POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Reliability Analysis of Convolutional Neural Network through Soft Error Mitigation Controller

Supervisors

Prof. Luca STERPONE

Prof. Sarah AZIMI

Candidate

Manuel CAPACCIO

DECEMBER 2023

Abstract

The progresses of Deep Neural Networks (DNNs) in several disciplines like image processing, system monitoring and decision making, continue to accelerate, making them appealing for space applications, in particular if implemented on SRAM-based FPGAs, which offer several advantages, like low cost manufacturing, CPUs-like performance and field programmability. Therefore, radiation effects have to be considered in design phase. Ionizing particles can modify the state of gates in electronic devices, leading to permanent faults (Hard Errors) or temporary ones (Soft Errors). Single Event Upsets (SEUs), one kind of soft error, can modify the value of one or more bits stored in the FPGA configuration memory, potentially causing design failures. It is therefore important to be able to easily and cheaply verify the behavior of the design subjected to SEUs so as to be able to mitigate their effects if needed. In this Thesis the Xilinx Essential Bits technology and the capabilities of the Soft Error Mitigation (SEM) Controller, an IP developed by Xilinx, are exploited to perform fast and cheap fault injection campaigns to simulate SEUs and to deeply analyze the behaviour of a single convolutional neuron, implemented in the programmable logic of the Xilinx System on Chip Zynq XC7Z020. The neuron belongs to the input layers of the ZFNet Convolutional Network present in the Alpha Data CNN Library. Critical sections of the neuron architecture are identified and then mitigated with a selective Triple Modular Redundancy (TMR) to reduce resources overhead. Finally, the ZFNet input layers with the modified neuron in them are submitted to a fault injection campaign to understand if the mitigation results obtained for a small block of a redundant architecture, such as DNNs, where several neurons are present, are propagated to the entire design.

Acknowledgements

Grazie a Claudio e Leonardo per avermi aiutato nella correzione

Grazie ad Alessio, Davide, Luca e Pier, amici da una vita, per avermi sempre sostenuto e consigliato nelle mie scelte.

Grazie a Marica, per avermi sopportato con pazienza negli anni a Torino.

Grazie ad Anita, per essermi stata accanto anche nei momenti più difficili, senza mai lasciare la mia mano.

E grazie a mia madre, per i suoi sacrifici, per il suo supporto, per il suo affetto... per tutto, GRAZIE.

Table of Contents

List of Tables

List of Figures

Acronyms

ACK Acknowledgement

AI Artificial Intelligence

AMBA Advanced Microcontroller Bus Architecture

APU Application Processing Unit

AXI Advanced eXtensible Interface

BRAM Block Random Access Memory

CME Coronal Mass Ejection

CNN Convolutional Neural Network

CR Carriage Return

CRAM Configuration Random Access Memory

DDD Displacement Damage Dose

DMR Dual Modular Redundancy

DNN Deep Neural Network

DSP Digital Signal Processing

DUT Device Under Test

ECC Error Correcting Code

FPGA Field Programmable Gate Arrays

GPIO General Purpose Input/Output

HW Tb Hardware Testbench

IC Integrated Circuit

ICAP Internal Configuration Access Port

IDE Integrated Development Environment

IP Intellectual Property

LED Light Emitting Diode

LF Line Feed

LFA Linear Frame Address

LSB Least Significant Bit

LSFR Linear-Feedback Shift Register

LUT Look-Up Table

MBU Multiple Bit Upset

ML Machine Learning

MOS Metal-Oxide-Semiconductor

NLP Natural Language Processing

OCM On-Chip Memory

PCAP Processor Configuration Access Port

PL Programmable Logic

PS Processing System

RX Receiver

ReLU Rectifier Linear Unit

SEE Single Event Effect

SEFI Single Event Functional Interrupt

SEM Soft Error Mitigation

SET Single Event Transient

SEU Single Event Upset

SLR Super Logic Region

SOC System on Chip

SOI Silicon on Insulator

SOS Silicon on Sapphire

SRAM Static Random Access Memory

SSI Stacked Silicon Technology

TID Total Ionizing Dose

TMR Triple Modular Redundancy

TX Transmitter

XSA Xilinx Support Archive

Chapter 1

Introduction

Over the past few years, Artificial Intelligence (AI) and Machine Learning (ML) have undergone a remarkable surge in popularity, captivating the interest of researchers, industry leaders, and enthusiasts alike, owing to their diverse and impactful applications [1]. The pervasive integration of smart devices into our daily lives further emphasizes this trend. These devices, characterized by constant inter connectivity, exhibit an ever-growing ability to autonomously interact, exemplified by the now tangible reality of autonomous driving.

The ubiquity of wearable devices, a commonplace aspect of contemporary life, exemplifies this technological shift. These devices, ranging from fitness trackers to health monitors, diligently collect vast amounts of data about our daily activities.Then, AI and ML algorithms not only analyze the collected information but also draw meaningful insights, revealing patterns in our behavior and lifestyle.

Consider, for instance, the increase of smart home ecosystems, where AI plays a pivotal role in optimizing energy consumption, enhancing security, and personalizing user experiences. The convergence of AI and ML technologies has thus transformed our surroundings into intelligent, adaptive environments, fostering a new era where technology not only serves as a tool but actively contributes to the augmentation of our daily lives.

Given the boundless possibilities, it would be imprudent not to consider leveraging these new technologies in environments and applications where human intervention

is truly challenging or altogether impossible. Examples include aerospace applications or hostile environments. In such scenarios, the implementation of Artificial Intelligence (AI) and Machine Learning (ML) proves invaluable. These technologies can autonomously navigate and process vast datasets in real-time, offering solutions that transcend the limitations of human capabilities. In aerospace applications, for instance, AI-powered systems can enhance autonomous navigation, optimize mission planning, and analyze complex sensor data more rapidly than traditional methods.

Deep Neural Networks (DNNs) are certainly the most widely used Machine Learning algorithms. In recent years, these deep neural networks have demonstrated their remarkable effectiveness in solving a wide range of problems. Their applications span from image recognition to Natural Language Processing (NLP), showcasing their versatility and power [2].

Among the models of DNNs, certainly, one of the most well-known is the Convolutional Neural Network (CNN). Their rise began around 2012, with models like AlexNet, which demonstrated the capability to autonomously recognize crucial features from input data streams and proved to be computationally efficient [3]. CNNs emerge as an attractive option for space applications and the prospect of implementing them on SRAM-based Field Programmable Gate Arrays (FPGAs) adds to their appeal. Indeed the system offer not only a cost-effective solution, but CPUs-like performance too. Moreover, the flexibility of on-the-field programming enhances their adaptability to the dynamic and evolving requirements of space missions. This combination of features positions CNNs on SRAM-based FPGAs as a compelling and efficient choice for real-time image processing and decision-making in the challenging environments of space exploration [4].

Deploying Convolutional Neural Networks on FPGAs poses several challenges, notably the constraint of limited hardware resources on FPGAs, which contrasts with the intricate and sizable nature of CNN architectures. While efforts can be directed towards creating more resource-efficient architectures, another persistent challenge is the impact of high energy particle radiation [5]. Particularly in space applications or harsh environments, radiation can introduce various errors into the configuration memory of the FPGA, necessitating mitigation strategies.

The impact of radiation on electronic devices can be both destructive and irreversible, causing malfunctions in critical applications and undermining the extensive design efforts put into FPGA-based systems. Hence, it is imperative not only to prevent potential radiation effects on the device but also to implement mitigation solutions, thereby increasing the system's reliability. To achieve this goal, an effective methodology for studying the radiation effects of interest is crucial from the early stages of design development.

This forms the foundation of the work presented in this thesis. In the following pages, a potential methodology for investigating design tolerance to one of the various radiation effects, specifically Single Event Upsets (SEUs), is outlined. SEUs involve bit switching in the configuration memory (CRAM) of the FPGA, potentially resulting in faults in the implemented circuit. The focus of this thesis is on a hardware implementation of a Convolutional Neural Network, the ZNFnet, made available by Alpha Data [6].

1.1 Radiation Effects

In this section the radiation effects are detailed.

1.1.1 Radiation Overview

Radiations are nothing but subatomic particles traveling through space at extremely high speeds and can originate from various sources [7]:

- **Cosmic Rays**: These particles, generally atomic nuclei, originate outside the solar system. Cosmic rays are particularly relevant for aerospace and space applications.
- **Coronal Mass Ejection (CME)**: These are protons and other high-energy particles emitted from the solar corona.
- **Solar Wind:** In contrast to CMEs, solar wind is steady and consists mainly of electrons and protons coming from the Sun

- **Alpha Particles:** These particles originate from the decay of certain materials and can impact even terrestrial applications.
- **Neutrons:** They are primarily generated in the atmosphere, and due to their neutral charge, it is difficult to fully shield them.

1.2 Physical Effects

The effects of radiation come in different types and, initially, can be categorized based on whether they are cumulative events, such as Total Ionizing Dose (TID) and Displacement Damage Dose (DDD), or single events, namely Single Event Effects (SEE). In turn, these single events can cause destructive effects, known as *hard errors*, or transitory effects, referred to as *soft errors*.

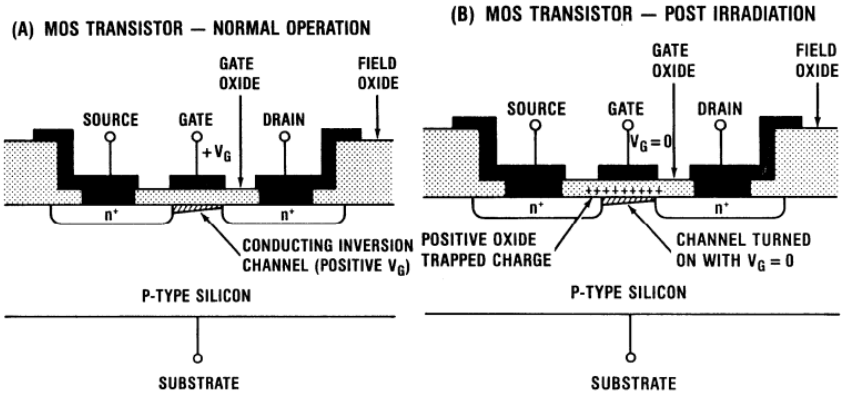

Figure 1.1: Radiation-induced charging of Gate oxide in a n-channel MOS Transistor [8]

Total Ionizing Dose (TID): When ionizing particles strike a Metal-Oxide-Semiconductor (MOS) transistor, they release electrons from the material, leaving holes that act as positive charges. Electrons and holes tend to recombine, generating current peaks (photocurrents). However, holes have low mobility and can become trapped in the gate lattice, as shown in Figure 1.1. As these events accumulate, the holes build up, creating a charge high enough to alter the threshold voltage of the transistor. This complicates the control of the MOS, and a significant accumulation

of charges can lead to n-channel MOS transistors always being on and p-channel MOS transistors always being off [5][8].

