POLITECNICO DI TORINO

Master Degree Course in Electronic Engineering

Master Degree Thesis

Development of Methods for the reliability analysis of RISC-V architectures

Supervisors

Prof. Luca STERPONE

Ph.D. Corrado DE SIO

Eng. Daniele RIZZIERI

Candidate

Giorgio CORA

ACADEMIC YEAR 2022/2023

Summary

The concept of space exploration has become more and more popular over recent years and many companies approaching this field are requesting hardware and software components that are reliable and efficient even when working in harsh environments.

SoC and, in particular, FPGA are representing an interesting solution that allows extremely good performances combined with a high level of re-programmability and fast computations. Whether it is efficient executions or power management and routing optimization, FPGA are what the field of space exploration is nowadays demanding.

Another big step forward was made thanks to the introduction of the RISC-V ISA. Being based on an open source license and a reduced instruction set architecture, it has quickly become a leading processor in thousands of different fields; many versions have been developed to respond to a wide variety of demands.

However, simply implementing a fast and efficient processor onto an FPGA is not sufficient to satisfy space requirements: reliability have to be ensured in an environment where electromagnetic fields and radiations can influence the behavior of every kind of electronic device, especially the programmable ones. Particular attention has to be payed to SEE and SEU: these faults can affect the configuration memory, thus modifying the internal architecture of the programmed FPGA and its routing, possibly leading to catastrophic results that cannot be tolerated in a safety-critical environment.

This is the starting point of my thesis, in which I analyzed a specific and relatively new RISC-V architecture, the NEORV32. My research addressed the radiation-induced effects on the processor architecture when implemented over an SRAM-based FPGA. Specifically, I focused on the SEU faults modelled as bitflip in the CRAM of the device, targeting the modules composing the NEORV32 CPU: the Arithmetic and Logic unit, the Register File, the Bus and the Control unit. For this purpose, an exhaustive fault injection campaign has been performed over specific portion of the FPGA where the NEORV32 has been mapped. To perform these operations a new fault injection platform, based on python programming language, has been developed.

Every time a fault was injected, a specific benchmark program was run on the processor and its output signature was recorded to analyze the fault propagation up to the application level. Then, correction of the fault takes place and if everything went smooth, the system will move on with the next injection; otherwise the board was reprogrammed from scratch.

Results have been classified into different categories depending on the type of response given by the executed program. Also different reports were generated according to the specific unit in which the fault injection was performed, thus allowing us to understand which portion of the CPU will misbehave the most in case of errors. This whole procedure was performed using different benchmark programs and a total of almost 1 million faults were injected during the whole process. This analysis allowed to understand that the overall error rate of our

NEORV32 architecture is in line with what expected in the field of space applications; the actual outcome of the faulty system will depend on where the injection was performed. For example when the ALU was under test most of the errors lead to a mismatch in the output signature with respect to the expected one. In the case of the Control Unit, most of the injected faults will lead to processor halts or generic SDC errors; for the Register file or Bus unit, mixed outcomes were registered. Obviously, suitable hardening actions such as TMR modules or software redundancy techniques will allow reliability improvements.

General Structure

Chapter 1 introduces the RISC-V general structure, the reasons behind its success and the NEORV32 architecture.

Chapter 2 defines the state-of-the-art: other available works on the same subject are analyzed to understand what was already done and what is required to further explore in a deeper way.

Chapter 3 summarizes the radiation effects theory and the background technology/knowledge needed to carry on this thesis work.

Chapter 4 is a description of the developed fault injection platform, the programs that were selected for evaluating RISC-V performances and the ways in which the injection has been performed.

In chapter 5 the obtained results of the fault injection campaign are analyzed, compared with already existing data, and discussed.

Chapter 6 is related to conclusions and potential future research work.

Acknowledgements

First, I would like to thank my supervisor, prof. Luca Sterpone for supporting me during these months of intense work with its patience and positive attitude, and all the research group members, particularly Daniele and Corrado, for helping me and allowing to expand my knowledge.

A special thanks to my parents, Ivo and Marina, whose presence and moral support has allowed me to surpass every difficulty that stood in front of me for all those years. You will always represent a milestone in my life.

Thanks to my brother, Sandro, who was present for me despite not always being able to stay in touch with each other.

Thanks to my dearest friends, Luca, Scila and all the persons who helped me to alleviate everyday pressure by always standing by my side and supporting me.

Table of Contents

List of Tables

List of Figures

Abbreviations

ALU Arithmetic Logic Unit

ADD Addition

CU Control Unit

DMEM Data Memoryt

FPGA Field Programmable Gate Array

FPU Floating Point Unit

GPIO General Purpose Input/Output

HDL Hardware Description Language

IMEM Instruction Memory

ISA Intruction Set Architecture

LSB Least Significant Bit

MSB Most Significant Bit

PC	Program Counter
PL	Programmable Logic
RF	Register File
SEU	Single Event Upset
SUB	Subtraction
UART	Universal Asynchronous Receiver-Transmitter

Chapter 1

Introduction

Nowadays microprocessors are some of the most used devices all over the world, especially in the embedded system environment. Stemming from cars, phones and many other commonly used devices, up to more critical and specific applications, the overall low-cost and high level of versatility of these devices is what allowed them to become so popular. Given that the first prototypes of microprocessors were formerly introduced in 1969, this technology has come a long way since then. From 8 up to 32- and 64-bits designs were developed until, in the late 80s and early 90s, a new and promising technology was introduced: the Reduced Instruction Set Computer (RISC) architecture. The aim was to reduce the complexity of the executed instructions and improve the efficiency of the system executing them.

Stemming from this, another big step forward was achieved with the development of the RISCV ISA. This technology opened a lot of doors in the electronic world thanks to its open-source availability and modular reconfigurability. A lot of companies started to utilize this technology by modifying it according to their requirement and providing the public with different versions based on the same simple pipelined architecture. Space exploration is one of the fields in which the versatility of this technology has come in handy and has allowed to take a big step forward. This thesis work mixes the utilization of one of the open-source models available to us with the analysis of how it will behave under specific conditions that simulate space environment.

1.1 RISCV General architecture

The RISC-V ISA was formerly introduced in 2015; however, this project already began in 2010 at the University of California, Berkley [1]. It is a simple Load-Store architecture with a reduced instruction set, meaning that no direct manipulation can be performed on data stored inside the memory but only operations between registers are allowed. The instruction set specifications provides either a 32-bit or 64-bit address space variants. Also, some 128-bit versions have been developed as an adaptation of the 32-bit or 64-bit ones. Every instruction in the ISA is encoded on 32-bit and classified according to 5 distinct groups: R-type (register-register operations), I-Type (Operations with immediate), S-type (Store operations), SB-type (Branches), U-type (Load) and UJ-type (Jumps). It is based on a modular design, meaning that each element inside the architecture can be singularly taken, analyzed and modified in a straightforward way; modularity also provides a way to add and remove components (those that are not essential to task execution) to optimize either power consumption or computational efficiency. Standard design only implements simple ALU operations, such as ADD, SUB and shifting operations; extensions are available, such as Multiplication/Division (M-Extension), Floating-Point (single F extension, double D extension, quad Q extension precision), compressed instructions (C-extension) and Control & Status register (Zicsr Extension) support and many others.

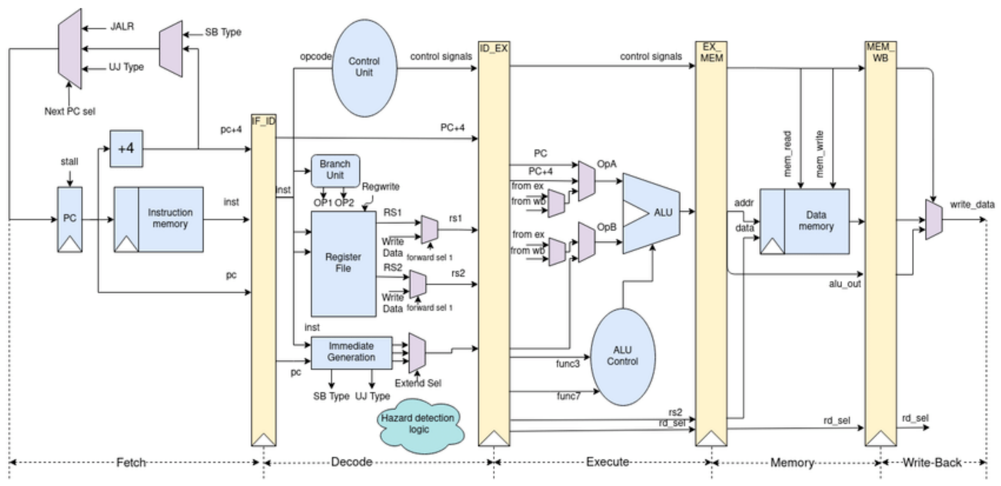


Figure 1.1: RISCV Pipeline

RISC-V is based on a 5-stage pipeline: Fetch, Decode, Execute, Memory and Writeback; however, certain versions allow further stages to be introduced to optimize computations capabilities.

- *Instruction Fetch Unit (F)*

  In this stage the instructions are fetched from the memory, addressed by the program counter, namely the PC; its value is updated with either the next address, which is computed by adding 4 to the previous one due to memory alignment requirements, or with the destination address in case a jump/taken branch instruction was executed.

- *Decode Stage (D)*

  Here the 32-bits instruction is decomposed and information are extracted: the op-code, which is used to understand what type of instructions must be executed, the destination and sources registers (expressed though 5-bit addresses called respectively rd, rs1 and rs2) or the immediate in case of I-type operations, the memory off-set in case of load/store operations or the offset to be added to the PC for jump/branch instructions.

- *Execute Stage (EX)*

  In this stage the ALU executes the actual operation between the 2-input operands, identified as OpA and OpB in the figure; depending on the available extension of the RISC-V, multi-cycle operations might also be executed. In this stage there is also a forwarding logic, implemented by using a series of multiplexers at the input of the ALU. Func3 and func7, meaning functions on 3 and 7 bits, are a series of bits used to decide which type of operations must be executed.

- *Memory Stage (MEM)*

  This stage is of interest only when load/store operations have to be executed. Memory is accessed using the address computed by the ALU and alignment on output value is performed. Alu_out bus is used when this stage has to be bypassed.

- *Write-Back Stage (WB)*

  The outcome of executed operation, namely write_data, is written back inside the destination register (identified by rd_sel signal) of the RF and the return address is updated.

1.2 NEORV32 Processor and CPU

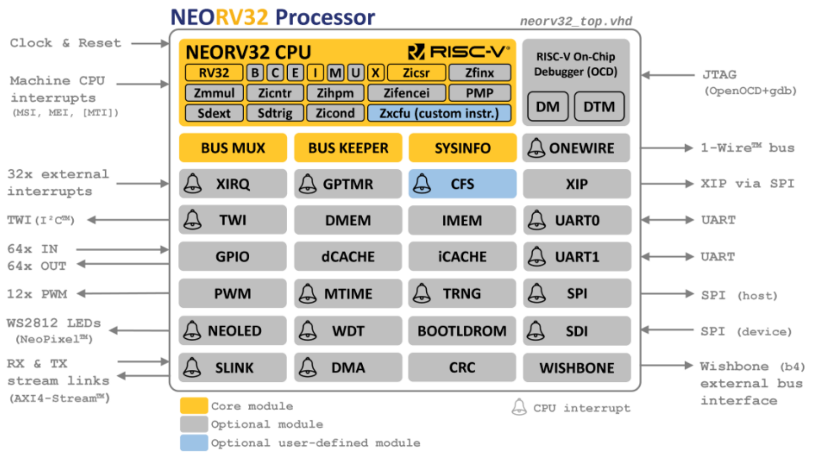


Figure 1.2: NEORV32 SoC Architecture [2]

One of the main and most recent implementations of the RISC-V architecture is the NEORV32 processor [2], developed by Stephan Nolting. This version provides a highly customizable SoC platform designed using VHDL language, in which modules can be easily enabled or removed according to the requirements of the user. The ones of interest for my case are:

- The **NEORV32 CPU:** based on a Von-Neumann machine built over a combination of multi-cycle and pipelined execution schemes

- The **UART:** 2 universal asynchronous receiver-transmitter channels are available, UART0 and UART1. This module is not active by default and its parameters can be set by defining parameters values. Baud rate is configurable via prescaler while the transmission frame is fixed to 8 bits, no parity and 1 stop bit. Specific blocking C-functions are defined, which can be used to setup and utilize the UART. Its control and data registers are memory mapped.

The only difference between UART0 and UART1 are the addresses of their corresponding registers.

