Faculty of Electrical Engineering,
Mathematics & Computer Science

Radiation resilience evaluation of a Flash-based FPGA with a soft RISC-V Core

Kevin Böhmer

M.Sc. Thesis - Embedded Systems
September 2023

Supervisors:
dr. ir. M. Ottavi
B. Endres Forlin

Computer Architecture for Embedded Systems
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Acknowledgements

I am grateful for the guidance, mentorship, and support I have received throughout the process of completing my thesis. This academic journey has been made possible with the invaluable contributions of several individuals, and I would like to express my appreciation to each of them.

I want to thank Dr. Ir. M. Ottavi for being willing to supervise this thesis. Thank you for your insightful expertise, advice and the opportunity to work under your guidance. I want to thank Bruno Endres Forlin for his role as my daily supervisor. Bruno, your patience in answering my numerous questions and your support have been invaluable throughout this process. Both Bruno and Marco introduced me to the academic world and guided me in the publication of an academic paper, an achievement of which I am incredibly proud.

I would like to extend my gratitude to Prof. Dr. Ir. A.B.J. Kokkeler and Dr. Ir. N. Alachiotis for their roles as members of my graduation committee. Thank you for your willingness to serve as evaluators through the trajectory of my thesis.

The graduation committee consists of the following members:

- Dr. Ir. M. Ottavi (University of Twente & University of Rome Tor Vergata)
- Dr. Ir. N. Alachiotis (University of Twente)
- Prof. Dr. Ir. A.B.J. Kokkeler (University of Twente)
- B. Endres Forlin (University of Twente)

Lastly, I extend my appreciation to you, the reader, for dedicating your time to read my thesis.

Abstract

Highly reliable and customizable micro-processors are critical enablers for future intelligent space platforms. From an architectural point of view, the RISC-V architecture is the current best option for adaptability, with its modular ISA and a multitude of contributors. To implement such a processor at a low price range, companies are looking at reprogrammable Field-Programmable Gate Arrays (FPGAs), which can extend the mission lifetime. SRAM FPGAs are known to be susceptible to low Linear Energy Transfer Single-Event Upsets (SEUs) in the configuration memory, Flash FPGAs on the other hand, are in general immune to such errors.

This thesis performs for the first time characterization of the open-core NEORV32, a lightweight yet representative RISC-V SoC, and provides insights into the tradeoffs of protection mechanisms against neutron-induced SEUs when this core is implemented in a Flash-based FPGA. The Unmodified core is compared against an ECC-protected version and a register-level Triple Modular Redundancy (TMR) with an Error Correction Code (ECC) version. All versions execute the CoreMark benchmark.

The Unmodified NEORV32 instances mainly experienced exceptions arising from Single Event Upsets (SEUs) that affected stored pointers in the data memory. These altered pointers, when employed as addresses, resulted in Load and Store exceptions, stemming from the pointers now residing outside the valid memory range. The incorporation of ECC swiftly mitigated these disparities and reduced Store and Load exceptions to zero. Introducing TMR on the Flip-Flop level further advanced the outcome by eliminating all exceptions, including those tied to Illegal instructions. These Illegal instructions are likely the fallout of SEUs influencing control logic, culminating in Single-Event Functional Interrupts (SEFIs).

Contents

Acknowledgements **iii**

Abstract **v**

List of acronyms **xi**

1 Introduction **1**
1.1 Motivation 3
1.2 Research goal 4
1.3 Report organization 4

2 Background **5**
2.1 Single Event Effects 5
2.1.1 SEE causes 5
2.1.2 Physical origins of SEU 7
2.1.3 Effects of SEEs 8
2.1.4 Architecturally Correct Execution (ACE) 10
2.1.5 Architectural Vulnerability Factor (AVF) 12
2.1.6 Metrics 13
2.2 Redundancy techniques 14
2.2.1 Physical Redundancy 14
2.2.2 Temporal Redundancy 15
2.2.3 Information Redundancy 15
2.3 RISC-V and FPGA Integration for Single Event Effects Mitigation . . . 21
2.3.1 RISC-V Instruction Set Architecture 21

3 Related Work **23**

4 NEORV32 **27**
4.1 NEORV32 Processor & CPU 27
4.2 RISC-V Standard Extensions Configurability 28
4.3 Pipeline 29

4.4 Memory Access 29
4.5 Execution safety 31
4.6 Wishbone interface 32

5 NEORV32 Implementation **35**
5.1 Target Device 35
5.2 Unmitiaged NEORV32 implementation 36
5.2.1 Memory access 36
5.2.2 AHBL-Wishbone Bridge 38
5.3 Fault-tolerant enhancements 41
5.3.1 Design of Hsiao Encoder & Decoder 41
5.3.2 ECC Implementation 42
5.3.3 TMR implementation 43
5.3.4 FPGA resource usage 44
5.3.5 Power Estimation using Microsemi's Smart Power Tool 48
5.4 Testing of the Fault-Tolerant adjustments 50

6 Neutron Beam Experiment **53**
6.1 Experimental setup 53
6.2 Software 54
6.3 Error Model 55
6.4 Characterization Results 56

7 Discussion **61**
7.1 Implications of findings 61
7.1.1 Unmodified 61
7.1.2 ECC-enhanced 63
7.1.3 ECC+TMR 64
7.2 Program analysis 65
7.2.1 Affected functions 65
7.2.2 Vulnerability Factor 66
7.2.3 CoreMark Linked List Algorithm 67
7.2.4 Analysis of the crash traces 69
7.3 Summary 71

8 Simulation **73**
8.1 Simulation setup 73
8.2 Fault injection strategy 74
8.3 Manual injected faults for PC value 0x600012cc 76
8.4 Manual injected faults for PC value 0x60001676 76

8.5 Summary 78

9 Conclusions and recommendations 81

9.1 Future work 83

References 85

Appendices

A 91

A.1 Assembly functions 91

A.2 Top-level design in Libero SoC 95

A.3 FPGA resources 96

A.4 Parity check matrices 97

List of acronyms

ACE	Architecturally Correct Execution
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuits
AVF	Architectural Vulnerability Factor
BRAM	Block RAM
CNN	Convolutional Neural Networks
COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DMEM	Data Memory
DFF	D-type Flip-Flop
DMR	Dual Modular Redundancy
DUE	Detected Unrecoverable Error
ECC	Error-Correction Code
EDC	Error-Detecting Code
ESA	Eureopean Space Agency
FDD	First-Level Dynamically Dead
FIT	Failure in Time
FPGA	Field-Programmable Gate Array
PWM	Pulse-Width Modulation
HPM	Hardware Performance Monitors
IC	Integrated Circuit
IP	Intellectual Property

ISA Instruction Set Architecture

LET Linear Energy Transfer

LUT Lookup Table

LSB Least Significant Bit

MBU Multiple Bit Upset

MEBF Mean Execution Between Failures

MSS Microcontroller Subsystem

MTBF Mean Time Between Failure

MTTF Mean Time to Failure

MWBF Mean Workload Between Failures

NMR N-Modular Redundancy

PC Program Counter

PVF Program Vulnerability Factor

RTL Register Transfer Level

SDC Silent Data Corruption

SEB Single Event Burnout

SECDED Single-Error Correcting and Double-Error Detecting

SEC Single Error Correcting

SED Single Error Detecting

SEE Single Event Upset

SEFI Single Event Functional Interrupt

SEGR Single Event Gate Rupture

SEL Single Event Latchup

SEMBU Single Event Multiple Bit Upset

SER Soft Error Rate

SET Single Event Transient

SEU Single Event Upset

SPI Serial Peripheral Interface

SRAM Static RAM

SoC System on Chip

TDD Transitively Dynamically Dead

TMR Triple Modular Redundancy

TRL Technology Readiness Level

VCD Value-Change Dump

WDT Watchdog timer

eNVM Embedded Non-Volatile Memory

Chapter 1

Introduction

In recent years, there is an increasing interest in Artificial Intelligence (AI) by the entire aerospace community [1]. For example, the communication delay between Mars and Earth could range from 6.5 to 44 minutes, to enable such a mission, individual space entities need autonomous decision-making abilities. Nevertheless, the majority of tasks performed by processors within space data systems primarily involve non-demanding control and housekeeping operations. Consequently, facilitating more resource-intensive tasks, such as running AI algorithms on embedded systems in space, necessitates a fundamental shift in the design of space-grade processors. This transformation is particularly crucial given that processors on satellites frequently operate under stringent power constraints.

The aerospace industry has highlighted the limitations of existing space-grade computing systems. Full rad-hard processors, which have hardening in both silicon process and microarchitecture, typically lag more than a decade behind their commercial counterparts in terms of performance, because of their niched-size market. Unfortunately, this gap is widening every year [2]. These architectures are built upon the Sparc V8 Instruction Set Architecture (ISA), which was initially introduced in 1992. Despite the decline of SPARC processors in terrestrial applications, they continue to be a prominent choice for ongoing and upcoming missions in the space sector, although these systems heavily depend on outdated architectures. In addition to the inadequate computing power for AI tasks, this persistence has also led to challenges in sourcing developers with expertise in these specific technologies.

Given these circumstances, opting for an architecture that provides a flourishing software environment and ISA extensions, like RISC-V, can become highly beneficial. Embracing the RISC-V architecture offers a multitude of benefits to the space sector. The Eureopean Space Agency (ESA) believes that the utilization of RISC-V presents the opportunity to leverage an expanded and robust software ecosystem in the years to come [2]. This prospect holds particular importance for essential functions such as the maintenance of the software toolchain. Furthermore, the in-

troduction Of RISC-V in space will contribute to providing a range of alternatives to proprietary solutions. Open standards allow designers to access multiple Intellectual Property (IP) sources or even create their own, resulting in a diverse ecosystem of innovation and competition. This is of particular significance as concerns continue to mount regarding monopolistic positions within the embedded market [3], [4], as well as the potential impact of sanctions and the desire for greater independence in the product chain. In addition, the RISC-V ISA is divided into different standard extensions, each targeting specific functionalities, covering low-end to high-performance applications, such as AI. RISC-V's modular design enables engineers to tailor the processor's architecture to meet these requirements more effectively.

To effectively assess the suitability of a RISC-V processor for space applications, a dedicated target platform is essential, which can take the form of either an Application-Specific Integrated Circuits (ASIC) or Field-Programmable Gate Array (FPGA). The advantages of employing an FPGA include its cost-effectiveness and the ability to be reprogrammed. With satellite operational lifetimes extending well over a decade, exceeding the validity of existing telecom standards, the demand for in-flight re-programmability becomes increasingly critical [5]. If software-based solutions are not viable, Repgrogammable FPGAs (RFPGA) could arise as the only method to address this need. The objective of the Heinrich Hertz (H2Sat), launched in 2018, is to validate new hardware, software and communication technologies in space, aimed at addressing the ever-evolving landscape of telecommunication standards [6]. The transponder of this satellite, integrated with an in-flight reconfigurable FPGA-based module, will possess the capability to receive updates while in-flight, allowing for adaptations to new telecommunication standards as they are introduced.

Utilizing Commercial Off-The-Shelf (COTS) electronic components offer an attractive advantage over their Radiation Tolerant or Hardened counterparts, primarily in terms of performance enhancement [7]. In fact, the adoption of COTS devices is often advocated as the optimal approach to meet the escalating performance demands of space applications [8]. However, it is important to acknowledge that COTS devices are more susceptible to external disturbances, such as radiation, owing to their lower power consumption and faster switching characteristics. In the challenging space environment, electronics are subjected to diverse forms of radiation, containing high-energy particles. One particular concern is the occurrence of Single Event Upsets (SEUs), wherein ionizing radiation impacts a memory cell, inducing a change within it. This alteration can lead to bit-flips. One of the most effective strategies for mitigating these effects involves introducing redundancy at the Register Transfer Level (RTL).

FPGAs come in two primary types: SRAM-based and Flash-based, each offering unique benefits. SRAM-based FPGAs employ Static RAM (SRAM) cells for

configuration, enabling high performance and rapid reconfigurability. However, they are susceptible to SEUs that can induce bit flips in the configuration memory, leading to malfunctions and unpredictable behaviour. In contrast, Flash-based FPGAs are notably well-suited for space applications due to their immunity to configuration memory upsets and low power consumption. Despite this advantage, certain components within Flash-based FPGAs, like Block RAMs (BRAMs) and D-type Flip-Flops, remain vulnerable to SEUs. Thus, incorporating redundancy into the design becomes necessary even for Flash-based FPGAs.

Future space could benefit from these new developments, provided that research is done to satisfy the stringent requirements in space, especially in terms of fault tolerance and assessing of Technology Readiness Level (TRL) for COTS devices. The process of testing and certifying novel cores and platforms before their deployment in space is a crucial step in ensuring the success and safety of space missions. Subjecting cores and platforms to simulated space environments on Earth helps to uncover any vulnerabilities and assess the TRL. A powerful approach to replicating the space environment involves the usage of beam experiments. Through such experiments, the cross-section of devices can be estimated under conditions that closely resemble the space environment, achieved by exposing the device to accelerated particle fluxes.

1.1 Motivation

In the ever-evolving landscape of the space sector, the imperative to embrace COTS devices for space applications has grown exponentially. Building upon the challenges and opportunities outlined in the introduction, this research addresses a critical research gap in the domain of space-grade processors. While substantial progress has been made in the domain of soft cores implemented in SRAM-based FPGAs, a significant research gap exists in the utilization of Flash-based FPGAs, despite their immunity to configuration memory upsets and lower power consumption.

In light of these considerations, this research attempts to delve into two corresponding domains. Firstly, it seeks to deepen our comprehension of the behavioural dynamics of Flash-based FPGAs when subjected to neutron radiation. This aims to unlock insights into the resilience and adaptability of Flash-based FPGAs in the hostile space environment.

Concurrently, this study undertakes the task of exploring the viability of integrating open-source RISC-V cores into a Flash-based FPGA. By doing so, the research tries to match the complexity of fault-tolerant enhancements with the cutting-edge capabilities of RISC-V architecture.

The main driving force behind this thesis is to contribute to the new era of space. By investigating the complex behaviours of Flash-based FPGAs when exposed to radiation and aligning them with the reliability of RISC-V cores, this study aims to make a substantial contribution to the progress of aerospace technology.

1.2 Research goal

The previous section has highlighted the importance of fault-tolerant processors. To investigate this issue, this thesis aims to achieve the following research goals:

Investigate different trade-offs between area, power, performance, and resilience for errors in different techniques of fault tolerance in a soft processor implemented on a Flash-based FPGA, and perform a radiation test to measure the performance of the redundancy techniques.

This goal can be achieved by answering the following research questions:

RQ1: How do different fault tolerance techniques impact the trade-off between area, power, and performance in a soft processor implementation on a Flash-based FPGA?

RQ2: What is the resilience performance of different fault tolerance techniques in the presence of errors, particularly radiation-induced faults, in a soft processor implemented on a Flash-based FPGA?

1.3 Report organization

This thesis aims to provide a comprehensive analysis of the radiation resilience of Flash-based FPGA with a soft RISC-V core through the exploration of eight key chapters. The remainder of this report is organized as follows. In chapter 2, the groundwork is provided by providing an overview of the necessary background. In Chapter 3, the related work about radiation tests of FPGAs and Asics is presented. In chapter 4, the chosen soft-core and its architecture are presented.

The implementation of this softcore on the chosen FPGA is explored in chapter 5. The setup for the radiation beam experiment and the results of this experiment are shown in Chapter 6. Chapter 7 provides a discussion of the conducted experiment. In chapter 8 the possible exception causes presented in the Discussion are validated through an RTL simulator. Lastly, in Chapter 9, conclusions and recommendations are given.

Chapter 2

Background

The current Integrated Circuit (IC) technology is vulnerable to Single Event Upset (SEE) caused by particle strikes. These SEEs can influence the functionality of a chip or the computational results, therefore there is a need for fault-tolerant in mission-critical applications, like avionics, space or medical applications. This chapter gives an overview of different forms of SEEs. In addition, this chapter will present common redundancy techniques to mitigate the errors caused by SEEs. Lastly, the RISC-V ISA will be introduced as the target platform.

2.1 Single Event Effects

The effects of subatomic radiation particles on ICs are frequently referred to as Single Event Upsets (SEEs). These SEEs can lead to randomly appearing glitches in electronic errors resulting in annoying system responses or catastrophic system failures. Due to the IC devices with higher density and smaller feature sizes, these devices are more vulnerable to SEEs. Smaller feature sizes, result in faster processing and also require a smaller quantity of electrical charge. Because the charges have been decreased, these charges can be generated in the IC device by the passage of cosmic rays or alpha particles.

2.1.1 SEE causes

Satellites in geosynchronous orbit and corresponding regions outside Earth's radiation belts experience upsets due to heavy ions from either cosmic rays or solar flares [9]. The cosmic ray heavy ion flux has approximately 100 particles/cm2 per day. For very sensitive devices this can result in daily upsets. Cosmic rays consist of mostly protons, but also alpha particles and heavy ions, they mainly originated outside the solar system. The solar particles come from the sun and are high-energy protons and heavy ions.

Upsets can also occur within the proton radiation belts. The Van Allen belts are two vast regions of intense radiation that encircle the Eart. It is named after the scientist James Van Allen who discovered the belts in 1958. These belts consist of charged particles, primarily electrons and protons. These are trapped by the magnetic field of the Earth, they surround our planet like concentric doughnut-shaped zones. The inner belt is closer to the Earth, while the outer belt is further from Earth, reaching into a region where satellites orbit. The outer belt consists mainly of high-energy electrons and is located 13,000 to 60,000 kilometres from sea level. The lower belt lies 1,000 km to 12,000 km above Earth and consists mainly of high-energy protons. The outer belt affects satellites and the lower belt affects high-altitude aircraft.

The cosmic rays in space and those reaching Earth's atmosphere are similar in their origin and general characteristics, but there are some differences between them. When cosmic rays interact with Earth's atmosphere, several factors come into play that influence their behaviour and composition. The atmosphere provides shielding, it reduces the intensity and energy of cosmic rays reaching the surface compared to those in space. The intensity and energy spectrum of cosmic rays varies with altitude. At a higher level in the atmosphere, there is less atmospheric shielding, therefore the cosmic ray flux is generally higher at higher altitudes. Because cosmic rays penetrate the atmosphere, there is a chain of nuclear reactions that produce high-energy neutrons and protons. At 12,000 kilometres feet altitude and 45 degrees latitude, there are 6000 neutrons per square centimetre. High-energy cosmic rays and solar particles react with the upper atmosphere generating high-energy protons and neutrons that shower to the ground. Neutrons are particularly troublesome because they are able to penetrate a concrete wall. This effect depends on both the latitude and altitude.

The problems of radiation were already discovered in 1962. During the period from 1962 to 1970, the initial satellite electronics showed unreliability, requiring the incorporation of significant redundancy [10]. The major satellite problem was differential satellite charging in the solar wind, which resulted in noise and arcing between satellite modules. Furthermore, data transmission to Earth suffered from high levels of noise, making it difficult to distinguish electronic soft failures from transmission errors. To address this, these transmissions were divided into smaller data streams accompanied by parity checks and handshaking procedures.

There are more sources of radiation besides cosmic rays, solar particles and the Van Allen belts. Package material of a chip can obtain radioactive isotopes which decay, causing alpha particles right on the device. By carefully selecting the materials, this cause can be greatly reduced. Besides space and avionics, there also exist radiation damage sources on Earth. For instance, nuclear reactors need sensors

and control circuits. Particle accelerators also need electronics for controlling and detector devices. In 1978, the cause of the errors in Intel's 2107 series 16Kb DRAM was found. The problem of trace radioactivity in the memory packaging materials was discovered. A new factory was built on the Green River in Colorado due to the large increase in demand for LSI ceramic packaging in the 1970s. However, this factory was built downstream from the tailings of an old uranium mine. The ceramic LSI packaging was contaminated due to the factory's utilization of water containing a significant amount of radioactive elements [10].

2.1.2 Physical origins of SEU

SEUs can be caused by charge deposition or by charge collection. Firstly **charge deposition** will be explained. There are two main ways in which ionizing radiation causes charge release in a semiconductor device: direction ionization caused by the particle itself, and ionization caused by secondary particles generated through nuclear reactions between the particle and the semiconductor device. Both mechanisms result in the malfunctioning of the integrated circuits.

Direct Ionization: When an energetically charged particle moves through semiconductor material it frees electrons along its path as it loses energy. The particle is at rest when all its energy is lost. The total path length of the travelled particle is referred to as the particle's range. The Linear Energy Transfer (LET) describes the energy loss per unit path length of a particle as it passes through the material [11]. LET has units of $MeV/cm^2/mg$ because the energy loss per distance (MeV/cm) is normalized by the density of the target material (mg/cm^3). Therefore the LET can be used for different targets. A LET of 97 $MeV/cm^2/mg$ corresponds to a charge disposition of 1 pC/um. Direction ionization is the primary charge deposition mechanism for upsets by heavy ions. Lighter particles such as protons do not usually produce enough charge by direct ionization to cause upsets in memory circuits. However, because SEE devices are becoming more susceptible, upsets by protons may occur.

Indirect ionization: Protons and neutrons can both produce significant upset rates due to indirect mechanisms. A high-energy proton or neutron can cause several nuclear reactions when it enters the semiconductor lattice. These reaction products can deposit energy along their path by direct ionization because these particles are much heavier than the original proton or neutron.

SEUs can also be caused by **charge deposition**. Usually, the most sensitive region is the *reverse-biased p/n junction*. These regions can very efficiently collect the particle-induced charge through drift processes, resulting in a transient current at the junction. This current spike has two components: a short component that

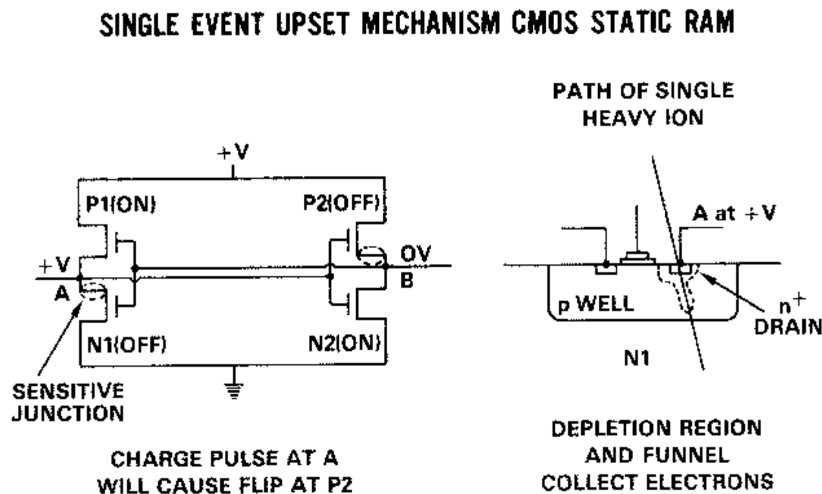

Figure 2.1: Flip-flop circuit with a sensitive junction highlighted. The right picture shows the sensitive junction hit with a single heavy ion. *Source: Petersen et al. [12]*

lasts for hundreds of picoseconds after the ion strike and a delayed component that may last hundreds of nanoseconds [9]. Figure 2.1 shows an **SRAM** cell in a valid logic hold state on the left. On the right, the reverse-biased drain N1 is hit with an ion strike. The inverter which is formed by P1 and N1, with input node B held low, and output node A is pulled to VDD by P1. The inverter formed by P2 and N2 has input node A forced high, therefore output node B is pulled low by N2. If an ion strikes and enough charge is collected at the drain of N1, which lowers the potential below the threshold of P2 and N2, the logic stored in the SRAM cell may flip.

When a charged particle, like an ion from radiation, interacts with the floating gate transistor of a **Flash** cell, it needs a significantly higher amount of energy to actually change the stored charge in the floating gate. This is because the charge is stored in the form of trapped electrons, which are very well insulated.

2.1.3 Effects of SEEs

SEEs happen in space, in avionics and even on Earth. When it comes to avionics systems, the main focus is on the neutrons generated by high-energy particles as they penetrate the atmosphere. While approximately 20% of atmospheric particles are protons, their impact is comparable in nature [9]. These SEEs can manifest in the following ways:

1. **SEU**: A change in the state of the the memory. The susceptibility depends on the type of memory, SRAM is more susceptible than Flash memory.