Displacement Damage Dose (DDD): When heavy particles collide with the silicon lattice, they can displace atoms, creating vacancies in the crystal structure. These defects can become traps for carriers and recombination centers. They can also alter the semiconductor's band structure, disrupting the normal functionality of the transistor [5][8].

Single Event Effects (SEE): As we have seen, ionizing particles that strike the device generate electron-hole pairs, which, due to the different mobility of the two carriers, lead to undesired phenomena (Figure 1.2).

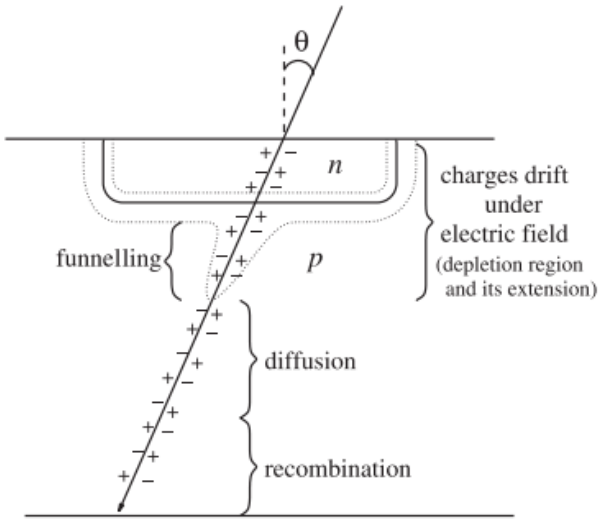

Figure 1.2: MOS Ionization caused by incident particle [9]

In addition to the cumulative effects described earlier, they also lead to immediate effects, the so-called Single Event Effects. These are further divided based on their effects on the device, as reported in Figure 1.2.
Soft errors tend to be critical, especially in the context of space applications with SRAM-based FPGAs [10][11]. For this reason, this thesis focuses on one specific type of soft error, namely Single Event Upsets (SEUs).
Taking a quick look at soft errors:

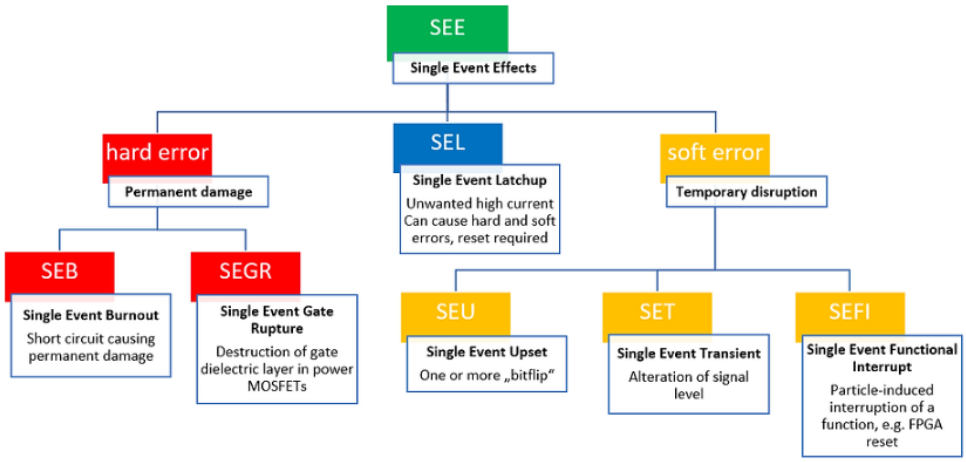

Figure 1.3: SEEs classification [7]

- **Single Event Transient (SET):** It is a temporary and unintended change in the state of a digital circuit, resulting in short-duration glitches or pulses.
- **Single Event Upset (SEU):** Termed as a bit-flip, it is triggered by a SET affecting a memory element within the circuit. This disturbance can modify the stored charge, leading to a change in the output state. When it influences more than one bit, it is termed a Multiple Bit Upset (MBU). The significance of SEUs in SRAM-based FPGA systems is now evident.
- **Single Event Functional Interrupt (SEFI)**: it is a condition in which the device abruptly ceases its normal functions, often necessitating a power reset to restore regular operations.

Chapter 2

State of Art

In this chapter, various techniques of radiation testing and radiation hardening are outlined, providing an overview of the state of the art upon which this Thesis is built.

2.1 Radiation-Hardening Techniques

As introduced in Chapter 1, radiation-effects pose a significant challenge in aerospace and space environments. For this reason, over the years, various radiation-hardening techniques have been developed to counteract and mitigate these effects. These countermeasures can be broadly divided into two main groups: *physical techniques*, such as radiation shielding of devices, and *logical techniques*, which involve logical solutions such as hardware and software redundancy or the use of error correcting code (ECC) memories.

This section will outline some of the main techniques currently available to address the impacts of radiation.

2.1.1 Physical Techniques

Shielding: This is certainly the most straightforward technique to implement. If radiation damages the device, simply blocking it should be sufficient. True in theory, but in practice, things are much more complex.
The fundamentals of shielding involve the use of protective materials capable of

absorbing or deflecting ionizing particles, such as lead, tungsten, and charged polymers. The effectiveness of these materials is closely related to their density and composition, being higher for denser materials with a greater capacity to absorb energy. Numerous studies are ongoing to discover new and optimal shielding materials, like lunar soil [12].
Material selection is crucial since each material can shield only certain types of ionizing particles. For example, shielding alpha particles is easier compared to the neutron flux from cosmic radiation [13].
In addition, other limitations are present. Two crucial aspects are weight and space, particularly important in space applications, where the choice of shielding materials must balance effectiveness in protection with the need to maintain compact weight and dimensions. Then costs are a determining factor too, as the use of advanced materials and shielding techniques can lead to increased production costs, often requiring a compromise between the desired level of protection and the economic sustainability of the project.
These challenges mean that the shielding of integrated circuit (IC) is generally not sufficient to eliminate SEUs but only to reduce their rate.

Insulating substrate: Hardened chips are often produced on insulating substrates instead of the typical semiconductor wafers. Silicon on Insulator (SOI) and Silicon on Sapphire (SOS) are commonly adopted.
SOI CMOS technology involves the insertion of an insulating layer between the transistor junction and the silicon substrate, significantly improving performance compared to classic bulk transistors. This technique eliminates potential parasitic transistors in the substrate, making it immune to latch-up. Additionally, it reduces the volume for charges collection from ionizing particles, thereby increasing tolerance to SEEs [14].

2.1.2 Logical Techniques

Error correcting code (ECC): It is a technique employed in computer memory systems to identify and correct errors in stored or transmitted data. It uses additional parity bits to detect and correct errors and so it can recover from bit-flip caused by SEUs.

ECC is commonly used in memory and storage systems to ensure data integrity, though it comes with an overhead in terms of additional bits. Hamming codes are a common form of ECC, and its implementation can be in hardware or software [15].

Redundancy: Redundancy techniques are often employed as a countermeasure against ECC to enhance the reliability of electronic systems. Redundancy techniques aim to detect and correct errors due to SEEs, primarily SEUs, minimizing their impact.
This technique can be implemented in various ways, and these can be primarily divided into two approaches:

- **Spatial Redundancy:** It involves using duplicate components or systems to perform the same task concurrently. One of the most widespread techniques following this approach is Triple Modular Redundancy (TMR) [16], also implemented for the SEUs mitigation on the design analized in this Thesis.

- **Temporal Redundancy:** It involves re-executing a task if an error is detected, potentially with different data or a different algorithm. An example is software redundancy, which involves executing the same instruction multiple times in succession and then comparing the results [17]

Triple Modular Redundancy (TMR)

TMR approach entails the deployment of three identical components or systems, each functioning simultaneously, applying spatial redundancy.

In a TMR system, the three components operate concurrently, each performing the same set of tasks or computations. To ensure the correct output under normal circumstances, a voting mechanism is integrated. This voting mechanism scrutinizes the outputs of the three redundant components and selects the result that is either agreed upon by a majority or adheres to a predefined consensus algorithm.

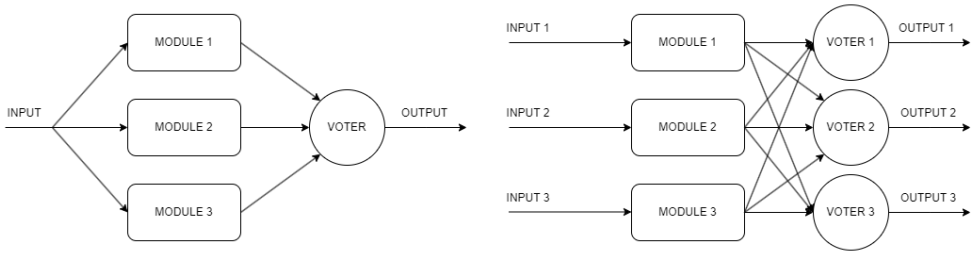

Figure 2.1: Partial TMR Implementation (left), Full TMR Implementaion (right)

While TMR significantly enhances fault tolerance [16], offering the capacity to withstand the failure of one of its three components, it introduces certain trade-offs. The triplication of components increases the complexity of the system, adds weight, and incurs additional costs.

As depicted in Figure 2.1, TMR implementation can vary [7]. These variations often arise due to physical constraints, especially in the context of resource-limited FPGAs-based system. The flexibility of TMR allows designers to adapt its configuration to meet the unique demands and limitations of the particular system or application. To reduce resource overhead, the Dual Modular Redundancy (DMR) can be adopted [18].