- The **GPIOs:** This module implements up to 64 general purposes input-output connections; the number of required pins can be set via parameters (8 GPIO are available in our architecture).
- The **BOOTLDROM:** Read-only memory holding executable image of the bootloader. If the bootloader is enabled, then boot address is automatically set at the beginning of the bootloader ROM. The configuration of the bootloader can be defined by compiling the corresponding C file into an application image. It allows interaction with the NEORV, allowing to upload an executable in memory, store/load it to/from flash memory, boot via XIP (requires pre-programmed flash) and launching the application. The bootloader will launch at system startup. After n seconds (n set as parameters through the C file) autoboot sequence will start. If n is set to 0, bootloader is disabled.
- The 2 memories, **DMEM and IMEM:** Processor internal instruction and data memories that can be enabled by setting the corresponding generic values to true (MEM_INT_IMEM_EN and MEM_INT_DMEM_EN). Also, their size can be modified accordingly to user requirements; however, when changing ram size, it is important to remember also to change the linker script accordingly. Largest ROM size is 2048Mbyte, this value is already set in the linker script and does not have to be modified even if IMEM size changes. If they are disabled, external memories are needed; in this case also the wishbone should be implemented.

All the other modules were not used for this thesis work. Those includes:

- **Wishbone:** used to interface the neorv CPU to external devices such as memories, IO devices and so on; load/store are delegated to wishbone(and so external components) if the access does not target an internal memory (IMEM,DMEM or bootloader ROM)

- **iCaches and dCaches:** Direct mapped or 2-way set associative cache, depending on the generic ICACHE_NUM_SETS or DCACHE_NUM_SETS value. Replacement policy is based on LRU(LRU). These memories can be enabled or not by setting the I-CACHE/D-CACHE generic value to true.

- **BUS MUX:** Allows bus access by 2 different controller port.

- **BUS KEEPER:** Internal bus monitor. Ensure that bus operations are executed properly. Inform the CPU in case of anomaly behavior of the bus itself.

- **XIP module:** used to boot a program image directly from a pre-programmed SPI flash memory (hence SPI module has to be implemented).

- **Custom Function Subsystem (CFS):** The custom function sub system is used if one wants to implement custom co-processors, interfaces or external modules, such as HW accelerators, signal processing, AI applications and so on. It provides up to 64 32-bit memory mapped registers accessed via load/store operations.

- **Watchdog Timer(WDT):** last-resort timer in case of system stalls or severe problems. Hardware reset of the entire system. Optional interrupt can be triggered when this timer reaches half of its countdown. Timeout value can is set accordingly to a generic parameter (WDT_CTRL_TIMEOUT).

- **SPI and SDI:** Up to 8 dedicated Chip-Select signals with configurable FIFO size. Phase and polarity of the clock can be set by configuring the CSR dedicated to it. SPI frequency can be set via prescaler.
- **PWM:** up to 12 independent PWM channels with a maximum resolution of 8 bits each. The actual number of channels can be implemented via parameter (IO_PWM_NUM_CH) which, if set to 0, disable PWM module. Activation of PWM is done by setting a bit into a dedicated CSR. 3 registers are dedicated to defining the prescaler of each channel.
- **TRNG:** physical properties of the system (voltage, thermal or semiconductor manufacturing fluctuations) are used in order to generate random numbers. It is a platform independent architecture.
- **External Interrupt Controller(XIRQ):** Up to 32 external interrupt channels can be configured. Both trigger type and polarity can be modified.
- **General Purpose Timer(GPTMR):** 32-bit general purpose timer that can work in interrupt. It can work in 2 ways: single-shot (when it reaches the threshold, it generates an interrupt and then stops) or continuous mode (as it reaches the threshold, an interrupt is generated and the timer is resetted and restarted). Threshold value can be decided by setting its value into a memory mapped register. Another register stores instead the counter value.
- **NEOLED Interface:** Single wire interface that uses asynchronous serial communication for transmitting data (color data).
- **Two-Wire Interface (TWI):** I2C interface with configurable clock frequency (via prescaler). This interface can also work in interrupt mode, raising a

signal every time 8 bits are transmitted. Synchronization is performed via ACK/NACK signals (generated either by the peripheral or by the noerv module).

- **ONEWIRE:** Asynchronous half-duplex bus interface. External pull-up resistance and a tri-state driver (to be located in the top entity) for the line are required.

- **Machine System Timer(MTIME):** Memory mapped timer that can be used to set the CPU's machine timer interrupt.

- **Direct Memory Access(DMA):** this module allows to implement a direct connection with the data memory independently of the CPU. A single channel for both read and write operations is implemented

- **Stream Link Interface(SLINK):** Allows external information exchange using 2 independent RX and TX channels. It provides higher bandwidth and lower latency with respect to the external BUS interface.

- **Cycle Redundancy Check(CRC):** one of the most common error-detecting code Algorithm that can work on 8, 16 or 32 bits.

- **On-Chip Debugger:** Allows execution-based debugging through J-TAG port. It provides run-control of the CPU (halt, single-step and resume functionalities), indirect access to all core registers and address space, trigger module for HW breakpoints and execution of arbitrary programs during debugging.

- **SYSINFO:** Implemented by default, contains all the informations regarding the current configuraton of the system. All its registers are read-only

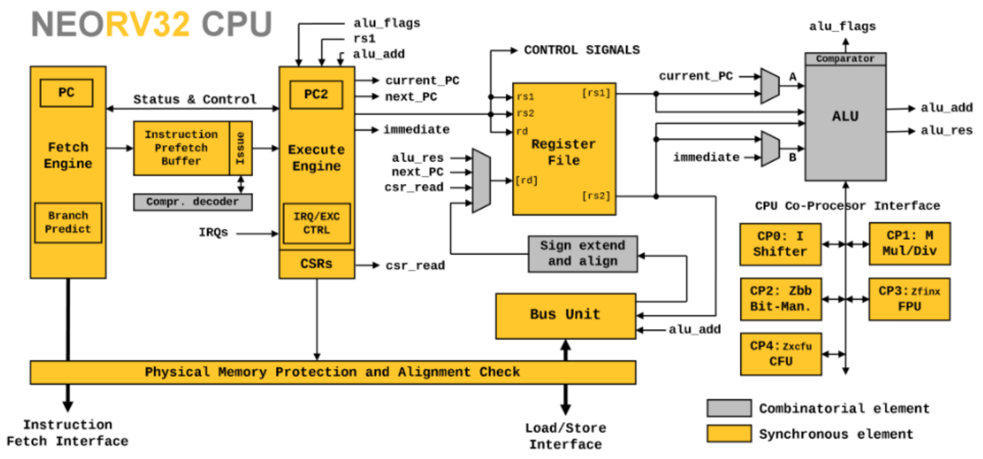


Figure 1.3: NEORV32 CPU Architecture [2]

The CPU resembles the typical RISC-V architecture described above but some modifications were performed.

The CPU is divided into only 4 sections:

- *CPU Control Unit*

  Contains the most essential elements of the system, oversees instructions fetching and sends out control signals to the entire system. Is divided into 2 main parts:

  Front-End: Instructions are fetched and fed to a FIFO (instruction prefetch buffer), whose size can be decided by the user. This FIFO allows a speculative approach, as well as splitting operations between front and back end, allowing them to run in parallel and improving performances.

  A simple branch prediction unit is also present in the front-end part, allowing the system to stop fetching further instructions while a jump/branch/call is in progress.

Back-End: All control and status registers (CSR), together with the trap controller, are here. The state machine controlling all CPU modules is described in this portion of the CPU too.

- *BUS Unit*

  Is in charge of managing access to DMEM when load or store operations are involved. It handles all the data adjustment when accessing sub-word data quantities (for example 16 or 8-bits data) and includes the optional PMP extension for checking all accesses to the memory.

- *RF Unit*

  32 entries (reduced to 16 if E extension is enabled), synchronous register file with 2 access ports: 1 read-only for retrieving rs2(second operand) and the other one for r/w operations of rs1 or rd(read first operand or destination register). The 32 registers are designed according to the regular RISCV register file, thus: x0 hardwired to 0, x1 for return address, x2 stack pointer, x3 as global pointer. The register for the Program counter is located inside the main Control Unit instead of an additional register in the register file. For floating point operations, integer register file is used and there is not a dedicated one.

  The register file can be mapped to an FPGA Ram block.

- *ALU Unit*

  Contains all the components necessary for performing integer operations on data. It allows the possibility to execute MUL and DIV operations if the corresponding extension is enabled (M extension, allow instantiation of a multiplier and a divider) as well as floating-point operations. The floating-point represents a subset of the actual and real F extension of the RISCV and in order to utilize it, a set of specific instructions have been written in C/asm language. The base integer ISA is implemented by default. Each extension has its own co-processor managing internal signal and correct synchronization.

The computation of addresses and other elements related to jumps/branches are also performed inside the ALU, so no adder/comparison logic is present outside of the ALU.

1.3 Processors Environment

The term "processor" is generally used to indicate an electrical device with computing capabilities. It can assume different forms, from a simple microprocessor to more specific devices, such as Graphic Processing Unit, also known as GPU, Deep learning processors, physics processing unit and so on. In fact, processors are used in different environments; these can be mainly subdivided into 2 categories: non-safety-critical environments and the safety-critical ones. The latter one represents those situations in which a failure in the computing unit does not lead to significant or catastrophic consequences. This means that processors can be developed to optimize processing capabilities, power consumption or efficiency.

The former one refers instead to those situations in which a failure of the system might cause either death or serious injury to people, loss or severe damages to properties or any environmental harm. In this case, processors must be developed in a specific way to avoid the consequences listed above; hardware or software redundancy are some of the most commonly adopted solutions. Some examples of a safety-critical environment/system includes medicine applications, life support systems, telecommunications and space applications.

1.3.1 RISCV in Space

When speaking about the space environment, both from the point of view of hardware and software, reliability has to be ensured. Having a technology that is resistant to the common problems that can affect devices in space is a requirement and the deeper we know and can interact with this technology, the better we can ensure good performances. Taking into account these considerations, an open-source technology, such as the RISCV one, is exactly what is required. This is an extremely fundamental step forward with respect to those technologies and IP that are restricted and cannot be inspected, thus limiting improvements. Modern SoC processors might be extremely efficient and fast, but when it comes to space applications these characteristics are not enough. Instead, using a technology in which we can figure out what kind of module is more essential than others, perform suitable hardening actions on their internal structure and improve their performances under specific conditions is exactly what is needed.

Also being the RISCV based on an ISA that will not change in the future is an important aspect for space applications, which are usually based on long term expeditions: executable codes and other types of software running on the RISCV ISA will be directly reusable in new technologies, allowing a high level of portability and compatibility with more efficient solutions.

Moreover, the utilization of FPGA technologies allows processors to be mapped in a precise and reconfigurable way, thus ensuring versatility and extremely high level of efficiency. Having the possibility to directly modify the bitstream used to program the device through the utilization of specific tools allow us to select specific portion of the processor to be analyzed; being the ALU, the RF and the CU some of the most essential elements in a processor, directly interacting with them is what allows to achieve the highest possible level of reliability at the minimum cost and in the most efficient way possible.

This thesis works aims exactly at understanding how an FPGA mapped processor such as the NEORV32 will behave when SEU affect the configuration memory by changing its content and so the routing of the processor itself; in particular, understanding which of the modules inside the CPU of the RISCV are the most affected and especially how they will respond in case of errors, either halting the system or simply leading to wrong computation results. This will allow us to understand what kind of software or hardware hardening actions have to be adopted.

Chapter 2

State of the Art

In this chapter, an overview related to all the previous work on this subject is performed. Analysis on works related to reliability of open-source processors, with particular attention to the RISC-V technology, when applied to the space environment is evaluated in a deep way.

Many research centers and companies have started to analyze the impact of the introduction of the RISC-V in space applications but only a few of them have explored the fault injection concept and the reliability analysis of this architecture in this field.

A preliminary analysis has been conducted in [3] where authors have evaluated the possible impact that RISCV might have in the space exploration world from the security point of view. Also, in [4] a roadmap related to the application of the RISCV in space is presented.

Few works have conducted a detailed analysis on the reliability of the RISCV architecture. Authors in [5] surely proposed a perfect starting point. A fault injection campaign similar but less invasive than the one performed in this thesis work has been conducted in collaboration with ESA. However, only a small number of injections have been performed, thus not evaluating in a deep way how resistant the device is. Also, no specific portion of the processor CPU were targeted, making it impossible to understand what hardening actions might be preferrable to improve the architecture. Understanding when a software rather than a hardware hardening technique is preferrable (or vice-versa) might lead to significant improvements. Also, in [6] the reliability of the RISCV architecture in presence of an operating system has been investigated. In my case, however, a BareMetal scenario is considered.

The work described in [7] provides a further theoretical analysis on the security and reliability of the RISCV architecture while authors in [8] proposed a simulation of how a RISCV device will behave in presence of SEU faults, providing new measurement method for improving the reliability of software operations.