2. **Single Event Multiple Bit Upset (SEMBU)**: a single particle strike, that causes multiple changes in memory. An SEU with multiple bits changed.
3. **Single Event Functional Interrupt (SEFI)**: unexpected output results that are produced by SEUs originated in the device, for instance, SEUs in the configuration memory of an FPGA [13].
4. **Single Event Transient (SET)**: A voltage pulse in combinational logic which can result in erroneous results if captured by a memory cell at the correct moment.
5. **Single Event Latchup (SEL)**: a short circuit which disrupts the functioning of the IC, it could lead to permanent destruction.
6. **Single Event Burnout (SEB)**: A destructive burnout of the drain-source in a power MOSFET.
7. **Single Event Gate Rupture (SEGR)** caused by high-energy particles damaging the gate oxide of MOS transistors, leading to malfunction or permanent failure.

SEL, SEB and SEGR are destructive forms of SEEs, which means they involve the physical destruction or irreversible changes in the device's component. On the other hand, Non-Destructive Single Event Effects refer to temporary or transient disruptions in the regular operation of an electronic device. SEUs, SEMBUs and SETs are forms of non-destructive SEEs and will be the focus of this thesis. An example of a particle hitting a memory cell can be seen in figure 2.2. A particle has hit a 16-bit word and depending on the technology this can result in a SEU or SEMBU.

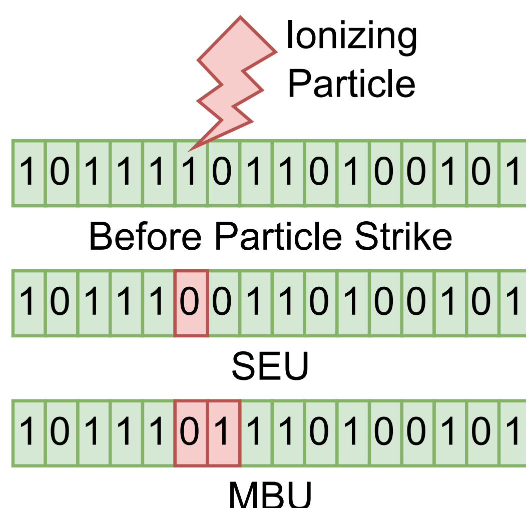

Figure 2.2: Example of a 16-bit memory array hit by an ionizing particle

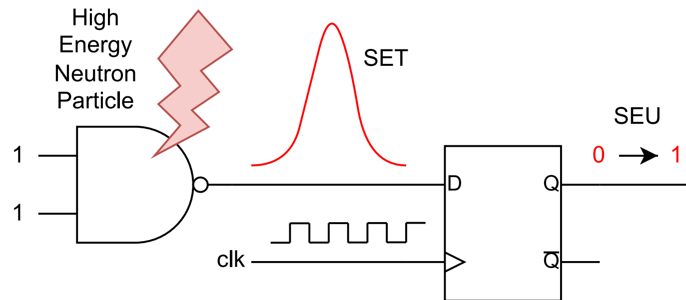

Figure 2.3: A SET in a combinational logic cell resulting in a SEU

If a particle hits a sensitive node of a logic gate cell it produces a SET [14]. This SET can propagate through the logic data path and be captured by a flip-flop or latch. The duration of the SET pulse is important since it must be sufficiently long to ensure its capture and precisely timed to be captured at the correct moment. Once the SET is stored within a memory cell, it transforms into a Single Event Upset (SEU). An illustration of this concept can be seen in Figure 2.3. Consider a NAND gate with both inputs set to 1, a situation that ordinarily produces an output of 0. However, an unexpected scenario unfolds when a high-energy neutron particle collides with the NAND gate. This collision generates a short voltage peak at the output of the gate. Notably, the connected D-type Flip-Flop (DFF) records this voltage wave, resulting in the transformation of what was originally a SET into a SEU.

Detected errors are called Detected Unrecoverable Error (DUE), and undetected errors are called Silent Data Corruption (SDC). Not every faulty bit will result in a visible as shown in figure 2.4. A faulty bit which is never read is not classified as an error, because it does not have any effect on the outcome of the system. When a faulty bit is read and there is error detection and correction, the bit can be corrected and the faulty is not classified as an error. When the faulty bit can only be detected, the question needs to be answered if it has any effect on the program outcome. When it has no effect, it will be classified as a false DUE, otherwise as a true DUE. When there is no error detection and it does not affect the program outcome the faulty bit is classified as no error, because it is never detected. When it changes the program outcome and is not detected, the classification SDC is used.

2.1.4 Architecturally Correct Execution (ACE)

Not all faults in a microarchitectural structure affect the final outcome of a program. For instance, a single-bit fault in the branch predictor will not affect the sequence or results of any committed instructions. The AVF is defined as the probability that a fault in that particular structure will result in an error [16]. So the AVF of the branch predictor is 0%. In contrast, the AVF of the program counter is effectively 100%.

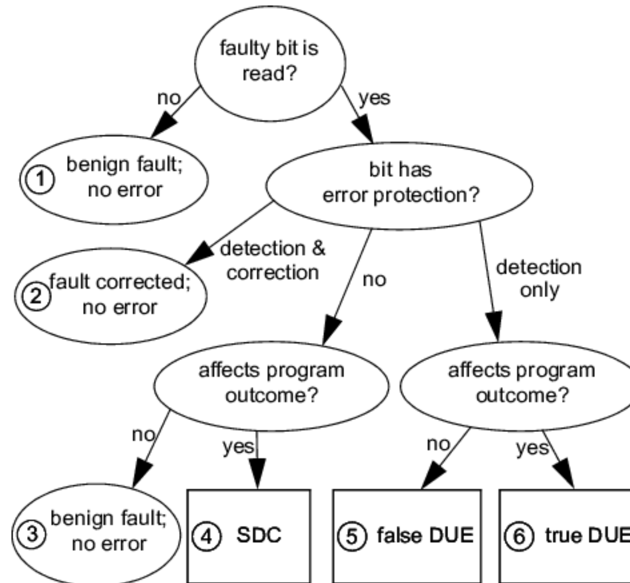

Figure 2.4: Possible outcomes of a faulty bit in a microprocessor. *Source: Mukherjee et al. [15]*

The AVFs can also be calculated for storage cells. An error in a storage cell that causes a visible error in the final output of the program in the absence of error correction techniques is an ACE bit [16]. The remaining processor state bits are called un-ACE bits. A fault in an un-ACE bit will not cause a visible error. The AVF for a single-bit storage cell is simply the fraction of the time that it holds ACE bits.

The AVF of a branch predictor is 0% because the branch predictor is always un-ACE bits. All the bits in the PC are always ACE bits, resulting in an AVF of 100%. Processor bits that do not affect the committed instruction path are called **microarchitectural un-ACE bits**, examples of these are:

1. **Idle or Invalid State**: there exist instances in a microarchitecture when a status bit is idle or does not contain any valid information. Such data and status bits are un-ACE bits.
2. **Mispeculated state**: a microprocessor can perform different forms of speculative operations. When these speculative operations are found to be incorrect, the bits of these incorrectly speculated operations can be classified as un-ACE bits.
3. **Predictor Structures**: A modern microprocessor can have many predictor structures. A fault in these structures will cause a misprediction. This will affect the performance, but the execution remains correct, therefore these bits can be classified as un-ACE bits.
4. **Ex-ACE State**: are ACE bits that become un-ACE bits after their last use. This

classification contains architecturally dead values, for instance, registers, as well as architecturally invisible states.

Architectural un-ACE bits affect correct-path instruction execution, however, it does not change the output of the system. There are five sources of architectural un-ACE bits [16].

- **NOP instructions**: most instruction sets have NOP instructions. These NOP instructions are used for instance for aligning instructions to address boundaries or filling VLIW-style instruction templates. The only ACE bit of a NOP instruction is the bits distinguishing the NOP instruction from a non-NOP instruction.
- **Performance-enhancing instructions** non-opcode bits are un-ACE bits, because a fault there may cause the wrong data to be prefetched or may cause an invalid address, in both cases the prefetch will be ignored.
- **Predicated-false instructions** are instructions which will not be committed and discarded. Clearly, all these bits are un-ACE Bits.
- **Dynamically dead instructions** are those whose results are not used. Transitively Dynamically Dead (TDD) instructions are instructions whose results are not read by other instructions. First-Level Dynamically Dead (FDD) instructions are instructions whose results are only read by other TDD or FDD instructions. Suppose, instruction A writes to register X1. After this write, instruction B writes to the same register, but X1 was not read, so the output of instruction A is not used. Therefore instruction A is a FDD instruction. FDD and TDD instructions can be counted as ACE bits.
- **logical masking** there exist bits that do not influence the output. For instance an OR operation on specific bits with 1's.

2.1.5 Architectural Vulnerability Factor (AVF)

Mukherjee et al. introduced the concept of Architectural Vulnerability Factor (AVF) and ACE analysis [16]. The AVF of a processor is the probability that a fault in this structure will result in a visible error in the program output. It is based on the concept of Architecturally Correct Execution (ACE) bits and un-ACE bits. ACE bits are needed for correct operation, while un-ACE bits are not. The definition of AVF can be found in Equation (2.1).

$$AVF_H = \frac{\sum_{n=0}^{N}(ACE \text{ m-bits in } H \text{ at cycle } n)}{B_R \times N} \quad (2.1)$$

The Program Vulnerability Factor (PVF) is a metric that measures the vulnerability of an architectural resource to errors [17]. The PVF is given in Equation (2.2) [17]. The difference between PVF and AVF is that PVF focuses on instructions, whereas AVF focuses on clock cycles.

$$PVF_R = \frac{\sum_{i=0}^{I}(ACE \text{ a-bits in } R \text{ at instruction } i)}{B_R \times I} \quad (2.2)$$

Fang et al. extended the PVF metric and created a dynamic model for predicting whether a particular fault will cause a crash. By performing fault injection they found that the majority of crashes are caused by illegal memory addressing [18]. They observed four types of exceptions resulting in crashes: Segmentation fault, Abort, Misaligned memory access and Arithmetic errors. They discovered that segmentation faults are the predominant source of crashes with a 99% average frequency and a 96% minimum frequency.

2.1.6 Metrics

Both SDC and DUE rates are typically expressed in Failure in Time (FIT). One FIT specifies one failure in 10^9 hours, which is a billion hours. FIT rates are additive, so the FIT rate of a system can be computed by summing all the FIT rates of the components. This sum is often referred to as Soft Error Rate (SER). The Mean Time to Failure (MTTF) is often more intuitive but is not additive. MTTF is the average time before the system fails and is inversely related to FIT. A FIT rate of 10,000 is equivalent to an MTTF of: $10^9/(10,000 * 24 * 365) = 11.42$ years.

The cross-section (σ) is the standard metric to evaluate the sensitivity to radiation of a device [19]. To understand this metric, it is important to first understand flux and fluence. Flux and fluence are used to describe the radiation environment or the particle beam used for testing. Flux is the rate of particles or energy across or onto a given area. The neutron flux (φ) is the number of incident particles per unit area and per unit time ($n/(cm^{-2}s^{-1})$).

Fluence is a measure of the quantity of light or radiation falling on a surface, expressed in terms of either particles or energy per unit area. The neutron fluence is defined as the neutron flux integrated of a particular time period, this results in a unit of n/cm^{-2} (neutrons per centimetre squared).

The cross-section (σ) is the standard metric to evaluate the sensitivity to radiation of a device [19]. The cross-section represents the radiation-sensitive area of the device. By performing a beam experiment the cross-section can be derived by

dividing the number of errors by the total particle fluence. The **fluence** is the number of particles hitting the device per unit area. The cross-section is calculated with the following formula:

$$\sigma = \frac{\text{number of errors}}{\text{fluence}}$$

The Mean Time Between Failure (MTBF) of a system, is defined as the average time between two radiation-induced failures on the system continuously executing a given task. The MTBF is defined as follows:

$$\text{MTBF} = \frac{1}{\sigma * \text{flux}}$$

A more reliable system has a higher MTBF, which means the system can run for a longer time before experiencing a radiation-induced error. However, this metric does not take into account the workload. Therefore, the authors of [20] propose two new metrics. The Mean Execution Between Failures (MEBF), is the number of successful executions of an application between two radiation-induced failures. This value can be computed by dividing the MTBF, with the execution time t:

$$\text{MEBF} = \frac{\text{MEBF}}{t}$$

The author of [20] also proposes the Mean Workload Between Failures (MWBF). Every system is characterized by a workload w, which is the amount of data that needs to be processed for one execution. The MEBF can be even further generalized by multiplying with the workload.

$$\text{MWBF} = \text{MEBF} * w$$

2.2 Redundancy techniques

There are two radiation-hardening techniques: at a physical or a logical level. The key to detecting and correcting errors is redundancy, a processor without redundancy cannot detect any errors. Therefore the question becomes what kind of redundancy should be used. There are three classes of redundancy: physical, temporal, and information [21].

2.2.1 Physical Redundancy

Physical (or spatial) redundancy is commonly used for providing error detection. The simplest form of physical redundancy is Dual Modular Redundancy (DMR). With DMR, the module is duplicated and a comparator is added, when the output of the

Table 2.1: Single bit majority voting for TMR

Input	Output
000	0
001	0
011	1
111	1

modules mismatch an error has occurred. A Triple Modular Redundancy (TMR) system can be created by three modules and a voter. For a single error fault, TMR adds error recovery. A general redundancy scheme is N-Modular Redundancy (NMR), which for odd values of N greater than three provides better detection and protection than TMR. In table 2.1 an example of a single-bit majority voting for TMR is shown.

Physical redundancy can be implemented at various granularities. At a coarse level, the entire processor or core can be replicated. For a more finer grain, the ALU or a register can be triplicated. For a finer grain, TMR can be applied on a flip-flop level. The primary cost of physical redundancy is the hardware cost and power and energy consumption. TMR uses three times more hardware compared to an unprotected system.

2.2.2 Temporal Redundancy

Temporal redundancy can be achieved by performing an operation twice or more after each other and comparing the result afterwards. Therefore, the total time is doubled, but the hardware remains the same in contrast to the physical redundancy. However, the energy consumption is doubled, because twice as much work is performed. This redundancy can be implemented in both hardware and software. Pipelining can be used to reduce the latency, however, the throughput will still be penalized. Note that this strategy only protects the pipeline against SETs. De Sio et al. propose a software solution for replicating data and computations to cope with SEUs affecting the memory where the binary is stored [22].

2.2.3 Information Redundancy

The idea of information redundancy is adding redundant bits to a dataword to detect when an error has occurred. Error-Correction Code (ECC) detect and can sometimes correct single or multiple-bit errors. Single Error Correcting (SEC) codes can detect and correct one single-bit in a n-bit codeword. An Error-Detecting Code (EDC) checks the data and the parity bits for errors. There also exist Single-Error

Correcting and Double-Error Detecting (SECDED) codes which can detect and correct a single bit and are able to detect double-bit errors. The codeword bits n are constructed by adding parity bits p to the dataword bits m.

$$n = m + k$$

The distance between any two codewords is called the *Hamming Distance (HD)*. The Hamming Distance is denoted as $dist(x, y)$ and is equal to the number of places where they differ. For instance:

$$dist(1001, 1000) = 1,\ dist(1100, 1010) = 2$$

The *hamming weight* of a vector $x = x_1x_2...x_n$ is the number of $x(i) \neq 0$ and is defined as $wt(u)$. For instance:

$$wt(1000) = 1,\ wt(1011) = 3$$

The *minimum distance* of codeword, is the minimal Hamming distance between its codewords. The distance of a codeword can be computed with the Hamming weight:

$$dist(x, y) = wt(x - y)$$

The minimum Hamming Distance of a block code is equal to the minimum Hamming weight among all non-zero codewords. Figure 2.5 shows two different Hamming Distances for the same codeword. Figure 2.5 (a) shows a Hamming Distance of 2, every axis represents a single bit in the three-bit codeword. With a Hamming Distance of 2, there are three valid codewords. These codewords both share a Hamming Distance of 2. Figure 2.5 (b) shows a Hamming Distance of 3, now there are only 2 valid options, namely 000 and 111.

The minimum distance tells something about the error detection and correction ability. A minimum Hamming Distance of 2, belongs to a Single Error Detecting (SED) code, for instance, Parity encoding. With this code, an error can be detected, but cannot be corrected, because the position of the invalid bit is unknown. With a Hamming Distance of 3 a SEC, code can be realized, which can correct a single-bit error. An example of a SEC code already discussed is TMR. With TMR a single bit is replicated, and this is also shown in Figure 2.5 (b). There are only 2 valid words with a codeword of 3 bits. Another famous SEC code is the Hamming encoding. With a minimum distance of 4, a SECDED code can be achieved, for instance, Extended Hamming.

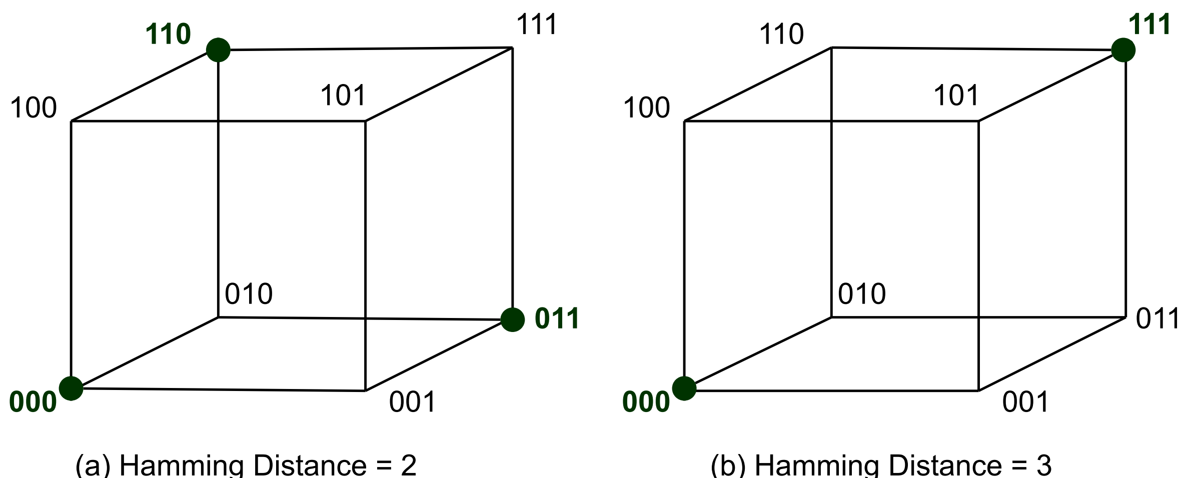

Figure 2.5: Hamming Distance 2 (a) and 3 (b) for a 3-bit codeword. The codewords marked with green are valid codewords.

Parity encoding

The simplest and most common form of EDC is parity. Parity adds one parity bit to a data word to convert it to a codeword. The parity bit indicates whether the amount of 1's is even or odd. When the number of 1's is even, the parity bit is 0, otherwise the parity bit is 1. Parity has a Hamming Distance of 2, resulting in single-bit error detection. An illustration of HD=2 with an example can be seen in Figure 2.5 (a)). Consider a valid codeword '110'. If a Single Event Upset (SEU) occurs, affecting the first bit, it results in '010'. If the SEU affects the second and third bits, we get '100' and '111' as possible outcomes. Importantly, these altered codewords are invalid, this invalidity allows the fault detection.

While parity-based EDC can detect single-bit errors, it cannot correct them. For instance, '010' can be connected to three valid codewords, making it impossible to determine which one it was originally.

Hamming

The Hamming code was invented by Richard W. Hamming in 1950 to address errors caused by punched card readers. This code is considered perfect because it achieves the highest possible efficiency for codes of their block length and minimum distance, which is three. Unlike simple parity codes, Hamming codes have the ability to not only detect but also correct errors. The extended Hamming code, on the other hand, incorporates an additional parity bit, resulting in a Hamming Distance of four. This expanded capability allows the decoder to differentiate between scenarios where only one bit has been corrupted and cases where two bits have been corrupted. Consequently, extended Hamming codes are known as SECDED codes.

Table 2.2: Hamming (7,4) Code: Parity coverage of the transmitted bits

Bit number	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Transmitted bit	*p1*	*p2*	*d1*	*p3*	*d2*	*d3*	*d4*
p1	T	F	T	F	T	F	T
p2	F	T	T	F	F	T	T
p3	F	F	F	T	T	T	T

Table 2.3: Hamming (7,4) Code: Parity coverage of the data bits

	d1	*d2*	*d3*	d4
p1	T	T	F	T
p2	T	F	T	T
p3	F	T	T	T

Hamming (7,4) Encoding & Decoding example

This example shows the encoding and decoding of Hamming (7,4). This Hamming encoding can detect and correct every single-bit error. Table 2.2 shows for every parity bit which transmitted bits are covered. For instance, parity bit p_1 provides an even parity bit for the transmitted bits: 1 (p_1), 3 (p_3), 5 (d_2) and 7 (d_4). Every data bit (d_1,d_2,d_3,d_4) is covered by 2 parity bits. Table 2.3 is created by removing the parity bits column from table 2.2. The parity-check matrix **H** is defined as table 2.2:

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The code generator matrix **G** is constructed as follows, rows 1, 2 and 4 of the matrix **G** correspond to the contents of Table 2.3. This table is placed at those rows because p_1 is transmitted as b_1, p_2 is transmitted as b_2 and p_4 is transmitted as b_4. The remaining rows map the data to their position in encoded form. Therefore rows: 3,5,6 and 7 form the identity matrix. This results in the following matrix:

$$G = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Suppose we want to transmit the message (**p**) ”1001”. The bits for the transmission codeword (**x**) can be computed by multiplying the generator matrix **G** with the

message (**p**):

$$x = G \times p = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Modulo 2 is taken of the product of **G** and **p** to determine the transmitted codeword **x**. This means that ”0011001” would be transmitted instead of ”1001”.

When no errors occur during the transmissions the received codeword (**r**) is identical to the transmitted codeword **x**.

$$r = x$$

The receiver multiples the parity-check matrix **H** with the received codeword **r** to compute the syndrome vector (**s**). This syndrome vector indicates whether an error has occurred. When an error has occurred this syndrome vector also indicates which codeword bit is wrong.

$$s = H \times r = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

In this case, the syndrome vector is a null vector, which means no error has occurred during the transmission. The data bits at positions (3,5,6 and 7) of the received codeword correspond to the original message ”1001”.

Suppose there is a single-bit error at b_2, the received codeword therefore becomes: ”0111001” instead of ”0011001”:

$$r = x + e_5 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$S = H \times R = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The syndrome vector is non-zero, which means that there was an error. The value of the syndrome is the bit place where the error is, namely 2.

Hamming codes can also be used for detecting dual-bit errors, by adding one overall parity bit as can be seen in Figure 2.6. In this figure a fourth parity bit has been added, resulting in an extended Hamming (8,4) presentation. To decode this code, first, the overall parity bit needs to be checked. If the parity bit indicates an error, the single error correction will indicate the error location.

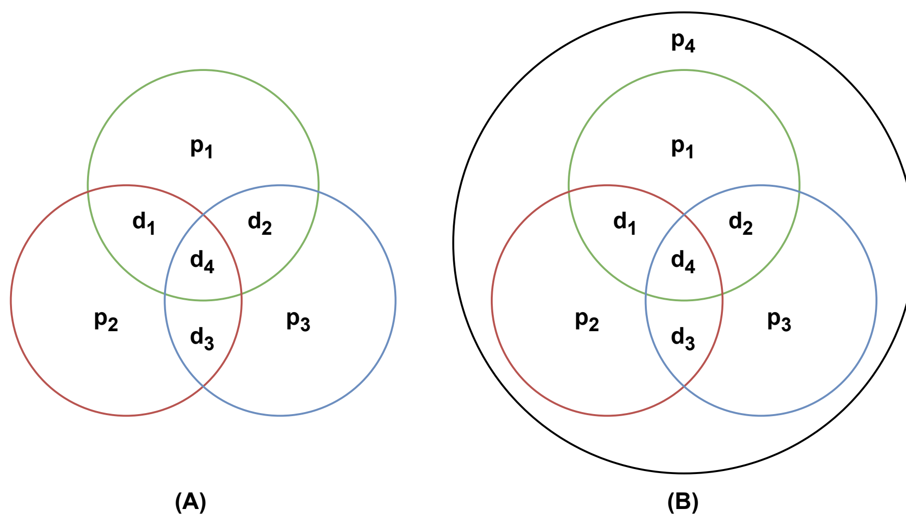

Figure 2.6: Venn diagram of the four data bits d_1 to d_4 and the parity bits p_1 to p_3

Hsiao

M. Y. Hsiao optimized the Hamming SEC-DED codes in 1970 [23]. The Hsiao SEC-DED code is an error correction code that introduces specific conditions for constructing the parity-check matrix.

The conditions of the **H** matrix for a SECDED code are:

1. There are no columns that contain only 0's.
2. Each column is distinct from every other column.
3. Each column contains an odd quantity of 1's.
4. The total count of 1's is minimized.
5. Each row in the H-matrix should have a number of 1's that is equal to or as close as possible to the average number, which is calculated as the total number of 1's in H divided by the number of rows.