2.2 Radiation Effects Testing

At this point, it is clear how radiation effects are dangerous and need to be countered. In the previous section, some mitigation techniques currently used were introduced. However, these need to be tested to be considered valid. Therefore, the methodology for testing the effects of radiation on integrated circuits also becomes significantly important.

To validate a device's compliance with imposed radiation hardness requirements, various approaches can be employed. The most straightforward is the radiation test, involving exposing the Device Under Test (DUT) to a radiation beam for a predefined time under controlled environmental conditions (e.g., vacuum, controlled temperature). Functional tests are conducted during this process, and the outcomes are analyzed to understand how the device performs under radiation influence. An

example of a beam radiation test can be found here [19], where a radiation hard EM Calorimeter has been analyzed.

However, radiation testing has significant drawbacks since it is expensive in terms of both time and budget resources, requiring a specialized facility and months of preparation. To address these challenges, alternative approaches, such as simulation and fault emulation, are considered.

This Thesis relies on fault emulation, a distinct approach from simulation, aiming to recreate the effects of the operational environment. Among the SEEs, the most common events are SEUs, an alteration of a single bit status (1 to 0 or viceversa) in a memory or register. These kind of fault can be emulated through different methods.Among these, to test RTL design reliability, the prevalent method involves the utilization of SRAM-based FPGA for implementing the DUT. This strategy leverages the Configuration RAM (CRAM), which stores both the circuit design and the memory element contents. By intentionally reprogramming (bit-flipping) the CRAM bits, it becomes feasible to precisely emulate the effects SEUs. In this context, a potential implementation of this methodology is reported here [20], where the inherent structure of the FPGA is harnessed to introduce faults systematically into all flip-flops of the DUT.

Such solutions offer exceptional fault tolerance characterization with notable reductions in time and costs compared to traditional radiation tests. In this thesis, a comparable approach has been embraced. An IP provided by Xilinx, the LogiCORE™ IP Soft Error Mitigation (SEM) Controller, was employed, granting real-time access to the CRAM of the implemented FPGAs. This facilitated the development of a fault injection environment aimed at characterizing and validating the DUT [21]. The methodology is exemplified in detail within this thesis, showcasing its effectiveness in evaluating fault tolerance in electronic systems.

Chapter 3

System Description and Methodology

The thesis goals are introduced in this chapter. Then, the system and the methodology implemented for the reliability analysis are described.

3.1 Goals

The main goals of this thesis are:

- Demonstrate the capabilities of SEM Controller for reliability analysis, by emulating SEUs and by validating SEU-tolerant design solutions;
- Illustrate a possible methodology to mitigate SEU effects in a CNN, reducing design time and critical FPGA resources overhead.

The implementation of DNNs on SRAM-based FPGAs is appealing for avionic and space applications. However, in such harsh environments, electronic devices can be damaged by ionizing particles of electromagnetic radiations. As a consequence, they must be subjected to radiation-hardening techniques and this leads to huge production time and costs. One of the main advantages of commercial SRAM-based FPGAs is precisely their low cost manufacturing, thus it became useful being able to perform reliability analysis in the cheapest and fastest way.

Here comes into play the Xilinx SEM Controller. It is an IP able to write directly to the configuration memory of the FPGA, giving the possibility to injects error and emulate SEUs. If used together with the Xilinx Essential Bit technology, it should grant detailed and fast fault injection campaigns.
In addition, trying to further reduce the time needed for reliability analysis and SEUs mitigation, it was decided to exploit the redundant structure of the CNN under test. Instead of analyzing the entire architecture, the single neuron is analyzed first. Thanks to the SEM Controller, its most sensitive parts to SEUs are researched in order to improve its reliability through selective redundancy, avoiding to increase the number of BRAMs and DSPs needed, critical resources for the FPGA. Only if and when the new architecture for the single neuron has been validated, the reliability of the entire neural network is analyzed to verify if the solutions adopted for the single neuron increase the tolerance to the SEUs of the entire architecture.

3.2 System Description

In this section the experiments setup and the main functional block and resources employed are described

3.2.1 Xilinx Zynq-7020 SoC

The two designs analyzed in this thesis have been developed and implemented on a development board, the TUL PYNQ-Z2, shown in Figure 3.1, which has the System on Chip (SoC) Zynq-7020, developed by Xilinx.

The SoC is the reason for which the PYNQ-Z2 has been used. It grants the possibility to use the SEM Controller and it implements all the elements of an elaboration system on a single chip and, in particular, it is composed by two main blocks:

- The Processing System (PS);
- The Programmable Logic (PL);

The PS contains the Application Processing Unit (APU), based on a Dual-core ARM Cortex-A9 MPCore, cache memories, On-Chip Memories (OCM), external

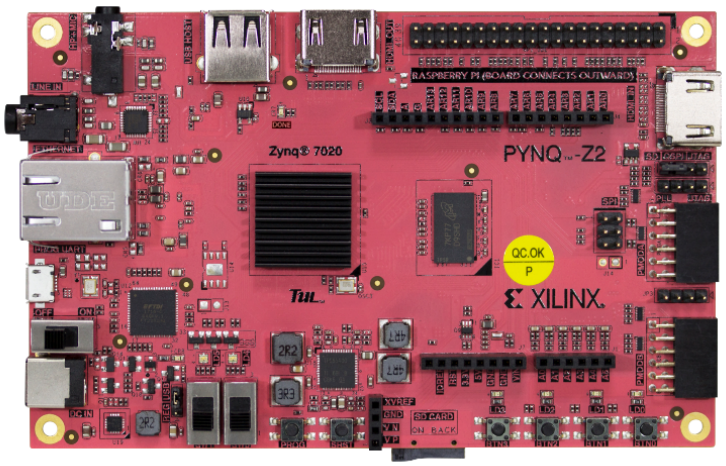

Figure 3.1: TUL PYNQ-Z2 [22]

memory interfaces, I/O peripherals and interfaces and PS-PL high bandwidth interconnections.
The PL, based on the Xilinx FPGA Artix-7 [23], is composed by:

- 85K Programmable Logic Cells;
- 53.200 Look-Up Tables (LUTs);
- 106.400 Flip-Flops;
- 280 18Kb Block RAM (BRAM);
- 220 DSP Blocks.

The PS-PL structure offers the possibility to run an application software on the PL, easily exploiting all the hardware potential implemented in the PL. In addition, both PL and PS are able to write directly to the CRAM. This ability is usually exploited to perform partial and dynamic reconfiguration. However, in this thesis, it will be used by the SEM Controller to perform faults injection.

The PS and PL are interconnected through Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) [24]. The AXI interface is composed by:

- **AXI4**: used for memory mapped interfaces. It allows data transfer burst up to 256 cycles.

- **AXI4-Lite**: similiar to AXI4 but bursting it's not supported. In this thesis it will be used to exchange control and data signal between PS and PL.

- **AXI4-Stream**: infinite bursting is supported. Useful to transfer large amount of data.

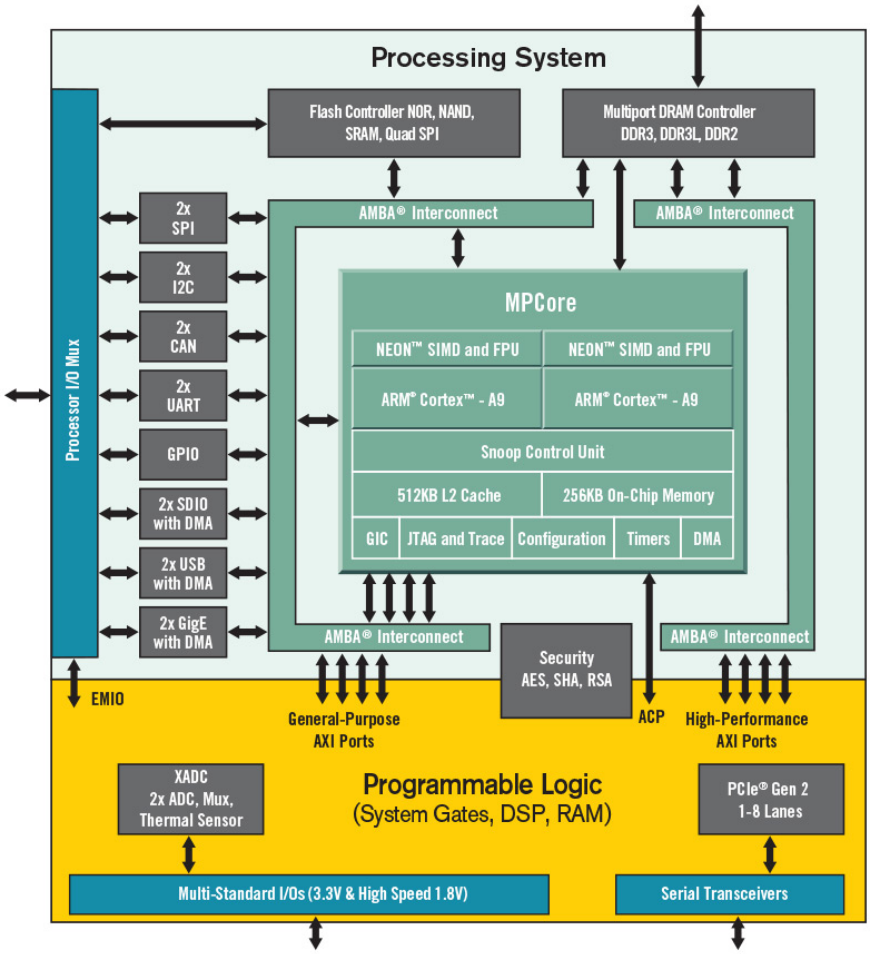

Figure 3.2: Xilinx Zynq-7000 SoC [23]

3.2.2 Vivado Design Suite

Xilinx offers designers several powerful software to work efficiently with its devices. In particular, the software suite called *"Vivado Design Suite - HLx Editions - 2020.1"* has been used for this thesis. It is composed by four different software:

- **Vivado 2020.1;**
- **Xilinx Vitis 2020.1;**
- **Vivado HLS 2020.1;**
- **Vitis HLS 2020.1;**

3.2.2.1 Vivado

Vivado [25] is an Integrated Development Environment (IDE) useful to develop and simulate hardware and programming FPGA. It grants the possibility to:

- **Describe RTL**: Hardware can be described by Verilog or VHDL languages. In addition, Vivado contains a graphic editor, *IP Integrator*, which allows designers to create Block Diagrams, where Xilinx IP or custom ones are seen as blocks and the architecture is described by simply connecting them with the mouse;
- **Simulate RTL**: If a testbench is provided, up to three simulations can be performed: behavioural, post-synthesis functional and timing and post-implementation functional and timing. Moreover, Vivado embeds a waveform viewer, essential to detect design errors during simulation phase;
- **Synthesize RTL**: RTL description is translated in a netlist of logic gates;
- **Implement RTL**: After synthesis, the netlist can be implemented in the FPGA. Place and Route operations are performed;
- **Generate Bitstream**: Only when implementation is complete, the configuration files for the FPGA can be generated and the device can be programmed.