Some works [9, 10] have been carried out on the evaluation of the reliability of RISCV architectures using different simulation or fault injection tools. Also, in [11–13] a characterization of the FPGA mapped RISC-V, under specific radiation effect, has been performed.

Further analyses and solutions were also proposed taking into considerations hardened version of the RISCV architecture: authors in [14] designed a TMR hardened solutions RISCV suitable for space applications while in [15] another new architecture, based on the Cobham open-source NOEL-V design, has been developed. Specific analyses were further conducted in [16–18] on the effectiveness of the adopted hardening solutions, either hardware or software ones.

A fault injection campaign has also been conducted in [19] where the effective improvements given by the utilizations of these hardening techniques are underlined.

Obviously, being the RISCV mapped onto a SRAM-based fpga, further analysis has been conducted in this field. Authors in [20] provides a survey on fault tolerance in FPGA devices while an innovative approach for evaluating error rates has been proposed in [21]. The work of [22–24] instead provides analysis of the SEU effects in SRAM-based FPGA and platforms that can be used to physically induce SEU errors in configuration memory. In particular, PyXEL software has been used inside this thesis work.

Chapter 3

Background

3.1 Radiation Effects

When operating in harsh environments, being the space one a perfect example, electronic devices are often subjected to physical phenomena that might modify their behavior. In this section I will introduce and describe the radiation concept and how they can affect our circuits.

A radiation is defined as the emission or transmission of energy between particles when they interact one with the other. It is often categorized in two groups: ionizing and non-ionizing.

The latter ones are characterized by particles with lower energy levels, thus causing a simple excitation instead of generating charged ions. Examples are microwave, infrared and radio waves. Those particles are generally characterized by lower frequency ranges and their consequences have been study in relation to human body. Given these considerations, they are not usually considered in the space field.

Ionizing radiations, on the other hand, are a more interesting case of study. They include all those particles that possess enough energy to ionize atoms by removing electrons from them. Some examples are:

- *Alpha particles:* composed by two protons and two neutrons bounded together, those are usually generated by the alpha decay process of heavier atoms, usually coming from highly radioactive nuclei.
- *Beta particles:* high-energy and high-speed electron or positron emitted through beta decay process.
- *Cosmic particles:* commonly produced by cosmic rays interacting with the earth's atmosphere. Those are high-energy particles that can move at nearly the speed of light. Cosmic particles include muons, mesons and positrons.

When radiations occur, several consequences might affect the digital device and its physical silicon layer [25]. In particular, high-energy particles can cause a modification in the arrangement of the atoms in the crystal lattice, thus modifying the internal structure and properties of the material at the base of the integrated circuit. This phenomenon is also commonly known as **Displacement ionizing Dose**. Also, these particles, especially those with lower energy, can lead displacement of charges in the CMOS technology. When a particle passes through a device, it can cause electrons to move away, thus leaving in their place positive charges, also known as holes. When this happens inside the gate of a CMOS transistor, due to the technology structure, holes can find their way towards the oxide and remain trapped there due to the technology structure, holes can find their way towards the oxide layer and remain trapped there, causing a gate biasing and modifying the threshold voltage levels; depending on the severity of the radiation, this consequence can also cause the transistor to be permanently active and never switching off. This phenomenon is called **Total Ionizing Dose**.

3.2 Single Event Effects

Single Event Effects, also known as SEE [26–28], are one of the most common fault models that can be used to represent previously discussed radiation consequences.

SEE are further classified into 2 categories: Hard and soft errors. Hard errors are the ones that can permanently damage the device structure. Those are SEL and SEGR [27]. SEL represent a modification in the silicon structure of the device; this can lead to variation in the flow of current and so in a permanent damage inside the integrated circuit itself if the power is not turned down sufficiently fast.

A SEGR is caused by a high energy particle that strikes a transistor, MOSFET in particular, and cause a break in the insulating oxide portion of the device. This can lead to breakdowns of the entire system due to high quantities of currents that can flow in the newly created path.

Soft errors can cause instead an unwanted behavior that will disappear after a certain amount of time. They are further divided into SET, SEU and SEFI.

SET [29] can be described as voltage spikes in a specific point of an integrated circuit. If this fault propagates to a memory element, such as a latch, it becomes a SEU.

A SEFI is a soft error that can cause a malfunction, such as a reset or a lock-up, in the device but does not necessary require a hard reset of the system to be solved.

3.2.1 Single Event Upset

SEU faults were first experienced along some nuclear experiments in the 50s while, during the 60s, further electronic anomalies were experienced in space environment. The first information related to this kind of error were gathered and then published in the first paper related to SEU in 1975.

SEU refer to the alteration of the content of a memory element due to a high energy particle interacting with it. They can be solved by re-writing the data inside the memory while suitable techniques, such as hardening or error correcting memories, can be adopted to prevent them.

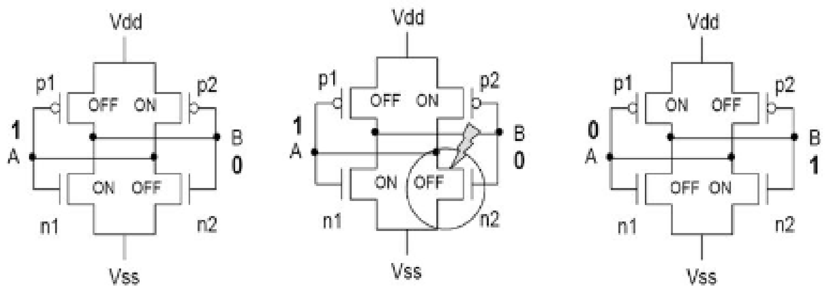


Figure 3.1: SEU effect

These errors can impact in a severe way an SRAM-based FPGA since they can modify one or more bit of the configuration memory, thus leading to a modification in the routing of the device itself. Given that each bit in the bitstream will be used to configure a specific CLB in the FPGA, a change in one of them can lead to 4 consequences [30]:

- *Antenna Fault:* a path is connected to a floating segment, leading to an unknown value at the output.
- *Open Fault:* the SEU will cause a path to become open, creating a disconnection

in the circuit.

- *Conflict Fault:* it refers to a short circuit generated between two signals normally disconnected. Unidentified value is generated at the output.

- *Bridge Fault:* The selection bit of multiplexer is compromised; the output will be connected to the wrong input value.

3.3 Technology Background

The NEORV32 processor has been provided both with a VHDL and a Verilog description, both implementable on almost any kind of FPGA [31]. Different versions of the SoC are available and can be freely modified according to the user requirements. FPGA, which stands for Field Programmable Gate Array, is a highly reprogrammable integrated circuit whose configuration is specified through an HDL language. Stemming from Programmable logic devices and programmable ROMs, FPGA were introduced on the market during the 80s and, thanks to their innovative features and high level of re-programmability, they immediately started to dominate the digital electronic market. Starting from telecommunications and networking fields, their areas of application have rapidly expanded, especially during the 90s. Nowadays, they are often used as hardware accelerators and in safety critical environments. FPGAs are internally composed by bi-dimensional structures of combinational logic called configurable logic blocks, which are interconnected together through switch matrixes. I/O blocks are present to allow the fpga to communicate with the external world.

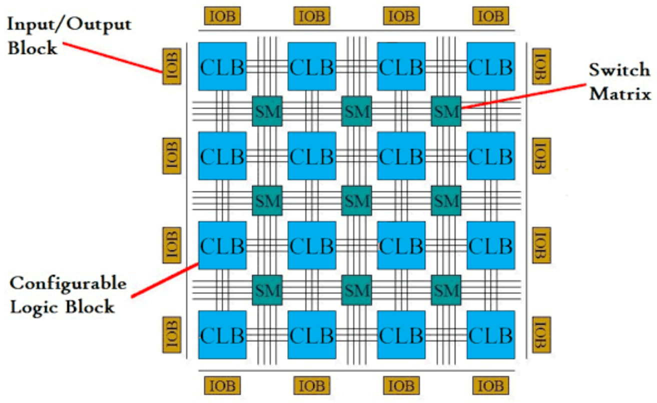


Figure 3.2: FPGA structure

In Xilinx FPGAs, each CLB has a switch matrix associated to it and is internally subdivided into 2 slices; these are further composed by: 4 Look-Up Tables of 6 inputs each, 8 flip-flops, some logic gates and carry logic blocks. An example of Xilinx FPGA CLB is reported below.

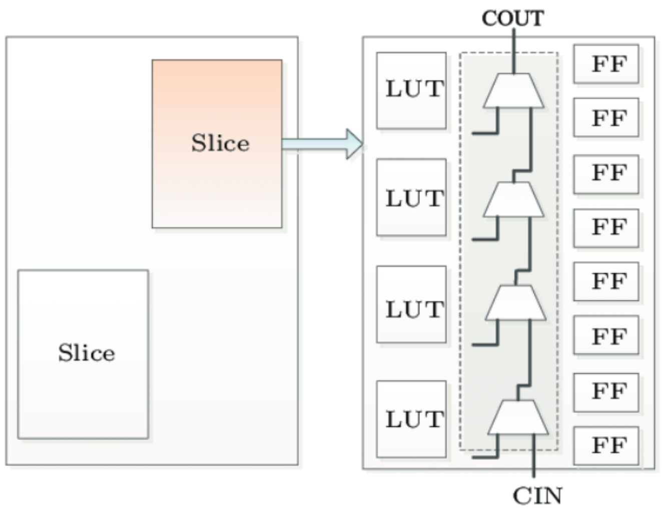


Figure 3.3: Xilinx CLB structure

In addition to CLB and switch matrixes, Xilinx FPGA also includes special blocks such as memory elements, RAM, and digital signal processors.

3.3.1 PYNQ-Z2 Development Board

The PYNQ-Z2 [32], based on Xilinx Zynq SoC, is a board designed for the Xilinx University Program to support Python Productivity for Zynq framework and for embedded system development.

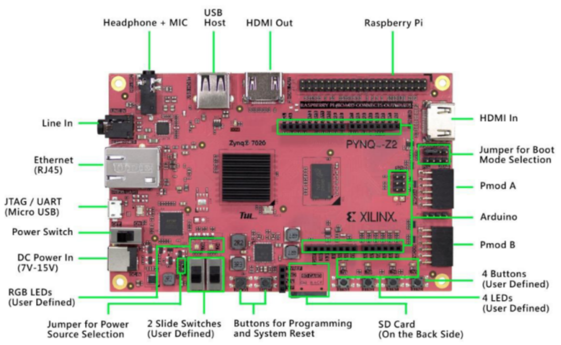


Figure 3.4: PYNQ-Z2 Board [32]

It provides the following features:

- The ZYNQ XC7Z020-1CLG400C SoC, which it includes a dual-core ARM Cortex-A9 processor running at 650MHz and a programmable logic element with up to 13300 logic slices, 220 DSP slices, 630KB RAM and one on-chip Xilinx analog-to-digital converter. The PL can be programmed via JTAG, Quad-SPI flash and MicroSD card.
- 512MB DDR3 RAM with a 16-bit bus and 16MB Quad-SPI Flash.
- 1 Ethernet port, 2 HDMI ports, 1 jack port for audio and microphone data and 1 USB host port.
- Arduino, 2x Pmod ports and Raspberry-Pi expansion connectors.

- Switches, pushbuttons, LEDs and RGB LEDs.

The PYNQ-Z2 board can be supplied using either a Micro-USB, an external power supply or a battery; the power source is selected by setting specific jumper on the board. Another jumper, located on the right side of the board, can be used to select one of the 3 boot modalities. In this thesis work, JTAG modality is adopted. A 50MHz oscillator is present on the board and used to supply the PS subsystem. This allows the processor to operate at a maximum frequency of 650MHz. Also, an external 125MHz is connected to one of the pins of the PL, allowing it to work independently from the PS.

The PS also incorporates an AXI memory port interface, a DDR controller, the associated PHY, and a dedicated I/O bank.

3.3.2 Xilinx Design Flow

Vivado Design Suite is a software program designed by Xilinx for synthesis and analysis of HDL designs. It provides different design management functionalities, such as logic simulation, I/O and clock planning, design validation, power and timing analysis and modification of implementation results, programming and debugging.

Vivado allows two different ways of achieving all these tasks [33]: RTL-to-bitstream design flow or System-level integration flow, which concentrates on the utilization of IP-centric design and C-based design to achieve same results as the first procedure

The approach used during this thesis work is the first one; the Bitstream is a binary file that contains the programming data corresponding to the design that we want to implement on the FPGA itself. Specifically, it is used to inform the FPGA about all the routing information, the connection between blocks and the internal configuration of each CLB.