The advantage of Hsiao over Hamming is that the number of 1's in the parity-check matrix is minimized resulting in faster calculations. Another advantage is when the odd-weight column-vectors are close to the average number of 1's in a row, therefore the amount of logic gate levels can be reduced, resulting in less delay. G. Tshagharyan et al. [24] demonstrated that Hsiao is more effective in terms of logic levels and area for larger word sizes.

2.3 RISC-V and FPGA Integration for Single Event Effects Mitigation

In addition to implementing redundancy techniques to mitigate SEEs, another critical aspect of ensuring the reliability and robustness of digital systems lies in the choice of the processor architecture and hardware platform. This consideration becomes particularly important when targeting Field-Programmable Gate Arrays (FPGAs), which offer flexibility and reconfigurability for various applications.

2.3.1 RISC-V Instruction Set Architecture

The RISC-V processor is an open-source instruction set architecture (ISA) designed for computer processors. This instruction set is based on the Reduce Instruction Set Computer (RISC) principles, which prioritize simplicity, efficiency, and modularity in processor design. The key concept of a RISC processor is that each instruction performs only one function, for instance, copying a value from memory to a register.

Compared to a complex instruction set architecture (CISC), a RISC processor might require more instructions in order to compute a particular task, but due to the simpler instructions these instructions can be executed at a higher speed.

The RISC-V project was created at the University of California, Berkeley, in 2010. In contrast to other academic designs, which are typically optimized for simplicity for teaching purposes, the designers of RISC-V intended it to be used in real-world processors.

One of the key advantages of RISC-V is its open standard, therefore anyone is able to access, use, modify and contribute to its specifications without any licensing fees or restrictions. This openness resulted in a vibrant community of developers, researchers, and companies actively contributing to the architecture's evolution. The RISC-V Foundation, a non-profit organization, manages and promotes the RISC-V ISA. Its mission is to drive the adoption of RISC-V by facilitating collaboration, providing education and support, and managing the architecture's specifications. Numerous companies, ranging from established semiconductor manufacturers and emerging startups, have embraced RISC-V and developed compatible processors and related technologies.

Chapter 3

Related Work

This chapter shows the related work about neutron radiation testing of FPGAs with softcores and ASICs. Neutron beam experiments of custom RISC-V soft-cores and COTS SoCs have been performed thoroughly in the literature.

The work of Wilson et al. [25] compares the neutron soft-error reliability of an unmitigated and TMR version of a Taiga RISC-v soft processor on a Xilinx SRAM-based FPGA. The TMR version showed a 33x reduction in the neutron cross section for a cost of around 5.6x resource utilization. The authors introduce two experimental designs on the Xilinx Kintex Ultrascale KU040 FPGA. One design contained 20 unmitigated processors and the other contained 20 TMR processors. The Dhyrstone benchmark was run during the neutron radiation tests. After every iteration, the checksum and iteration count were reported as the CPU status over a JTAG interface. These values were compared to a golden for correctness. The BYU BL-TMR tool was used to triplicate the Taiga processor.

Besides RISC-V there also exist literature about the LEON3. Keller et al. [26] compares a variety of mitigation techniques of the LEON3 soft processor with fault injection and neutron radiation testing. They show that fault injection can be a good way to estimate the cross-section of a design before going to a radiation test. They have tested 5 mitigation variants: an unmitigated version, TMR without scrubbing, TMR with BRAM scrubbing, TMR with CRAM scrubbing and TMR with BRAM & CRAM scrubbing. For the fault injection, the TMR & CRAM scrubbing resulted in a 27.28x improvement, by adding BRAM scrubbing the improvement was even increased to 51.30x. The numbers of the neutron radiation tests are closely related, the TMR & CRAM scrubbing resulted in a 26.94x improvement and the fully mitigated variant has an improvement of 48.85. During the test, the Dhyrstone 2.1 benchmark was run in a continuous loop. Fine-grain TMR was applied, which means that all flip-flops are triplicated.

In addition to utilizing SRAM-FPGAs for radiation tests, flash-based FPGAs have been used. Santos et al. [27] tested their own developed low-cost Riscv CPU Hard-

ened Risc-V (HARV). HARV has some basic hardening features to provide an alternative with competitive silicon and power overheads. It is a single-cycle processor and applies ECC on all internal registers and data, increasing the width by 7 bits. TMR is applied on the Arithmetic Logic Unit (ALU) and control unit. CoreMark was used as a benchmark during the radiation test. The design was implemented on the Microchip M2S010 SmartFusion2 Flash-FPGA. The authors have tested four configurations: without hardening, only hardening the processor, only hardening the memory, and hardening the processor and the memory. The latter one executed the CoreMark benchmark without errors in all 221 CoreMark executions. Each execution took 12 minutes to compute. The non-protected variant has a success rate of 73.08.

Besides FPGAs, ASICs also have been tested with neutron radiation. Dos Santos et al. investigate the error rate of a commercial RISC-V ASIC to a neutron beam [28]. They tested the GAP8 platform from GreenWaves, it has a cluster of 8 RISC-V cores. They show that code with more synchronization actions between the main core and the cluster of cores and more memory operations, for instance, FIR, have a higher DEU rate. They show that in computations for Convolutional Neural Networks (CNN) the error rate can be 3.2x higher than the average error rate. In addition, the majority of the errors (96.12%) on the CNN do not generate misclassifications.

Canizzaro et al. [29] evaluate the SEU susceptibility of two commercial RISC-V processors, the Microchip PolarFire SoC and the SiFive HiFive Unmatched, in the presence of neutron radiation. Both devices are compared with the flight-proven Xilinx Zynq-7020 system-on-chip, which has an ARM Cortex-A9 processor integrated. The ARM Cortex-A9 architecture, released in 2008, has been used in flight-proven devices such as the CHREC Space Processor (CSP) and the SHREC Space Processor (SSP). Bot CSP and SSP have performed missions in the ISS successfully. Both systems are not radiation-hardened but are fault-tolerant devices that use rad-hard power management and a watchdog subsystem. Both the RISC-V SoCs have parity detection for the L1 caches and SECDED and ECC available for their L2 caches. The DDR memory ECC capabilities were not enabled by the platform vendors. The ARM Cortex-A9 provide parity detection for its L1 and L2 caches, which was enabled for both memories. The ECC on the DDR memory is available but like the RISC-V boards not enabled. The EEMBC CoreMark and SHREC SpaceBench benchmarks were used to evaluate the presence of data and execution errors. CoreMark is a synthetic benchmark developed by the Microprocessor Benchmark Consortium (EEMBC) used to evaluate single-core performance. Originally, it was designed to be an improvement of the popular Dhrystone benchmark. The authors did not use CoreMark for performance but only counted the successful

execution of the benchmark. SHREC was developed by Dr Tyler Lovelly and is a suite of nine kernel benchmarks, containing matrix multiplication, addition, convolution and transposition. The datatype selection is configurable and provides parallel processing support. All the benchmarks ran on the Linux operating system, which was booted from an SD card. The PolarFire and the Unmatched had no errors in 99.70% and 99.59% of the operations. The Zynq had only 65.23% error-free operations. The PolarFire, Unmachted, and Zynq experience data errors in 0.02%, 0.03%, and 16.10% of the operations. These results and the radiation data of the following cross sections were found for PolarFire, Unmatched and Zynq: $8.033 * 10^{-12} cm^2$, $8.342 * 10^{-12} cm^2$, and $3.759 * 10^{-9} cm^2$. The cross-section of the Zynq is much larger compared to the RISC-V platforms, therefore the commercial RISC-V devices have much lower SEU susceptibility compared to the flight-proven Zynq platform.

The related work in table 3.1 provides a comprehensive overview of the key elements discussed in this section. This table summarizes the tested cores, platforms, enhancements, tests and evaluations from the various research works.

Table 3.1: Schematic summary of the related work for radiation experiments with FPGAs and ASICs

Core & Platform	**Redundancy**		**Test**		**Communication**			**Benchmark**		
	TMR	**ECC**	**Neutron**	**Simulation**	**Uart**	**Jtag**	**Other**	**CoreMark**	**Dhrystone**	**Other**
Taiga on Xilinx Kintex Ultrascale KU040	X		X			X			X	
LEON3 on Xilinx KC705	X	X	X	X		X			X	
HARV on Microsemi M2S010	X	X	X		X			X		
GreenWaves GAP8			X				X			X
Microchip PolarFire SoC & SiFive HiFive		X	X		X			X		X

Chapter 4

NEORV32

The NEORV32 processor is an upcoming customizable System on Chip (SoC) built around the RISC-V-compatible NEORV32 CPU [30]. This open-source RISC-V processor was released in 2020 and is written in platform-independent VHDL. Platform-independent code does not use any vendor-specific primitives, attributes, macros, or libraries. The goal of NEORV32 is to offer a RISC-V core with execution safety in mind. There are many open-source RISC-V implementations available, and most of them focus on performance or area. Because safety features play a crucial role in ensuring reliable options, particularly in demanding and harsh environments, the NEORV32 has been chosen as the core. Besides safety, it also focuses on documentation, platform independence, portability, RISC-V compatibility, extensibility & customisation, and ease of use.

4.1 NEORV32 Processor & CPU

The NEORV32 CPU is a 32-bit RISC-V processor based on the rv32i instruction set architecture (ISA). It is designed to be fully compatible with the RISC-V architecture and has successfully passed the official architecture tests, ensuring its compliance and reliability. The NEORV32 CPU offers a rich set of customization options, allowing the user to tailor its functionality to their specific needs. These options include adding privileged architecture, ISA extensions and custom options for design goals. One of the key design goals of the NEORV32 CPU is to support Full Virtualization capabilities for the CPU and SoC, which increases security. The CPU carries an official RISC-V open-source architecture ID for recognition within the RISC-V ecosystem [31].

NEORV32 Processor (SoC) is a complete microcontroller-like processor system with high configurability. It is built upon the NEORV32 CPU architecture. The processor system offers optional serial interfaces, including UARTs, TWI, and SPI, allowing

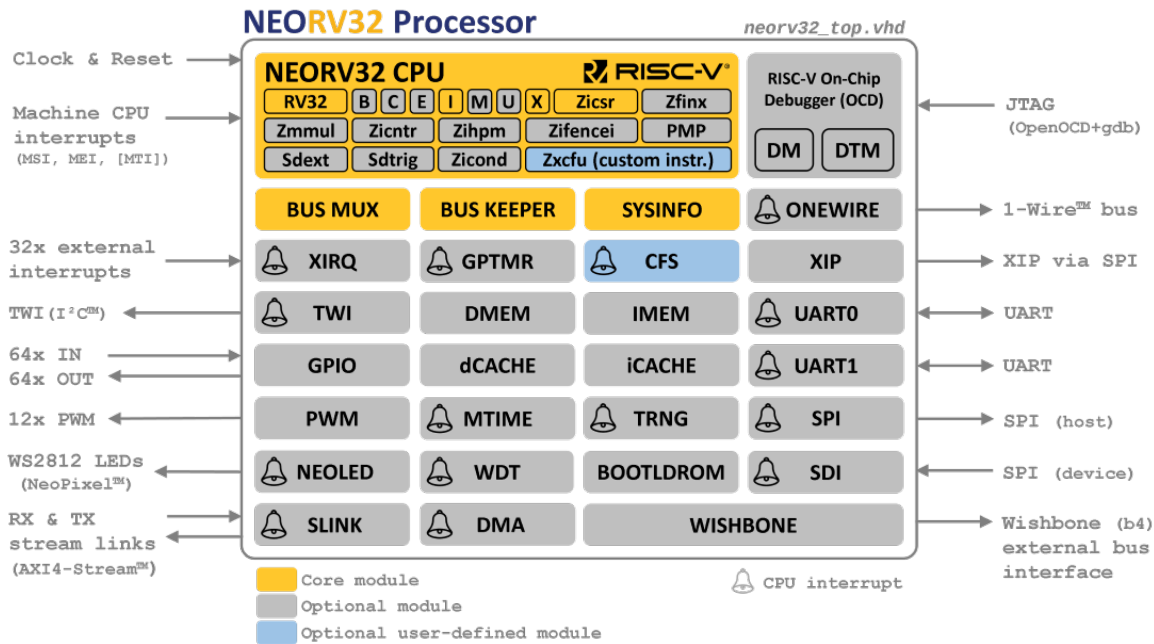

Figure 4.1: Overview of the NEORV32 Processor. *Source: S. Nolting [30]*

for versatile communication capabilities. In addition, it provides optional timers and counters, such as the Watchdog timer (WDT) and MTIME. To enhance its functionality, the NEORV32 Processor incorporates optional features like general-purpose IO and Pulse-Width Modulation (PWM). These features empower users to interface with external devices and control various peripherals. Furthermore, the processor system allows for optional caches for the data memory, instruction memory, and bootloader memory. For integration with external memory systems and custom connectivity, the NEORV32 Processor supports an optional external memory interface, namely Wishbone. Additionally, it provides a stream link interface (AXI4-Stream) to accommodate specialized data streaming needs. An optional execute-in-place (XIP) module is also available, enabling the execution of instructions directly from external memory, thereby improving performance. To facilitate debugging and troubleshooting processes, the NEORV32 Processor features an on-chip debugger that is fully compatible with OpenOCD and gdb. An overview of the CPU and SoC can be seen in figure 4.1.

4.2 RISC-V Standard Extensions Configurability

With the Application-Specific Processor Configuration, the SoC can be tailored to application-specific requirements. These configuration options are specified through generics within the top-level entity. By leveraging this flexibility, the SoC can be op-

timized for performance, size, area, and clock speed. Firstly, the B extension introduces additional instructions to support bit-manipulation operations. Compressed instructions (C) are available to reduce program size, thereby optimizing memory utilization. The Embedded (E) extension minimizes the register file size from 32 entries to 16, enabling a more compact design. The hardware-based integer multiplication and division (M) extension, offers dedicated hardware support rather than relying on software-based emulation. Moreover, the user-mode (U) extension adds a second less-privileged operation mode. Furthermore, the following Z extensions are available to enable: Zifencei, Zmull, Zxcfu, Zicond, Zfinx, Zinctr, PMP, Zihpm, Sdext, and Sdtrig. The Zicsr extension, which enables access to Control and Status Registers (CSRs), is enabled by default. Besides these extensions, NEORV32 also offer two microarchitecture features, the *FAST_MUL_EN* feature implements a faster multiplier by using DSP blocks and *FAST_SHIFT_EN* implements faster pull-parallel barrel shifters by using more area.

4.3 Pipeline

The CPU's architecture is a pipelined multi-cycle approach. Each instruction is executed through a series of consecutive micro-operations. To enhance performance, the CPU separates its front-end (instruction fetch) and back-end (instruction execution) using a FIFO known as an instruction prefetch buffer. This allows the front-end to fetch a new instruction while the back-end is still processing the previously fetched instruction. An overview of the NEORV32 CPU can be seen in figure 4.1. The CPU's microarchitecture lies between a traditional pipeline design, where each stage requires exactly one processing cycle (unless stalled), and a classical multi-cycle design, where each instruction is executed in a series of consecutive micro-operations. This combination results in an increased instruction execution compared to a pure multi-cycle approach because the fetch and execution operations can be overlapped. The hardware footprint is still reduced due to the multi-cycle concept. It is important to note that the CPU does not perform any speculative or out-of-order operations. Therefore, the CPU is not susceptible to security issues caused by speculative execution, such as Spectre or Meltdown [32].

4.4 Memory Access

The CPU has a unified 32-bit address space, where all memory addresses, including those for peripheral devices, are mapped. The instruction fetch interface and the data access interfaces are multiplied by a simple switch, into a single processor-

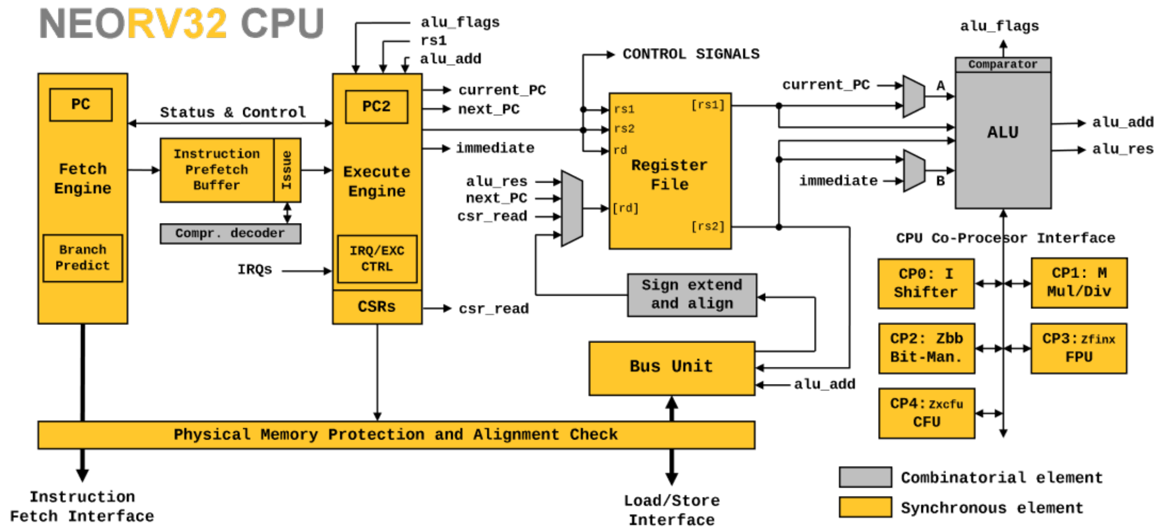

Figure 4.2: Overview of the NEORV32 CPU. *Source: S. Nolting [30]*

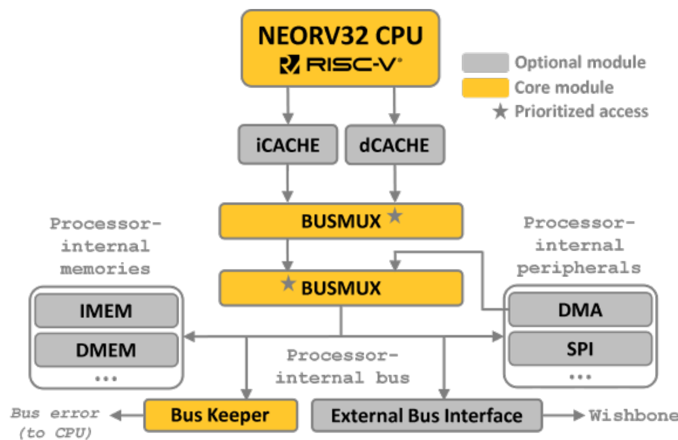

Figure 4.3: Processor-internal bus architecture. *Source: S. Nolting [30]*

internal bus, via a bus multiplexer. This bus multiplexer prioritizes data access. Additionally, this bus multiplex also multiplexes between peripherals like Direct Memory Access (DMA) and Serial Peripheral Interface (SPI). The bus multiplexer prioritizes the instruction fetch and data access above the peripherals. An overview of this bus is shown in 4.3. Because both the instruction fetch and data access have access to the same identical address space, this processor can be classified as a modified von-Neumann architecture.

The default NEORV32 linker script uses all the available RAM for several sections, some of these sections can be empty. At the beginning of the RAM, the constant data (.data) is stored. This section is used for explicitly initialized global variables, which are initialized by the executable. The dynamic data (.bss) section is placed after the constant data which contains uninitialized data without explicit

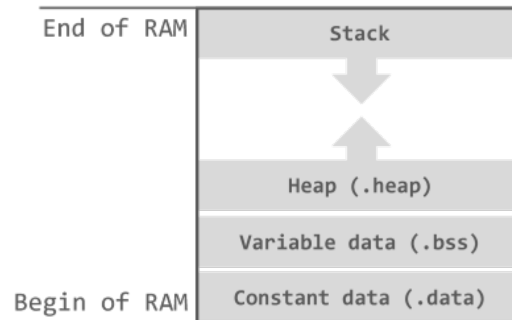

Figure 4.4: Default RAM layout of NEORV32. *Source: S. Nolting [30]*

initialization, this section is cleared by the start-up code. After the dynamic data the heap (.heap) is placed, this location is dynamic and grows to the end of the RAM. The heap is used for functions like *malloc()* and *free()*. This section is not initialized. The stack starts at the end of the RAM and grows backwards. See figure 4.4 for an overview of the used memory layout.

4.5 Execution safety

The NEORV32 has a special focus on execution safety to provide defined and predictable behaviour at any time. Therefore, the CPU ensures that all memory access is acknowledged and no illegal or malformed instructions are executed. When an unexpected situation occurs, the software is notified via a hardware exception. The Bus Keeper is a crucial component of the processor's internal bus system, responsible for ensuring proper bus operations while maintaining execution safety. It closely monitors every bus transaction initiated by the CPU. If a device being accessed responds with an error condition or fails to respond within a specific access time frame, a corresponding bus access fault exception is triggered. The following exceptions can be raised by the bus keeper:

1. TRAP_CODE_I_ACCESS: error during instruction fetch bus access
2. TRAP_CODE_S_ACCES: error during data store bus access
3. TRAP_CODE_L_ACCESS: error during data load bus access

By default, the access time frame is set to 15 clock cycles. The Bus Keeper's control register can be used to retrieve further details of the exception. There is a flag that indicates an actual bus access fault has occurred. This flag is cleared when a read or write occurs to the control register. There is an additional bit which indicates the type of bus fault. A 0 indicates a device error, and a 1 indicates a timeout error. In addition to these 3 exceptions, there are other exceptions, which are shown in Table 4.1. The NEORV32 supports all traps specified by the RISC-V.

Table 4.1: List of NEORV32 Exceptions

Prio.	mcause	Cause	mepc	mtval
1	0x00000000	instruction address misaligned	I-PC	0
2	0x00000001	instruction access bus fault	I-PC	0
3	0x00000002	illegal instruction	PC	CMD
4	0x0000000B	environment call from M-mode	PC	0
5	0x00000008	environment call from U-mode	PC	0
6	0x00000003	software breakpoint / trigger firing	PC	PC
7	0x00000006	store address misaligned	PC	ADR
8	0x00000004	load address misaligned	PC	ADR
9	0x00000007	store access bus fault	PC	ADR
10	0x00000005	load access bus fault	PC	ADR

4.6 Wishbone interface

To communicate with external memories, the NEORV32 processor utilizes a Wishbone interface. Figure 4.5 depicts a timing diagram for a Wishbone read transaction from the master's perspective. The read operation spans two clock cycles. In the second clock cycle, the read operation begins. The address for the read is set during this cycle, along with the *wb_sel_o* signal that indicates the specific bytes the master expects to receive data from. Throughout the read transaction, the *wb_cyc_o* signal remains asserted, indicating an ongoing transaction. The *wb_stb_o* signal is also asserted during the entire transaction, indicating a valid data transfer cycle. To signify that it is a read transfer, the *wb_we_o* signal remains low throughout the transaction. In the third clock cycle, the *wb_ack_i* signal is set high to confirm the completion of the read transaction. The requested data is available on the *wb_dat_i* signal. After the third clock cycle, some signals return to a low state, indicating the end of the transaction, while others may become undefined.

Figure 4.6 illustrates a write transaction using the Wishbone interface. This write operation spans two clock cycles and initiates in the second clock cycle. To announce a write transaction, the *wb_we_o* signal is asserted. In the third clock cycle, the slave acknowledges this write request by asserting the *wb_ack_i* signal. It is crucial that the data, address, and *sel* signal remain stable during this write request to ensure proper data transfer. After the third clock cycle, in the fourth clock cycle, some signals are set to a low state, indicating the end of the transaction, while others may become undefined. When the *wb_err_i* signal is asserted, it means that an error has occurred. The specific nature of the error can vary depending on the implementation and context. It could indicate an invalid operation, a data corruption issue, or any other error condition that has been defined for the particular system.

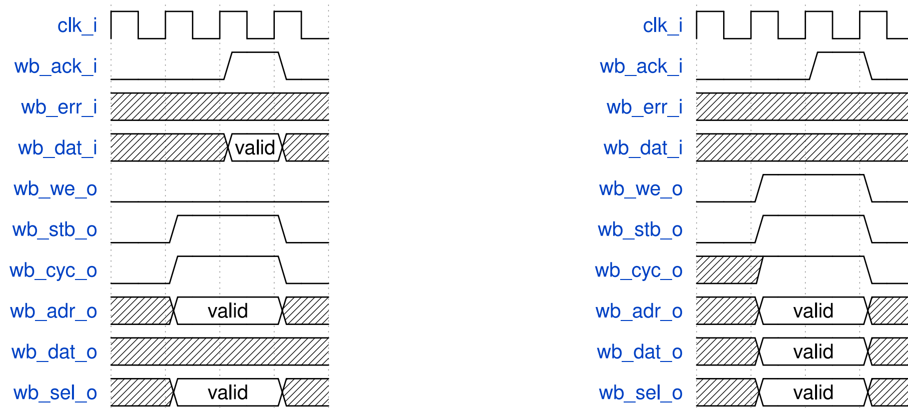

Figure 4.5: Wishbone read transaction **Figure 4.6:** Wishbone write transaction

The error signal is typically used to inform the master that the requested transaction could not be completed successfully. In the case of NEORV32, a bus exception will be raised.