Furthermore, several reports with crucial information, such as about timing, power and resource utilization, can be generated during the different design steps.

3.2.2.2 Xilinx Vitis

Vitis [26], based on *Eclipse*, allows designers to develop and debug software applications and to load them in the destination device CPU. One of the Vitis main advantages, exploited in this thesis, it is represented by the possibility to create applications based on Vivado projects. In this way the software, which is run by the PS, can interface perfectly with the hardware implemented in the FPGA, since drivers are automatically generated.

3.2.2.3 Vivado HLS and Vitis HLS

Vivado HLS and Vitis HLS [27] both grant the possibility to describe hardware blocks through the use of high level programming languages, such as C or C++, by generating their corresponding descriptions in VHDL or Verilog.
Neither software was used in this thesis.

3.2.3 Experiment Setup

As anticipated, two different blocks will be analyzed (single neuron and ZFNet input layer) but the same experiment setup has been used for both of them.

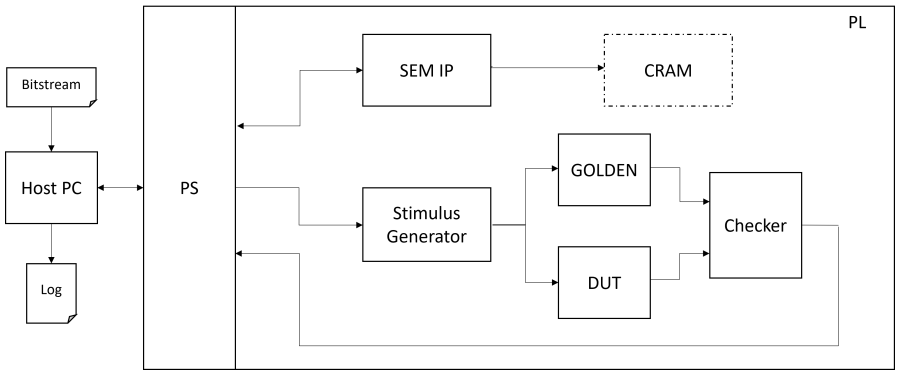

Figure 3.3: Experiment Setup

As shown in the figure 3.3, the setup is composed by:

- **Device Under Test (DUT)**: It is the block being analyzed and subjected to fault injection. In the first analysis it is the convolutional single neuron, in the second it is represented by ZFNet Input Layer. Unfortunately, due to the

limited number of resources and area available in the Zynq-7020, the entire ZFNet design could not be analyzed.

- **Golden Model (GOLD)**: It is a working copy of the DUT but it is not subjected to fault injection. It is necessary to compare the DUT outputs, that could be affected by fault injection, with correct ones in order to understand if the DUT is working correctly as its reference copy does.
- **Input Generator**: As the name suggests, it generates the input signals for both the DUT and the GOLD.
- **Checker**: It receives both the DUT and GOLD output signals, compares them and check if an error occurred.
- **SEM IP**: As anticipated, it is the main block for this thesis. Thanks to its capability to overwrite the configuration memory of the FPGA, it is used to inject errors to emulate SEUs, allowing to verify how they affect the DUT behavior.
- **PS**: The Processing Element of the Zynq-7020 SoC is mainly used as a bridge between the host pc and the architecture implemented in the PL. According to the commands received from the host PC through a serial interconnection, it enables the Input Generator, reads and send back the Checker outputs and drives the SEM IP.
- **Host PC**: The host PC runs a python script developed for this thesis. It manages the generation of the injection addresses for the SEM IP (as explained in section 3.3.5) and generates log files which contain information on the errors encountered. The python script and the application running on the PS are synchronized exploiting the blocking capability of their serial write and read functions, used to exchange data, which implement a sort of hand-shake protocol.

3.2.4 Input Generator

The Input Generator, called also Hardware Testbench (HW Tb), is a custom IP, described in VHDL, developed to correctly stimulate the DUT. Two versions of

it have been implemented, one for the single neuron and one for the ZFNet input layers.
Its main advantages are:

- Simplify the application running on the PS.
- Allow to easily perform fault injection while the DUT is working.

The first idea was to stimulate the DUT directly from the PS. In this way, a detailed control over the input generated would have been achieved, but, at the same time, it would have lead to a more complex application for the PS and to the development of a custom AXI interface, able to exchange data between the PS and the DUT. To obtain a correct timing would have been not trivial.
To reduce the effort needed to bring up a working experiment setup, a different solution has been explored: by analyzing the VHDL testbenches provided by Alpha Data, it has been noted that an equivalent test algorithm can be described through a simple state machine, as shown in the Figure 3.4. In both versions, inputs are generated pseudo-randomly through the use of a Linear-Feedback Shift Register (LFSR).

The algorithms implemented for the two versions are practically the same, except for some small variations due to the different number of neurons present in the architectures. As described in Chapter 5, the ZFNet input layers has two layers of neurons and the corresponding weights are stored in memory one layer at a time.

Taking into account this variation, the Input Generator implemented function can be summarized as:

1. Wait for start.
2. Generate and send the correct number of needed weights to DUT.
3. Send features continuously, until the block is enabled.

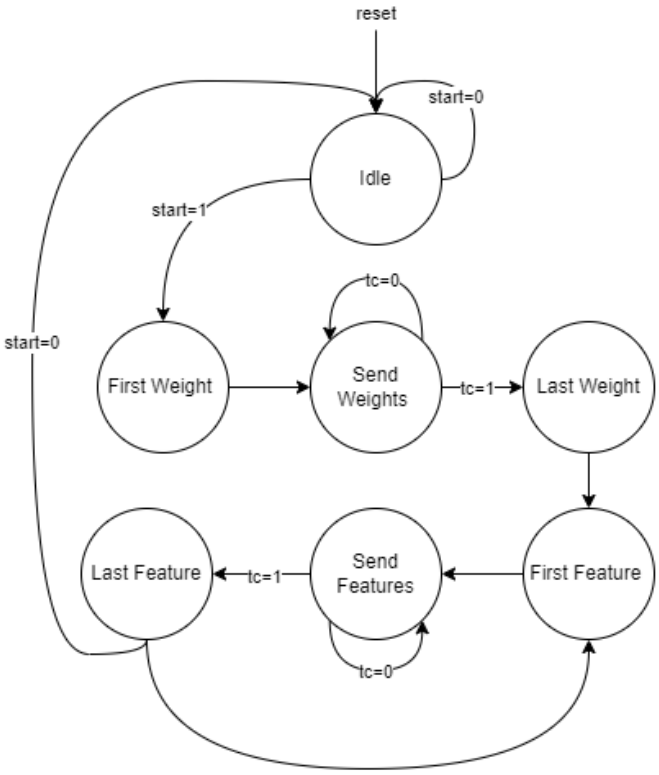

Figure 3.4: Input Generator FSM

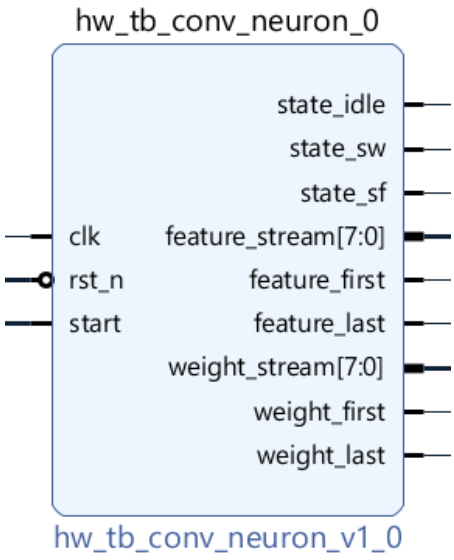

Figure 3.5: Input Generator in Vivado IP Integrator

Furthermore, as shown in Figure 3.5, the Input Generator needs only three input signals: a clock, a reset and an enable. With this configuration, the PS has simply to enable or disable the block and the rest of the work is done by the block itself. This allows to easily perform fault injections while the DUT is running, emulating a more real scenario.

Some status signal have been implemented and connected to some LEDs of the board, just to be able to easily observe the state of the circuit while running.

3.2.5 Checker

The Checker is a custom IP, described in VHDL, developed to compare the DUT and GOLD outputs, looking for mismatches between compared signals. Two versions of it have been implemented, one for the single neuron and one for the ZFNet input layers.

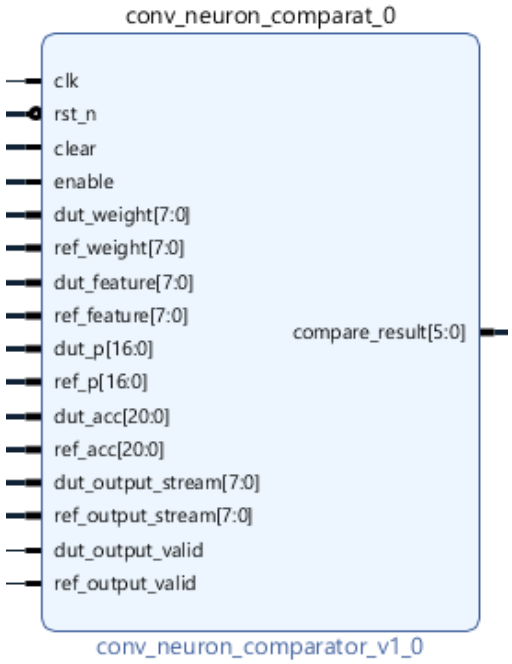

Figure 3.6: Checker in Vivado IP Integrator

The two versions of the Checker perform the same function, that is to compare the signals coming from the DUT and the GOLD at each clock cycle. They differ

only in the number of signals to be compared. While for the ZFNet only output data and output valid signals are compared, in the case of the single neuron some internal signals are also compared, so as to be able to trace the origin and possible propagation of the error (see section 4.2.1 for details).

