, and V. Kamakoti. Detecting SEU-caused routing errors in SRAM-based FPGAs. 2005.

[19] H. R. Zarandi, S. G. Miremadi, D. K. Pradhan and J. Mathew. Soft error mitigation in switch modules of SRAM-based FPGAs. 2007.

[20] S. Srinivasan, A. Gayasen, N. Vijaykrishnan and M. Kandemir. Improving soft-error tolerance of FPGA configuration bits. 2004.

[21] H. Ebrahimi, M. S. Zamani and A. Razavi. A switch box architecture to mitigate bridging and short faults in SRAM-based FPGAs. 2010.

[22] E. S. S. Reddy, V. Chandrasekhar, M. Sashikanth and V. Kamakoti. Novel CLB architecture to detect and correct SEU in LUTs of SRAM-based FPGAs. 2004.

[23] E. Vacca. Study and development of a radiation-hardened implementation of the RISC-V processor on reconfigurable devices. Master's thesis, Politecnico di Torino, 2021.

[24] S. Azimi, C. De Sio, D. Rizzieri and L. Sterpone. Analysis of single event effects on embedded processor. 2021.

[25] C. De Sio, S. Azimi and L.Sterpone. PyXEL: Exploring bitstream analysis to assess and enhance the robustness of designs on FPGAs. 2023.

[26] E. Vacca, C. De Sio and S. Azimi. Layout-oriented radiation effects mitigation in RISC-V soft processor. 2022.

[27] C. De Sio. SEU evaluation of hardened-by-replication software in RISCV soft processor. 2021.

[28] S. Azimi. A radiation-hardened CMOS full-adder based on layout selective transistor duplication,. 2021.

[29] S. Azimi. A new single event transient hardened floating gate configurable logic circuit. 2020.

[30] A. Portaluri, C. De Sio, S. Azimi and L. Sterpone. A new domains-based isolation design flow for reconfigurable SoCs. 2021.

Appendix A

Python Fault Injection code

A few notes on the fault injection Python script:

- In lines 14/18 lies the Customization Section, where the user can set whether to target only the bitstream active parts and how many tests to perform for each injected-SEUs value. Moreover, it is possible to adjust start, end and increment amounts of the number of simulated SEUs. Last but not least, a randomseed parameter can be added to generate always the same sequence of bits to be corrupted in the bitstream: this can be useful for more advanced testing.
- Line 28 configures a timeout for the serial port used to communicate with the NEORV32 processor: this proved mandatory in order to avoid never-ending loops while waiting for the results to be read.
- Lines 65 and 66 were used once to create the bitstream viewable file, thanks to PyXEL Python library.
- The exception clauses in lines 118 and 120 were implemented as a backup plan to terminate communication with the processor: many times before such implementation, especially with heavy corrupted bitstreams, the program would exit abruptly or remain stuck in these sections.
- Note that lines 126 and 127 are necessary to restore the default, uncorrupted bitstream before the following corruption and injection.

```
import serial
import shutil
import math
import os
import numpy as np
import random
from pyxel.core.xbitstream.xbitstream import Xbitstream
from pyxel.core.xtools.xxsct.xxsct import XXsct
from serial import EIGHTBITS
from serial import STOPBITS_ONE
from serial import PARITY_NONE

# MODIFIABLE VALUES
only_active_parts = True  # select only parts actually written and used in the bitstream
start = 0
stop = 250  # not reached
step = 10
n_attempts = 10000
# END OF CUSTOMIZATION

e = 0
a = 0
b = 0
test_number = 1
dim = (stop - start) / step
err = np.zeros(math.ceil(dim), dtype=int)
correct_results = np.array([443, 428, 533, 603, 584, 725, 666, 647, 798])
ser = serial.Serial(timeout=0.1)  # specified for readline
ser.port = 'COM3'
ser.baudrate = 19200
ser.bytesize = EIGHTBITS
ser.parity = PARITY_NONE
ser.stopbits = STOPBITS_ONE
ser.rtscts = False
ser.dsrdtr = False
ser.open()

shutil.copyfile('C:/Users/lupol/Desktop/PoliTO/TESI/Backup_Bootloader_Bitstream/TripleCoreDesign_wrapper.bit',
                'C:/Users/lupol/Vivado_project/TripleCore/TripleCore.runs/impl_1/TripleCoreDesign_wrapper.bit')
```

```
40
41 xsct = XXsct()
42
43 x = xsct.communicate('connect')
44 print(x)
45
46 x = xsct.communicate('target 1')
47 print(x)
48
49 x = xsct.communicate('source C:/users/lupol/ps7_init.tcl')
50 print(x)
51
52 x = xsct.communicate('ps7_init')
53 print(x)
54
55 x = xsct.communicate('ps7_post_config')
56 print(x)
57
58 for k in range(start, stop, step):
59     temp_err = 0
60     for x in range(n_attempts):
61         results_array = np.zeros(9, dtype=int)
62         xbs = Xbitstream(
63             'C:/Users/lupol/Vivado_project/TripleCore/
    TripleCore.runs/impl_1/TripleCoreDesign_wrapper.bit')
64
65         # create visual file to know used zones of the
    bitstream
66         # xbs.cm2pbm('C:/Users/lupol/Vivado_project/SingleCore/
    SingleCore.runs/impl_1/SingleCore_design_wrapper.bit.pbm')
67
68         # Load .bit in-memory
69         xbs.remove_crc_from_tail()  # disable crc control when
    .bit is downloaded in the fpga
70         for i in range(k):
71             if only_active_parts:
72                 select = random.randint(0, 4)
73                 if select == 0:
74                     a = random.randrange(600, 650)
75                     b = random.randrange(0, 3232)
76                 if select == 1:
77                     a = random.randrange(1800, 2600)
78                     b = random.randrange(1600, 1650)
```

```
79                  if select == 2:
80                      a = random.randrange(3200, 3900)
81                      b = random.randrange(0, 3232)
82                  if select == 3:
83                      a = random.randrange(4300, 7500)
84                      b = random.randrange(0, 3232)
85                  if select == 4:
86                      a = random.randrange(8700, 10007)
87                      b = random.randrange(0, 3232)
88              else:
89                  a = random.randrange(0, 10008)
90                  b = random.randrange(0, 3232)
91
92              xbs.cmem.flip_bit(a, b)  # modify .bit (in memory)
93
94          xbs.store_to(
95              'C:/Users/lupol/Vivado_project/TripleCore/
    TripleCore.runs/impl_1/TripleCoreDesign_wrapper.bit')
96
97          # implementing RISCV on PYNQ Z2
98          x = xsct.communicate(
99              'fpga C:/Users/lupol/Vivado_project/TripleCore/
    TripleCore.runs/impl_1/TripleCoreDesign_wrapper.bit')
100         print(x)
101
102         try:
103
104             read = ser.readline(45)
105
106             # Why max size in readline()? Easy to explain
107             # Despite timeout being set, the program can
    sometimes remain stuck here since the function
108             # readline() reads a variable number of bytes until
     a newline is encountered. Internally,
109             # it does this by calling read() repeatedly. The
    timeout parameter applies individually to each read()
110             # call, but not to the overall readline() -- it
    will keep trying to read() until it gets a newline,
111             # no matter how long it takes. So no EOL (maybe due
     to corruption) and program gets stuck forever
112
113             if 'Printing results '.encode() in read:
114                 for n in range(9):
```

```
115                     try:
116                         value = ser.readline(6)  # same error
    as above could happen here
117                         results_array[n] = value
118                     except Exception:
119                         pass
120         except Exception:
121             pass
122         if not np.array_equal(results_array, correct_results):
123             temp_err += 1
124 
125         # restoring the original bitstream before next
    injection
126         shutil.copyfile('C:/Users/lupol/Desktop/PoliTO/TESI/
    Backup_Bootloader_Bitstream/TripleCoreDesign_wrapper.bit',
127                         'C:/Users/lupol/Vivado_project/
    TripleCore/TripleCore.runs/impl_1/TripleCoreDesign_wrapper.
    bit')
128         print('Executed test n. ' + str(test_number) + ' of ' +
     str(dim * n_attempts))
129         print(temp_err)
130         test_number += 1
131 
132     err[e] = temp_err
133     e += 1
134 
135     # saving data so far in case there is a problem not to
    restart the analysis
136 
137     with open(
138             'C:/Users/lupol/Desktop/PoliTO/TESI/run_results'
139             '/
    TEMP_SINGLE_CORE_matrix_multiplication_fault_injection_errors
    .txt',
140             'a') as f_temp:
141         print('N. attempts per n. of simulated SEUs =',
    n_attempts, file=f_temp)
142         print('Only active parts of the bitstream have been
    corrupted =', only_active_parts,  file=f_temp)
143         print('Start =', start, file=f_temp)
144         print('Stop (not reachable) =', stop, file=f_temp)
145         print('Step =', step, file=f_temp)
146         print('N. of errors =', err, file=f_temp)
```

```
147         print('Execution worked until k =', k, file=f_temp)
148
149 print('DONE')
150 print('Results obtained: n. of errors is equal to ')
151 print(err)
152 with open('C:/Users/lupol/Desktop/PoliTO/TESI/run_results/
    SINGLE_CORE_matrix_multiplication_fault_injection_errors.txt
    ',
153           'a') as f:
154     print('N. attempts per n. of simulated SEUs =', n_attempts,
        file=f)
155     print('Only active parts of the bitstream have been
    corrupted =', only_active_parts, file=f)
156     print('Start =', start, file=f)
157     print('Stop (not reachable) =', stop, file=f)
158     print('Step =', step, file=f)
159     print('N. of errors =', err, file=f)
160
161 # removing temporary file
162
163 if os.path.exists('C:/Users/lupol/Desktop/PoliTO/TESI/
    run_results'
164                    '/
    TEMP_SINGLE_CORE_matrix_multiplication_fault_injection_errors
    .txt'):
165     os.remove('C:/Users/lupol/Desktop/PoliTO/TESI/run_results'
166               '/
    TEMP_SINGLE_CORE_matrix_multiplication_fault_injection_errors
    .txt')
167 ser.close()
```