Chapter 5

NEORV32 Implementation

This chapter describes the implementation of the NEORV32 on the Microsemi SmartFusion2 FPGA. The first section provides details about the target device. The second section will explain how the eNVM memory is connected to the Wishbone output of the NEORV32. The next section explains the implementation of the fault tolerance enhancements. Additionally, it provides the FPGA resource usage and it shows a power estimation of these enhancements. The fourth section discusses the testing of the implemented fault-tolerant adjustments. The complete overview of this design created in Libero SoC 2022.3 can be seen in A in Figure A.1.

5.1 Target Device

The Microsemi SmartFusion 2 SoC FPGA combines a flash-based FPGA fabric and a microcontroller subsystem (MSS) on a single chip. The MSS provides an ARM Cortex-M3 processor, Embedded Non-Volatile Memory (eNVM), embedded SRAM, and a high-performance interface [33]. For this research, the M2S010 variant of the SmartFusion 2 product line was used. This FPGA provides 256 KB of eNVM and 64 KB of embedded SRAM in the Microcontroller Subsystem (MSS). The FPGA fabric has 12K logic elements, 22 Math blocks (18x18), 2 PLLs and CCCs and 400K bits of RAM. The Trenz SMF2000 board, which can be seen in figure 5.1, includes this FPGA on a PCB with communication interfaces.

The FIT rate of the flash FPGA configuration memory is zero, therefore this FPGA is immune to SEUs in the configuration memory [34]. In addition during the testing of flip-flops, LSRAM blocks, and uSRAM blocks, no multiple-bit upsets were detected within any word. There are two reasons for this, one is the physical distance between adjacent bits in the 65 nm manufacturing node and the other reason is due to the interleaving of logical bits in the physical implementation of the memory blocks. High energy particles cause multiple-bit upsets, only cause single-bit upsets

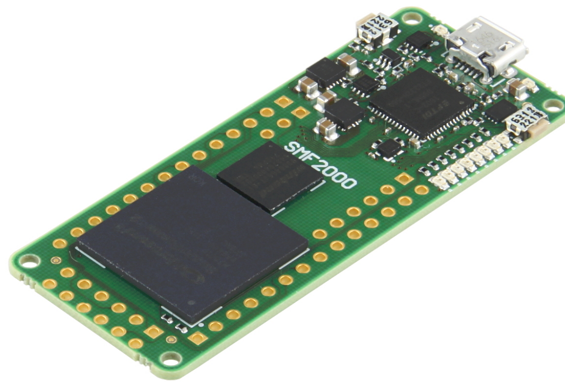

Figure 5.1: SMF2000 FPGA Module with Microsemi SmartFusion2

to logical words [35]. This makes this FPGA suitable for safety-critical and mission-critical systems. In the provided diagram, labelled as Figure 5.2, the floorplan of the M2S050 device is depicted. Although this particular device is not the intended target, it belongs to the same family and gives an idea of how the fabric is constructed. The diagram reveals that the LSRAM and uSRAM blocks, along with the math blocks, are situated within the FPGA fabric.

5.2 Unmitiaged NEORV32 implementation

This section explains how the NEORV32 is implemented on the SmartFusion 2 FPGA. In the first subsection, the memory access is explained. The second subsection explains the bridge interface between the NEORV32 Wishbone and the AHBL-Lite eNVM interface. The NEORV32 CPU is configured as RV32IMCZihpm, which means that a 32-bit architecture is used together with Integer instructions (I), Multiply and divide instructions (M), Compressed instructions (C) and Hardware performance counters (Zihpm).

5.2.1 Memory access

The data memory was implemented in VHDL with the provided VHDL file by NEORV32. By default, the address of the data address was set to 0x80000000. The data size of the Data Memory (DMEM) was 16kB and has a word length of 32 bits. The synthesis tool recognised the RAM implementation and implemented this memory block as BRAMs. Because BRAMs use SRAM technology, this memory is susceptible to SEUs. The data memory is implemented in VHDL as 4 blocks of one byte because RISC-V supports the writing of half-words and bytes to memory. When only one byte is written, there is only one write to one block and the other 3

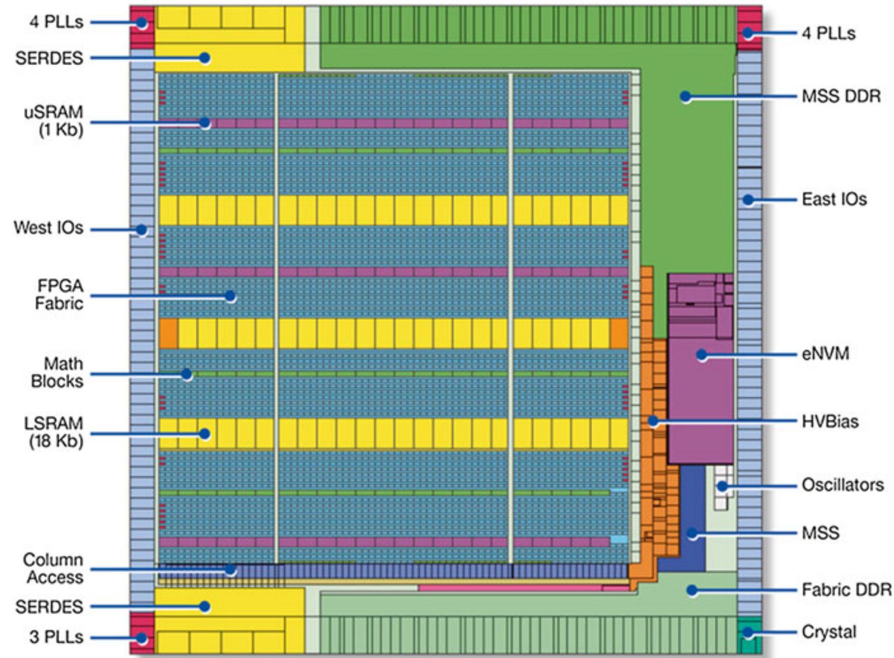

Figure 5.2: Floor plan of M2S050 device and the location of each functional block. *Source: Rezzak et al. [36]*

blocks remain unchanged.

The main goal of this thesis is to test the RISC-V architecture in a radiation-filled setting. The choice to save instructions in eNVM is important for reaching this goal. When instructions are stored in eNVM, which can withstand SEUs, the testing becomes more reliable. The other option is storing instructions in SRAM, but this can lead to SEUs causing many Illegal Instructions, which might be the main reason for errors. Opting for the more robust eNVM memory means the focus can be on assessing the architecture, not just the vulnerability of the instruction memory. In a similar study, Douglas et al. discovered that in the first radiation test using the same board, the instruction memory was the weakest point of the SoC [27]. For the second radiation test, they have decided to use the eNVM instruction memory.

During the execution, there are no writes to the memory, so the eNVM will act as a read-only memory. Because the Microsemi eNVM memory has an address of 0x60000000 in the MSS, the choice was made to use this address in NEORV32 as well, therefore no address mapping was necessary. The SmartFusion2 eNVM is a component of the MSS [37]. It is accessed through the eNVM Controller, which operates as a slave to the MSS AHB Switch Matrix, shown in figure 5.3. The Masters of the AHB Switch Matrix, specifically the MSS Cortex-M3 and a Fabric Master, have the ability to read from and write to the eNVM. The Fabric Master can access the Switch Matrix via one of the two Fabric Interface Controllers (FICs): FIC_0 and FIC_1 located in the MSS.

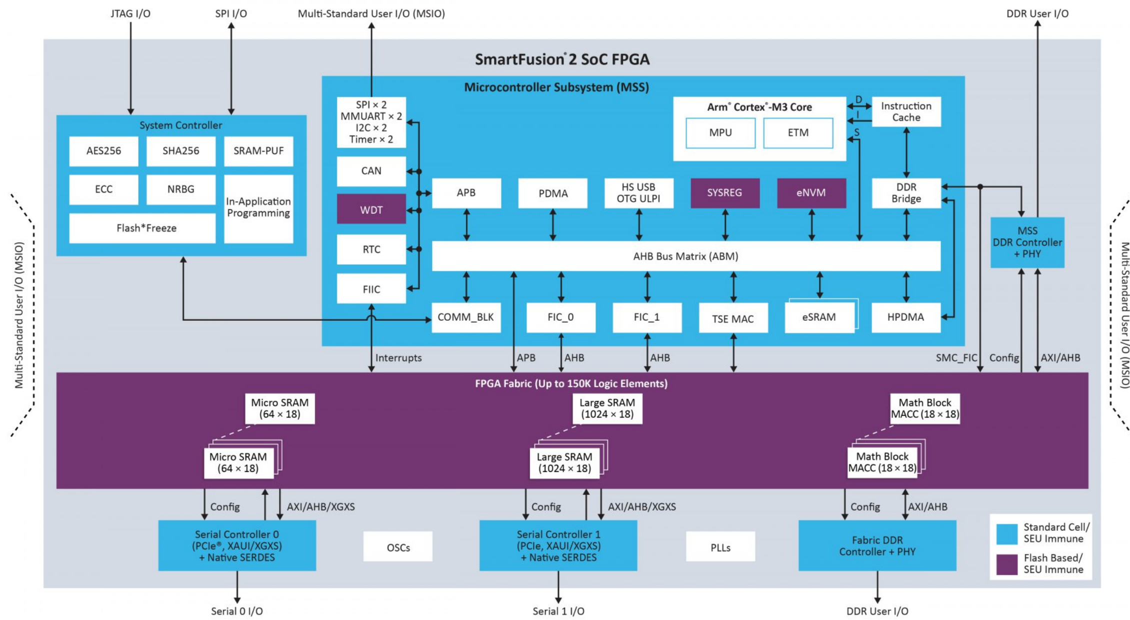

Figure 5.3: SmartFusion 2 FPGA Architecture

These FIC blocks provide an interface from the MSS AHB-Lite (AHBL) bus to user masters or user slaves in the FPGA fabric. Each FIC block performs an AHBL to AHBL or AHBL to APB3 bridging between the AHB Bus Matrix and AHBL or APB3 bus in the FPGA fabric. There are two bus interfaces for the FIC, the first one provides a master in the FPGA fabric and a slave in the MSS. The other options provide a slave in the FPGA fabric and a master in the MSS. The bus interfaces to the FPGA fabric are 32-bit AHBL or 32-bit APB.

As described earlier, the NEORV32 provides the Whisbone as an external bus interface. In addition, it also provides a wrapper which converts this bus interface to an AXI4Lite. Because the NEORV32 and the FIC do not provide the same bus interfaces, a bridge is necessary. The choice was made to use the Wishbone bus interface of the NEORV32 and the AHBLite of the FIC, because these busses are very similar, resulting in a simple Wishbone-to-AHBL bridge. A complete overview of the system can be seen in Figure 5.4. Everything is implemented in the FPGA fabric except the block marked in red which is located in the MSS.

5.2.2 AHBL-Wishbone Bridge

Table 5.1 displays the input and output signals for the Wishbone and AHB Lite bus interfaces. Some signals share the same encoding, allowing for a direct connection

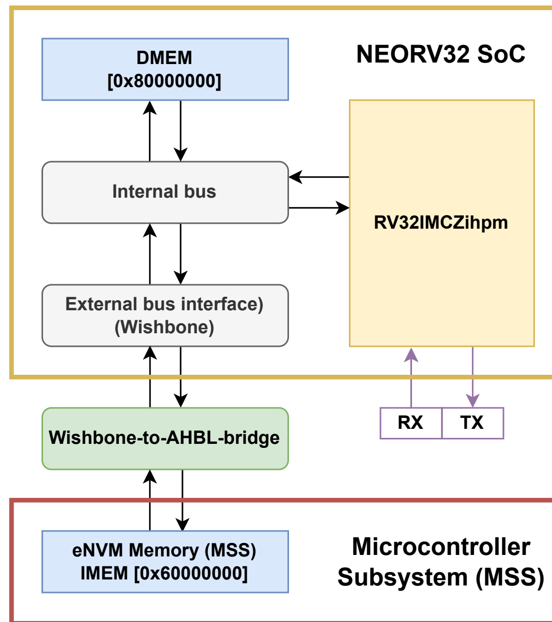

Figure 5.4: NEORV32 Configuration

between the corresponding signals of the two bus interfaces. For example, *HWRITE* and *we_i* have compatible encodings and can be connected directly. Similarly, *data_i* can be assigned to *HWDATA*, *we_i* can be assigned to *HWRITE*, and *HRESP* can be assigned to *err_o* because they have the same length and encoding, requiring no conversion.

The Wishbone STALL signal is not implemented in the bridge, as devices are not required to respond immediately to a bus response but within a specified time window.

According to the AMBA AHB protocol, transfers within a burst must be aligned to the address boundary equal to the size of the transfer for AHB-Lite [38]. Specifically, for a 32-bit transfer, *HADDR[1:0]* needs to be set to ”00”. Halfword transfers must be aligned to halfword boundaries, so *HADDR[0]* is set to ”0”. In the case of a word transfer, the bridge removes the two most significant bits of *addr_i* and adds two zeros at the end, assigning the resulting value to *HADDR*. For a 16-bit transfer, one MSB is removed, and a zero is added. For an 8-bit transfer, *addr_i* is directly assigned to *HADDR*.

The *stb_i* signal is delayed by one clock cycle and stored in *stb_i_dl*. This delayed signal is used to determine the values of *ack_o* and *HTRANS*. Since bursts are not supported, the bridge does not implement the busy and sequential transfer types. Therefore, *HTRANS* can be either idle or nonsequential. The bridge sets *HTRANS* to nonsequential when *HREADY* and *stb_i* are asserted, and *stb_i_dl* is 0. Otherwise, if there is no transfer, *HTRANS* is set to IDLE. A new transfer is indicated when *stb_i_dl* is 0 and *stb_i* is 1.

Table 5.1: Mapping of the signals in the AHBL-Wishbone bridge

AHB Lite input	Wishbone output	Mapping	Function AHB Lite input
HADDR	Addr_i	Mapping by appending zeros	Address
HWRITE	We_i	No conversion needed	Read/Write Transfer
HTRANS[1:0]	HREADY, stb_i, stb_i_dl	Special mapping needed	Transfer type
HSIZE[2:0]	Sel_i	Special mapping needed	Transfer size
HBURST[2:0]	000	No burst mode supported	Number of transfers and adress incrementation
HPROT[3:0]	0000	Protection control bus not used	Protection control
HWDATA[31:0]	data_i	No conversion needed	Data
Wishbone input	**AHB Lite output**	**Mapping**	**Function Wishbone input**
Ack_o	HREADY, stb_i, stb_i_dl	Logic AND operation of these 3 signals	Acknowledgement
data_o	HRDATA	No conversion needed	Data
Err_o	HRESP	No conversion needed	Error occurred

The transfer size is specified by *HSIZE*. This can be deduced from the *sel_o* signal, which indicates the used bytes. When *sel_o* is ”1111”, it represents a 32-bit word read or write, corresponding to ”010” in the AHB protocol. Both ”0011” and ”0011” are mapped to ”001” since they indicate a 16-bit transfer. Lastly, ”0001”, ”0010”, ”0100”, and ”1000” all specify an 8-bit transfer, which maps to ”000” in the AHB protocol.

The NEORV32 Wishbone interface currently does not support burst transfers, so *HBURST* is always set to ”000”. The protection control signals are not used, therefore *HPROT* is forced to ”0000”.

The *ack_o* signal is the logical AND operation of *HREADY*, *stb_i*, and *stb_i_dl*. The transfer is considered finished when *HREADY* is set high. However, since this cannot happen on the first cycle, the delayed *stb_i* is also checked.

5.3 Fault-tolerant enhancements

In order to evaluate the NEORV32 processor core, three different platforms were implemented: Unmodified NEORV32 processor, ECC-enhanced, and synthesis-level TMR with ECC-enhancements (TMR+ECC). All configurations are based on the RV32IMCZihpm implementation and use the eNVM for storing instructions. This section explains the ECC and TMR implementation, concluding with the resource usage by all implemented variants.

5.3.1 Design of Hsiao Encoder & Decoder

Because the register file and data memory are implemented as BRAMs, redundancy is needed to mitigate the errors caused by SEEs. Because applying DMR and TMR on memory elements is very costly in terms of hardware, information redundancy was used. The register file and data memory in the system were protected using Hsiao ECC. The encoder and decoder are written in Verilog.

The **encoder** takes a 32-bit input `enc_in` and produces a 39-bit output `enc_out`. The encoding process is performed in an always block, which is a combinational logic block that generates the encoded output based on the input. The `enc_out` is initially assigned the 39-bit value of `enc_in`. Following that, the individual bits of the `enc_out` are calculated for error detection using bitwise operations. Each bit of the `enc_out` is computed by performing a bitwise AND between the `enc_out` and a specific 39-bit mask. The resulting bits are then XORed together to obtain the parity bits. The specific 39-bit masks used for error detection are hardcoded in the module. These masks are applied to the `enc_out` to calculate the parity bits. The XOR operation of each mask with the corresponding bits of `enc_out` generates the parity bits for $b_{32}...b_{38}$ of `enc_out`. The encoded output `enc_out` contains the original 32 bits of data and an additional 7 parity bits that facilitate error detection and correction during decoding.

The **decoder** takes a 39-bit input `dec_in`, and produces a 32-bit output `dec_out`. Additionally, it provides outputs for the `syndrome`, `dec_syndrome_out`, and the error status `dec_errorout`. The syndrome calculation is performed using bitwise AND operations between the input `dec_in` and specific 39-bit masks. The resulting bits are then XORed together to obtain the syndrome bits. The correct output calculation is done by comparing the syndrome bits to specific values and XORing the corresponding input bits with the syndrome comparison result. Each bit of the output `dec_out` is calculated independently based on the syndrome bits and the corresponding input bits. The error status is determined by checking for a single error and a double error. The `single_error` signal bit is calculated by taking the logical XOR of all the

syndrome bits. When this signal is high, there was a single error detected and corrected. There is a dual error detected when the `singe_error` signal is low and the OR operation of `dec_syndrome_out` is high. The single-bit error detection signal is wired to `dec_errorout[0]` and the dual error detection to `dec_errorout[1]`.

5.3.2 ECC Implementation

To accommodate the (39,32) Hsiao encoding, the register file length was extended to 39 bits. When a read request occurs, both the memory addresses of operands `a` and `b` are read each time. Therefore, two decoders are necessary to handle the read operations. On the other hand, when writing, only a single encoder is required.

The RISC-V ISA supports writing bytes, half-words, and words. Therefore, applying ECC to the entire word length becomes more complex. In such a case, every write request for a byte or half-word would require a read request for the remaining non-written portion because it is needed for calculating the ECC bits. One solution to this problem is to extend writes with an additional clock cycle for reading, but this approach would lower performance. Alternatively, all write requests can be buffered in a cache before being written to the memory. However, this method requires more logic and is complicated to implement. For this study, the decision was made to apply ECC on each individual byte, which simplifies the implementation but requires four encoders and decoders. Hsiao (13,8) encoding was used for the data memory. An overview of the ECC implementation can be seen in Figure 5.5, there are 6 decoders and 5 encoders used. The parity check matrixes for the Hsiao (13,8) encoding can be found in Appendix A in Table A.4. The Hsiao (39,32) matrix can be found in Table A.5.

To prevent error accumulation inside the data memory, periodic scrubbing was applied. Every scrubbing operation reads all the words inside the memory one by one and writes back the corrected value. Because of the hardware decoder and encoder, there are only 2 assembly instructions necessary for performing the scrubbing. The *lw* loads a word from memory and the *sw* stores the word back to the memory. For simplicity, in this implementation, every word is written back to memory even though there was no error correction.

It is possible that the scrubber may encounter the reading of uninitialized memory. This uninitialized memory is highly likely to contain incorrect ECC bits. Because we are interested in the number of detections, it is crucial that these instances of uninitialized memory are not accounted for. Therefore, prior to activating the scrubber during each iteration, the entire data memory is initialized with a value of 0. The encoded representation of 0 is also 0, making the initialization to 0 effective approach. The register file is smaller and data is written more frequently, therefore

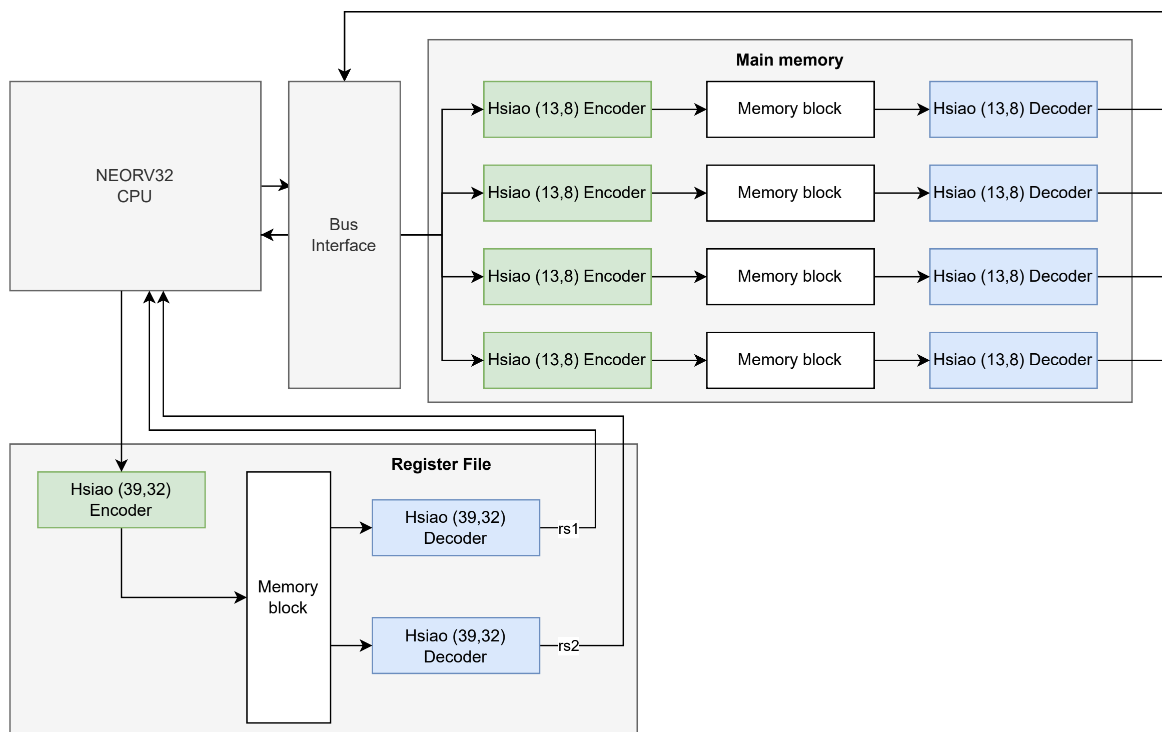

Figure 5.5: Overview of the ECC implementation

no scrubbing was used for the register file.

To provide observability, custom hardware-performance counters were integrated to track the ECC correction occurrences. A total of four counters were implemented, with two dedicated to counting single errors and dual errors separately within the DMEM. Additionally, the Register file also features two counters four counting the single errors and dual errors.

5.3.3 TMR implementation

Initially, the approach was to triplicate the entire NEORV32 processor to achieve fault tolerance. However, it was found that the number of 4-input lookup tables (4LUTs) required for this exceeded the available resources on the FPGA. As a result, the TMR option in the Synplify synthesis tool was utilized as an alternative solution. This option performs triplication at the flip-flop level by triplicating every flip-flop in the design.

Synplify, developed by Synplicity and acquired by Synopsis in 2008, is a synthesis tool for producing high-performance and cost-effective FPGA designs. It supports a variety of FPGA vendors, including Microsemi FPGAs. Synplify created specific features for Microsemi FPGAs and Radiation-Hardened FPGAs. It offers a feature that automatically infers either C-C (combinational cell), TMR (Triple Modu-

Table 5.2: MS2S010 Resource usage

	4LUT	DFF	uSRAM 1K	LSRAM 18K
Unmodified	6239	3014	3	8
ECC-enhanced	7086	3478	4	16
TMR+ECC	11666	8643	4	16

lar Redundancy), or TMR_CC (a combination of C-C and TMR) implementations in place of regular flip-flops [39]. This inference occurs during synthesis, eliminating the need for post-processing the netlist for flip-flop substitutions. The C-C implementation utilizes combinational cells with feedback to provide storage functionality, while the TMR setting applies triplication at the register level, with each register being implemented using three flip-flops or latches. TMR_CC is a combination of both implementations, where voting registers are composed of combinational cells with feedback. In the case of the NEORV32 CPU and the AHBL-Wishbone bridge, the TMR setting was chosen as a trade-off between protection and overheads.