3.2.6 SEM Controller IP

The LogiCORE™ IP Soft Error Mitigation (SEM) Controller is an automatically configured, pre-verified solution to detect and correct (not prevent) soft errors in Configuration Memory of Xilinx FPGAs [28]. In addition, it provides error injection capability to better evaluate SEM Controller operations.

In this thesis, the latter functionality has been exploited to perform Fault Injection campaigns, offering the possibility to emulate and analyse the effects of Single-bit error in FPGA Configuration Memory.

In the next lines, a brief description of the SEM Controller and of how to correctly work with it is provided.

3.2.6.1 System Level Design Example

Xilinx provides an example design for SEM controller. It is strongly suggested to use it to operate SEM IP properly [29]. This example design encapsulates the controller and provides:

- The FPGA configuration system primitives and their connection to the controller.
- The **MON shim**, a bridge between controllers and a standard RS-232 port. The resulting interface can be used to exchange commands and status with controllers. It grants the possibility to easily drive the SEM controller by a host pc.
- The **EXT shim**, a bridge between controllers and a standard SPI bus. The resulting interface can be used to fetch data by controllers. This shim is only present when error correction by replace and/or error classification are enabled. Used for connection to standard SPI flash.

- The **HID shim**, a bridge between controllers and an interface device. The resulting interface can be used to exchange commands and status with controllers. It is only present when error injection capability is enabled.

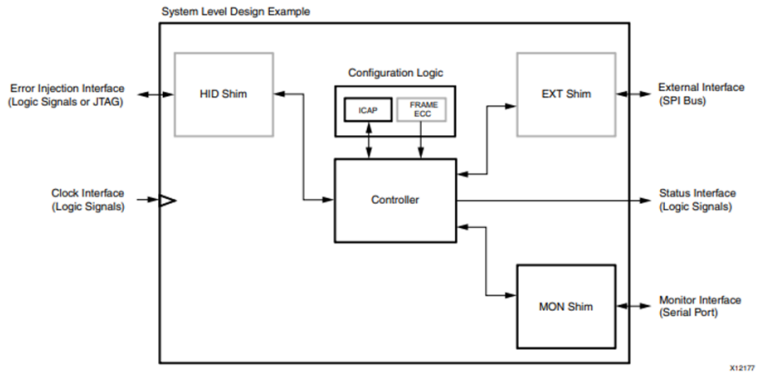

Figure 3.7: SEM Controller System Level Design Example Block Diagram [29]

As anticipated before, the grey blocks are present only in certain controller configurations. The system-level example design is verified along with the controller. It is not a "reference design," but an integral part of the total solution. While users do have the flexibility to modify the system-level example design, it is recommended to use it as delivered [29].

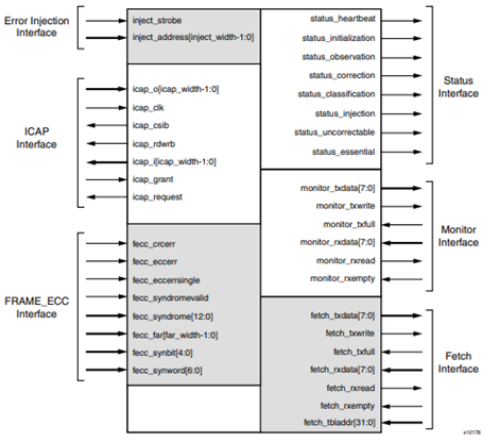

Figure 3.8: SEM Controller Interfaces [29]

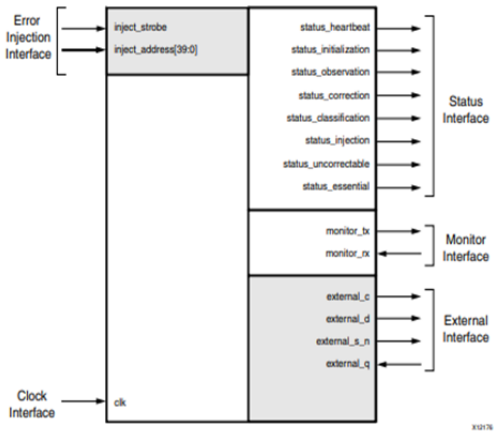

Figure 3.9: SEM Controller System-Level Design Example Interfaces [29]

The system-level design example encapsulates the SEM controller and, as showed in the Figures 3.8 and 3.9, it gives the possibility to manage the controller in a simpler way, by reducing the number of ports and interfaces and their complexity.

As suggested by the documentation, in this thesis the SEM Controller is used through its example design and therefore only its interfaces will be described below.

The **Status Interface** is composed by eight signals: five to indicate in which working state the controller is, two to provide information about the last error detected (if essential and if correctable), one to signal the core is alive while in observation state. All the eight signals are active high. In details:

- **status_heartbeat:** Active while status_observation is asserted. This output issues a single-cycle high pulse at least once every 150 clock cycles. It can be used to implement an external watchdog timer to check if the core is still scanning. When status_observation is deasserted, the behavior of the heartbeat signal is unspecified.

- **status_initialization:** Active only during controller initialization, which occurs one time after the design begins operation.

- **status_observation:** Asserted when the controller is observing for errors detection signals.

- **status_correction:** Active during controller correction of an error or during transition through this controller state if correction is disabled.

- **status_classification:** Active during controller classification of an error or during transition through this controller state if classification is disabled.

- **status_injection:** Active during controller injection of an error. When the injection is complete and the SEM can receive another injection command, the signal is deasserted.

- **status_essential:** It is an error classification status signal. Before leaving the correction state, the controller asserts this signal if the error occurred on an essential bit.

- **status_uncorrectable:** It is a correction status signal. Before leaving the correction state, the controller asserts this signal if the error was correctable. This signal is updated only after an error correction and it maintains its state until the next correction.

From the first five state signals it is possible to decode two additional controller states. If all five signals are low, the controller is in **Idle**. If all five signals are high, the controller is halted (due to **Fatal Error**).

The **Error Injection Interface** is one of the two interfaces through which commands can be sent to the SEM IP. It is usually used to drive the core by the FPGA Processing System (an additional Custom IP to manage communications is needed). This interface is present only if error injection is enabled during IP Customization in Vivado. It is composed by:

- **inject_strobe:** After a valid data is present on inject_address, it should be pulsed high for one clock cycle, synchronous to icap_clk. When sampled high, the value present on inject_address port is captured.
- **inject_address [39:0]:** Used to specify commands and error injection parameters.

The **Clock Interface** is used to provide a clock to the example design. It must not exceed the ICAP maximum frequency. For Zynq-7020, FMAX=100 MHz [29].

The **Monitor Interface** is always present, and it can be used to send commands to the SEM controller and receive back useful information about the working state of the IP. As shown in figure 3.9, the example design Monitor Interface is composed only by an RX signal and a TX signal, used to realize a RS-232 serial port. This interface is controlled by the MON shim provided by the example design. The MON shim implements a bridge between the SEM controller monitor interface and the example design one.

The default configuration parameters for the implemented RS-232 are the following:

- Baud: 9600
- Settings: 8-N-1
- Control: None
- Terminal Setup: VT100
  - TX Newline: CR (Terminal transmits CR [0x0D] as end of line).
  - RX Newline: CR+LF (Terminal receives CR [0x0D] as end of line, and expands to CR+LF [0x0D, 0x0A]).
  - Local Echo: NO

The default bit rate is small, and it can slow down the controller. Fortunately, higher bit rates can be implemented, including 115200, 230400, 460800, and 921600 baud. In the MON shim system-level example design module, the parameter V_ENABLETIME sets the communication bit rate:

$$V\_ENABLETIME = roundtointeger\left(\frac{inputclockfrequency}{16 * nominalbitrate}\right)^{-1}$$

This parameter can be found in the MON shim VHDL/Verilog source file and for this thesis it has been set to obtain a baud rate of 115200.

The **icap_grant** signal, if asserted, tells to the SEM Controller that it has permission to write to the CRAM.

The **External Interface** consists of four signal implementing a SPI bus protocol compatible, full duplex serial port. This interface is present only when one or both Error Classification and Correction by Replace are enabled. If External Interface is implemented, an external SPI flash is required.

3.2.6.2 Generating and Packaging the System-Level Design Example

Few steps are needed to correctly implement the SEM Controller Example Design. First of all the SEM Controller IP must be located and added to a blank Vivado project (figure 3.10):

1. Under Flow Navigator, click **IP Catalog**.
2. In the IP Catalog, look for **FPGA Features and Design > Soft Error Mitigation**.
3. Double click on Soft Error Mitigation IP.
4. The **Customize IP** window will appear. It is now possible to customize and configure the SEM Controller. Set the SEM Controller parameters as desired and then click ok.
5. The SEM Controller IP is now present under the **Design Sources** folder.

Then, to generate all the source files of the example design (figure 3.11):

1. Go to **Sources > Design Sources** and Right-click on the SEM Controller IP generated before.
2. In the drop-down menu click on **Open IP Example Design**.
3. Follow the instructions. At the end of the procedure, a new Vivado Project with the Example Design is open.