To allow even greater compatibility with the industry world, Vivado Design Suite supports the following industry design standards: Tcl, on which Vivado itself is based, AXI4, IP-XACT, Synopsys design constraint (SDC), Verilog, VHDL, VHDL-2008, SystemVerilog, SystemC, C, C++.

Also, it allows two further possibilities: independently from the selected flow, Vivado can be used either through a Graphical User Interface (GUI), also known as Vivado Integrated Design Environment (IDE), or through Tcl scripting and commands. The former one allows Higher level of interaction and a detailed view of each step of the design flow of the entire project. The latter allows an overall faster design procedure, especially useful when performing repetitive operations.

RTL-to-Bitstream Design Flow

The first step requires the creation of a new project, the board characteristics and the specification of the sources. These include the files containing the HDL description of our design (either Verilog, VHDL or SystemVerilog files can be submitted) and the constraint files in the XDC format. These are particular types of files used to impose rules that must be respected during the physical implementation of the design, for example I/O connection specifications, specific cell and block placement in the FPGA, output files constraints and so on.

Vivado also allows the possibility to specify source and constraint files after the project creation; however, boards type and specifications have to be defined immediately.

The next step is to define the block diagram of the device, which allows the integration of IP as standalone modules and simplify the final design definition. Some predefined IP blocks are always available in the Vivado IP catalog; however, it is possible to package custom IP following the IP-XACT protocol, export and have them available to be used in other projects. IP can include different kinds of logic, from combinational elements to DSP, embedded processors, modules or C-based algorithm designs. They use AXI4 interconnection protocol to communicate one with the other to allow faster system-level integration.

This step is performed using the Vivado IP integrator environment, in which it is possible to interconnect together different IP, in an autonomous or manual way, set their properties and characteristics. The final block design is then validated to ensure that all connections are performed properly, and no essential element is floating or configured in a wrong way. Also, the final block design, as well as single IP blocks, can be packaged and exported.

The third step consists of the functional simulation. An ad-hoc testbench must be designed by the user to test the functionalities of our design and verify the correctness of its behavior. In case of memory elements, Vivado provides a way to display their content; for other generic signals, a waveform displaying their behavior over simulation time is given.

Subsequently, the synthesis step is performed. This is where the XDC file is used, and the physical constraints are imposed on the design. This process allows to move from a simple RTL description, using an HDL language, to a physical design, in which every element is mapped into logic gates and Boolean expressions. Together with constraint files, also technological libraries are used during this step, returning at the output the final netlist of the design. Different synthesis processes are available, with higher or lower levels of optimization and computation capabilities; by default, Vivado utilizes out-of-context type, also known as bottom-up design flow. This approach allows to synthesize complex designs in an easy way, characterized by a large number of modules and IP. The netlist that is generated at the output of the synthesis process is then used during top-level implementation to fasten the procedure and reduce runtime. It is important to notice that Vivado Design Suite also allows the utilization of external synthesized netlists; however, if IP cores from the vivado catalog are present in the design, then they must be synthesized using Vivado synthesis tool.

Following the synthesis, Placement & route procedure takes place. During this phase, the netlist is physically mapped onto the FPGA resources of the specific board selected at the beginning of the procedure. If any of the constraints, either physical, logical or timing ones, cannot be satisfied, then the implementation procedure fails. In this case it is up to the user to solve any problem and rerun the synthesis process again. During the implementation phase, Vivado Design Suite allows to physically map specific portions of the netlist onto specific elements of the board. This feature represents one of the elementary points on which this thesis work is based on.

At the beginning of the implementation process, suitable optimizations are performed onto the netlist to optimize power, timing, reduce wiring at the minimum and avoid excessive congestion, spreading as much as possible the cells over the available area while still meeting other constraints. Then the design is placed onto the target device. Before the routing procedure, further power and physical optimization procedure can be performed, however these are optional steps. A last optimization phase is started when the design is fully routed. This concludes the implementation phase.

It is important to notice that the output of any of the 2 previous stages is still a netlist that can also be exported into a Verilog or VHDL format. This is extremely important, since it allows to further perform logic simulation on the design that will then be physically placed onto the FPGA. Suitable testbenches have to be generated and adapted to the circuit.

After the design has been implemented, Vivado proceeds by generating the actual Bitstream, which will then be uploaded onto the FPGA through the Hardware Manager.

ARM cores programming

This thesis work required, outside the FPGA, also the utilization of the ARM processor part. Its programming is not performed by Vivado but uses a specific program called Xilinx Vitis. Stemming from the Vivado design, after the bitstream generation, it is possible to export the hardware platform, containing all the information related to our design, such as the memory mapping, and feed it to Vitis. It will automatically create an environment that allows to program the cores, upload all the files onto the board and define all the parameters required by the system to work in the correct way. The ARM cores can be programmed using C-language and several examples' programs are already present when the platform is created. Vitis will also generate all the Tcl scripts for programming the board; these files can be launched through the utilization of XSCT program.

Chapter 4

Fault injection Platform and NEORV32 reliability analysis

In this chapter I am going to describe the fault injection platform that has been developed to analyze the reliability of the RISCV. The objective was to design a fast and efficient program able to verify how single SEU in configuration memory affects the behavior of the FPGA mapped NEORV32. In particular, the internal elements of the CPU, these being the ALU, CU, RF and BUS, were tested during this thesis work; additionally, a tool for evaluating the results obtained from the fault injection campaign was designed. Both programs were implemented using python3 programming language.

4.1 Vivado Block diagram and FPGA setup

The first and most fundamental step of this work was the definition of the NEORV32 design that had to be tested, together with the generation of the bitstream for programming the FPGA. Depending on the displacement of the CPU modules in the FPGA, specific addresses had to be generated; perhaps, it was fundamental to define the design structure. Also, specific elements outside of the NEORV32 were required to be added to allow the possibility to interact with the FPGA and perform the injections in an interactive way instead of modifying the bitstream before uploading it. In the image below, the final Block Diagram realized using Vivado is reported.

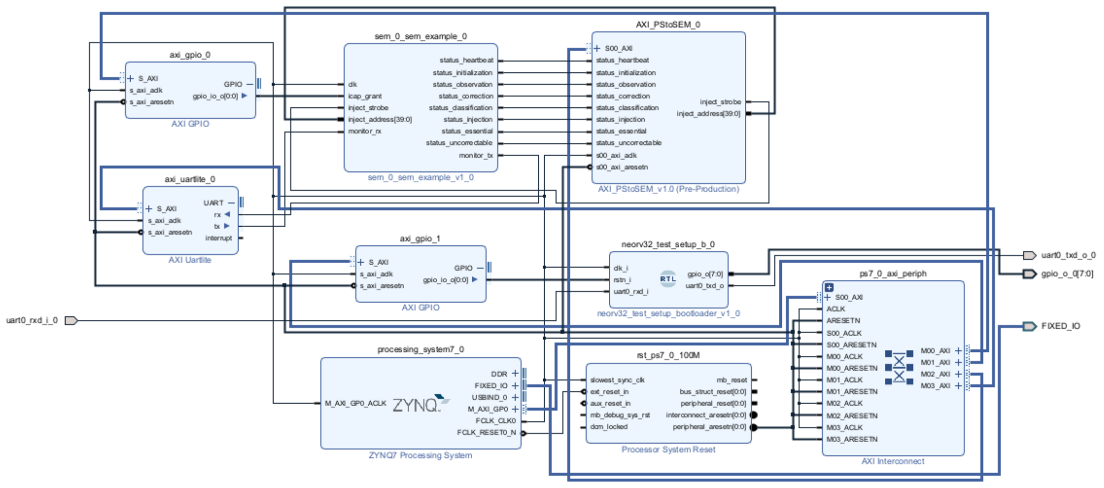


Figure 4.1: Final Vivado Block Design

It is possible to observe that the NEORV32 only represents a small portion of the whole design; most of the components are either required for AXI configurations or for fault injection purposes.

4.1.1 NEORV32 Block

The design of the NEORV32 selected for this case of study included only a small portion of the available modules; other than the CPU, the Bootloader, GPIO, and UART components were used. Their purpose has been exhaustively described in the first chapter.

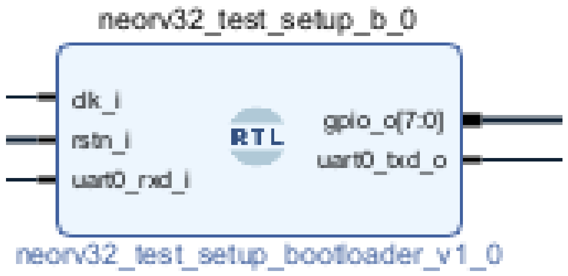


Figure 4.2: NEORV32 IP

The 8 GPIOs were externally connected to the LEDs already present on the PYNQ-Z2 board. They were mainly used for debugging purposes when the CU module was under test.

The reset signal was directly connected to an AXI peripheral; in this way it was possible to manage it using internal AXI peripheral registers and no direct interaction from the user was required when the NEORV32 had to be resetted.

The clock signal was supplied directly from the ZYNQ processing system, this being the clock of the board. In this case the NEORV32 was running on a 100MHz clock signal, as required by the user guide specifications [31].

The 2 UART signals, which represented one of the most fundamentals requirements for this project, were mapped onto external pins; for this purpose, pin1 and pin2 of PMODA ports of the PYNQ-Z2 board were used, according to [32].

4.1.2 AXI Interfaces

The AXI interfaces represent, together with the SEM-IP, one of the 2 most essential elements required by this thesis work. The AXI is an on-chip communication protocol developed by ARM and used by Xilinx in most of its boards. It allows exchange of information between 2 devices, the AXI Master and the AXI Slave; for this purpose, 5 channels are defined: 2 are used for reading operations of addresses and data, 3 are used for writing operations of address, data and response.

The AXI interface allowed to map pins and control signals onto specific memory mapped registers; these locations can be accessed through the usage of XSCT software. By mapping bits of these registers through the usage of specific masks or by reading them, it was possible to control the reset signal of the NEORV32 and the control signals of the SEMPIP or verify its status thanks to the monitor, using serial communication. The developed fault injection platform made use of this feature to allow performing completely autonomous operations on the whole device; a total of 4 AXI interfaces were used:

- 2 AXI-GPIO for managing the icap_grant signal of the SEMPIP and the Reset signal of the NEORV32.
- 1 AXI-UARTLITE for managing the monitor signal coming out of the SEM-IP.
- 1 AXI-PStoSEM for managing the control signals of the SEMIP.

Many AXI IP are already present and ready to be used in the Vivado catalog; however, the one used to interact with the SEM-IP, required some modifications to be adapted to our design.

In particular, the inject_strobe signal, the one controlling the actual state evolution as described in the following section, had to be active only for one single clock cycle, otherwise unwanted operations might be performed. Since the communication through XSCT is slower than the actual sampling frequency of the inject_strobe signal by the SEM, it was possible that the bit controlling this signal remained at one for a longer period of time; thus, the VHDL prototype of the AXI block had to be adapted to this situation. In particular, if the corresponding bit in the AXI register goes to 1, independently on the duration of this event, only a single pulse is sent to inject_strobe input of the SEM itself, guaranteeing the correct behavior. This was achieved using VHDL processes.

4.1.3 SEM IP Block

The Soft Error Mitigation controller, also known as SEM, is a device used to automatically detect and correct soft errors in configuration memories of Xilinx FPGAs [34].

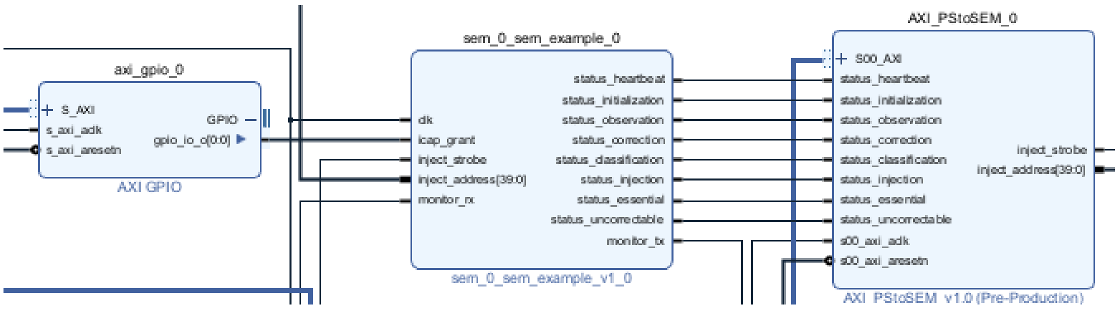


Figure 4.3: SEM Interface

It provides the following functionalities:

- Typical detection latency of 25 ms.
- Error correction, selected among 3 different methodologies: repair, enhanced

repair or replace.

- Error classification capabilities.
- Error injection capabilities.