Figure 5.6 provides an example of the TMR setting in the hierarchical netlist level. The flip-flop *fifo.r_pnt* is implemented as a Sequential Logic Element (SLE), which consists of one DFF (Data Flip-Flop) and one LE (Logic Element). These LEs are triplicated, and all the outputs are marked in red. Additionally, TMR utilizes a single 4-input lookup table (4LUT) for implementing the voter per flip-flop. In Figure 5.6, the voter is implemented using the CFG4 macro, while CFG1, CFG2, CFG3, and CFG4 are post-layout 1-input, 2-input, 3-input and 4-input LUTs used for implementing various combinational logic functions. The TMR table, as shown in Table 2.1, is stored in this CFG4.

5.3.4 FPGA resource usage

Table 5.2 shows the resource usage for the 3 different variants. The synthesis was performed with Libero SoC V2022.3. The TMR+ECC version needed the *High Effort Layout* option to be enabled in order to create a successful design.

BRAMs usage

The data memory size has been configured as 16KB, with a data width of 32 bits, enabling it to store 4069 entries. In the unmodified variant, the memory system utilizes 8 blocks of LSRAM 18K. Each LSRAM block is configured as 4KX4_4KX4 [40], which means it is a dual-port RAM with 4K entries of 4 bits on each port. The DMEM consists of 4 arrays of 8 bits, and during synthesis, each array is mapped to 2 LSRAM blocks configured as 4KX4_4KX4. As a result, the memory effectively pro-

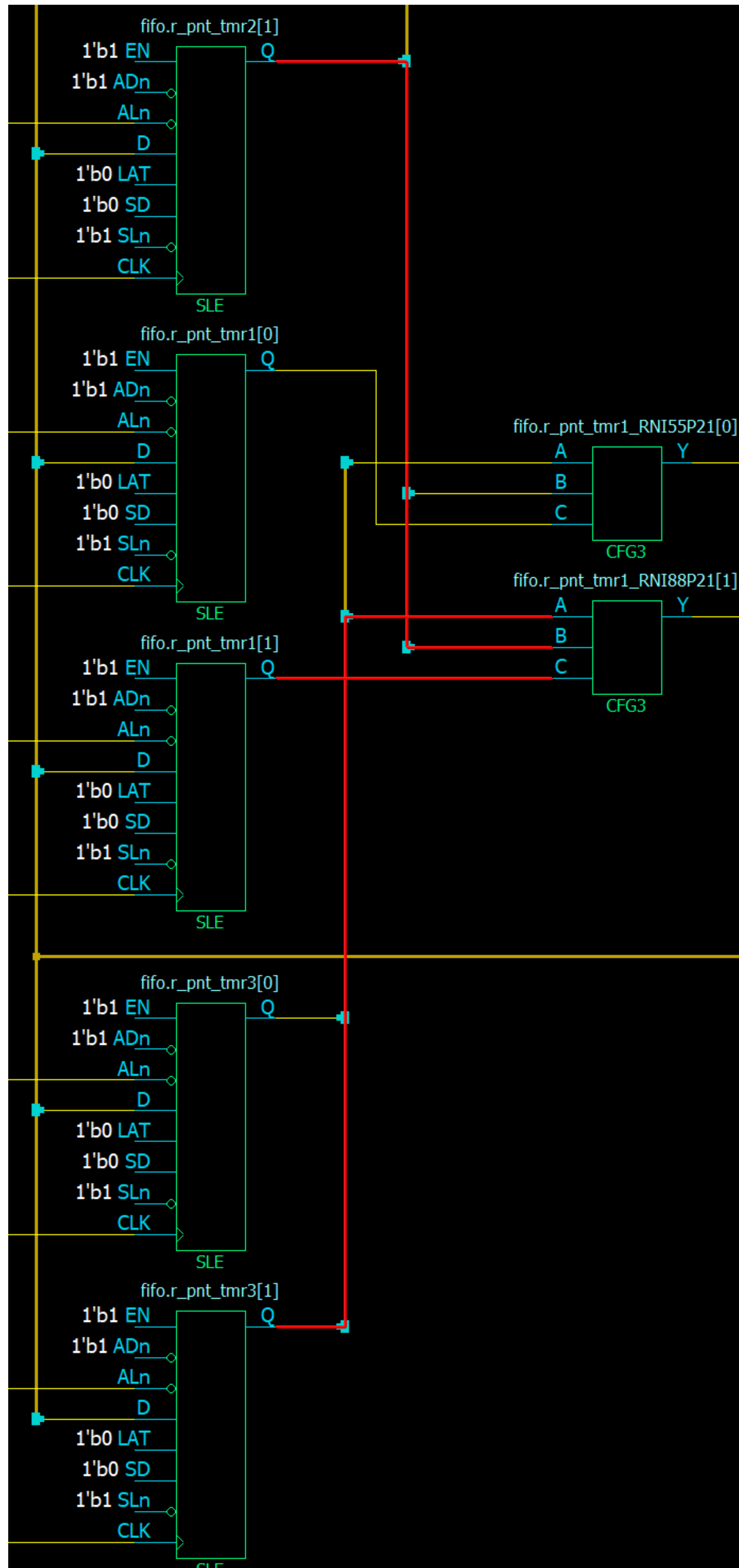

Figure 5.6: Example of Synplify TMR setting on hierarchical netlist level

vides 4K entries of 32 bits, as expected. This mapping ensures efficient utilization of the BRAMs, with all bits being utilized. Although there are alternative configurations possible, such as 4 blocks of 512x32, the synthesis tool chooses the current mapping due to the definition of byte-addressable banks. All memories are implemented as dual-port RAM.

The ECC-enhanced and the TMR+ECC both share the same LSRAM block usage because they use the same ECC implementation. This LSRAM block is doubled compared to the unmodified version because each of the 4 arrays stores now 13 bits instead of 8. These are mapped to 4KX4_4KX4 blocks, and these ECC bits cannot fit in one block, a second additional block is necessary, resulting in an increase of 2 blocks per bank. For 4 banks this results in an increase of 8 BRAM blocks. Notably, not all bits of the BRAMs are utilized, and a more efficient option could have been 10 blocks of 2Kx8. However, the synthesis tool does not identify this option due to the defined bank configuration.

The register file and TX FIFO of the UART module are mapped to uSRAM blocks. In the unmodified variant, the register file is stored in 2 uSRAM blocks configured as 64x18. The register file consists of 32 entries with a data size of 32 bits. The synthesis tool prioritizes larger block mappings, selecting 64x18 over 64x16, which would also fit the requirements. Additionally, the TX FIFO of the UART module meets the necessary threshold for mapping to uRAM and is therefore mapped to a 128X9 uSRAM block. In the ECC-enhanced and TMR+ECC versions, an extra uSRAM block is required for the register file due to the extension of its data width to 39 bits.

Logic usage

The original NEORV32 architecture utilizes 6239 4LUTs and 3014 flip-flops as specified in Table 5.2. Among these resources, the NEORV32 processor consumes 98% of the Fabric 4LUTs and 96% of the Fabric DFFs, see Table 5.3. Additionally, the bridge requires 94 4LUTs and 1 flip flop, indicating a small resource footprint. The top-level, responsible for interfacing with the MSS, uses 128 Fabric 4LUTs and 105 Fabric DFFs. Furthermore, the design involves the utilization of interface 4LUTs and DFFs, primarily for interfacing with BRAMs. This is evident in Table A.1 in the Appendix A, where various blocks such as the CPU, DMEM, and UART0 use interface logic and BRAMs. DMEM consumes the most interface logic but also instantiates the most BRAM blocks.

Upon implementing the ECC, there is a noticeable increase of 13.6% in 4LUT utilization and 15.4% in DFFs, see Table 5.2. Table 5.4 illustrates that the ECC processor uses 6228 Fabric 4LUTs and 2652 Fabric DFFs, representing an increase

Table 5.3: Resource usage Unmitigated NEORV32

Module Name	Fabric 4LUT	Fabric DFF	Interface 4LUT	Interface DFF
Primitives	2	0	0	0
WB2AHBL_O	9	1	0	0
Neorv32_Processor	5704	2512	396	396
Top_sb_0	128	105	0	0
Total	**5843**	**2618**	**396**	**396**

Table 5.4: Resource usage ECC NEORV32

Module Name	Fabric 4LUT	Fabric DFF	Interface 4LUT	Interface DFF
Primitives	2	0	0	0
WB2AHBL_O	8	1	0	0
Neorv32_Processor	6228	2652	720	720
Top_sb_0	128	105	0	0
Total	**6366**	**2758**	**720**	**720**

of 524 4LUTs and 140 DFFs. This increase is mainly attributed to the usage of encoders and decoders. A closer look at Table A.2 in Appendix A reveals that the interface logic for the CPU has grown by 1.5 times, driven by the adoption of 3 BRAMs for the register file in the ECC version (versus 2 BRAMs in the original). Similarly, the interface logic for the DMEM has doubled due to the doubled BRAM usage as discussed earlier. However, the UART module's resource usage remains unchanged as ECC is not applied to this block. Additionally, there are minor deviations in fabric resource usage across other blocks due to a different design and therefore a different place and route.

In the ECC-enhanced variant with TMR added, the 4LUTs experience a substantial 61.5% increase, while the DFFs surge by 148.5%, as indicated by the data in Table 5.2. Although a typical TMR solution involves a 200% increase, not all DFFs are triplicated in this case, because interface DFFs are not triplicated by the Synplify tool. This could be due to physical placement in memory blocks or strict timing constraints that do not allow for the additional voter propagation time. As shown in Table 5.5, the NEORV32 processor now utilizes 7814 Fabric DFFs, representing an almost expected triplication. Fabric 4LUTs have increased by 73.5%. TMR is also applied to the Wishbone-AHBL bridge, evident in the tripling of the single fabric flip-flop in the TMR design compared to the original design. The resource usage of the subparts of the ECC+TMR version can be seen in Table 5.5 in Appendix A.

In summary, the addition of ECC and TMR to the NEORV32 architecture impacts resource utilization significantly, with notable increases in both 4LUTs and DFFs. However, some DFFs remain non-triplicated, likely due to tool limitations or strict

Table 5.5: Resource usage ECC NEORV32

Module Name	Fabric 4LUT	Fabric DFF	Interface 4LUT	Interface DFF
Primitives	2	0	0	0
WB2AHBL_O	9	3	0	0
Neorv32_Processor	10806	7814	720	720
Top_sb_0	129	106	0	0
Total	**10946**	**7923**	**720**	**720**

timing constraints, resulting in a slightly different increase than expected for TMR.

5.3.5 Power Estimation using Microsemi's Smart Power Tool

In this section, the focus is on power consumption in the Active mode, which is the most relevant aspect of this analysis. Microsemi's Smart Power tool within the Libero environment was employed for power estimation. The tool has support for the Value-Change Dump (VCD) format generated by well-known simulators like ModelSim. It leverages switching activity over time to assess average power consumption. One encountered challenge was the size of VCD files, as they tend to become quite large, mainly because they must encompass all signals in the circuit after post layout for the most accurate power estimation. Consequently, the decision was made to use the default prediction settings provided by the tool. These default settings make certain assumptions about how frequently signals switch and the likelihood of such switching events. The outcomes of this predictive approach are detailed in Table 5.6.

Static power represents power consumption even when the circuitry is not actively switching. The data in Table 5.6 reveals that all configurations exhibit identical levels of static power. However, distinctions exist in dynamic power usage. Notably, the component that consumes the most dynamic power is the Built-in Blocks. These Built-in Blocks are integral parts of the MSS, such as the eNVM. It is worth noting that this part remains consistent across all configurations, as there are no alterations in the MSS between variants. Types like I/O, Core Static, Banks Static, and VPP Static contribute to dynamic power consumption and remain constant across all three configurations. Differences in I/O power are negligible. The key variations are observed in Nets, Gates, and Memory.

When comparing the ECC-Enhanced and ECC+TMR configurations, both utilise the same amount of memory, therefore their memory consumption is identical. However, compared to the Unmodified variant, both ECC-Enhanced and ECC+TMR configurations show nearly a doubling of memory consumption, corresponding to the increased number of hardware memory blocks. In particular, the ECC-Enhanced

Table 5.6: Estimated power consumption of the different configurations

	Unmodified		ECC-Enhanced		ECC+TMR	
	Power (mW)	Percentage	Power (mW)	Percentage	Power (mW)	Percentage
Dynamic Power	**69.941**	**86.4%**	**72.529**	**86.8%**	**82.120**	**88.1%**
Type Net	3.681	4.5%	3.942	4.7%	10.330	11.1%
Type Gate	8.854	10.9%	9.246	11.1%	12.449	13.4%
Type I/O	0.363	0.4%	0.359	0.4%	0.359	0.4%
Type Memory	2.122	2.6%	4.061	4.9%	4.061	4.4%
Type Core Static	8.262	10.2%	8.262	9.9%	8.262	8.9%
Type Banks Static	2.161	2.7%	2.161	2.6%	2.161	2.3%
Type VPP Static	0.625	0.8%	0.625	0.7%	0.625	0.7%
Type Built-in Blocks	54.921	67.8%	54.921	65.7%	54.921	58.9%
Static Power	**11.048**	**13.6%**	**11.048**	**13.2%**	**11.048**	**11.9%**
Total Power	**80.989**	**100.0%**	**83.577**	**100.0%**	**93.168%**	**100.0%**

configuration experienced a 7.09% increase in Net usage and a 4.43% increase in Gate usage compared to the Unmodified variant. Furthermore, when comparing the ECC+TMR version to the ECC-Enhanced, there is a significant difference of 162.05% in Net power consumption. There is also an increase of 34.64% in Gate usage due to the replication of flip-flops, resulting in a greater number of interconnections.

In conclusion, the ECC variant exhibits a mere 3.2% increase in overall power consumption, which is a slight increase. This marginal rise can largely be attributed to power-hungry components like the built-in blocks, whose power usage remains consistent across all configurations. However, only considering the power consumption of Nets and Gates does reveal a slight uptick in these power consumptions. While each of these components may individually have a relatively small impact, their cumulative effect becomes evident when assessing the ECC variant as a whole. In contrast, the ECC+TMR configuration demonstrates a considerably higher increase in power consumption, with a substantial 15.04% rise when compared to the unmodified version. This increase becomes particularly prominent when we consider only the additional blocks introduced, resulting in a striking 162.05% difference in Net power consumption and a significant 34.64% increase in Gates. These findings emphasize the need for careful consideration of specific components' power implications when implementing enhancements such as ECC+TMR, especially for space applications where power can be scarce.

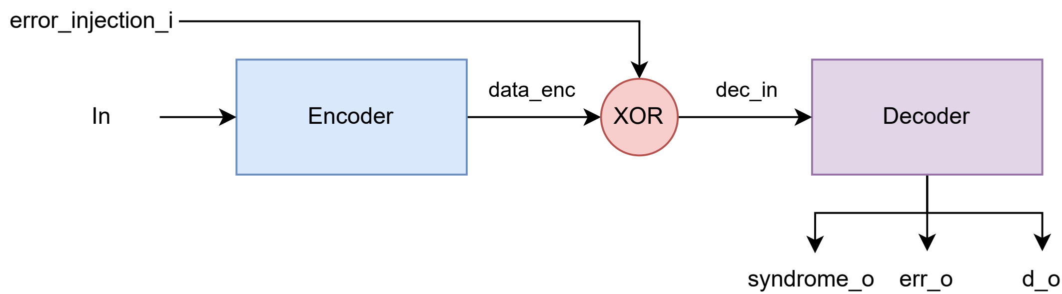

Figure 5.7: Schematic overview of the Encoder & Decoder Testbench

5.4 Testing of the Fault-Tolerant adjustments

The Hsiao(13,8) encoder and decoder, as well as the Hsiao(39,32) encoder and decoder, were manually tested using ModelSim. The test bench schematic, shown in Figure 5.7, instantiates both the decoder and encoder and connects the encoder's output to the decoder's input. In the absence of any fault injection, the output should match the input, and no errors should be detected. The test module includes an error injection signal in addition to the input signal. An XOR operation is performed on the encoder's output and the *error_inject_i* signal. The result of this XOR operation is forwarded to the decoder for decoding.

The purpose of this test bench is to evaluate the encoder and decoder's functionality and resilience against errors. By injecting errors using the *error_inject_i* signal, the test bench can assess how well the decoder handles and corrects these errors, if applicable. The XOR operation simulates potential errors in the transmitted data, and the decoder's output can be compared to the original input to verify error correction capabilities. Throughout the testing process, various test vectors may be used to assess different scenarios and potential edge cases.

A specific manual test case, shown in Figure 5.8, provides an example of this evaluation. Initially, when both the input and error inject signal are set to 0, the encoded value and error signal should also be 0, as expected. At 100ps, the input value is set to 0 with a single-bit flip in the Least Significant Bit (LSB) position. Although the encoded data remains 0, the error signal correctly indicates a single-bit error, confirming the decoder's error detection capability. The decimal value of the syndrome (13) indicates that the error occurred at position 13, which aligns with the actual injected error. At 200ps, a double-bit error is injected with the input value still set to 0. The error signal indicates a double-bit error, showcasing the decoder's ability to detect multiple errors. Subsequently, at 300ps, 400ps, and 500ps, tests are performed with non-zero input values without error injection, with a single-bit error, and with a double-bit error injection, respectively.

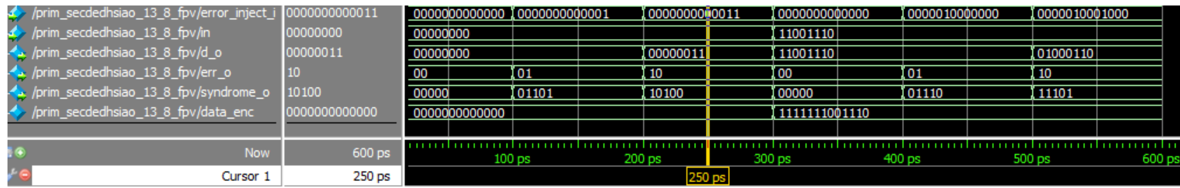

Figure 5.8: ModelSim simulation of the Hsiao(13,8) encoder and decoder with fault injection

Besides the ModelSim simulations, the ECC on the data memory has been emulated. A single-bit flip was introduced at the output of every encoder of the data memory in VHDL. Therefore all the data that is written to the memory will include a single bit error. This was tested with a small C program which sets up the Hardware Performance Monitorss (HPMs) and a while loop that writes the values 1 to 100 to an array of 100 items. After this writing, the data is read. As expected the HPM counter of the ECC DMEM single error had stored value 100. By removing the single-bit flip, the counter was 0.

The byte addressable bug was found with this emulation. By changing the data type of the array from *int32_t* to *int16_t* and without a single-bit flip, the ECC HPM showed errors.

Chapter 6

Neutron Beam Experiment

In order to study the impact of Single Event Effects (SEEs) on the NEORV32 System-on-Chip (SoC) implementation, we subjected the Device Under Test (DUT) to a high-energy neutron beam. Our experiments were conducted at the ChipIR beamline, located at Rutherford Appleton Laboratories in the United Kingdom. This facility accurately replicates the energy spectrum of atmospheric neutrons [41], making it a suitable approximation for simulating SEUs (SEUs) caused by protons. To evaluate the NEORV32 designs, we utilized two Trenz SMF2000 boards that contained SmartFusion2 [34] devices. Both boards were positioned in line with the neutron beam, and we accounted for the variation in distance from the source when calculating the total fluence.

6.1 Experimental setup

The experimental setup consisted of two Trenz SMF2000, which can be seen in figure 6.1. Both devices were programmed with the same bitstream, therefore the fluence of each experiment was doubled. The UART receive (Rx) and transmit (Tx) pins of the UART module within the FPGA fabric were routed to the external PMOD connector. This connecter was connected to the Digilent Pmod USB UART module, which converts this UART signal to a USB output. This module was connected to the Raspberry Pi through a USB cable.

The Raspberry Pi acts as a host computer and was responsible for processing the UART data and determining when a power reset was necessary. This Raspberry Pi was connected to a Lindy IPower Switch via an ethernet cable. This switch provides power to the 2 DUTs via a USB connection. The Raspberry Pi was able to enable and disable the power of individual FPGAs by sending commands to the IPower Switch when there was a hang or an exception. Because the configuration memory and the program memory were stored in Flash, reprogramming the FPGA

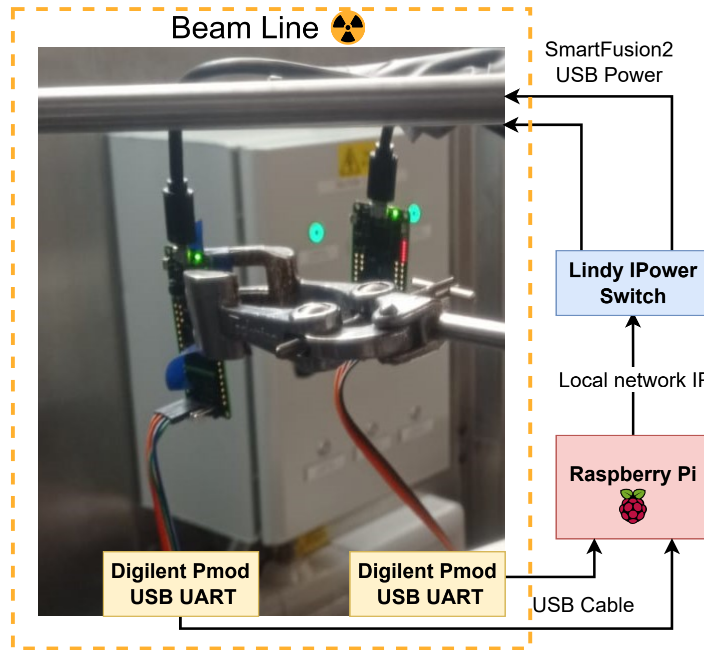

Figure 6.1: Diagram of the experimental setup

after a power reset is not necessary, assuming that the eNVM memory is immune to SEUs.

6.2 Software

The software layer consisted of bare metal code, comprising the UART peripheral communication, scrubbing routine, and the initialization of the HPMs. In this context, an execution or run refers to the execution of CoreMark, which involves multiple iterations.

Regarding **UART communication**, the transmission involved sending a burst of four characters. At the beginning of each execution, a unique identifier was sent twice to indicate the start of an execution. If an incorrect CRC (Cyclic Redundancy Check) was encountered during the computation of the list, matrix, or state, a specific value was sent. In cases where the CoreMark execution finished too quickly, which is not possible without an illegal jump, a predefined value was printed. In the event of errors such as mismatched CRC or excessively fast execution time, a distinct value was printed twice to indicate the execution has errors and the following prints contain the following information. Subsequently, the CRC values for the list, matrix, and state computations were printed, followed by the final CRC. The number of errors was then printed. Afterwards, the ECC counters were printed.

Finally, two specific prints were sent to indicate the end of the execution. Since only bytes were printed without any accompanying text, it is important to associate

each print with its corresponding statement for parsing the correct result.

The **CoreMark** benchmark was used as workload. To guarantee that the benchmark utilizes most of the resources of the device, the CoreMark data size was increased from 2K to 12K. 20 computations needed 1 minute and 44 seconds to compute. Initialization and outputting only take a small amount of time, therefore increasing the iterations is not necessary.

The **exception trap handler** was changed to output the `mcause`, `mepc`, and `mtval` registers, in order to find the cause and origin of an exception. The `mcause` contains the machine-level exception codes. These codes are shown in Table 4.1. When a trap is taken, `mepc` contains the virtual address of the instruction that encountered the exception. The `mtval` CSR provides additional information on why a trap was entered.

6.3 Error Model

The possible outputs of the experiment consist of:

1. **Match**: when the CRC output matches with the expected value.
2. **SDC**: indicates a difference between the CRC output and expected value. The program has finished but with an incorrect output.
3. **DUE**: The program does not finish correctly, either via a crash that triggers a hang or because of a detected exception.
4. **ECC corrections**: the number of single-bit corrections and double-bit detections provided by the ECC unit in the DMEM or register file. Either by the scrubber software, or normal data access.

The ECC SEC-DED is represented as a single category, but in reality, we had separate counters for single and double-bit errors and for the DMEM and register file.