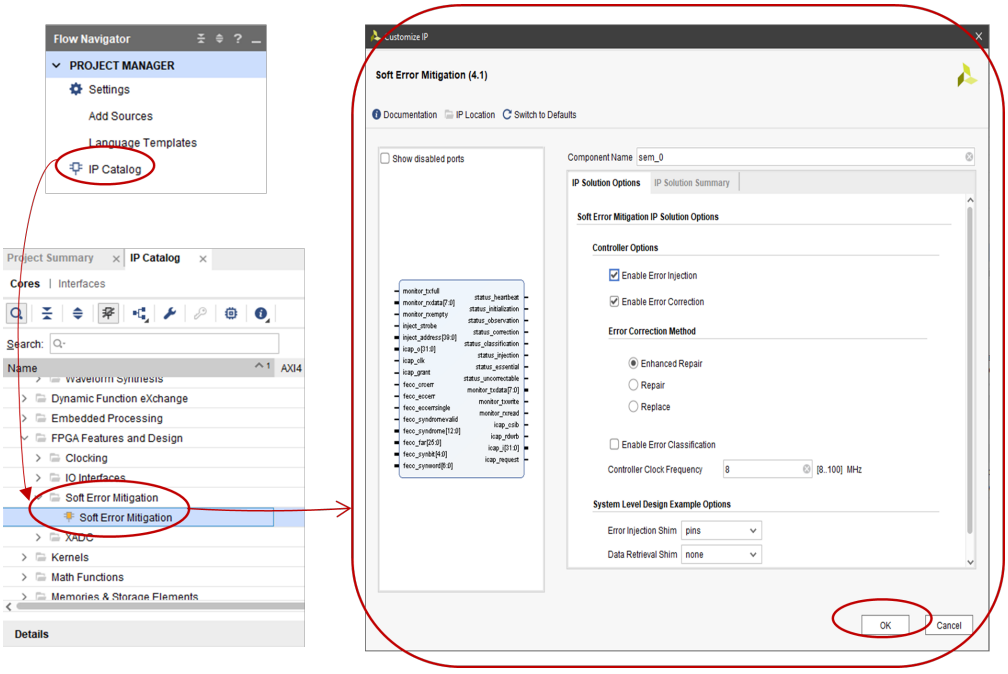

Figure 3.10: Steps to add SEM Controller to a Vivado project

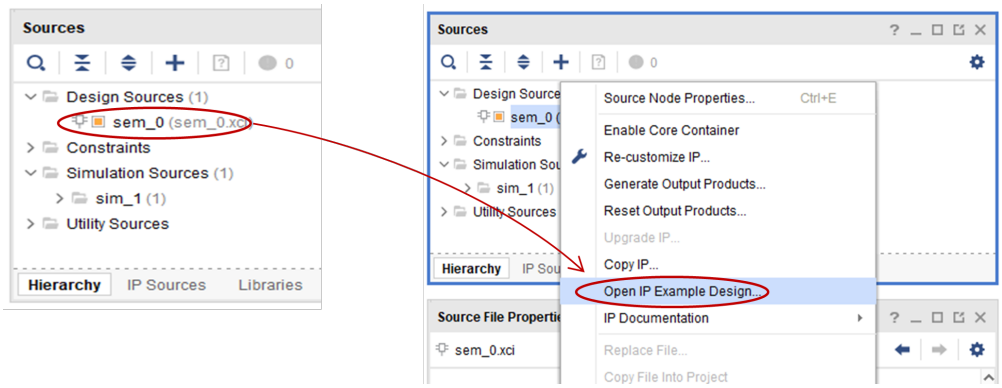

Figure 3.11: Steps to generate the SEM Controller Example Design in Vivado

Now all the requested files for the system-level design are available. Since for this thesis Error Classification and Error Correction by Replace are both disabled, the EXT shim is not needed, and its files are not generated. Files can be generated in VHDL or Verilog depending on the target language set in Vivado.
A simulation source is also provided by the example design.

In Vivado, FPGA designs are usually made through Block Designs filled by different IPs. For that reason, the SEM Controller Example Design just generated has been packaged and a working IP that can be added to the IP Catalog and used when desired has been created.
In Vivado, with the Example Design project open, go to **Tools > Create and Package New IP** and follow the instructions. At the end, the SEM IP is saved in a repository defined by the user and finally ready to be used in Vivado projects. When added to a Vivado Block Design, it should look like the following:

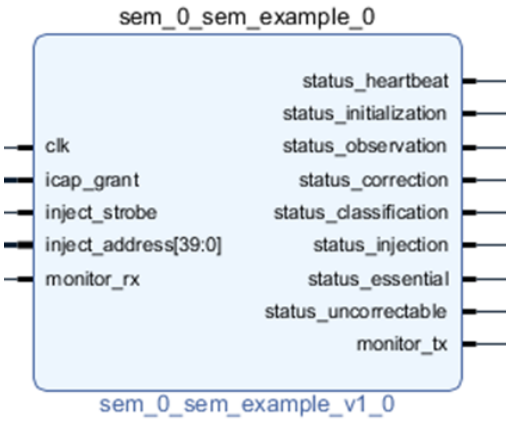

Figure 3.12: Working SEM IP in Vivado IP Integrator

In this thesis, the SEM Controller has to be driven by the Processing Element, therefore two others IP have been implemented, one to manage the *icap_grant* signal and another one to connect the SEM IP to the PS.

3.2.6.3 Managing icap_grant Signal

As anticipated, *icap_grant* is a very critic signal since it tells if SEM Controller can access or not to the FPGA Configuration Memory.

In Zynq-7000 devices, during boot of the Processing System, the access to the configuration logic is given to the PS through the Processor Configuration Access Port (PCAP) and the ICAP is locked [29]. The Controller has no simple method to sense the ICAP is locked out and during initialization state it simply polls the ICAP attempting to read the IDCODE register, until the expected identification

value is observed. However, to switch from PCAP to ICAP the PCAP_PR bit (bit 27 of the DEVCFG CTRL register, address 0xF8007000) must be cleared by the PS and if this operation is performed during a SEM controller attempt to access ICAP, the configuration logic might receive a malformed ICAP transaction, leading to unpredictable behaviour of the configuration logic. To avoid this possibility, icap_grant signal must be held inactive (low) until the PS has completed all necessary PCAP activity. In this way the Controller is prevented from entering the Initialization state and polling the ICAP. The PS must drive the icap_grant through the GPIO. The GPIO used might either be and EMIO from the PS or a GPIO in the Programmable Logic (PL), as in figure 3.14, but it must be initialized so that the Controller observes icap_grant deasserted immediately upon completion of PL configuration.

In the figure 3.14, a possible solution to drive the icap _grant is provided, derived from here [30]. To be sure the signal will be deasserted right after the PL configuration, the AXI GPIO default output value must be set to 0.

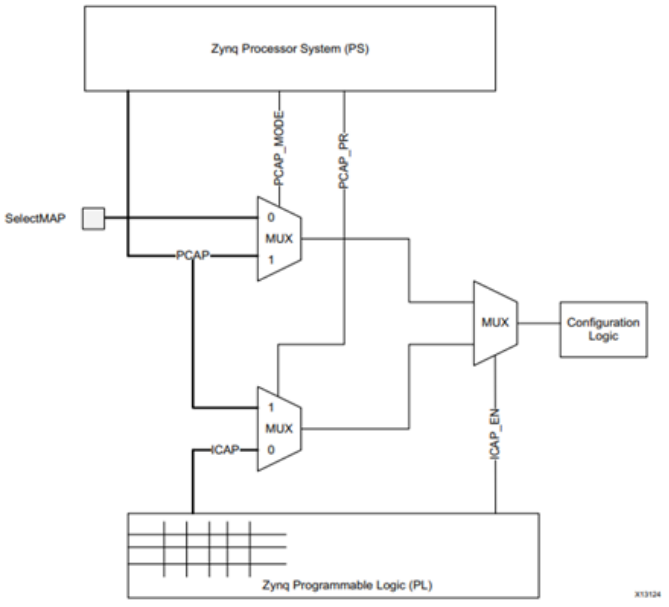

Figure 3.13: Configuration Logic Access in Zynq-7000 [29]

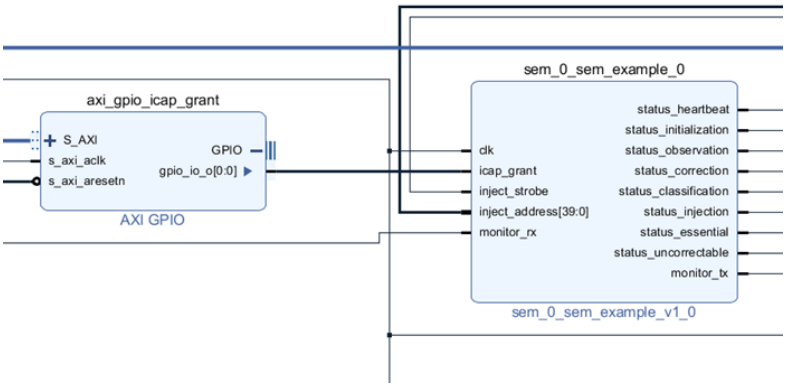

Figure 3.14: SEM IP with AXI GPIO to Drive icap_grant signal

3.2.6.4 Connecting SEM IP to PS

When Error Injection is enabled, the Error Injection Interface, composed by the inject_strobe signal and the inject_address bus, is present. In this situation, SEM Controller can receive commands directly from the PS. However, since the Error Injection Interface does not implement a standard communication protocol, an additional Custom IP is needed. In figure 3.15 a possible solution is provided, derived from here [30].

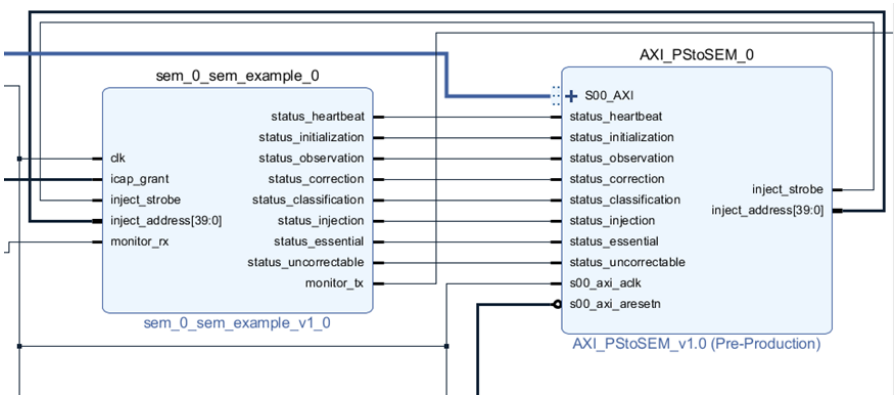

Figure 3.15: SEM IP and AXI_PStoSEM

The AXI_PStoSEM IP, is a very simple custom IP developed to interface the PS to the SEM IP. It implements a Slave AXI4-Lite interface, with four memory mapped 32-bit registers, structured as follow:

Register	Address	Operation	Description
slv_reg0[31:0]	Base Address+0x0	Write Only	Used to implement inject_address [31:0] output bus.
slv_reg1[31:0]	Base Address+0x4	Write Only	The least significant 8 bits of slv_reg1 are used to implement inject_address [39:32] output bus.
slv_reg2[31:0]	Base Address+0x8	Write Only	The LSB of slv_reg2 is used to implement inject_strobe signal.
slv_reg3[31:0]	Base Address+0xC	Read Only	slv_reg3 gives the possibility to the PS to read back the SEM IP status signals.