Error correction, injection and classification are optional features and they can be enabled or not during the IP customization. Vivado directly provides a design example for the SEM-IP; it is the one used during this thesis work.

The SEM controller operates by switching between different states, in an FSM style, depending on the command it receives.

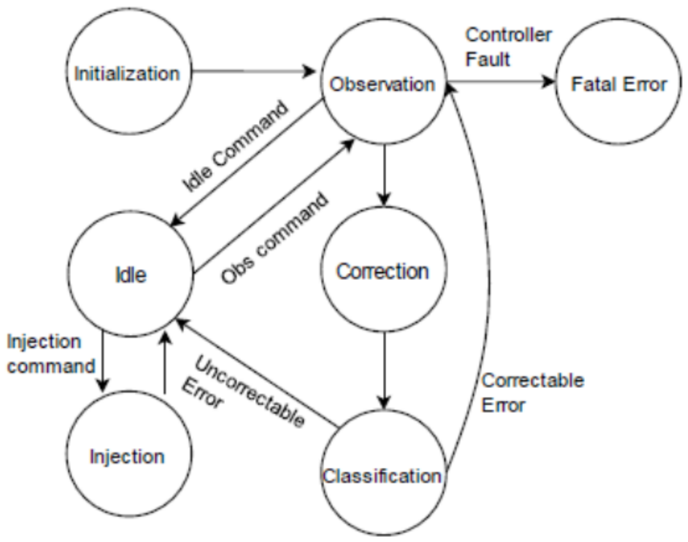


Figure 4.4: SEM State Diagram

- **Initialization:** this phase is executed only once when the system is initialized, or in case of a software reset request. The controller initially polls the icap_grant signal to determine if it can access the Internal Configuration Access Port (ICAP). Zynq-7000 devices need to handle the icap_grant signal in a specific way to initialize the SEM correctly and avoid unwanted behaviors. To correctly manage this signal, the AXI_gpio_icap_grant block of the Block Design is used.

- **Observation:** during this stage, the SEM monitors the FPGA and looks for errors. There are 2 possible scenarios: if no error is detected, then it can only move to the Idle stage or report diagnostic information using the status report command; any other command is ignored. If an error is detected, the controller gathers further hardware information and then attempts to correct it by moving into the correction state.

- **Idle:** this state is pretty like the observation one, without FPGA checking for errors. However, error injection and software reset commands are available only in this stage. The latter one allows to move into the injection state; the former one allows instead to perform a complete reset of the SEM-IP, initializing again the device.

- **Injection:** during this state, the controller performs an error injection in the configuration memory. The injection is a simple bit-flip at a specific address in the FPGA. The address must be specified as part of the command. While in this stage, it is also possible to perform multiple error injection at the same time; this is achieved by launching the injection command multiple times with different addresses. If the same address is given to 2 consecutive injection commands, then 2 bit-flip are performed at the same location, thus restoring the initial configuration. This error will obviously not be detected. From the injection state it is only possible to go back to the Idle state and then, eventually, to the Observation state for correction.

- **Correction:** during this stage the controller attempts to correct the error. The action that takes place depends on the type of correction that is enabled. In any case, a report is generated and then the system moves, if enabled to do so, to the classification state. If the error cannot be corrected, specific messages are reported on the monitor interface.

- **Classification:** this stage allows to classify errors depending on the outcome of the correction phase. All uncorrectable errors, meaning those that cannot be located, are classified as essential errors. In case of this kind of error, FPGA must be reconfigured. In case of correctable errors, classification depends on the configuration of the controller: if the classification stage is disabled, all errors are classified as essential; otherwise, the SEM retrieve further information from external memory and then it attempts to find the essential ones. After this procedure is completed, the controller automatically switches again to the observation stage.

- **Fatal Error:** this represents one of the worst cases for the SEM operational phase. The controller detects an internal inconsistency that cannot be solved. In this case, the whole FPGA has to be reconfigured. It is a very unlikely condition that will cause the SEM to halt; it can happen when the configuration memory portion in which the controller is mapped has been corrupted, for example due to a fault injection. This condition has also been managed in the developed platform.

Furthermore, the SEM is provided with different interfaces and ports:

- *Clock interface:* used to provide the clock to the design. Different frequencies can be supplied depending on the logic used. For Zynq-7000 the maximum operating frequency is 100MHz.

- *Error injection interface:* it consists of an input bus and a strobe. It is used when the error injection feature is enabled. The strobe signals that a valid data has arrived on the bus line, which contains the address and other configuration bits for performing the injection.

- *Status interface:* it is composed of 8 bitsand signals the state in which the controller is; additional information about previously corrected errors are reported here.

- *Monitor Interface:* this module is always present and can be used to interact with the SEM, sending commands and analyze its response. It is composed by 2 signals implementing a RS-232 compatible transmission. No parity bit is present and only 1 stop bit is used. The baud rate is set to 9600 baud; however, this value can be set by modifying a parameter in the VHDL description of the SEM, according to the following formula:

  $$V\_ENABLETIME = rnd\left(\frac{InputClockFrequency}{16 * Nominalbitrate}\right) - 1$$

  For this thesis work, this parameter was set to 324, leading to a baud rate of 19200. This allows the injection procedure to be executed in a fast way while also allowing the ARM core not to lose any of the information transmitted by the SEM.

To control the SEM, specific commands must be sent to it; this was done by setting the bits of the AXI interface with specific masks, according to the documentation. These commands can also be sent by using the monitor, but in this case the AXI approach was a simpler and faster solution.

The 3 AXI peripheral registers to control the SEM are located at 0x41210000, 0x41210004 and 0x41210008 and are called slv_reg0, slv_reg1 and slv_reg2 respectively. They are composed of 32 bits each. Out of all bits of slv_reg2, only the LSB is used. It is the one that controls the inject strobe signal and so it is switched to 1 and 0 every time a new command is sent to the SEM. Slv_reg0 implements the first 32 LSB of the inject_address signal. The remaining 8 bits are instead contained in the LSBs of slv_reg1 register.

Of all the 40 bits contained between slv_reg0 and slv_reg1, the 4 MSBs of the address signal are used to specify the command that must be executed, thus the state in which the SEM has to move. The remaining bits represent a "don't care" condition, except for the inject phase in which they represent the injection address.

- **Entering IDLE State:** To enter the idle state, it is necessary to set the 4 MSBs of the address signal to "1110" or "e" in hexadecimal form.

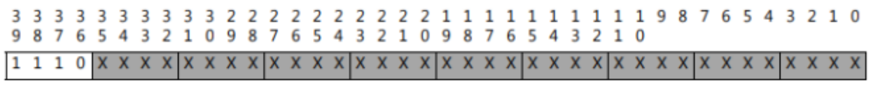


Figure 4.5: Idle command [34]

- **Entering OBSERVATION State:** To enter the observation/correction state, the bits must be set to "1010" or "a" in hexadecimal.

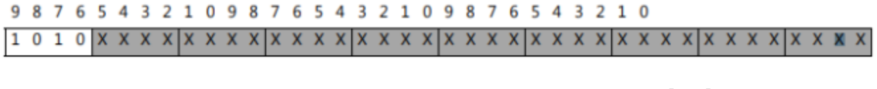
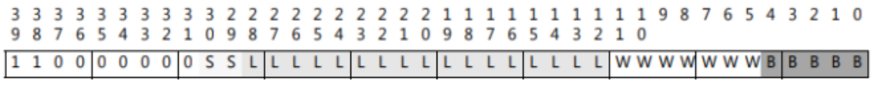


Figure 4.6: Observation command [34]

- **Entering INJECTION State:** The injection phase is reached when the bits are set to “1101” or “c” in hexadecimal. Also, in this case all the other 36 bits, specifying the injection address, have to be defined. The bit defined as S, L, W or B specify respectively the SLR number, the linear address frame, the word address and the bit address. Those refer to address generation and extraction.



39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	S	S	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	W	W	W	W	W	W	W	B	B	B	B	B

Figure 4.7: Injection command [34]

- **Performing SEM reset:** The reset state is used to restart the SEM in case of Halt. The bits are set to “1011” or “b”.

The response of the controller, displayed through the monitor, is then captured by the AXI_UartLite and read by the ARM core. It is then analyzed in the python program to verify that each step was performed correctly.

4.1.4 Processing system and Reset IPs

The processing system IP is the software interface around Zynq 7000 PS. In this thesis work it is mainly used to deliver the 100 MHz clock to the entire system; however, it provides different functionalities such as enabling or disabling peripherals and AXI ports, MIO and DDR configurations, security and isolation configuration. The reset IP is used for managing reset signals inside the system.

4.1.5 NEORV32 FPGA Implementation

Another important step in the implementation flow was to map the NEORV32 CPU onto specific portions of the FPGA, as described in previous sections. This was achieved by using a tool provided by Vivado Design Suite, the PBlocks. In fact, it is possible either to let Vivado auto manage cells placement over the whole area of the FPGA or to specify, using PBlocks, specific portion onto which cells have to be inserted. PBlocks are defined by imposing specific constraints in the XDC file; however, for simplicity, Vivado allows the user to utilize the GUI to create them over the area of the FPGA. In this case, Vivado will automatically update the constraint file once the design is saved and the bitstream generation procedure is started. Each PBlock is a rectangle with variable dimensions and characterized by X and Y coordinates over the implementation area. For this project, different PBlocks have been used: 4 of them contain the 4 elements of the CPU to be analyzed, 1 contains the SEM-IP cells and the remaining ones contains all the cells and memory elements of the design that were not tested. When designing using PBlocks it is extremely important to consider timing constraints. If those are not met because elements are displaced too far one from the other, implementation can fail.

In the image below, the final implementation of the design is reported. It is possible to see that the whole structure occupies only a small portion of the total space available. The units under test are highlighted as follows: CU in green, RF in red, BUS in purple and ALU in yellow.