For the calculation of the average cross section and the 95% confidence intervals, the methodology by Quinn [42] is used. Including confidence intervals in all experimental data is crucial as they offer valuable context to comprehend the uncertainty in the performed measurements. The radiation effects community conventionally follows this practice and uses two-sigma or 95% confidence intervals [19]. For counting statistics the Poisson distribution is used. If there are 50 or more events, the Poisson distribution can be approximated with the Normal distribution with the following formula:

$$\sigma = \frac{2 * \sqrt{\text{events}}}{\text{fluence}}$$

For this experiment, the Poisson distribution was used to calculate the DUE and SDC cross-sections of the experiments, because all errors were below 50. The normal

Table 6.1: Overview of the errors counted during the beam experiment

	Execution time (HH:MM:SS)	**Mismatch**	**Timeout**	**Exception**
Unmodified	11:48:41	5	1	31
ECC-enhanced	10:30:50	0	0	3
ECC+TMR	5:01:15	0	0	0

distribution was used to calculate the cross-section of the LSRAM. The SDC and DUE cross-sections are calculated for every experiment. The formula for computing the cross-section is:

$$\sigma = \frac{\text{number of errors}}{\text{fluence}}$$

The number of errors (N) is the sum of exceptions and timeouts for the DUE cross-section calculation. The SDC cross-section calculation only uses the number of mismatches. When the number of errors is 0, a value of 1 is used because a division with 0 is not possible.

The fluence was calculated by a Python script which receives the start and end time of the experiment. It also receives the timestamps when the beam was off. Lastly, it needs the flux of the beam, which is a constant value. This script produces fluence. Because two boards were tested at the same time, this fluence needs to be multiplied by 2 to get the total fluence. The boards were not angled, so the fluence does not need to be compensated for that

6.4 Characterization Results

The 2 boards run for a cumulative time of 27 hours for all 3 configurations. The unmodified and ECC-enhanced both received the most beam time because they were likely to receive the most errors. The classification of the counted errors can be seen in Table 6.1. Table 6.2 shows the counted errors, lower and upper limits and the amount of fluence used for the cross-section calculation.

Unmodified NEORV32 The unmodified implementation had a total run time of 11 hours and 49 minutes, counting both boards. During the test, the DUT suffered 1 timeout, 5 SDCs and a total of 31 exceptions. The found cross-section for this configuration is $1.96 * 10^{-10} cm^2$ for DUE and $3.06 * 10^{-11} cm^2$ for SDC, see Table 6.3. Table 6.4 shows which exceptions occurred and in which C function of the benchmark.

ECC-enhanced: The SEC-DED implementation resulted in a clear reduction in the number of errors present in the ECC version. There were no SDCs found. However, there were still 3 DUEs caused by exceptions, see table 6.4. There was an

Table 6.2: Upper and lower limits for the cross-section calculation

	N	95% Lower Limit	95% upper limit	Fluence	Total Fluence
Unprotected DUE	32	21.8	45.1	8.16E+10	1.63E+11
Unprotected SDC	5	1.6	11.7	8.16E+10	1.63E+11
ECC-enhanced DUE	3	0.6	8.8	8.25E+10	1.65E+11
ECC-enhanced SDC	0	0	3.7	8.25E+10	1.65E+11
ECC+TMR DUE	0	0	3.7	3.96E+10	7.94E+10
ECC+TMR SDC	0	0	3.7	3.96E+10	7.94E+10

access fault resulting from the bus getting timeout on the instruction fetch, which may be caused by an error in the bus connecting to the eNVM. The illegal instruction could represent a malformed instruction or in this case an unintended privilege access violation. Despite no DUEs being encountered, the cross-section represents a worst-case scenario, where it is assumed that the following particle would cause an event [42]. Table 6.3 shows the cross-sections for this configuration, which is $1.82 * 10^{-11} cm^2$ for DUE and $6.06 * 10^{-12} cm^2$ for SDC, which is a reduction of 10.7x and 5x respectively.

DMEM: The cross-section of the 12kB LSRAM was calculated by evaluating the total of both ECC-protected configurations. The user memory accumulated a significant amount of single-bit errors. No double-bit errors were detected in both configurations. The cross-section for a single-bit error is $1.23 * 10^{-13} cm^2$. This is comparable to the cross-section found in a characterization study performed on different SmartFusion2 models [34], they found a cross-section of $2.53 * 10^{-14} cm^2$ for a single bit in the LSRAM memory. When we divide the found cross-section by the used memory, which is 12K, we get a cross-section of $9.85 * 10^{-13} cm^2$. This found cross-section is a factor 39x bigger than the cross-section found in the characterization study of Dsilva [34].

ECC+TMR: The TMR version ran for a total of 5 hours and did not have any mismatches timeouts or exceptions. The cross-section again represents a worst-case scenario for this configuration. Since this version received a smaller total of fluence, it can be seen in Figure 6.2 that the ECC+TMR has a higher cross-section than the ECC-enhanced. The cross-section for the DUE and SDC is $1.26 * 10^{-11} cm^2$ with an upper limit of $3.40 * 10^{-11}$

There were no faults detected for the ECC+TMR solution. This result is in line with the results from the SoC hardening configuration in both the processor and the memory from HARV [27]. Applying ECC to the data memory resulted in a large reduction in errors, this also aligns with the finding of HARV [27].

Table 6.3: Cross Sections of the tested devices and the 12kB of LSRAM with ECC coverage.

	Number of Errors	Fluence (n/cm^2)	Cross Section (cm^2)	95% Confidence
LSRAM 12kB	2948	2.44E+11	1.21E-08	±4.44E-10
Unprotected DUE	32	1.63E+11	1.96E-10	+8.02E-11 -6.25E-11
Unprotected SDC	5	1.63E+11	3.06E-11	+4.10E-11 -2.08E-11
ECC-enhanced DUE	3	1.65E+11	1.82E-11	+3.51E-11 -1.45E-11
ECC-enhanced SDC	0	1.65E+11	6.06E-12	+1.64E-11
ECC+TMR DUE	0	7.94E+10	1.26E-11	+3.40E-11
ECC+TMR SDC	0	7.94E+10	1.26E-11	+3.40E-11

Fang et al. extended the PVF metric and created a dynamic model for predicting whether a particular fault will cause a crash. By performing fault injection they found that the majority of crashes are caused by illegal memory addressing [18]. They observed four types of exceptions resulting in crashes: Segmentation fault, Abort, Misaligned memory access and Arithmetic errors. They discovered that segmentation faults are the predominant source of crashes with a 99% average frequency and a 96% minimum frequency. This is in line with the exceptions found in the Unmodified variant because Table 6.4 shows that the majority of the exceptions are Load or Store Access Faults or Load Address Misaligned, which can all be classified as Segmentation faults.

Table 6.4: Causes of exceptions, separated by the function of origin according to *mpec* register.

Implementation	Coremark Function	Exception	Occurrences in function
Unmodified	core_bench_list()	Load Access Fault	10
		Store Access Fault	1
		Load Address Misaligned	2
		Illegal instruction	1
	core_bench_state()	Load Access Fault	1
	cmp_complex()	Load Access Fault	13
		Store Access Fault	1
		Load Address Misaligned	1
		Illegal instruction	1
ECC-Enhanced	matrix_test()	Illegal instruction	2
	core_state_transition()	Instruction Access Fault	1

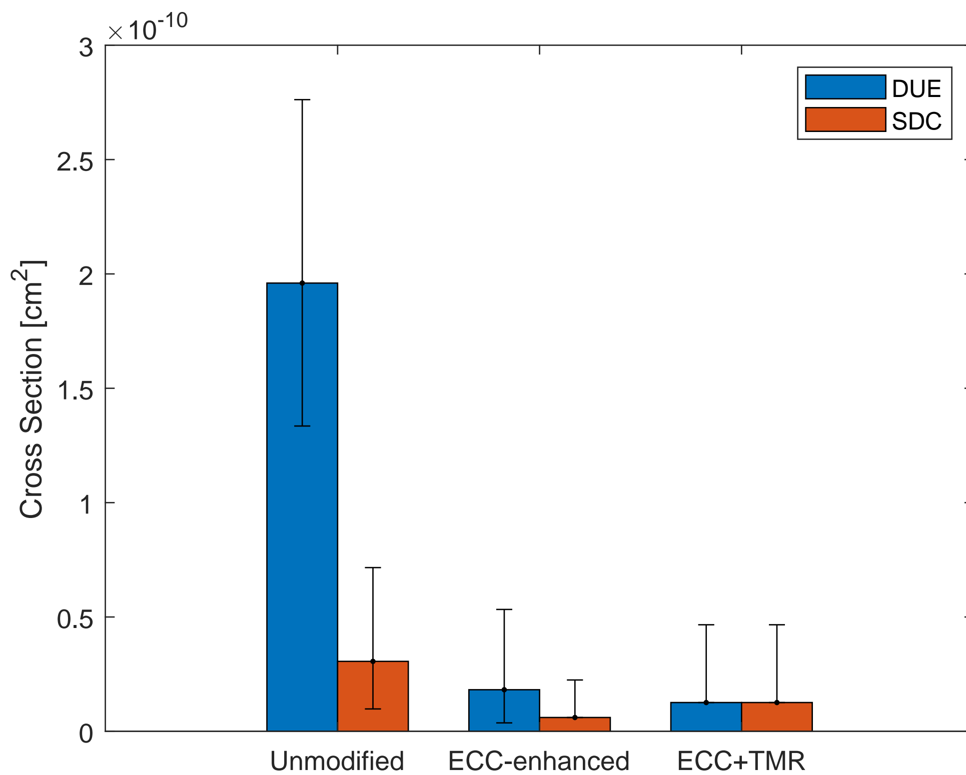

Figure 6.2: Measured cross section for the 3 different configurations of NEORV32

Chapter 7

Discussion

This discussion section offers a comprehensive analysis and interpretation of the results derived from the beam experiment. Within this section, we delve into the implications of the findings, while also delving into potential explanations for the observed exceptions. Furthermore, we extend our investigation to the assembly level, aiming to strengthen the proposed potential explanations with a detailed assembly-level analysis.

7.1 Implications of findings

This section discusses the findings of the beam experiment in Chapter 6 for every tested variant.

7.1.1 Unmodified

The Unmodified variant of the system, as shown in Table 6.1, showed 5 mismatches, a single timeout, and 31 exceptions. These mismatches are likely caused by SEUs inside the vulnerable 12kB SRAM data memory, which lacks any protection. When an SEU affects a memory location that stores a value used for computation, it can lead to a mismatch later in the program execution. However, the data memory not only stores values but also pointers. In the event of an SEU on a pointer, it can result in an exception during program execution.

Among the exceptions encountered, there were 24 load access faults and 2 store access faults. The exception handler reported load and store addresses outside the defined RAM memory space, with the upper address being 0x80003FFF. An interesting case was the load store access fault caused by a load from the memory location 0x804006d8. By replacing the '4' from this address with a '0', it becomes a valid address within the BSS section. This occurrence of '4' in the address could be caused by a bit SEU at bit b_{23}, flipping a bit from zero to one.

Table 7.1 provides an overview of all the requested memory locations at the instructions that caused Load Access Faults or Store Access Faults in the unmodified configuration. Among the 24 Load Access Faults, 20 memory addresses could be corrected by flipping one positive bit to 0, which may be caused by SEUs.

Moreover, three other memory locations could be made valid by correcting two bits. While Multiple Bit Upsets (MBUs) are considered unlikely to occur, as suggested by Microsemi's documentation on Igloo radiation test results [35], a more plausible explanation for these cases is the accumulation of two SEUs on the same word. This lower likelihood of SEUs occurring in the same word may explain why we only found three instances of such occurrences. However, there was one memory location, `0x30372e34`, that could not be easily traced back to a valid memory location, needing further investigation.

Regarding the two Store Access Faults, one of them could be traced back to a single-bit flip, but the other memory address that caused an exception, a store at address 0, is unusual. Since it is a zero value, it is unlikely to be a legitimate memory location such as `0x80000000`. The occurrence of a value of 0 could potentially be attributed to a SEU affecting a pointer. This SEU might alter the pointer to point to a valid address range within memory that has not been utilized yet. Consequently, the value retrieved from this location could be 0 due to the lack of prior utilization. Further investigation is needed to determine the cause of this exception.

In addition to the previously mentioned events, there were also two instances of illegal instructions encountered. At the memory address `0x60001340` a zero instruction was fetched. At address `0x60001628` the word `81000208` was fetched. Both these addresses are valid program memory locations; however, no valid instruction was fetched.

Moreover, 3 Load Addresses Misaligned Exceptions were found. A Load Adress Misaligned exception is raised when the processor attempts to read data from a memory location, but the address used for the load operation is not aligned properly with the data size being read. The address for byte loads and stores must be a multiple of 1 byte, addresses of halfwords must be aligned with a multiple of 2 bytes and for words, the addresses need to be a multiple of 4. The memory addresses of these encountered exceptions are not valid addresses, they are not in the memory region of the instruction or data memory, therefore these instructions can be classified as undefined bits. If these addresses matched the proper load size instruction, they would have become a Load Access Fault exception.

Lastly, a single timeout was encountered. When a timeout is encountered, the device does not present the results in the given timeframe. It is difficult to explain how a timeout could be caused. Because when for instance the program counter is affected by an SEU, a valid program instruction is fetched or an instruction fetch

Table 7.1: The load and store addressed which causes an exception for the unmodified NEORV32 configuration

Load Access Fault	Possible valid Address	Classification			Load Access Fault	Possible valid Address	Classification		
		Single	Dual	?			Single	Dual	?
804006d8	800006d8	X			8004096c	8000096c	X		
900007f8	800007f8	X			80004680	80000680	X		
8020096c	8000096c	X			804002dc	800002dc	X		
880004e0	800004e0	X			801007b0	800007b0	X		
8005206f	8000006f		X		800206a4	800006a4	X		
800107a8	800007a8	X			80109383	80001383		X	
840008dc	800008dc	X			80040398	80000398	X		
80200928	80000928	X			8000404f	8000004f	X		
808005b0	808005b0	X			80080778	80000778	X		
a0000354	80000354	X			8200045c	8000045c	X		
30372e34	?			X	800102dc	800002dc	X		
801008c8	800008c8	X			80408900	80000900		X	
Store Access Fault	**Possible valid Address**	**Classification**			**Store Access Fault**	**Possible valid Address**	**Classification**		
		Single	**Dual**	**?**			**Single**	**Dual**	**?**
00000000	?			X	80400598	80000598	X		

is done outside the program memory, which will be very likely to raise an Illegal instruction or Instruction Address Misaligned exception. SEUs might induce pipeline stalls or affect the content of crucial registers. Resulting in disruption of instructions and data handling contributing to executing delays and an eventual timeout. A SEFI is often associated with an upset in a control bit or register.

In conclusion, the mismatches and exceptions observed in the unmodified variant are likely the result of SEUs occurring in the vulnerable SRAM data memory, which lacks protection. The majority of Load and Store Access Faults can be attributed to pointers stored in the data memory, which were affected by SEUs, resulting in illegal pointers. On the contrary, the mismatches were likely caused by SEUs in memory locations that stored values. A comprehensive analysis is required to fully understand the cause of all exceptions and the single timeout.

7.1.2 ECC-enhanced

The addition of the ECC on the data memory and the register file, reduced the total mismatches to 0. It is expected that the ECC-enhanced version should have fewer

mismatches because the amount of SDCs on the data memory is reduced heavily. Besides the expected reduction of mismatches, we also expect a reduction of Load and Store Access Faults, because we claimed in the previous section that these are caused by an SEU striking a pointer. With the ECC and scrubber, the chance of a single SEU or multiple SEUs is very low. This is also what we see in the results, there were no mismatches found and there were no Load and Store Access Faults anymore.

However, it does not mean that mismatches will never occur with this configuration in a similar radiation environment. Memory is not fully tolerant, because accumulation of errors is still possible. Due to the scrubber, the time window where 2 SEUs need to happen to cause an MBU is reduced drastically. In addition, SDCs are still possible in the ALU; therefore, a wrong computation output could be written to memory, which can result in a later mismatch.

Although instances of timeouts were not detected, the situation introduces some complexities. The variant incorporating ECC was subjected to a shorter duration of radiation exposure in comparison to the Unmodified variant. In addition, the occurrence of just one timeout in the Unmodified variant does not provide a sufficient basis for a strong comparison.

Two Illegal Instructions were caused by executing a zero instruction, which is not a valid RISC-V instruction. The Program Counter (PC) of the instructions that caused the trap were `0x0001BF40` and `x60001BFA`, the first PC is not pointing to a valid instruction and therefore uninitialized data was fetched. The upper bound of the instruction memory is `0x6000352E`. However, the second instruction is valid and stores an addition. Lastly, the Instruction Access Fault was caused at `0x60000928`. This instruction is raised when the slave does not respond in time, which could be attributed to a SEFI.

7.1.3 ECC+TMR

The ECC+TMR implementation did not have any mismatches, timeouts or exceptions during the execution. Although these results look very promising, it does not mean that this device with the used architecture can be classified as fully fault-tolerant. Fault tolerance refers to a system's ability to continue functioning properly even in the presence of faults or failures. There are several reasons why a device could pass the beam test without errors. The beam test may not have covered all possible failure scenarios. One failure scenario could be the accumulation of multiple SEUs. This can happen inside the data memory, which will likely result in a mismatch. However, it can also happen at the flip-flop level, when the state of 2 flip-flops inside a single ECC+TMR solution is flipped, the ECC+TMR voter will provide

the wrong output. This will only cause a problem if the value of the voter is read after the accumulation of 2 errors and is not overwritten. As described not all storage bits result in a malfunction of the system, the non-ACE bits. One or two SEUs occur at one bit of the program counter, the ECC+TMR solution will fail when the timing is right. Although MBUs have not been detected before for the target device, there is still a probability that it can happen.

7.2 Program analysis

In the preceding section, a hypothesis was presented that the emergence of Load Access Fault and Store Access Fault exceptions can be attributed to store pointers in memory that have been influenced by SEU. Similarly, it was suggested that the origins of Illegal Instruction and Instruction Access Fault exceptions likely stem from SEUs affecting the control logic. In the current section, we concentrate solely on elaborating the hypothesis about Access Fault Exceptions. Given that the occurrence of these exceptions was exclusive to the unmodified NEORV32 configuration, this section only applies to this specific setup.

7.2.1 Affected functions

Table 6.4 lists the functions where the exception occurred. The `core_bench_list` function contains code for the linked list benchmark. This function has the purpose of identifying multiple data items within a list. It arranges the list in sorted order, conducts Cyclic Redundancy Check (CRC) operations on the data present in the list, and further manages the removal or reinsertion of items within the list. Importantly, at the end of the function's execution, the list is restored to its initial state. The function `core_bench_list` has been reported to encounter 14 exceptions, as outlined in Table 6.4.

The `cmp_complex` function plays a crucial role in the `core_bench_list` function by performing comparisons on data items within list cells. This function, described in Listing 7.3, appears to be a straightforward function with just three lines of code, yet it is remarkable for the frequency of exceptions it encounters. This function causes the most exceptions of all functions, namely 16 exceptions, several factors contribute to this phenomenon, needing exploration.

The `core_bench_state` function implements the state machine benchmark. This state machine tests types of string input and tries to determine whether the input is a number or something else. `core_state_transition` implements the actual state machine and is called by the `core_bench_state` function. The unmodified variant

had one exception in `core_bench_state`. The ECC-enhanced configuration had a single instruction Access Fault in the `core_state_transition`.

The `matrix_test` function performs the matrix manipulation. It adds a constant value to all elements of a matrix. Furthermore, the matrix is multiplied with a constant, followed by a vector multiplication. After this, the matrix is multiplied by a different matrix. Finally, a constant is added to all elements of a matrix, after this step the matrix is back to its original contents. Only the ECC-enhanced had exceptions in this function, namely 2 Illegal Instructions.

The `matrix_test` and `core_state_transition` will not be used in this section because they were caused by Illegal Instructions by the ECC-enhanced. This section only focuses on Load and Store faults. The Illegal instruction of the unmodified NEORV32 will also not be investigated. Therefore this subsection focuses on the `core_bench_list`, `cmp_complex` and `core_bench_state`. We refer to these exceptions as data memory exceptions or Segmentation Faults.

The function `core_bench_list` has 13 data memory exceptions, `cmp_complex` has 15 exceptions and `core_bench_state` has a single data memory exception. Because `cmp_complex` is a helper function of the `core_bench_list`, 28 data memory exceptions occurred in the linked list benchmark. A single exception occurred in the state benchmark. The matrix benchmark does not have any data memory exceptions.

Two key observations arise from these results. Firstly, it is noteworthy that nearly all the exceptions in the unmodified version are attributed to Segmentation Faults. Secondly, it is remarkable that the vast majority of these exceptions are raised within the list benchmark. This subsection delves deeper into the exploration of these two findings.

7.2.2 Vulnerability Factor

Section 2.1.4 in the background introduced the concept of ACE bits, AVF and PVF. The equations for AVF and PVF consider the number of ACE bits over a specified period of time. To illustrate this, let us analyze the PVF calculation for a single register in the context of two programs, each featuring a single store and load operation. In this scenario, both programs have an initial store operation at instruction 1. Program A continues with a load at instruction 5, while Program B performs a load at instruction 10, and in between NOP instructions. The PVFs for both programs can be computed as follows:

$$Prog_A = \frac{\sum_{i=0}^{I}(ACE \text{ a-bits in } R \text{ at instruction } i)}{B_R \times I} = \frac{(5-1)*32}{32*5} = 0.8$$

$$Prog_B = \frac{\sum_{i=0}^{I}(ACE \text{ a-bits in } R \text{ at instruction } i)}{B_R \times I} = \frac{(10-1)*32}{32*10} = 0.9$$

From this basic example, we can deduce that data stored for a longer duration in a register or in memory shows a higher vulnerability factor.

In order to say something about the vulnerability of instruction at a specific PC or C function, the presence of jumps and loops needs to be taken into consideration. These branches introduce a dynamic aspect to instruction execution, where certain addresses are executed multiple times, consequently boosting the probability of triggering exceptions.

7.2.3 CoreMark Linked List Algorithm

One of the CoreMarks benchmarks is the linked list operations. CoreMark uses C structures for creating the linked list, which is shown in Listing 7.1. For this experiment, dynamic memory allocation was not used, so calls such as `malloc` and `alloc` were not used. The application controls the memory for this list directly. The linked list will be initialized such that $\frac{1}{4}$ of the list pointers point to sequential areas in memory, and the other $\frac{3}{4}$ of the pointers are distributed in a non-sequential manner. In this way a list is emulated that had many adds and removes, disrupting the order of the list and is followed by a series of adds, emulating sequential memory locations.

Listing 7.1: CoreMark linked list structure

```
typedef struct list_data_s {
        ee_s16 data16;
        ee_s16 idx;
} list_data;

typedef struct list_head_s {
        struct list_head_s *next;
        struct list_data_s *info;
} list_head;
```

The linked list benchmark consists of the following:

1. Multiple find operations. The result of each find will be part of the output CRC chain.
2. Sorting the list based on the *data16* value. The CRC is derived from a part of the sorted *data16* items.
3. Sorting the list based on the *idx* value. This will revert the list back to the original state, therefore no initialization is needed for further iterations. The

CRC of the *data16* for part of the list will be calculated again and part of the output chain.

The calculation of the number of list items utilized for the benchmark is important for the analysis and can be seen in Listing 7.2. Given a memory size of 12kB for our experiment, each benchmark utilizes one-third of this memory, resulting in a block size (`blksize`) of 4kB. Considering that the size of the `list_data_s` structure is 32 bits, the per-item size (`per_item`) amounts to 48 bits.

Consequently, the amount of list items is $4000 * 8/48 - 2 = 664.67$, leading to a list size of 664 items, which is used for the benchmark.

Listing 7.2: CoreMark list size calculation

```
ee_u32 per_item = 16 + sizeof(struct list_data_s);
ee_u32 size     = (blksize / per_item) - 2;
```

For this analysis it is important to know how much of the data memory actually contains linked list pointers, this percentage is calculated as follows:

$$mem_{pointer} = \frac{N_{items} * size_{pointer}}{size_{mem}} * 100\% = \frac{664 * 16}{12000 * 8} * 100\% = 11.1\%$$

11.1% of main memory contains pointers for the linked list benchmark.

The `core_bench_list` function is the main function of the linked list benchmark and encountered 16 exceptions. One primary factor is the frequency of function calls. As an integral part of the sorting algorithm, `cmp_complex` is invoked repeatedly during comparisons. This repetitive nature increases the probability of encountering exceptions. It is a numbers game: the more a function is called, the higher the odds of encountering an exceptional case.