Table 3.1: AXI_PStoSEM Memory Mapped Registers

Additional information about slv_reg3 organization are necessary:

- It is divided into 8 nibbles, one per each status signal (*status_heartbeat* included).
- The LSB of each nibble stores the value of its status signal, obtaining the structure reported in table 3.2.

This structure gives the possibility to work with human-readable hexadecimal values in the PS source code. Indeed, if the PS read the slv_reg3 register while the SEM IP is in observation state, the read value is equal to 0x00000100. In this way, it is easy to understand which status signal is asserted.

Nibble	Slave Register Bits	Status Signal	Status Signal Bit
0	slv_reg3[3:0]	*status_heartbeat*	slv_reg3(0)
1	slv_reg3[7:4]	*status_initialization*	slv_reg3(4)
2	slv_reg3[11:8]	*status_observation*	slv_reg3(8)

3	slv_reg3[15:12]	*status_correction*	slv_reg3(12)
4	slv_reg3[19:16]	*status_classification*	slv_reg3(16)
5	slv_reg3[23:20]	*status_injection*	slv_reg3(20)
6	slv_reg3[27:24]	*status_essential*	slv_reg3(24)
7	slv_reg3[31:28]	*status_uncorrectable*	slv_reg3(28)

Table 3.2: AXI_PStoSEM slv_reg3 structure

3.2.6.5 SEM Controller States

To operate with the SEM Controller it is necessary to drive correctly its state machine 3.16. Depending on configuration and on commands received, SEM controller moves through seven states as shown in figure

- **Initialization** At the beginning, the controller is inactive due to FPGA global set/reset signal. After FPGA configuration, global reset signal is deasserted and the controller boots. In this state all the five status signals are inactive. During boot, the controller polls its *icap_grant* input to determine if it can access the ICAP. After ICAP access is granted, the controller computes the FPGA configuration memory readback reference codes, to later be able to detect and correct CRAM upset. After that, it moves to observation state and it will never enter initialization state again. If a new initialization process is needed (e.g., configuration memory is changed and new reference codes must be computed), a reset command must be sent to the controller.

- **Observation** This is the main controller state. While in observation, the controller scan continuously the configuration memory to detect errors. If an error is detected, the controller moves to correction state (even if error correction is disabled). While in observation, the controller can move to idle state or generate a status report if the respective command is received.

- **Correction** In this state the controller tries to correct the errors detected in the observation state. If the error is CRC-only, the *status_uncorrectable* signal

is asserted and a report to the Monitor Interface is generated. If the error is not a CRC-only error, then the correction attempt depends on the correction method chosen during IP Customization. When the correction procedures are completed, the controller moves to Classification state

- **Classification** In this state errors are classified. All uncorrectable errors are classified as essential. In this case, *status_essential* signal is asserted, a report is generated on the Monitor Interface and the controller moves to Idle state. After an uncorrectable error has occurred, the SEM IP stop scanning for errors. The FPGA must be reconfigured.
- **Idle** In this state the controller does not scan for errors. Error injection command and software reset command are supported only in this state. The "enter observation" command can be sent to transit to observation state. Status report command is also supported. The SEM Controller must be held in Idle when access to ICAP is not granted (e.g., other resources need to access ICAP).
- **Injection** If in Idle state a valid injection command is received, the controller moves to Injection state. Here it performs the error injection through a read-modify-write process to invert one configuration bit at an address specified in the injection command. When injection is complete, controller returns to Idle state. If multi-bit errors are desired, multiple injection must be performed, one after another, before entering Observation state.
- **Fatal Error** The controller enters Fatal Error state when it detects an internal inconsistency. Controller can be halted due to soft errors that affect the SEM IP configuration memory bits. Fatal Error state is indicated by the assertion of all five status signals.

In this thesis the SEM Controller is exploited to perform error injection but also error correction, so almost all states are crossed during the experiments.

3.2.6.6 SEM Controller Commands

Thanks to the AXI_PStoSEM IP developed for this thesis, the PS can drive the SEM Controller through the Error Injection interface.

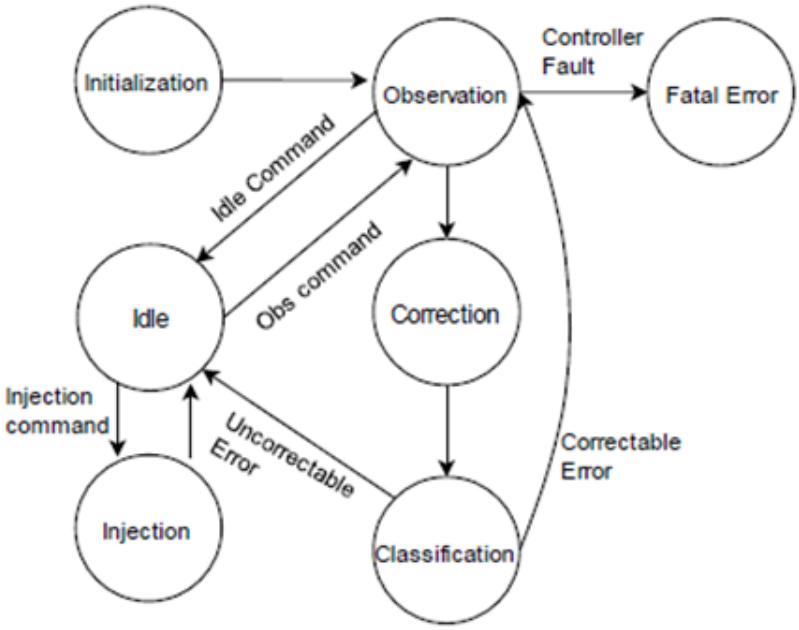

Figure 3.16: SEM Controller States Diagram

The controller can be moved between observation and idle states by a directed state change. The commands format is shown in the figures below:

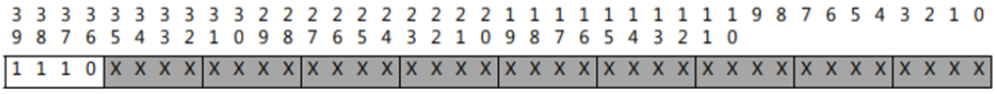

Figure 3.17: Enter Idle Command [29]

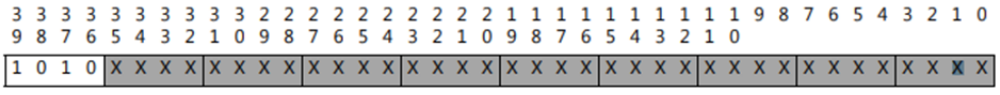

Figure 3.18: Enter Observation Command [29]

As can be noticed, the commands above are coded only by the four most significant bit of the inject_address bus.

The SEM IP, after Initialization state, moves to Observation state. Then, it is supposed to move to Idle state before sending an error injection command.

Remembering how the AXI_PStoSEM slave registers are structured, the code snippet below, extrapolated from the C application developed for the PS, shows a way to perform this state change.

Listing 3.1: PS code to send Enter Idle command

```
1   //Putting SEM in IDLE
2   semStatus=(Xil_In32(SEM_ADDR+0x0C)&0x00111110);
3   if(semStatus==0x00000100){
4       //Sending Idle Command
5       Xil_Out32(SEM_ADDR+0x04, IDLE);
6       Xil_Out32(SEM_ADDR, 0);
7       //Asserting Strobe
8       Xil_Out32(SEM_ADDR+0x08, 0x01);
9       Xil_Out32(SEM_ADDR+0x08, 0x00);
10      do{
11          //do nothing
12          usleep(1000);
13          semStatus=(Xil_In32(SEM_ADDR+0x0C)&0x00111110);
14      }while(semStatus!=0x00000000);
15  }
```

semStatus is a variable used to store the status signals read value, while *SEM_ADDR* is a macro constant equal to the AXI_PStoSEM base address.

SEM_ADDR+0x04 is the slv_reg1, whose first byte corresponds to *inject_address[39:32]*. For this reason, the *IDLE* macro constant is equal to 0xE0 (i.e., 0b1110_0000) as requested from the Enter Idle command format.
Since all the other bits are not used for this command, the write operation to slv_reg0 is unnecessary.

inject_address bus is now valid and the *inject_strobe* signal must be asserted (for one ICAP clock cycle is enough) to inform the SEM Controller that *inject_address* bus must be captured.

After asserting and deasserting the strobe, a do/while loop is implemented to wait for the SEM Controller being in Idle before moving on with the algorithm.
As it can be noticed, every time the status signals are read, a bitwise operation is performed. It implements a bit mask to discard *status_heartbeat*, *status_essential* and *status_uncorrectable* signals, not useful in this situation. Only the signals indicating the Controller state are desired.