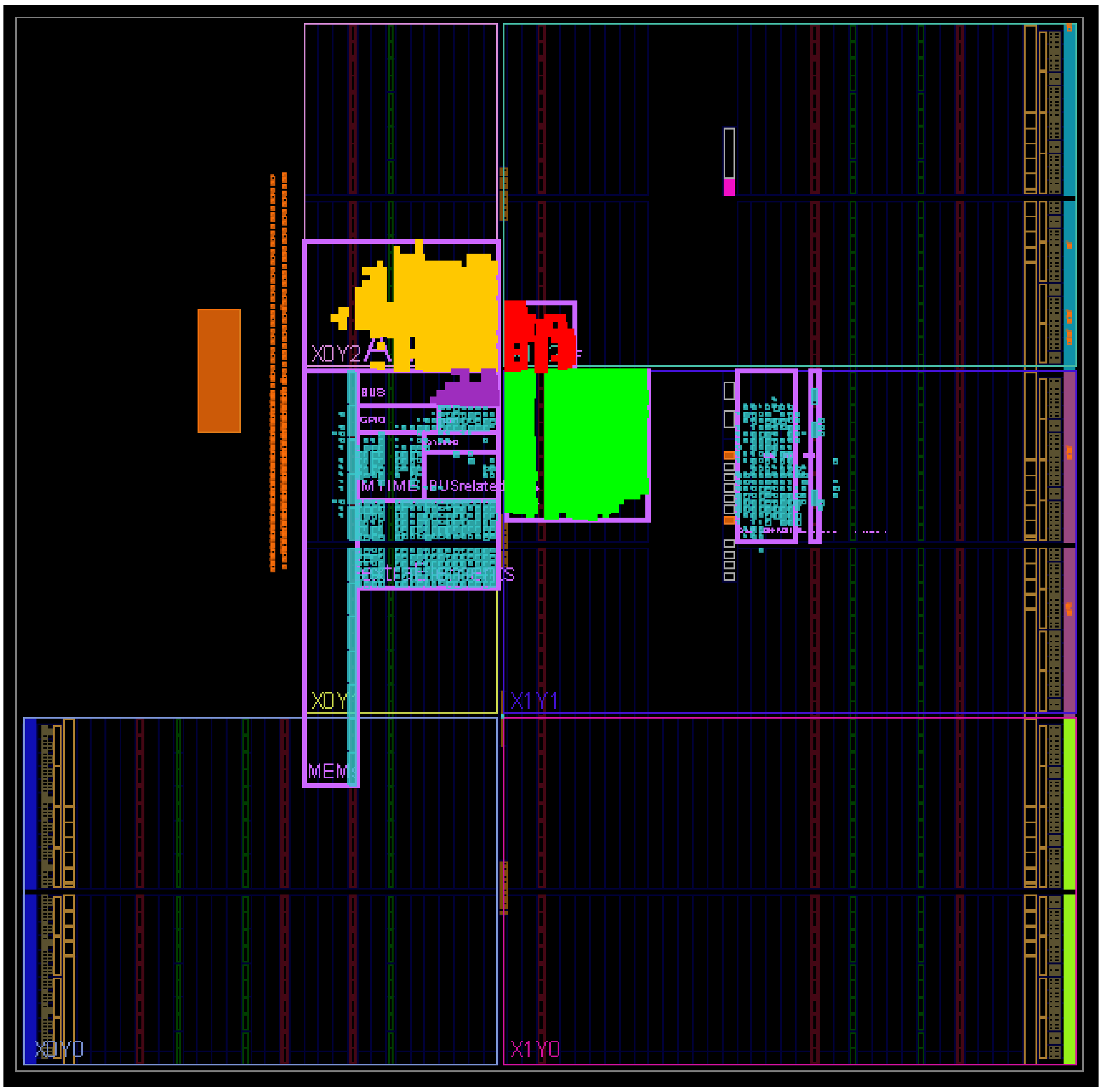


Figure 4.8: NOERV32 Implementation Design

4.2 ARM cores software and Vitis

As stated in previous charters, also one of the ARM cores of the PYNQ-Z2 board was used. Its purpose was to run a small portion of code able to continuously read data coming out of the monitor of the SEM-IP and delivering it to the PC to which the board was connected through USB cable. This procedure allowed me to use only one UART-USB converter. The software consists of a simple polling-based cycle: initially, the UartLite module is initialized and checked, then the main cycle starts. It continuously checks the buffer in which the data coming out of the SEM-IP is stored; if something is present, it sends it to the computer, 1 byte at a time. Since the ARM core works at an extremely high frequency compared to the UART transmission rate of the SEM, no data is lost even if interrupts are not used.

```
    int Status;
    unsigned int SentCount;
    unsigned int ReceivedCount = 0;
    int Index;
    /*
     * Initialize the UartLite driver so that it is ready to use.
     */
    Status = XUartLite_Initialize(&UartLite, DeviceId);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    /*
     * Perform a self-test to ensure that the hardware was built
correctly.
     */
    Status = XUartLite_SelfTest(&UartLite);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    /*
     * Initialize the send buffer bytes with a pattern to send and
    the
     * the receive buffer bytes to zero.
     */
    for (Index = 0; Index < TEST_BUFFER_SIZE; Index++) {
        SendBuffer[Index] = Index;
        RecvBuffer[Index] = 0;
    }
        /*
         * Receive the number of bytes which is transferred.
         * Data may be received in fifo with some delay hence we
        continuously
         * check the receive fifo for valid data and update the
        receive buffer
         * accordingly.
         */
        while (1) {
            ReceivedCount = XUartLite_Recv(&UartLite, RecvBuffer,
    TEST_BUFFER_SIZE);
            if (ReceivedCount != 0){
                if (RecvBuffer[0] == 10 || RecvBuffer[0] == 13){
                    putchar('\n');
                } else {
                    putchar((char)RecvBuffer[0]);
                }
            }
        }
```

4.3 Developed Fault injection platform

After defining the final NEORV32 design and implementations details over the FPGA, the injection platform was developed. For this purpose, python3 programming language was used and important tools from Pyxel[24] library were imported. This allowed the program to directly interact with XSCT, allowing me to program the board, start or reset the NEORV execution of the benchmark programs and send commands to the SEM controller.

XSCT is a Xilinx interactive and scriptable command line interface based on Tcl (Tools Command Language). Specific commands or even scripts can be launched to communicate with the PYNQ-Z2 board, program it and access its internal registers.

Combining XSCT with the tools from Pyxel allowed the platform to operate in a completely independent way, without any required interaction with the user.

Also, python serial communication was required. Two different ports were used: the first one was connected, using a UART to USB bridge, to the NEORV32 output; this allowed to capture the response of the processor when the benchmark was executed. The second port was instead connected to the USB cable supplying the PYNQ board. Here, the communication with the ARM core was established and the response from the SEM-IP captured.

Data coming from the board is analyzed into 2 different ways, depending on the source:

- *NEORV32 response:* once the output of the processor is received it is only compared with a golden response, the one without errors. If a discrepancy is present between the 2, meaning that the injected fault caused a misbehavior, this response was stored into a text file for further analysis. Otherwise, the response was labeled as good.

- *SEM response:* when the response of the controller to a new command is retrieved then it is immediately analyzed. If the output was the expected one, then the system can move one. On the other hand, depending on the outcome of the analysis, different outcomes might take place; either the board is reprogrammed, the injection is performed again, or the correction of the error is repeated.

By passing to the python program specific information regarding the serial ports used, the board identifier and the file(s) containing the injection addresses, the fault injection campaign is structured as described below:

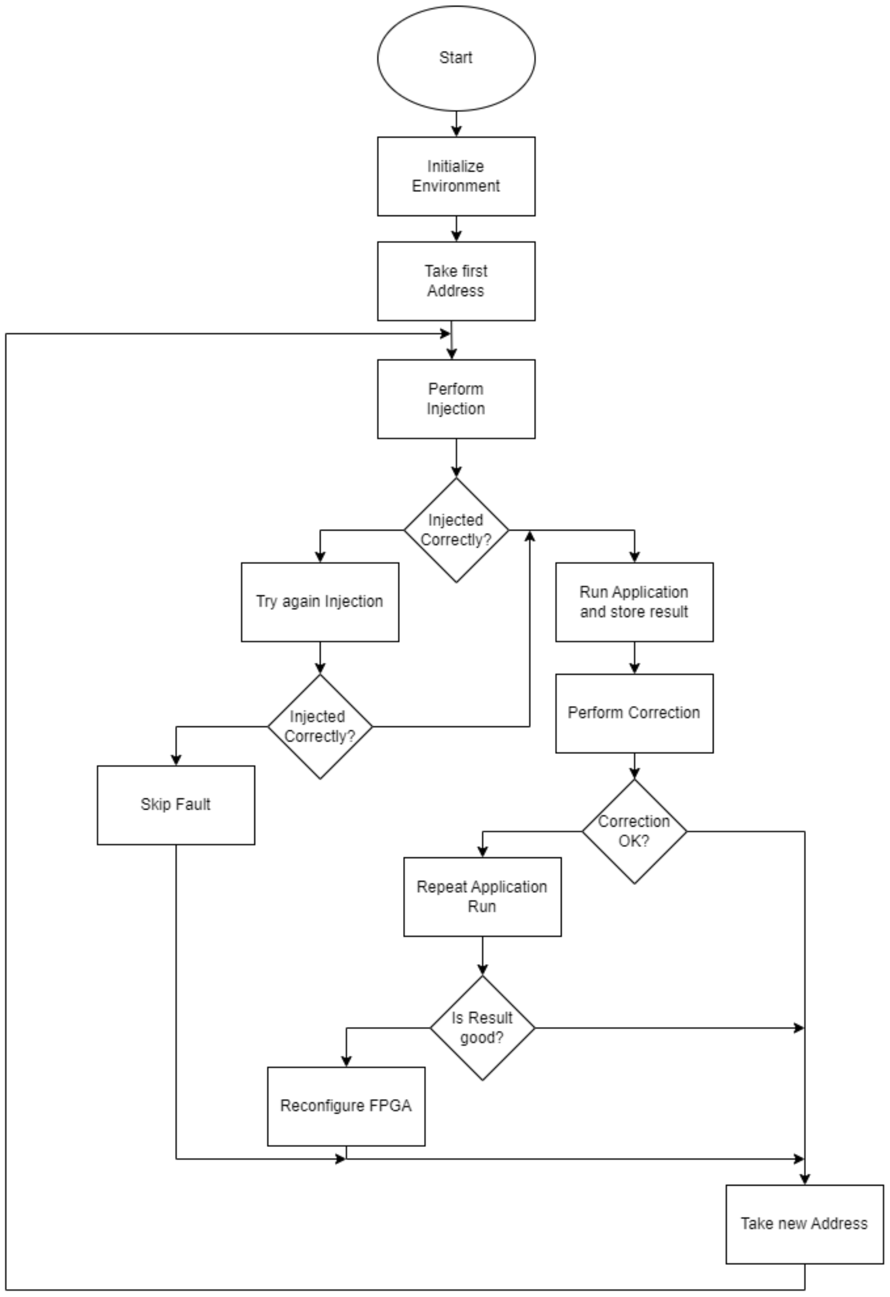


Figure 4.9: Fault injection campaign flow

1. As a starting point, the program sets the environment, prepares all the files for programming the board, initialize all the destination folders and analyzes if any previous injection was already performed; this was required to eventually restarts the campaign from a specific point in case any major problem was encountered.

2. A new injection address is fetched every new cycle, unless for some reason a previous injection was not performed correctly. In that case, the analysis was conducted again at the same address.

3. Once the injection is performed, the reset of the NEORV32 is turned off, the benchmark is executed, and its response is captured. A preliminary analysis is conducted on this outcome as described previously.

4. Now, the correction of the fault takes place; two outcomes are available: if everything was ok, then the system can move on with the new injection. Otherwise, the application is run again on the processor and the output is analyzed again. If the output is still different from the gold one, then the board is fully reprogrammed, meaning that the SEM has encountered some irreparable errors.

Once all the addresses have been analyzed, the program returns and the environment is ready to analyze in a deeper way the results obtained.

4.3.1 Address generation and validation

One additional step was performed to analyze that the injection addresses were compliant with the specific area of the FPGA where the processor was mapped.

Using specific PyXel[24] tools it was possible to take all the addresses from the original bitstream used to program the FPGA, those from the injected lists, map them over specific files and finally compare them. If the addresses of the inject lists were not contained in the ones of the original bitstream, then the injection campaign would have targeted the wrong portion of the FPGA; this would have led to inconsistency in the results. In the images below, the original bitstream and the addresses for the CU injections are reported.

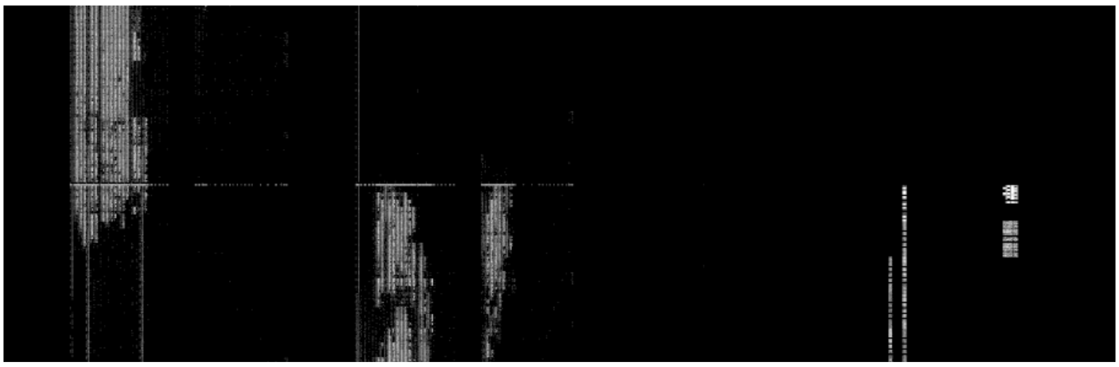


Figure 4.10: Original bitstream

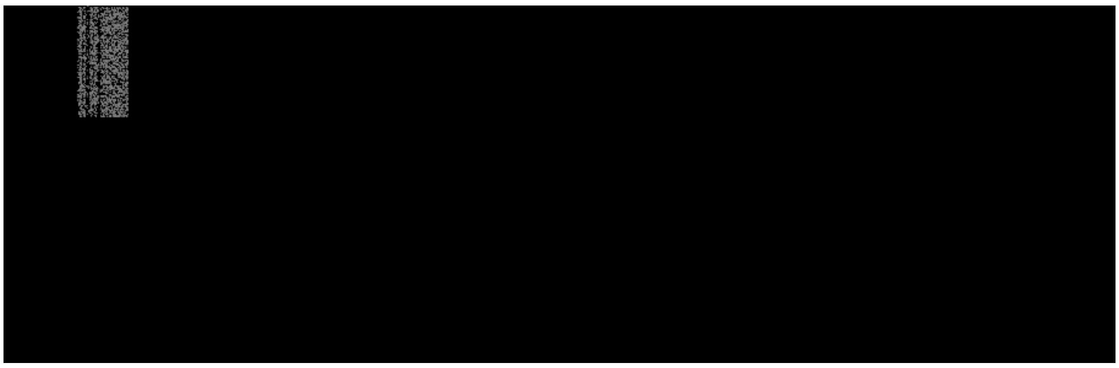


Figure 4.11: CU injection addresses

Chapter 5

Experimental Analysis

This section is divided into two parts: in the first one, I am going to analyze and describe the selected benchmark applications that were used to test the processor functionalities. In the second part, I will highlight the results obtained through the fault injection campaign and the most interesting considerations.