In contrast, `cmp_complex` is a small function with only three lines of code, see Listing 7.3. Table 6.4 reveals that the function `cmp_complex` for the unmodified variant had 14 load Exceptions and 1 store exception. Smaller functions have a lower probability of causing exceptions due to their reduced complexity. The *objdump* of the compiled binary shows that the function `calc_func` was inlined in `cmp_complex`. Listing A.1 in the appendix shows the compiled assembly code for this instruction. This function contains 127 lines of assembly.

Listing 7.3: CoreMark cmp_complex()

```
/* Function: cmp_complex
        Compare the data item in a list cell.
        Can be used by mergesort.
*/
ee_s32
```

```
cmp_complex(list_data *a, list_data *b, core_results *res){
    ee_s16 val1 = calc_func(&(a->data16), res);
    ee_s16 val2 = calc_func(&(b->data16), res);
    return val1 - val2;
}
```

One other interesting observation is the fact that the function `cmp_idx`, shown in Listing 7.4, is very similar to `cmp_complex`, but does not have any single exceptions. The reason for this is timing. When a list is sorted, every item in the linked list is accessed. For every item, the comparison functions are called. Because `cmp_complex` is called **before** `cmp_idx`, `cmp_complex` will raise an exception when one pointer in memory was affected by an SDC. Because `cmp_idx` is **directly** called after `cmp_complex`, the time window of an SEU affecting a pointer causing an exception in `cmp_idx` is small, therefore we did not detect any exceptions in `cmp_idx` during our beam experiment.

Listing 7.4: CoreMark cmp_idx()

```
/* Function: cmp_idx
        Compare the idx item in a list cell, and regen the data.
        Can be used by mergesort.
*/
ee_s32
cmp_idx(list_data *a, list_data *b, core_results *res)
{
    if (res == NULL)
    {
        a->data16 = (a->data16 & 0xff00) |
                    (0x00ff & (a->data16 >> 8));
        b->data16 = (b->data16 & 0xff00) |
                    (0x00ff & (b->data16 >> 8));
    }
    return a->idx - b->idx;
}
```

7.2.4 Analysis of the crash traces

Table 7.2 lists all relevant instructions affecting a particular exception. When an exception is caused by a load from an address stored in a register, the instructions before affecting this register are shown as well. Relevant instructions for these traces only consisted of load, store and move operations.

Table 7.2: Load and store traces of possible affected registers causing an exception

Adress	Instruction	Adress	Instruction
600012cc:	lh s1,0(s2)	**60001676:**	lw a1,4(s0)
60001280:	mv s2,a1	6000166c:	mv s0,a5
		6000165e:	lw a5,0(a5)
60001538:	lh a3,0(a7)	6000165a:	mv a5,s0
60001530:	lw a7,4(s0)	60001630:	lw s0,56(sp)
6000144c:	mv s0,a4		
6000143e:	lh a4,2(a4)	**60001330:**	sh s1,0(s2)
6000143c:	lw a4,4(a5)	60001280:	mv s2,a1
60001438:	lw a5,0(a5)		
60001434:	mv a5,s0	**6000168e:**	sw a5,0(s3)
60001414:	lw s0,36(a0)	60001630:	lw s0,56(sp)*
		60001652:	li s3,0*
6000165e:	lw a5,0(a5)		
6000165a:	mv a5,s0	**60000ba2:**	lbu a1,0(a6)
60001630:	lw s0,56(sp)	60000b9c:	sw a6,12(sp)
6000126c:	lh s1,0(a0)		

This subsection will explain every trace. Note that only the trace of instructions or given of the affected function. The following traces: `0x600012cc`, `60001538`, `0x6000165e`, `0x6000126c`, and `0x60001330` end at load or move from the function argument register, which means that the pointer has been passed to this function. The table shows 8 traces because multiple exceptions occurred at the same PC.

The exception at **0x60001676** was triggered due to an invalid address used in the `lw` instruction. By analyzing the assembly, we verify that the address for the load operation is stored in register `s0`. To identify the root cause, we trace back to the instruction at address `0x6000166c`. This instruction transfers the contents of register `a5` to `s0`. Now an investigation is needed for register `a5`. This brings us to the instruction at address `0x6000165e`, where a memory load is executed with both source and destination addresses set to the value of `a5`.

Two plausible explanations for the exception can be derived. In the first scenario, register `a5` initially held a valid pointer, and an external event, such as an SEU, affected the memory location pointed to by `a5`, leading to the exception. Alternatively, the second scenario declares that the pointer stored in `a5` was erroneous. In this case, instruction `0x6000165a` is relevant, which copies the contents of `s0` to `a5`.

Continuing at address `0x60001630`, where a load operation from the stack pointer to `s0` is observed. Considering the assumption that `s0` might contain an invalid

pointer, two potential explanations arise. The first is that an SEU impacted the memory location at address 56(sp), causing the exception-inducing behaviour. The second explanation involves the possibility of the stack pointer (sp) being affected by a SEU, leading to the exception.

The exceptions detected at address **0x600012cc** can be attributed to the `s2` register, which held an erroneous address. This address originated from the `a1` register, a function argument. The exception might have resulted from a wrong pointer passed to the function or an SEU the `a1` or `s2` registers. A straightforward explanation applies to the exception at PC **0x6000126c**. The `core_bench_list` function takes a list pointer and an ID as inputs. The most likely scenario involves an invalid list pointer stored in `a0`.

A similar pattern emerges for the exception at **0x60001330**, where a store exception arose. The address in `s2` was erroneous, obtained from function argument `a1`. Moving to the exception at address **0x60000ba2**, a load exception occurred using the address stored in `a6`. This address was loaded from memory with the stack pointer.

Address **0x6000168e** presents a more complex case due to jumps and branches. The content of `s3` might have been fetched from memory via the stack pointer or could still be 0. The former scenario, where the value was obtained from memory, is more likely.

The most extensive trace, found at address **0x60001538**, contains multiple loads and stores. In summary, a common thread among these traces is the involvement of pointers loaded from memory. The source alternates between function argument registers and memory-loaded pointers via the stack pointer.

7.3 Summary

The frequent occurrence of exceptions within the linked list benchmark is not coincidental but rather a predictable outcome. This phenomenon can be attributed to the benchmark's heavy reliance on pointers stored in the main memory. The linked list benchmark involves a sorting operation on the list, leading to the access of every list item. Consequently, any SEU affecting one of these pointers is highly likely to trigger an exception. In cases where the SEU impacts the least significant bits of a pointer, the memory location might still remain valid. However, the probability of an exception being raised upon accessing the affected structure is significantly high.

Subsection 7.2.1 introduced the concept of PVF and ACE bits. Since the list is initialized during the initial phase, and subsequent iterations of the CoreMark benchmark employ this same pre-initialized list, any SEU on these pointers consistently leads to an exception or mismatch. As a result, all these pointer bits are appropri-

ately classified as ACE bits. Notably, this observation aligns seamlessly with the findings presented in section 7.2.3.

The previous section showed that load and store exceptions mainly occurred due to memory operations based on pointer values stored in the main memory. These pointers occasionally stemmed from function arguments or, alternately, the stack pointer itself, employed for retrieving the pointer from memory.

In essence, the convergence of these insights underscores the critical role played by ACE bits, which account for approximately 11.1% of the memory. This insight is substantiated by the analysis presented in section 7.2.3, offering a comprehensive understanding of the exceptions arising during Coremark benchmark execution.

Chapter 8

Simulation

In the previous chapter, various factors and scenarios that could potentially trigger exceptions within the unmodified NEORV32 processor were discussed. This chapter serves as a direct extension of the discussions in Chapter 7 by conducting specific manual fault injections to validate the potential causes of these exceptions.

The focus of the chapter remains on simulating exceptions of the two instructions with the most exceptions, specifically those occurring at PC `0x600012cc` and `0x60001676`. These exceptions garnered attention due to their notably high frequency of occurrence, with 10 and 7 instances, respectively.

8.1 Simulation setup

The simulation of the NEORV32 processor and the MSS was carried out using QuestaSim, a popular commercial RTL simulator tool. Chapter 7 Discussion showed different reasons how a particular exception could be caused. One way to validate these reasonings is by performing a simulation. Simulation provides full access to all signals of the entire processor. However, a disadvantage of simulation is that it requires a long time to run [43].

The script for setting up the QuestaSim simulator was automatically generated by the Libero tool when a test diagram was provided. The used test design is straightforward and can be seen in Figure 8.1. The *top_0* block is the top-level design, which is shown in Appendix A in Figure A.1 in more detail.

The *RESET_GEN_C0_0* IP block from Microsemi is a straightforward reset generator. This generator is configured to operate with an active-low signal and introduces a delay of 1000 ns. Additionally, custom ports are utilized to connect the RX input, as well as the LED0, LED1, and TX output pins. These custom ports facilitate access and monitoring of signals during the simulation process.

During the beam experiment, one CoreMark execution was run with a configu-

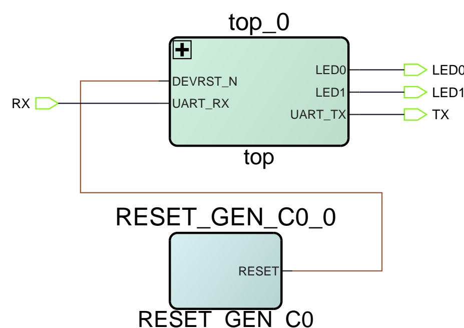

Figure 8.1: Libero top-level design of the testbench

ration of 12KBytes of memory and 20 iterations, which required approximately 1.5 minutes to complete. However, during the simulation phase, we encountered a significant increase in computation time. The CoreMark source code was recompiled with a reduced memory allocation of 2KBytes and set to a single iteration. This simulation was carried out using the ModelSim simulator, and it consumed approximately 20 hours for a single run.

The simulations were executed on an Intel Xeon Gold 6126 CPU, operating at a clock frequency of 2.60 GHz, in both QuestaSim and ModelSim environments. The act of recompiling the program introduced variations in instruction placement compared to the original binary, making it more challenging to directly compare the results between the simulation and the beam experiment.

While QuestaSim improved the simulation times slightly, it was still impractical to conduct a comprehensive fault-injection campaign within the available time frame. Consequently, the decision was made to manually reproduce the exceptions by injecting faults at the right place and time to validate the findings of Chapter 7: Discussion. For these manual fault injections, the same binary was used as during the beam experiment.

8.2 Fault injection strategy

In order to inject manual faults, the simulation needs to be stopped at a specific PC. The command shown in Listing 8.1 demonstrates how the simulator can be stopped before executing the instruction stored at address `0x60001630`.

Listing 8.1: Command to pause simulator at PC `0x60001630`

```
$ when {sim:/test_top/top_0/neorv32_ProcessorTop_Minimal_0/
neorv32_inst/neorv32_cpu_inst/neorv32_cpu_control_inst/
execute_engine.next_pc = 60001630} {Stop}
```

This approach uses the *next_pc* register, which holds the PC of the next instruction to be executed. By strategically stopping the simulation at the desired PC value, faults can be introduced before the intended instruction is executed.

To facilitate fault injection experiments, checkpoints were used to manage the simulator state. This technique allows the simulation to be stored at specific PC points, subsequently inject manual faults, evaluate the outcomes, and, if needed, restore the simulator to the checkpoint for further fault injections at different bit positions or registers.

The Questasim and ModelSim simulator provides the ”force” command which can be used to alter signal values [44]. The ”freeze” command option, sets the signal to a particular fixed value until the end of the simulation, this can be used for instance to simulate a stuck-at error. Additionally, the ”deposit” option releases the signal immediately after it is altered by the RTL model. The ”deposit” option is used for injecting manual faults, because this emulates for instance a storage cell hit by a high-energy particle, changing the current value.

The current signal value can be read with the *examine* command or GUI. The command shown in Listing 8.2 shows how the value of the eleventh register in the register file can be set to ’80100898’.

Listing 8.2: Command set the value of register x11 to 80100898

```
$ force -deposit sim:/test_top/top_0/
neorv32_ProcessorTop_Minimal_0/neorv32_inst/neorv32_cpu_inst/
neorv32_cpu_regfile_inst/reg_file(11) 80100898 0
```

Furthermore, the objdump of a program lists the used registers in ABI convention, for instance, *a0* or *s1*. Table 8.1 lists the index of each register in the register file, this is used to determine which registers need to be manipulated for a manual fault injection.

Table 8.1: The register utilization guidelines for RISC-V [45]

Reg	ABI	Reg	ABI	Reg	ABI	Reg	ABI
x0	zero	**x8**	s0/fp	**x16**	a6	**x24**	s8
x1	ra	**x9**	s1	**x17**	a7	**x25**	s9
x2	sp	**x10**	a0	**x18**	s2	**x26**	s10
x3	gp	**x11**	a1	**x19**	s3	**x27**	s11
x4	tp	**x12**	a2	**x20**	s4	**x28**	t3
x5	t0	**x13**	a3	**x21**	s5	**x29**	t4
x6	t1	**x14**	a4	**x22**	s6	**x30**	t5
x7	t2	**x15**	a5	**x23**	s7	**x31**	t6

During the simulation, the *trap_ctrl.cause* register is an important signal to monitor, because this register shows the exception if an exception occurs. By default, NEORV32 initializes this register to the lowest-priority exception: the Machine Timer Interrupt (MTI), with a value of '0x47'. When an exception arises, this trap cause register reflects the specific cause, as shown in Chapter 4: NEORV32 Table 4.1.

8.3 Manual injected faults for PC value 0x600012cc

The relevant stack trace for the root causes of the exceptions can be found in Table 7.2. In this example, a SEU will be introduced just before the PC value `0x600012cc`. Figure 8.2 shows the current execution state of the PC. Additionally, it shows the content of register `x18 (s2)` and the trap cause, which, in this case, is due to correction execution 0x47. At the cursor location, an SEU upset has occurred in register `x18`, altering bit b_{20} from 0 to 1. This register holds a value pointing to an illegal memory location in the data memory, resulting in a Load Access Fault in the future. Following the `0x600012c8` instruction, the `0x600012cc` instruction is executed, causing the trap cause register to change to 0x5, indicating the expected outcome of a Load Access Fault.

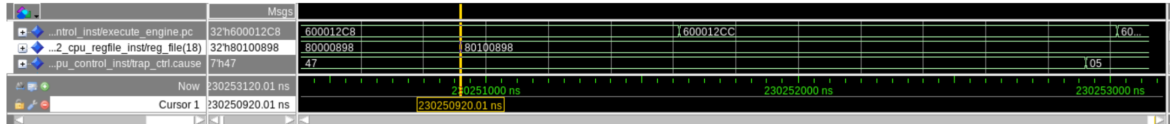

Figure 8.2: Simulation demonstrating a manual fault injected leading to a Load Access Store Fault exception at PC `0x600012CC`, by injecting a fault in register `x18`.

An alternative failure scenario could occur at PC value `0x60001280`, where the contents of register `a1 (x11)` are copied into register `s2 (x18)`. Before the execution of `0x60001280`, a manual fault was introduced into register `x11`. The outcome of this simulation is depicted in Figure 8.3. Notably, there was a time difference of 102700 ns between the fault injection and the emergence of the exception, which correlates to 1027 clock cycles. Throughout this duration, register `x18` retained its value.

8.4 Manual injected faults for PC value 0x60001676

The last example which will be investigated with QuestaSim are the exceptions caused at address `0x60001676`. The load instruction on this address generated

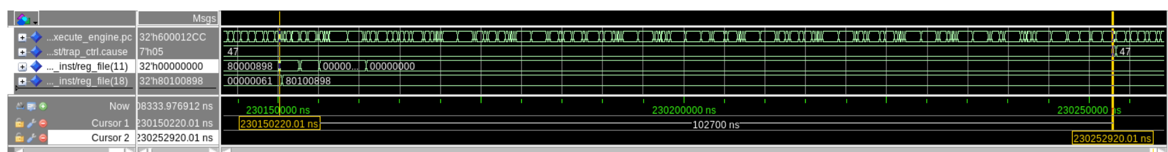

Figure 8.3: Simulation demonstrating a manual fault injected leading to a Load Access Store Fault exception at PC `0x600012CC`, by injecting a fault in register `x11`.

7 Load Access Faults in total. The trace in Table 7.2 shows 4 relevant instructions before the execution of `0x60001676`, namely: `6000166c`(move), `6000165e`(load), `6000165a`(move) and `60001630`(load). The first encounter of the address `0x60001676`, did not include the instruction at `0x60001630`, which could also possibly be the cause of the exception of 0x60001676. This instruction was skipped due to a jump at `0x6000164e` to address `0x600014ae`.

Therefore manual faults will be injected before the execution of address `0x6000165a`. This instruction copies the value of `s0` to `a5(x15)`. Suppose the `s0 (x8)` register value has been affected by SEU, this SEU could have happened in the data memory or in the register file itself. A fault has been injected before the execution of instruction `0x60001658`. A manual SEU was introduced at b_{14}, resulting in a new register value of '0x80002530'. With the introduced SEU, this pointer is still in the address range of the data memory. The simulation for this manual injected fault can be seen in Figure 8.4.

After the execution of instruction `0x6000165e`, it can be seen that register `s0` is copied to register `a5`. At the end of instruction `0x6000165e`, register `a5` contains the new data. At instruction `0x6000165e` the value which the pointer of `a5` is pointing to is stored in `a5` at address `0x6000165e`. The simulation in Figure 8.4 shows that '0x2E302B2C' is fetched from memory. At the next relevant instruction, this register is copied to `s0`. At instruction 60001676, the value in `s0` is used to fetch the data at the memory `s0+4`, and of course, this results in a load Access Fault, because '0x2E302B2C' is clearly not in the valid address range.

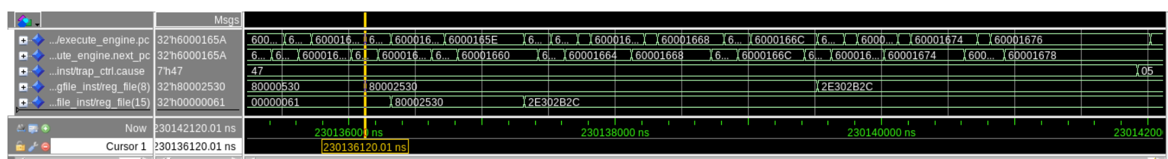

Figure 8.4: Simulation demonstrating a manual fault injected leading to a Load Access Store Fault exception at PC `0x60001658`, by injecting a fault in register `x8`.

The previous example showed how this exception could be caused by a SEU

affecting the register value contents, however not every change to the contents of register s0 results in the same behaviour. Figure 8.5 shows the simulation case for an SEU at position b_6, resulting in the new pointer value '0x80000570'. This simulation shows that this pointer is used to fetch the valid pointer '0x800004A0', because this is a valid pointer, there is no exception caused during instruction 0x60001676.

After many subsequent simulated instructions, it was observed that no exceptions were raised. Unfortunately, due to the substantial resource demands of the simulation, it was not possible to simulate a full CoreMark run. The address in question belongs to the *core_bench_list()* function, which contains tasks like searching, sorting, or removing and reinserting items within a list.

The hexadecimal difference between the original pointer and the new pointer is 0x40, equivalent to 64 bytes. Depending on the memory map, an increase of 0x40 in the address could point to the next item in the list or some other item.

If the skipped list items during this process do not contain the value being searched for, for example during a find operation, there may be no observable difference in the output. However, it is important to note that during a sorting operation, every list item typically needs to be accessed, which could potentially result in a mismatch in the output due to the altered memory access caused by the injected fault.

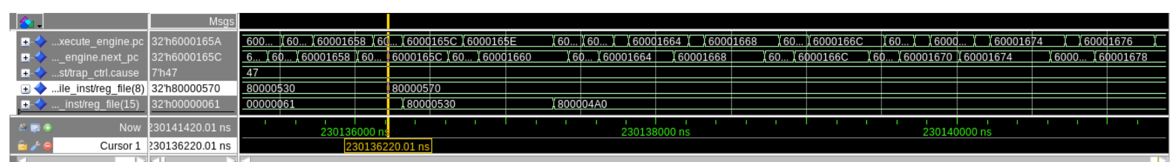

Figure 8.5: Simulation demonstrating a manual fault injected leading still to a successful execution at PC 0x60001658, by injecting a fault in register x8.

Another interesting simulation can be seen in Figure 8.6, where a SEU is introduced at bit b_{12}, altering the pointer value to '0x80001530'. This new pointer is a valid address and points to the value '0x00CEE00CD'. However, this value is used as a pointer, but because bit b_0 is 1, this pointer is unsuitable for fetching complete words. Consequently, this injected fault results in a Load Address Misaligned exception, which can be seen in in simulation in figure 8.6, because the Trap cause register becomes '0x4'.

8.5 Summary

These findings validate several theories presented in Chapter 7: Discussion. Nevertheless, they also emphasize the critical role of bit position in understanding and mitigating exceptions. Identifying all sources of a specific exception can be challeng-

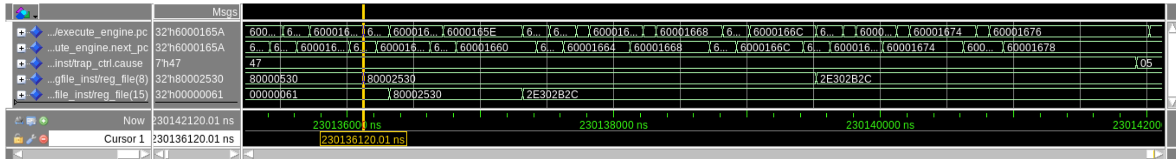

Figure 8.6: Simulation demonstrating a manual fault injected leading to a Load Address Misaligned exception at PC `0x60001658`, by injecting a fault in register `x8`.

ing, particularly in scenarios involving multiple relevant instructions beforehand. For instance, in the case of `0x60001676`, where a value from memory serves as a pointer for another load, SEU in a relevant register may result in exceptions at various other PCs. Furthermore, an SEU affecting a different bit can lead to a distinct exception or even no exception at all.

Another significant observation from our simulations is the substantial variation in the time elapsed between the fault injection and the triggering of an exception. Essentially, the longer a value remains stored in a register, the more vulnerable the register becomes. However, it is worth noting that most exceptions likely originate from SEUs within the data memory.

These simulations exclusively focused on SEUs within the register file. Expanding the scope to include the data memory would not significantly alter the results. To shift the focus to the data memory, the "force" command would need to be applied to the data memory instead of the register file. However, this aspect was disregarded due to the challenges associated with pinpointing the precise instruction address and memory address for data retrieval. In summary, these findings underscore the significance of bit positions and timing, highlighting the complex nature of exception sources.

Chapter 9

Conclusions and recommendations

The primary objective of this thesis was to comprehensively explore and evaluate diverse trade-offs among different fault tolerance techniques in the context of a soft RISC-V processor realized on a Flash-based FPGA. This investigation contains crucial factors, including area utilization, performance metrics and resilience to errors. In addition, an important part of this research involved conducting a radiation test to assess the efficiency and robustness of the various redundancy techniques implemented.

The first research question (RQ1) led us to delve into the various fault tolerance mechanisms available and their implications for the resource utilization, energy consumption, and computational speed of the FPGA-based soft RISC-V processor. Both research questions have been answered by a study comparing three different configurations of the NEORV32 processor: Unmodified, ECC-enhanced, and TMR. Chapter 5: NEORV32 Implementation addresses this question. When integrating ECC, including both an encoder and a decoder, approximately 14% more Flip-Flops and Lookup Tables (LUTs) are utilized. Additionally, the inclusion of extra ECC bits also results in a larger SRAM memory usage. Regarding the estimated power of the ECC version, a 4.43% increase in Gate usage and a 7.09% increase in Net usage are observed. Notably, the total estimated power consumption sees only a slight 3.2% increase. However, because ECC is part of the critical path in the register file, it does limit the maximum achievable clock speed. Nevertheless, the target of 10 MHz can still be met. The primary drawback of implementing TMR is the increased hardware cost, primarily due to the triplication of Flip-Flops. The number of Lookup Tables (LUTs) increases by 61.5%, while the DFFs grow by 148.5%. In terms of estimated power consumption, TMR+ECC results in a 162.05% increase in Net power consumption and an estimated increase of 34.64% for the Gate usage. Compared to the unmodified version, the total estimated power consumption only increases by 15.04%.