Listing A.1 Python code to perform fault analysis

Appendix B

NEORV32 TMR Module VHDL code

Showing how the core triplication had taken place and how the different signals had been connected seemed unnecessary: however, a quick look at the VHDL TMR entity and behaviour felt way more interesting:

- As shown in figure 4.6, the TMR module was connected to the three cores: this is why lines 48/50 assign to the different core-specific response signals the value obtained by the NEORV32 gateway.
- Line 73 implements an extra check: if all three cores show different results, than none of them is forwarded to the memories and a request termination is performed.
- An extra control signal - tmr_err - has been defined: it allows the system to know a priori whether everything is working fine or not. In fact, it is set to 1 only when the TMR module does not recognize two values as equal and thus detects the presence of errors.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 library neorv32;
6 use neorv32.neorv32_package.all;
7
8 entity neorv32_bus_tmr is
9   port (
```

```
10    -- global control --
11    clk_i         : in  std_ulogic;
12    rstn_i        : in  std_ulogic;
13    -- core complex ports --
14    core_a_req_i : in  bus_req_t;
15    core_a_rsp_o : out bus_rsp_t;
16    core_b_req_i : in  bus_req_t;
17    core_b_rsp_o : out bus_rsp_t;
18    core_c_req_i : in  bus_req_t;
19    core_c_rsp_o : out bus_rsp_t;
20    -- system port --
21    sys_req_o     : out bus_req_t;
22    sys_rsp_i     : in  bus_rsp_t
23  );
24 end neorv32_bus_tmr;
25
26 architecture neorv32_bus_tmr_rtl of neorv32_bus_tmr is
27
28  signal tmr_err : std_ulogic;  -- in future could be used to
      detect discrepancies
29
30   function cmp_bus_req_f(x : bus_req_t; y : bus_req_t) return
      boolean is
31   begin
32     if (x.addr = y.addr) and
33        (x.data = y.data) and
34        (x.ben  = y.ben)  and
35        (x.we   = y.we)   and
36        (x.re   = y.re)   and
37        (x.src  = y.src)  and
38        (x.priv = y.priv) and
39        (x.rvso = y.rvso) then
40       return true;
41     else
42       return false;
43     end if;
44   end function cmp_bus_req_f;
45
46 begin
47
48   core_a_rsp_o <=  sys_rsp_i;
49   core_b_rsp_o <=  sys_rsp_i;
50   core_c_rsp_o <=  sys_rsp_i;
```

```
51
52
53
54 write_tmr: process(rstn_i, clk_i)
55 begin
56   if (rstn_i = '0') then
57     sys_req_o <= req_terminate_c;
58     tmr_err   <= '1';
59   elsif rising_edge(clk_i) then
60     if (cmp_bus_req_f(core_a_req_i, core_b_req_i) = true) or
61        (cmp_bus_req_f(core_a_req_i, core_c_req_i) = true) then
62       sys_req_o <= core_a_req_i;
63       tmr_err   <= '0';
64     elsif (cmp_bus_req_f(core_b_req_i, core_a_req_i) = true) or
65           (cmp_bus_req_f(core_b_req_i, core_c_req_i) = true)
    then
66       sys_req_o <= core_b_req_i;
67       tmr_err   <= '0';
68     elsif (cmp_bus_req_f(core_c_req_i, core_a_req_i) = true) or
69           (cmp_bus_req_f(core_c_req_i, core_b_req_i) = true)
    then
70       sys_req_o <= core_c_req_i;
71       tmr_err   <= '0';
72     else -- discrepancy detected
73       sys_req_o <= req_terminate_c; -- no system bus access is
    performed at all
74       tmr_err   <= '1';
75     end if;
76   end if;
77 end process write_tmr;
78
79
80
81 end architecture neorv32_bus_tmr_rtl;
```

Listing B.1 VHDL definition and architecture of the TMR entity

Appendix C

C Test Program

This is a simple C two 3x3 integer matrix multiplication program, with only a couple highlights:

- Lines 11 and 17 are used as safety check to communicate via UART with the NEORV32 processor.
- Line 65 and 66, on the other hand, actually handle information communication through UART channel.