5.1 Benchmarks

After performing the injection over the FPGA area, a way of testing how the processor will behave under this condition is required; this is where benchmark comes into play. Selecting a specific program to be executed is not an easy task, considering that it has not only to be adapted to our case of study, but also that it must target, especially in this situation, a specific portion of the processor. For example, if we take a benchmark that mostly executes load and store operations, while arithmetical operations only cover a small portion of the whole code, then the ALU module of the processor will not be tested in a suitable way.

Also, two other important considerations have to be taken into account: memory occupation and C code instructions. For what concerns the former one, the bigger the code, the slower its execution, the slower the whole injection procedure. Also, since a bare metal scenario is considered, a portion of the executed code will be devoted to system setup operations; this will increase even more the memory occupation.

Another important consideration is related to the executed instructions: being the NEORV32 processor still in development, not all the extensions are already fully implemented. This is the case of the FPU; even if this module is present and can be tested, pure floatingpoint operations cannot be executed and they have to be substituted, in the C code, with pseudo instructions. Suitable arrangements have also to be performed all over the benchmark, for example, to allow the utilization of the UART communication. The output of the benchmark corresponds to a specific signature printed by the UART module. This is then used to classify the faults.

5.1.1 Whetstone

This program carries out a series of arithmetic and logic operations, from matrix convolution to angles computations and roots functions. It is subdivided into different modules that can be either executed or not, depending on the user requirements; each module is executed for a variable number of cycles. It mostly targets floatingpoint operations. Some portions of the code allow to test the integer unit: addition subtraction, shift operations, multiplications, divisions and conditional or unconditional jumps are often executed. The wide variety of operations made this software a perfect test program for this injection campaign.

5.1.2 Linpack

Linpack benchmarks are used to test the floating-point computational power of a device. They solve an nxn system of linear equations Ax=B and they are often used to test some of the world's most powerful supercomputers. These characteristics made the Linpack program one of the best benchmarks to be used, since FPU represents one of the biggest components in the design. Specific integer operations have been added to test the integer part.

5.2 Fault classification

To analyze the outcome of the injection, a specific tool classifying the faults has been developed. Depending on the behavior of the processor, faults were classified according to the following scheme:

Good: those faults that do not generate any misbehavior; the faulty response is equal to the gold one.

Skipped: those faults cause an irreparable crash in the device, requiring a reset of the FPGA; before being classified as skipped, multiple injections on the same address are performed to ensure that it was not a simple misbehavior of the SEM tool.

Faulty: this category is further subdivided into 5 sections:

- *Halted:* faults cause a processor halt, meaning that no response is sent out after a given amount of time (set as a UART parameter).
- *FPU errors:* only FPU results are wrong.
- *MUL/DIV errors:* only MUL/DIV results are wrong.

- *Integer errors:* integer operations were wrong.
- *Common mode errors:* multiple signatures (>=5) were wrong.
- *Generic SDC errors:* non-numeric characters were printed by the UART.

5.3 Results analysis

In this section I am going to analyze the obtained results, highlighting the foreshadowed outcomes and considerations while also underlining the unexpected results. Before entering into further details, it is important to understand how many injections were performed. Considering that an exhaustive fault campaign has taken place, all the addresses related to the essential bits, those calssified by Vivado as the bits defining the logical functionalities of the design, were tested using the SEU fault model as described in previous chapters. This led to around 450000 tested faults for each of the selected benchmarks. Almost 1 million injections were performed over the processor. To fasten the procedure, at least 2 boards with the same configuration, targeting different addresses, were used. The actual test time varied a lot not only depending on the number of faults tested for each unit, but also on the actual outcome of the injection itself. In fact, if more faults required a reconfiguration of the FPGA, then much more time would have elapsed. This is the case, for example, of the control unit.

Firstly, it is important to analyze the overall error rate of each unit, independently of the specific classification. This has been computed as the ratio between the number of addresses generating a faulty behavior and the total number of addresses tested for that specific unit. As it can be observed from the table below, Whetstone benchmark gave an overall higher error rate level. This is because the number of operations performed is more exhaustive than the Linpack one. This is exactly what was expected to happen.

Total Error Rate [%]				
Benchmark	ALU	CU	RF	BUS
Whetstone	14.92	12.58	9.77	13.05
Linpack	12.89	10.93	9.80	12.85

Table 5.1: Overall Benchmarks Error Rate

5.3.1 ALU

For what concern the ALU, results are reported in the graph below.

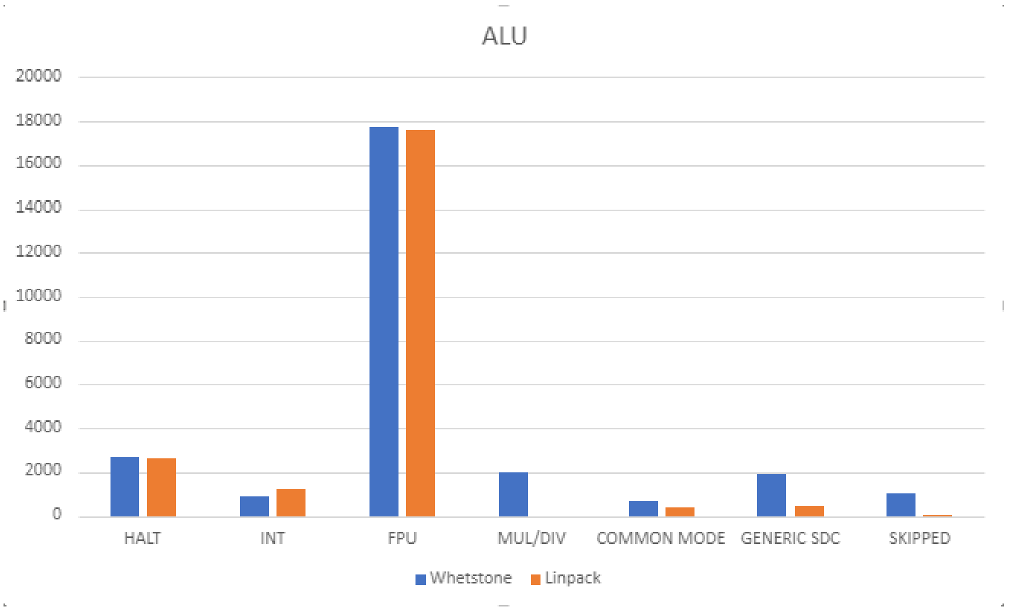


Figure 5.1: ALU fault distribution

As expected, the highest number of faults interested the FPU; in fact, its architecture covers more than 60% of the total ALU module. As we move towards smaller devices, such as the multiplier or the integer unit, the number of errors decreases. However, it is important to highlight that the relative frequency of errors for each of the components of the ALU is relatively higher than the absolute one. This immediately allows to understand that the bigger the device, the lower the probability that a generic error might affect smaller modules. This is also why solutions such as TMR can become extremely effective even if implemented in a simple way.

Whetstone ALU Relative Frequencies[%]			
Module	Faults	Relative Frequency	Absolute Frequency
Floating-point Unit	17738	14.20	10.24
Multiplier Unit	1972	10.70	0.53
Integer Unit	910	5.2	1.14

Table 5.2: Relative Frequency using Whetstone

Linpack ALU Relative Frequencies[%]			
Module	Faults	Relative Frequency	Absolute Frequency
Floating-point Unit	17617	14.10	10.16
Integer Unit	1234	14.55	0.71

Table 5.3: Relative Frequency using Linpack

5.3.2 Control Unit

The CU is mainly affected by halt type of faults. Obviously, being this module the one managing control signals and sending commands to the rest of the processor, the most common consequence of an error can be a stall inside the whole device. It is also important to highlight that in specific cases, misbehavior propagated to external modules, such as the GPIO one. This was verified while, during the injection campaign, some LEDs or RGBs were turned on and off when faults were injected. The overall error rate is lower than the ALU and BUS one because a portion of the CU contains the Control & Status registers, which were not used in this specific contest but still tested, leading to a higher number of tested addresses.

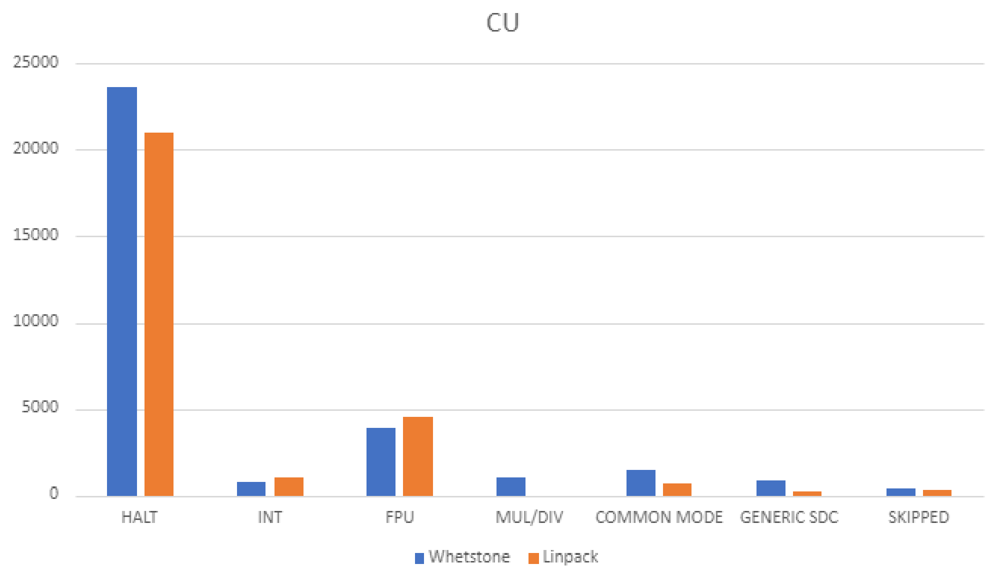


Figure 5.2: CU fault distribution

5.3.3 Regiter File and Bus Unit

The RF and BUS injection analysis led to expected outcomes since most of the faults are related to halts. In the register file cases, some FPU errors are also highlighted. This is because, as explained in previous sections, floating point operations in the NEORV32 use integer registers. One important thing to highlight is that, for both benchmarks, the RF and Bus analysis led to an almost equal total error rate percentage, even if the whetstone is in general more complicated. This is because the number of generated instructions does not affect in a heavy way these modules. Instead, the type of instructions, such as the presence of multiple Load/Store operations, or the internal register access functions is what really allows a deeper analysis. However, this is hard to achieve if programming while using C language. Writing code using ASM allows instead to bypass one step in the compiler phase and directly write specific operations to be executed. In this way a deeper analysis of these modules can be conducted.

The difference in the results obtained while using the 2 benchmarks was a bit more evident in the ALU and CU injection campaign, where a difference of 23% in the results is present.