The second research question (RQ2) evaluates the resilience performance of

different fault tolerance techniques in the presence of errors, particularly radiation-induced faults, in a soft processor implemented on a Flash-based FPGA. This research question has mainly been answered in Chapter 6: Neutron Beam Experiment. In the Unmodified configuration, numerous Load Access Faults and Store Access Faults were observed, which were attributed to SEUs affecting pointers stored in data memory. These errors resulted in illegal pointers and subsequent exceptions during program execution. Additionally, mismatches were attributed to SEUs in memory locations storing values, leading to incorrect computation results. These findings underscored the vulnerability of the default NEORV32 processor to radiation-induced errors. With the ECC-enhanced configuration, the addition of ECC on the data memory and register file significantly reduced the occurrence of mismatches, Load Access Faults, and Store Access Faults. This outcome aligned with the expectations, as ECC enhanced the error detection and correction capabilities of the processor. However, it was highlighted that while the ECC-enhanced configuration demonstrated improved resilience, it still remained susceptible to certain error scenarios, such as Multiple Bit Upset (MBU) orSEUs affecting the ALU, which could lead to incorrect computation results. The TMR configuration demonstrated the most robust behaviour among the tested configurations. It successfully eliminated mismatches, timeouts, and exceptions during execution. However, it is important to note that the TMR configuration's performance in the beam test does not necessarily translate to complete fault tolerance, as certain failure scenarios may not have been covered by the test conditions.

These findings of the experiment highlight the importance of error mitigation techniques such as ECC and TMR in improving the reliability and resilience of a soft processor under a neutron beam. Given the utilization of high-energy neutron particles, the obtained results hold direct relevance primarily to domains such as avionics and low-orbit satellites, including environments like the Van Allen belt. While this research utilised neutron radiation, the SEEs that were caused by neutron radiation are not bounded solely to high-energy neutron particles. High-energy proton particles, common in space environments and cosmic rays, also induce similar SEEs. Consequently, high-energy neutron particles can serve as an approximation to faults in space. These findings demonstrate a meaningful insight into the performance and applicability of fault tolerance techniques implemented on a flash-based FPGA in scenarios characterized by similar conditions.

The choice of the most suitable redundancy technique depends on the specific situation. When an absolute error-free operation is crucial and any exceptions are unacceptable, employing both TMR and ECC is the best option. However, in scenarios involving non-critical systems where the occurrence of exceptions is of lesser importance, relying solely on ECC could prove sufficient and efficient because the

area and power costs are low.

The recognition of these findings was underscored by the acceptance of this work as a paper titled ”Neutron Radiation Tests of the NEORV32 RISC-V SoC on Flash-Based FPGAs” at the 36th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems.

9.1 Future work

This section provides recommendations for further research. From the radiation experiments, various types of errors were detected, including mismatches, exceptions, and timeouts. Mismatches belong to SDC and can occur when a memory cell is impacted by a high-energy particle. On the other hand, SETs have the ability to propagate through logic and be captured by a memory cell at the right time. In this experiment, SETs may be detected as SDC, but it is challenging to determine how often SDCs are specifically generated by SETs. As described in the background, SETs are short transient pulses. The clock frequency of the device plays a crucial role in this context, as a higher clock frequency leads to smaller single cycles, resulting in an increased likelihood of a SET being captured by a memory cell. Therefore, it is reasonable to hypothesize that a higher clock frequency may result in more errors caused by SETs. Consequently, a valuable addition to this research would be to conduct tests with different clock frequencies and compare the results. This would help to determine whether a higher clock frequency indeed leads to an increase in errors caused by SETs and whether the classification of these errors as SDCs occurs in the same ratio. By exploring the impact of clock frequency on error rates and error classifications, insights will be provided about the behaviour of SETs and their influence on system reliability. This information can be used to improve error mitigation strategies and design more robust systems capable of withstanding the effects of SETs, particularly in critical applications where the consequences of data corruption can be severe.

The tested configurations in this research are likely vulnerable for SETs. This downside could be covered by a lockstep configuration. In a lockstep processor, two or more identical processor cores operate in synchronization, executing the same set of instructions simultaneously. This synchronized execution allows for real-time comparison of outputs, ensuring consistency and fault detection. While a lockstep processor may demand more hardware and potentially longer fault recovery time compared to TMR, it can offer additional protection through its broader error detection capabilities and resilience to common-mode failures.

ASICs and Flash-based FPGAs are similar digital hardware solutions. They both work similarly during beam experiments. The memory in a Flash-based FPGA,

which is used for configuration and is based on flash technology, is safe from errors caused by radiation. This makes it behave like an ASIC in such experiments. It would be valuable yet expensive to research and compare a Flash-based FPGA with an ASIC manufactured using the same technology, looking at the contrasts between them.

Bibliography

[1] D. Girimonte and D. Izzo, "Artificial intelligence for space applications," 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:109246556

[2] G. Furano, S. Di Mascio, A. Menicucci, and C. Monteleone, "A european roadmap to leverage risc-v in space applications," in *2022 IEEE Aerospace Conference (AERO)*, 2022, pp. 1–7.

[3] S. Di Mascio, A. Menicucci, G. Furano, C. Monteleone, and M. Ottavi, "The case for risc-v in space," in *Applications in Electronics Pervading Industry, Environment and Society*, 2019, international Conference on Applications in Electronics Pervading Industry, Environment and Society, APPLEPIES 2018, APPLEPIES 2018 ; Conference date: 26-09-2018 Through 27-09-2018.

[4] S. Di Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone, "Leveraging the openness and modularity of risc-v in space," *Journal of Aerospace Information Systems*, vol. 16, no. 11, pp. 454–472, 2019. [Online]. Available: https://doi.org/10.2514/1.I010735

[5] [Online]. Available: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/The_use_of_reprogrammable_FPGAs_in_space

[6] N. Montealegre, D. Merodio, A. Fernández, and P. Armbruster, "In-flight reconfigurable fpga-based space systems," in *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2015, pp. 1–8.

[7] M. Pignol, "Cots-based applications in space avionics," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, 2010, pp. 1213–1219.

[8] S. Esposito, C. Albanese, M. Alderighi, F. Casini, L. Giganti, M. L. Esposti, C. Monteleone, and M. Violante, "Cots-based high-performance computing for space applications," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2687–2694, 2015.

[9] E. Petersen, *Foundations of Single Event Analysis and Prediction*, 2011, pp. 13–76.

[10] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus, "Ibm experiments in soft fails in computer electronics (1978–1994)," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 3–18, 1996.

[11] P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583–602, 2003.

[12] E. L. Petersen, "Radiation induced soft fails in space electronics," *IEEE Transactions on Nuclear Science*, vol. 30, no. 2, pp. 1638–1641, 1983.

[13] M. Ceschia, M. Violante, M. Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori, "Identification and classification of single-event upsets in the configuration memory of sram-based fpgas," *IEEE Transactions on Nuclear Science*, vol. 50, no. 6, pp. 2088–2094, 2003.

[14] L. Sterpone and B. Du, "Analysis and mitigation of single event effects on flash-based fpgas," in *2014 19th IEEE European Test Symposium (ETS)*, 2014, pp. 1–6.

[15] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA '05. USA: IEEE Computer Society, 2005, p. 243–247. [Online]. Available: https://doi.org/10.1109/HPCA.2005.37

[16] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 29–40.

[17] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 117–128.

[18] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 168–179.

[19] L. A. Tambara, P. Rech, E. Chielle, and F. L. Kastensmidt, "Analyzing the failure impact of using hard- and soft-cores in all programmable soc under neutron-induced upsets," in *2015 15th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2015, pp. 1–5.

[20] P. Rech, L. Pilla, P. Navaux, and L. Carro, "Impact of gpus parallelism management on safety-critical and hpc applications reliability," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 455–466.

[21] D. J. Sorin, *Error Detection*. Cham: Springer International Publishing, 2009, pp. 19–59.

[22] C. De Sio, S. Azimi, A. Portaluri, 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)*, 2021, pp. 1–6.

[23] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[24] G. Tshagharyan, G. Harutyunyan, S. Shoukourian, and Y. Zorian, "Experimental study on hamming and hsiao codes in the context of embedded applications," in *2017 IEEE East-West Design Test Symposium (EWDTS)*, 2017, pp. 1–4.

[25] A. E. Wilson and M. Wirthlin, "Neutron radiation testing of fault tolerant risc-v soft processor on xilinx sram-based fpgas," in *2019 IEEE Space Computing Conference (SCC)*, 2019, pp. 25–32.

[26] A. M. Keller and M. J. Wirthlin, "Benefits of complementary seu mitigation for the leon3 soft processor on sram-based fpgas," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 519–528, 2017.

[27] D. A. Santos, L. M. Luza, M. Kastriotou, C. Cazzaniga, C. A. Zeferino, D. R. Melo, and L. Dilillo, "Characterization of a risc-v system-on-chip under neutron radiation," in *16th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2021.

[28] F. F. Dos Santos, A. Kritikakou, and O. Sentieys, "Experimental evaluation of neutron-induced errors on a multicore risc-v platform," in *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2022, pp. 1–7.

[29] M. J. Cannizzaro and A. D. George, "Evaluation of risc-v silicon under neutron radiation," in *2023 IEEE Aerospace Conference*, 2023, pp. 1–9.

[30] S. Nolting and ..., "The neorv32 risc-v processor," https://github.com/stnolting/neorv32, 2022.

[31] "Open-Source RISC-V Architecture IDs," https://github.com/riscv/riscv-isa-manual/blob/latex/marchid.md, [Online; accessed 13-June-2023].

[32] T. M. Conte, E. P. DeBenedictis, A. Mendelson, and D. Milojičić, "Rebooting computers to avoid meltdown and spectre," *Computer*, vol. 51, no. 4, pp. 74–77, 2018.

[33] *IGLOO2 FPGA and SmartFusion2 SoC FPGA*, Microsemi, 8 2018, dS01289.

[34] D. Dsilva, J.-J. Wang, N. Rezzak, and N. Jat, "Neutron see testing of the 65nm smartfusion2 flash-based fpga," in *2015 IEEE Radiation Effects Data Workshop (REDW)*, 2015, pp. 1–5.

[35] *IGLOO2 and SmartFusion2 65nm Commercial Flash FPGAs: Interim summary of radiation test results*, Microsemi corp., October 2014.

[36] N. Rezzak, j.-j. Wang, D. Dsilva, C. Huang, and S. Varela, "Single event effects characterization in 65 nm flash-based fpga-soc," 01 2014.

[37] *UG0498: SmartFusion2 and IGLOO2 Embedded Nonvolatile Memory (eNVM) Simulation User Guide*, Microsemi, 7 2018, revision 3.

[38] *AMBA AHB Protocol Specification*, ARM, 9 2021.

[39] *Using Synplify to Design in Microsemi Radiation-Hardened FPGAs*, Microsemi, 5 2012, application Note AC139. Rev 1.

[40] *SmartFusion2 and IGLOO2 Macro Library Guide*, Microsemi, 10 2019, revision 10.0.

[41] C. Cazzaniga and C. D. Frost, "Progress of the scientific commissioning of a fast neutron beamline for chip irradiation," *Journal of Physics: Conference Series*, vol. 1021, no. 1, p. 012037, may 2018.

[42] H. Quinn, "Challenges in testing complex systems," *IEEE Transactions on Nuclear Science*, vol. 61, no. 2, pp. 766–786, 2014.

[43] D. Asciolla, L. Dilillo, D. Santos, D. Melo, A. Menicucci, and M. Ottavi, "Characterization of a risc-v microcontroller through fault injection," in *Applications in Electronics Pervading Industry, Environment and Society - APPLEPIES 2019*, ser. Lecture Notes in Electrical Engineering, S. Saponara and A. De Gloria, Eds. SpringerOpen, 2020, pp. 91–101, international Conference on Applications in Electronics Pervading Industry, Environment and Society, ApplePies 2019 ; Conference date: 11-09-2019 Through 13-09-2019.

[44] I. Tuzov, J.-C. Ruiz, and D. de Andrés, "Accurately simulating the effects of faults in vhdl models described at the implementation-level," in *2017 13th European Dependable Computing Conference (EDCC)*, 2017, pp. 10–17.

[45] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

Appendix A

A.1 Assembly functions

Listing A.1: Assembly CoreMark cmp_complex()

```
60001268 <cmp_complex>
60001268:       addi    sp,sp,-32
6000126a:       sw      s1,20(sp)
6000126c:       lh      s1,0(a0)
60001270:       sw      s0,24(sp)
60001272:       sw      s2,16(sp)
60001274:       srai    a5,s1,0x7
60001278:       sw      s4,8(sp)
6000127a:       sw      ra,28(sp)
6000127c:       sw      s3,12(sp)
6000127e:       andi    a5,a5,1
60001280:       mv      s2,a1
60001282:       mv      s0,a2
60001284:       andi    s4,s1,127
60001288:       bnez    a5,600012cc <cmp_complex+0x64>
6000128a:       srai    a1,s1,0x3
6000128e:       andi    a1,a1,15
60001290:       slli    a3,a1,0x4
60001294:       andi    a4,s1,7
60001298:       lhu     a5,56(a2)
6000129c:       mv      s3,a0
6000129e:       or      a1,a1,a3
600012a0:       beqz    a4,60001352 <cmp_complex+0xea>
600012a2:       li      a3,1
600012a4:       beq     a4,a3,600013c6 <cmp_complex+0x15e>
600012a8:       slli    a0,s1,0x10
```

```
600012ac:   srli   a0,a0,0x10
600012ae:   mv     s4,s1
600012b0:   mv     a1,a5
600012b2:   andi   s4,s4,127
600012b6:   andi   s1,s1,-256
600012ba:   jal    60000c9c <crcu16>
600012bc:   or     s1,s4,s1
600012c0:   sh     a0,56(s0)
600012c4:   ori    s1,s1,128
600012c8:   sh     s1,0(s3)
600012cc:   lh     s1,0(s2)
600012d0:   srai   a5,s1,0x7
600012d4:   andi   a5,a5,1
600012d6:   andi   s3,s1,127
600012da:   bnez   a5,60001334 <cmp_complex+0xcc>
600012dc:   srai   a1,s1,0x3
600012e0:   andi   a1,a1,15
600012e2:   slli   a3,a1,0x4
600012e6:   andi   a4,s1,7
600012ea:   lhu    a5,56(s0)
600012ee:   or     a1,a1,a3
600012f0:   beqz   a4,6000138c <cmp_complex+0x124>
600012f2:   li     a3,1
600012f4:   bne    a4,a3,60001348 <cmp_complex+0xe0>
600012f8:   mv     a2,a5
600012fa:   addi   a0,s0,40
600012fe:   jal    ra,60001d9a <core_bench_matrix>
60001302:   lhu    a5,60(s0)
60001306:   slli   s3,a0,0x10
6000130a:   srai   s3,s3,0x10
6000130e:   bnez   a5,600013e8 <cmp_complex+0x180>
60001310:   lhu    a5,56(s0)
60001314:   sh     a0,60(s0)
60001318:   mv     a1,a5
6000131a:   andi   s3,s3,127
6000131e:   andi   s1,s1,-256
60001322:   jal    60000c9c <crcu16>
60001324:   or     s1,s3,s1
60001328:   sh     a0,56(s0)
```

```
6000132c:     ori     s1,s1,128
60001330:     sh      s1,0(s2)
60001334:     lw      ra,28(sp)
60001336:     lw      s0,24(sp)
60001338:     sub     a0,s4,s3
6000133c:     lw      s1,20(sp)
6000133e:     lw      s2,16(sp)
60001340:     lw      s3,12(sp)
60001342:     lw      s4,8(sp)
60001344:     addi    sp,sp,32
60001346:     ret
60001348:     slli    a0,s1,0x10
6000134c:     srli    a0,a0,0x10
6000134e:     mv      s3,s1
60001350:     j       60001318 <cmp_complex+0xb0>
60001352:     li      a3,34
60001356:     mv      a4,a1
60001358:     bge     a1,a3,60001360 <cmp_complex+0xf8>
6000135c:     li      a4,34
60001360:     lh      a3,2(s0)
60001364:     lh      a2,0(s0)
60001368:     lw      a1,20(s0)
6000136a:     lw      a0,24(s0)
6000136c:     zext.b  a4,a4
60001370:     jal     ra,60000b08 <core_bench_state>
60001374:     lhu     a5,62(s0)
60001378:     slli    s4,a0,0x10
6000137c:     srai    s4,s4,0x10
60001380:     bnez    a5,600013ee <cmp_complex+0x186>
60001382:     lhu     a5,56(s0)
60001386:     sh      a0,62(s0)
6000138a:     j       600012b0 <cmp_complex+0x48>
6000138c:     li      a3,34
60001390:     mv      a4,a1
60001392:     bge     a1,a3,6000139a <cmp_complex+0x132>
60001396:     li      a4,34
6000139a:     lh      a3,2(s0)
6000139e:     lh      a2,0(s0)
600013a2:     lw      a1,20(s0)
```

```
600013a4:    lw      a0,24(s0)
600013a6:    zext.b  a4,a4
600013aa:    jal     ra,60000b08 <core_bench_state>
600013ae:    lhu     a5,62(s0)
600013b2:    slli    s3,a0,0x10
600013b6:    srai    s3,s3,0x10
600013ba:    bnez    a5,600013e8 <cmp_complex+0x180>
600013bc:    lhu     a5,56(s0)
600013c0:    sh      a0,62(s0)
600013c4:    j       60001318 <cmp_complex+0xb0>
600013c6:    mv      a2,a5
600013c8:    addi    a0,s0,40
600013cc:    jal     ra,60001d9a <core_bench_matrix>
600013d0:    lhu     a5,60(s0)
600013d4:    slli    s4,a0,0x10
600013d8:    srai    s4,s4,0x10
600013dc:    bnez    a5,600013ee <cmp_complex+0x186>
600013de:    lhu     a5,56(s0)
600013e2:    sh      a0,60(s0)
600013e6:    j       600012b0 <cmp_complex+0x48>
600013e8:    lhu     a5,56(s0)
600013ec:    j       60001318 <cmp_complex+0xb0>
600013ee:    lhu     a5,56(s0)
600013f2:    j       600012b0 <cmp_complex+0x48>
```

A.2 Top-level design in Libero SoC

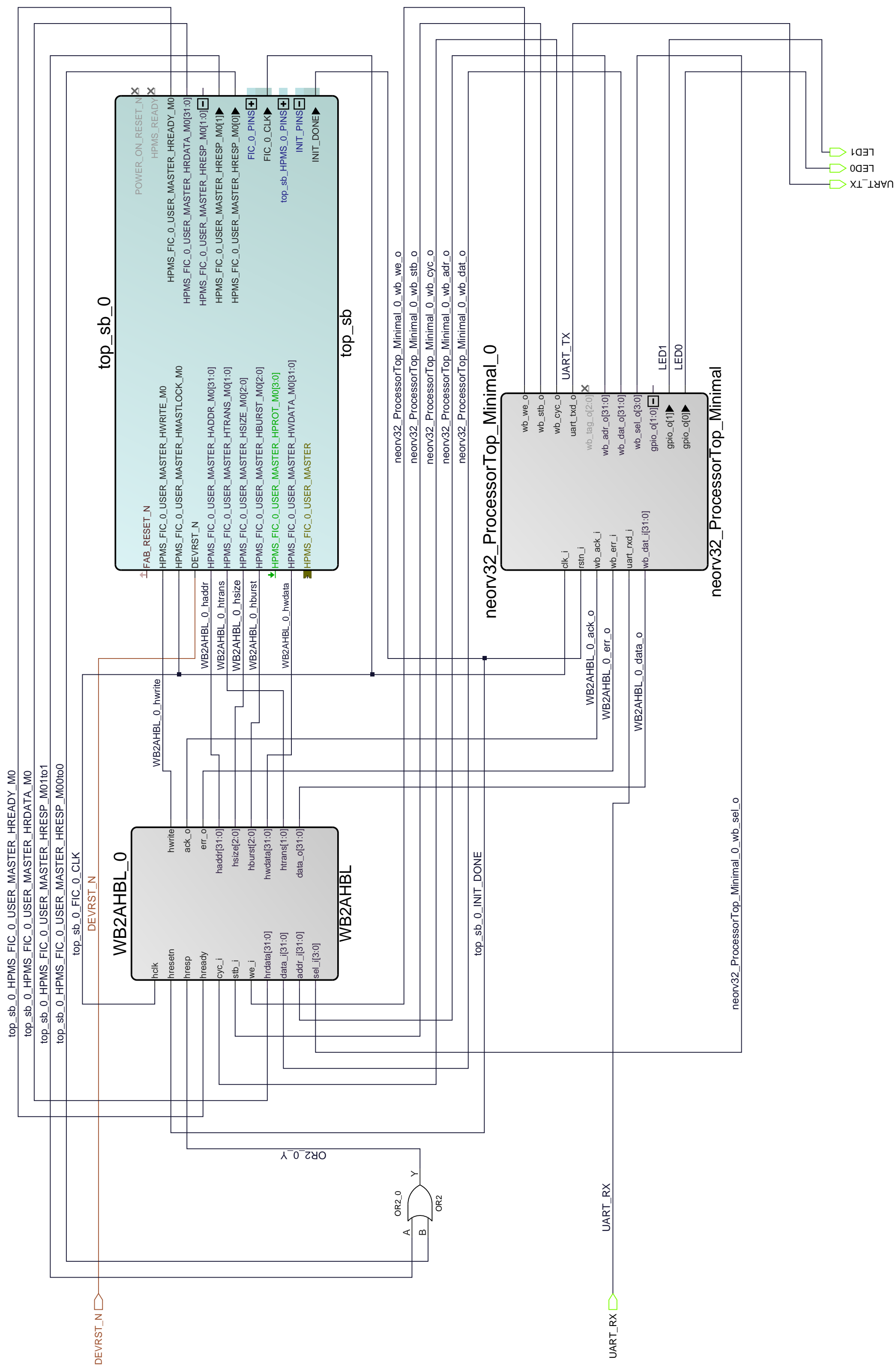

Figure A.1: Top level design created in Libero SoC

A.3 FPGA resources

Table A.1: Resource usage of the Unmitigated NEORV32 Processor subcomponents

Module Name	Fabric 4LUT	Fabric DFF	Interface 4LUT	Interface DFF
Primitives	67	25	0	0
Buskeeper	31	13	0	0
Busswitch	58	8	0	0
CPU	5014	2001	72	72
GPIO	10	5	0	0
DMEM	73	42	288	288
MTIME	258	166	0	0
Sysinfo	16	7	0	0
UART0	128	131	36	36
Wishbone	51	114	0	0
Total	**5704**	**2512**	**396**	**396**

Table A.2: Resource usage of the ECC-Enhanced NEORV32 subcomponents

Module Name	Fabric 4LUT	Fabric DFF	Interface 4LUT	Interface DFF
Primitives	80	25	0	0
Buskeeper	22	13	0	0
Busswitch	55	8	0	0
CPU	5352	2113	108	108
GPIO	9	5	0	0
DMEM	247	70	576	576
MTIME	256	166	0	0
Sysinfo	24	7	0	0
UART0	125	131	36	36
Wishbone	58	114	0	0
Total	**6288**	**2652**	**720**	**720**

Table A.3: Resource usage of the TMR-ECC NEORV32 Processor subcomponents

Module Name	Fabric 4LUT	Fabric DFF	Interface 4LUT	Interface DFF
Primitives	95	75	0	0
Buskeeper	38	39	0	0
Busswitch	67	24	0	0
CPU	9197	6333	108	108
GPIO	16	15	0	0
DMEM	256	74	576	576
MTIME	485	498	0	0
Sysinfo	31	21	0	0
UART0	317	393	36	36
Wishbone	304	342	0	0
Total	**10806**	**7814**	**720**	**720**

A.4 Parity check matrices

Table A.4: Partiy Check Matrix of Hsiao (13,8) code

syndrome[i]	c4	c3	c2	c1	c0	d7	d6	d5	d4	d3	d2	d1	d0	sum
0					1		1	1		1		1	1	6
1				1		1	1	1	1	1				6
2			1			1	1		1		1		1	6
3		1				1		1			1	1	1	6
4	1								1	1	1	1		5
	1	1	1	1	1	3	3	3	3	3	3	3	3	29

Table A.5: Parity Check Matrix of Hsiao (39,32) code

Syndrome[i]	c6	c5	c4	c3	c2	c1	c0	d31	d30	d29	d28	d27	d26	d25	d24	d23	d22	d21	d20	d19	d18	d17	d16	d15	d14	d13	d12	d11	d10	d9	d8	d7	d6	d5	d4	d3	d2	d1	d0	sum
0							1			1	1	1												1	1			1	1		1	1		1	1	1	1			14
1						1		1	1				1					1	1	1			1	1	1					1	1			1			1		1	15
2					1				1		1			1		1	1		1	1						1		1	1				1	1				1	1	15
3				1				1		1			1					1	1		1	1				1	1	1					1		1		1	1		15
4			1					1			1	1		1	1	1						1	1			1	1				1					1			1	14
5		1								1		1	1		1	1	1			1	1	1	1		1					1		1	1							15
6	1								1					1	1		1	1			1			1			1		1	1		1			1	1		1		14
	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	102