```
1  #include <neorv32.h>
2
3  #include <stdio.h>
4
5  #define BAUD_RATE 19200
6
7  int main() {
8
9    // capture all exceptions and give debug info via UART
10   // this is not required, but keeps us safe
11   neorv32_rte_setup();
12
13   // setup UART at default baud rate, no interrupts
14   neorv32_uart0_setup(BAUD_RATE, 0);
15
16   // check available hardware extensions and compare with
       compiler flags
17   neorv32_rte_check_isa(0); // silent = 0 -> show message if
       isa mismatch
```

```
18
19    int results[9];
20
21      int c[3][3];
22
23      int a[3][3] = {
24              {10, 11, 12},
25              {14, 15, 16},
26              {13, 19, 17}
27      };
28
29      int b[3][3] = {
30              {11, 12, 13},
31              {15, 16, 17},
32              {14, 11, 18}
33      };
34
35  int p = 0;
36
37  for (int i = 0; i < 3; ++i) {
38      for (int j = 0; j < 3; ++j) {
39          c[i][j] = 0;
40      }
41  }
42
43  for (int i = 0; i < 3; ++i) {
44      for (int j = 0; j < 3; ++j) {
45          for (int k = 0; k < 3; ++k) {
46              c[i][j] += a[i][k] * b[k][j];
47          }
48      }
49  }
50
51
52  neorv32_uart0_puts("Printing results \n"); // now we can start
        printing the results
53
54  for (int i = 0; i < 3; i++) {
55      for (int j = 0; j < 3; j++) {
56          results[p] = c[i][j];
57          p ++;
58      }
59  }
```

```
60
61 char buffer[16];
62
63 for (int i = 0; i < 9; i ++) {
64   //printf("%d \n", results[i]);
65   snprintf((char *)buffer, sizeof(buffer), "%d \n", results[i])
     ;
66    neorv32_uart0_puts(buffer);
67 }
68
69
70 return 0;
71 }
```

Listing C.1 C program to stress the NEORV32 processor

Acknowledgements

Vorrei ringraziare tutti quelli che mi sono stati accanto in questi impegnativi 5 anni di università. In primis vorrei ringraziare mia madre, in quanto si è sempre sacrificata affinchè io potessi avere il meglio e dare il massimo, e per il costante supporto.
Un grazie sincero va ai miei fratelli, Fabrizio Ferruccio e Stefano, e ai miei nipoti, Federico Elena e Sofia, per avermi sempre fatto sentire accettato e anzi desiderato. E avermi dato esempi costanti, in qualsiasi settore. Un abbraccio anche a tutta la mia famiglia, zii e cugini, per essere sempre stati al mio fianco (tra questi un pensiero speciale va a zia Anna, zia Lea e zio Gianpaolo).
Un profondo grazie va alla mia ragazza, Valeria, per avermi sempre sostenuto e stimolato a dare di più. Una vigorosa stretta di mano va poi ai miei amici, per avermi accompagnato e fatto distrarre e divertire ogni volta che c'era occasione, e per essermi invece stati vicini quando le cose andavano male. Anna, Fabio, Roberta, Marta, Peppe e Gaia (tra i tanti): grazie davvero!
Un grazie più "tecnico" va invece a Stephan Nolting, creatore del "programma" alla base di questa tesi, il NEORV32. Senza la tua disponibilità e gentilezza non sarei riuscito a venire a capo di tutti i problemi e le difficoltà che hanno costellato questo lavoro: grazie dal profondo del cuore.
Infine vorrei ringraziare il mio relatore, il professore Luca Sterpone, ed i suoi collaboratori, tra cui Corrado de Sio e Daniele Rizzieri, per avermi dato quest'opportunità ed accompagnato in questa fantastica esperienza.