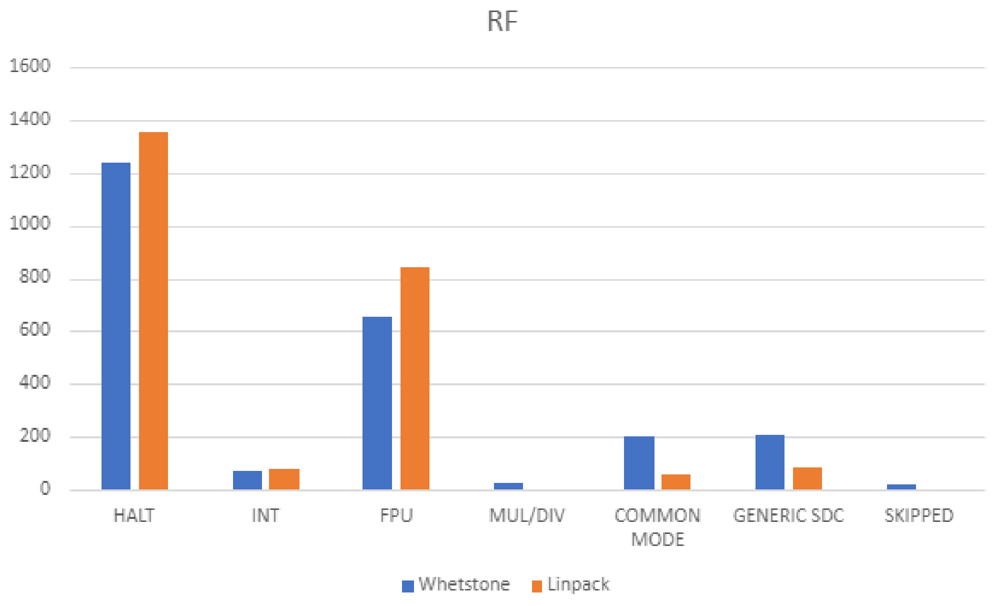


Figure 5.3: RF fault distribution

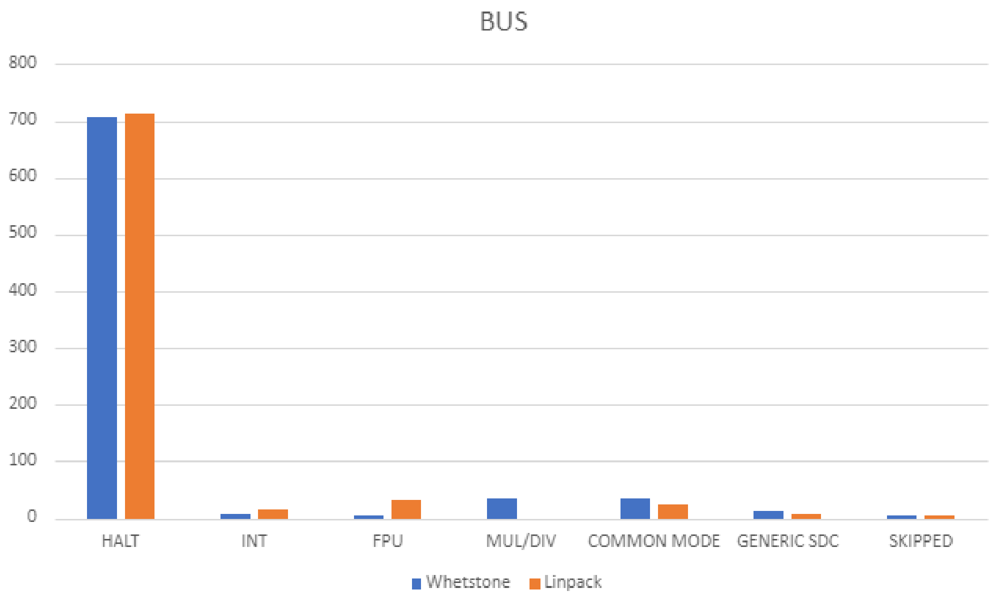


Figure 5.4: Bus faults distribution

Chapter 6

Conclusions

During this thesis work, an in-depth analysis on the reliability of a RISCV5 core, the NEORV32, has been conducted. By mapping the processor onto a Zynq-7000 FPGA, it was possible to emulate how SEU faults, when injected in configuration memory, will affect the behavior of the system. For this purpose, a new fault injection platform has been developed, exploiting all the features provided by Vivado design suite and python programming language. An important consideration is that this platform has been designed in a portable way: by passing slightly different parameters or changing some internal values and global variabales, it is possible to adapt it to different kind of situations; as highlighted in the following section, this feature will allow to conduce even further analysis on new technologies and processors, stemming from single core architectures to multi cores one mapped onto different kind of FPGAs. After defining the NEORV32 design and the fault injection environment, the injection campaign took place. For this purpose, thousands of addresses, targeting specific portions of the CPU, were tested; by storing all the faulty response delivered by the processor, and reprogramming the FPGA when needed, an exhaustive injection campaign took place. Moreover, at the end of this procedure, another tool for evaluating the results was developed, once again, in a portable fashion: depending on the benchmark that was used, faults were classified according to the type of error that was registered. Minor changes can be performed to adapt it to new programs. In this way we managed to have a clear idea of which unit requires more actions to make them suitable for space applications; also, it was possible to understand what type of actions can be performed, either hardware or software ones, to significantly improve system reliability in a fast and direct way.

6.1 Future works

As anticipated previously, the portability of the software developed will allow further research works. A first approach could be to understand how the device will respond when random multiple injections are performed over the whole area of the FPGA. This will allow to define a reliability curve for the processor under test, allowing to simulate in an even more realistic way the possible behavior of the system in the space environment. These results can be then further compared with respect to hardware or software hardened processors, to understand how efficient specific solutions are. By performing small changes to the fault injections platform, it will also be possible to move towards the testing of multi-cores architectures; few of these processors have been designed for the space environment, thus moving in this direction will probably lead to some interesting and significant results. Moreover, specific protocols and interconnections between all the cores must be defined and tested. Another interesting approach would be to understand how the routing and mapping over the FPGA will affect the behavior of the system; by imposing specific constraints when generating the bitstream, it will be possible to understand how faults can propagate and how the reliability of a non-hardened processor can be increased by simply organizing in a better way all the modules of the design. Finally, these injection platforms can also be easily adapted to simulate many other kinds of faults, as well as testing designs which are used for other applications outside the space ones; automotive industry, for example, is an interesting field in which FPGA and processor are rapidly spreading and becoming leading technologies.

Bibliography

[1] Krste Asanovi´c. Andrew Waterman. *The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2.* `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`. Nov. 2017 (cit. on p. 2).

[2] Stephan Nolting. *The NEORV32 RISC-V Processor: Datasheet.* `https://stnolting.github.io/neorv32`. 2023 (cit. on pp. 5, 10).

[3] L. Cassano et al. «Is RISC-V ready for Space? A Security Perspective». In: *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT).* Austin, TX, USA, 2022 (cit. on p. 15).

[4] A. Menicucci G. Furano S. Di Mascio and C. Monteleone. «European Roadmap to Leverage RISC-V in Space Applications». In: *2022 IEEE Aerospace Conference (AERO).* Big Sky, MT, USA, 2022 (cit. on p. 15).

[5] Cobham. *Introduction of Fault-Tolerant Concepts for RISC-V in Space Applications (RV4S).* `https://indico.esa.int/event/323/contributions/5045/attachments/3748/5204/15.15_-_Introduction_of_Fault-Tolerant_Concepts_for_RISC-V_in_Space.pdf`. Nov. 2019 (cit. on p. 15).

[6] A. Ramos I. Wali A. Sánchez-Macián and J. A. Maestro. «Analyzing the impact of the Operating System on the Reliability of a RISC-V FPGA Implementation». In: *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS).* Glasgow, UK, 2020 (cit. on p. 16).

[7] J. Abella et al. «Security, Reliability and Test Aspects of the RISC-V Ecosystem». In: *2021 IEEE European Test Symposium (ETS),* Bruges, Belgium, 2021 (cit. on p. 16).

[8] m. yu j. dong l. xi m. yu l. xi and z. zhang. «Software Simulation Error Injection in RAM on RISC-V of PolarFire FPGA». In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C),* Sofia, Bulgaria, 2019, pp. 499–502 (cit. on p. 16).

[9] N. Lodéa et al. «Early Soft Error Reliability Analysis on RISC-V». In: *in IEEE Latin America Transactions.* Vol. 20. Sept. 2022, pp. 2139–2145 (cit. on p. 16).

[10] F. Restrepo-Calle A. Aponte-Moreno and C. Pedraza. «Reliability Evaluation of RISC-V and ARM Microprocessors Through a New Fault Injection Tool». In: *2021 IEEE 22nd Latin American Test Symposium (LATS).* Punta del Este, Uruguay, 2021, pp. 1–6 (cit. on p. 16).

[11] M. J. Cannizzaro and A. D. George. «Evaluation of RISC-V Silicon Under Neutron Radiation». In: *2023 IEEE Aerospace Conference.* Big Sky, MT, USA, 2023, pp. 1–9 (cit. on p. 16).

[12] D. A. Santos et al. «Characterization of a RISC-V System-on-Chip under Neutron Radiation». In: *2021 16th International Conference on Design &*

Technology of Integrated Systems in Nanoscale Era (DTIS). Montpellier, Francey, 2021, pp. 1–6 (cit. on p. 16).

[13] Á. B. de Oliveira et al. «Evaluating Soft Core RISC-V Processor in SRAM-Based FPGA Under Radiation Effects». In: *in IEEE Transactions on Nuclear Science*. Vol. 67. July 2020, pp. 1503–1510 (cit. on p. 16).

[14] L. M. Luza et al. D. A. Santos. «A Low-Cost Fault-Tolerant RISC-V Processor for Space Systems». In: *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. Marrakech, Morocco, 2020, pp. 1–5 (cit. on p. 16).

[15] N. -J. Wessman et al. «De-RISC: the First RISC-V Space-Grade Platform for Safety-Critical Systems». In: *2021 IEEE Space Computing Conference (SCC)*. Laurel, MD, USA, 2021, pp. 17–26 (cit. on p. 16).

[16] S. Gerardin et al. A. Manuzzato. «Effectiveness of TMR-based techniques to mitigate alpha-induced SEU accumulation in commercial SRAM-based FPGAs». In: *2007 9th European Conference on Radiation and Its Effects on Components and Systems*. Deauville, France, 2007, pp. 1–7 (cit. on p. 16).

[17] L. Sterpone and L. Boragno. «Analysis of radiation-induced cross domain errors in TMR architectures on SRAM-based FPGAs». In: *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2017, pp. 174–179 (cit. on p. 16).

[18] A. Portaluri C. De Sio S. Azimi and L. Sterpone. «SEU Evaluation of Hardened-by-Replication Software in RISC- V Soft Processor». In: *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. Athens, Greece, 2021, pp. 1–6 (cit. on p. 16).

[19] A. E. Wilson and M. Wirthlin. «Fault Injection of TMR Open Source RISC-V Processors using Dynamic Partial Reconfiguration on SRAM-based FPGAs». In: *2021 IEEE Space Computing Conference (SCC)*. Laurel, MD, USA, 2021 (cit. on p. 16).

[20] B. Harikrishna and S. Ravi. «A survey on fault tolerance in FPGAs». In: *2013 7th International Conference on Intelligent Systems and Control (ISCO)*. Coimbatore, India, 2013, pp. 265–270 (cit. on p. 17).

[21] L. Sterpone et al. «A Novel Error Rate Estimation Approach forUltraScale+ SRAM-based FPGAs». In: *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. Edinburgh, UK, 2018, pp. 120–126 (cit. on p. 17).

[22] H. Yang et al. T. Li. «Investigation into SEU Effects and Hardening Strategies in SRAM Based FPGA». In: *2017 17th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. Geneva, Switzerland, 2017, pp. 1–5 (cit. on p. 17).

[23] Z. Jing et al. «Study of an Automated Precise SEU Fault Injection Technique». In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. Shanghai, China, 2012, pp. 277–281 (cit. on p. 17).

[24] L. Sterpone L. Bozzoli C. De Sio and C. Bernardeschi. «PyXEL: An Integrated Environment for the Analysis of Fault Effects in SRAM-Based FPGA Routing». In: *2018 International Symposium on Rapid System Prototyping (RSP)*. Turin, Italy, 2018, pp. 70–75 (cit. on pp. 17, 45, 49).

[25] R. C. Baumann. «Radiation-induced soft errors in advanced semiconductor technologies». In: *in IEEE Transactions on Device and Materials Reliability.* Vol. 5. Sept. 2005, pp. 305–316 (cit. on p. 19).

[26] R. Gaillard. «Single Event Effects: Mechanisms and Classification.» In: *In: Nicolaidis, M. (eds) Soft Errors in Modern Electronic Systems. Frontiers in Electronic Testing.* Vol. 41. Springer, Boston, MA, 2011 (cit. on p. 20).

[27] F. W. Sexton. «Destructive single-event effects in semiconductor devices and ICs». In: *in IEEE Transactions on Nuclear Science.* Vol. 50. June 2003, pp. 603–621 (cit. on p. 20).

[28] Daniela Munteanu Jean-Luc Autran. «Single Event Effects: the effects of Single particles on electronics». In: *https://amu.hal.science/hal-01788355/file/ANIMMA2017_Li%C3%A8ge.pdf.* June 2017 (cit. on p. 20).

[29] K. J. Hass and J. W. Ambles. «Single event transients in deep submicron CMOS». In: *42nd Midwest Symposium on Circuits and Systems (Cat. No.99CH36356).* Vol. 1. Las Cruces, NM, USA, 1999, pp. 122–125 (cit. on p. 20).

[30] Luis Alberto Aranda et al. «Analysis of the Critical Bits of a RISC-V Processor Implemented in an SRAM-Based FPGA for Space Applications». In: *In: Electronics 9.1.* 2020 (cit. on p. 21).

[31] Stephan Nolting. *The NEORV32 RISC-V Processor: User Guide.* https://stnolting.github.io/neorv32/ug/. 2023 (cit. on pp. 22, 32).

[32] *PYNQ-Z2 Reference Manual v1.0.* `https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf`. May 2018 (cit. on pp. 24, 32).

[33] *Vivado Design Suite User Guide, Design Flows Overview.* `https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_1/ug892-vivado-design-flows-overview.pdf`. 2022 (cit. on p. 25).

[34] *Soft Error Mitigation Controller v4.1.* `https://www.xilinx.com/support/documentation/ip_documentation/sem/v4_1/pg036_sem.pdf`. May 2022 (cit. on pp. 34, 39, 40).