While the SEM Controller is in Idle, the Error Injection command is supported. The command can be sent in the same way as for the enter idle command. Depending on the address scheme used, two different formats are possible:

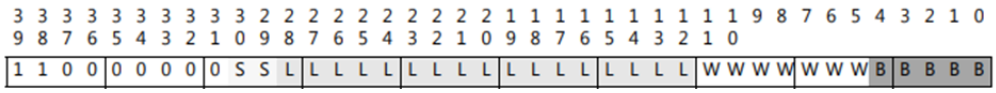

Figure 3.19: Error Injection Command (Linear Frame Address)

Where:

- SS = Hardware SLR number for SSI (2-bit) and set to 00 for non-SSI
- LLLLLLLLLLLLLLLLL = linear frame address (17-bit) [0..Maximum Frame]
- WWWWWWW = word address (7-bit) [0..100]
- BBBBB = bit address (5-bit) [0..31]

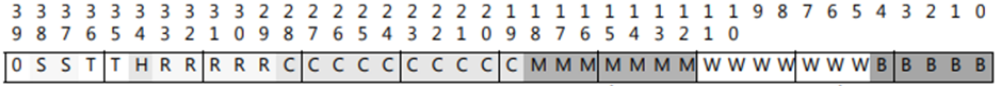

Figure 3.20: Error Injection Command (Physical Frame Address)

Where:

- SS = Hardware SLR number for SSI (2-bit) and set to 00 for non-SSI
- TT = block type (2-bit)
- H = half address (1-bit)
- RRRRR = row address (5-bit)
- CCCCCCCCCC = column address (10-bit)
- MMMMMMM = minor address (7-bit)
- WWWWWWW = word address (7-bit) [0..100]

- BBBBB = bit address (5-bit) [0..31]

For this thesis, the Linear Frame Address (LFA) address scheme has been used. When the injection address has been calculated the PS can drive an error injection thanks to the following code:

Listing 3.2: PS code to send Error Injection command

```
1     scanf(" %d", &lfa);
2     //Sending Injection Command
3     Xil_Out32(SEM_ADDR+0x04, INJECTION);
4     Xil_Out32(SEM_ADDR, lfa);
5     //Asserting Strobe
6     Xil_Out32(SEM_ADDR+0x08, 0x01);
7     Xil_Out32(SEM_ADDR+0x08, 0x00);
8     do{
9         //do nothing
10        usleep(1000);
11        semStatus=(Xil_In32(SEM_ADDR+0x0C)&0x00111110);
12    }while(semStatus!=0x00000000);
```

Through *scanf()* function the PS receives the complete LFA from the host PC. After that, injection is performed (SEM Controller is already in Idle state). When an error injection command is received, SEM Controller moves to Injection state. After completion, returns to Idle state, ready to receive new commands. A do/while loop is implemented to wait for the SEM IP to be in Idle again.

3.3 Experiment Methodology

In this section, the methodology exploited for the reliability analysis and SEU effects mitigation is discussed.

At this point it should be clear that the design under test are two: the convolutional single neuron and the ZFNet Input Layers. For this reason two different reliability analysis have been performed. However, the strategy behind them it is the same for both design and it can be summarized through the following steps:

1 Design and Synthesis

2 Placement and Implementation

3 Bitstream Generation

4 Application Development for PS

5 Error Injection Campaign

6 Results Analysis and Re-design

7 Repetition of the steps from 1 to 6 until the results satisfy reliability requirements and other desired specs

3.3.1 Design and Synthesis

In this step the RTL introduced in the Experiment Setup section [3.2.2.2] is developed. Thanks to the Vivado IP Integrator, custom IPs and Xilinx IPs are connected and the entire architecture is described as a block design (figure 3.21).

As can be noticed, in addition to the blocks already illustrated (Checker, SEM IP, PS, Input Generator, DUT and GOLDEN) other blocks are present:

- **AXI Interconnect**: IP automatically generated by Vivado when AXI interfaces are present in the design and must be managed [24]
- **Processor System Reset**: IP automatically generated by Vivado when resets must be managed. It provides customized resets for an entire processor system [31].

- **AXI GPIOs**: Xilinx IPs inserted to exchange data and manage control signals present in the design [32]. Thanks to their memory mapped interface, they can be easily read and written through the PS application developed through Vitis.

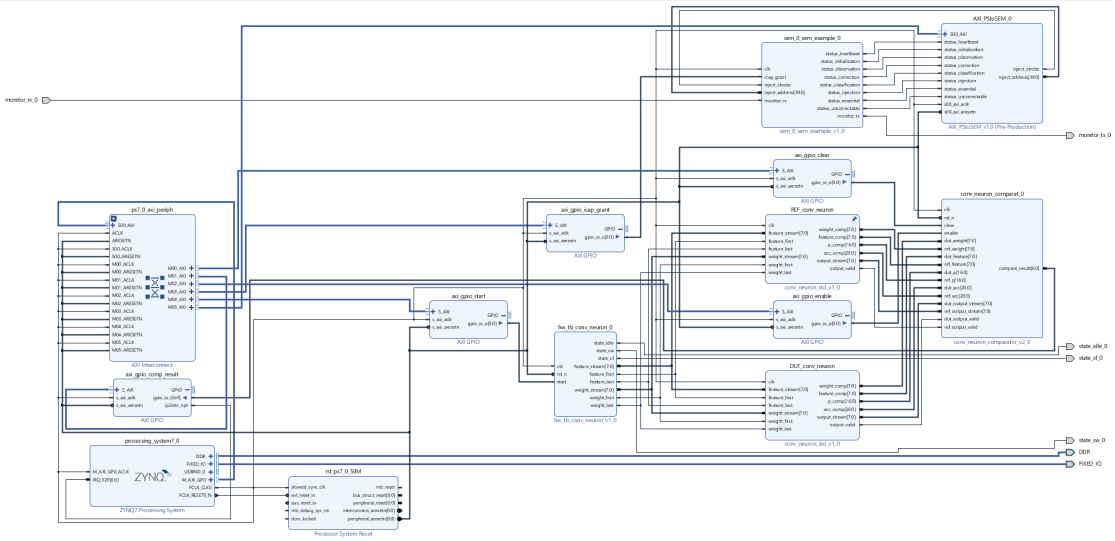

Figure 3.21: Block Design for Convolutional Single Neuron Reliability Analysis

After the design phase, the RTL is synthesized by Vivado. If no errors occur during this process, placement and implementation can be performed.

3.3.2 Placement and Implementation

In this step all the blocks present in the design are physically allocated inside the FPGA and than implemented.

This is a fundamental step to carry out the reliability analysis in the fastest and most accurate way. Thanks to SEM IP ([3.2.6]) it is possible to inject an error at a specific location in the CRAM that corresponds to a particular region of the FPGA. For the analysis in this thesis, it is necessary that the injected errors only affect the DUT. Therefore, it should be physically separated from the rest of the design to minimize the likelihood of injecting faults into CRAM locations that do not correspond to the DUT.

This can be done with the Vivado software which grants the possibility to define the so called *"Pblocks"* [33].

Each Pblock is a physical block that can contain logic modules and cells from anywhere in the design hierarchy. For this thesis, two Pblocks have been defined to conduct the reliability analysis:

- *pblock_DUT_conv_neuron*, which contains the DUT
- *pblock_ctrl_circuit*, which contains all the other components of the system

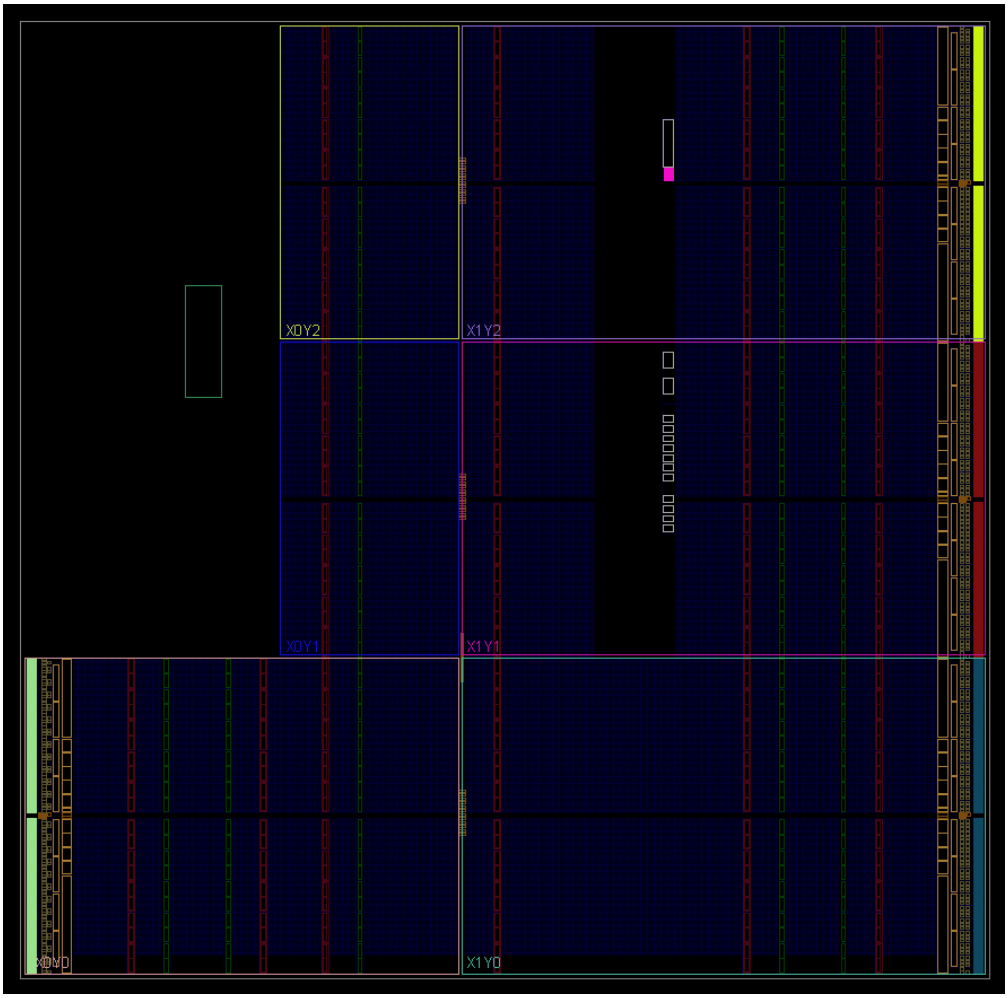

Figure 3.22: Zynq 7020 in Device Window

In figure 3.22, you can observe the Zynq 7020 SoC through the Device Window of Vivado, which is a graphical interface primarily used for floorplanning [25]. It also displays all the resources of the device, such as device logic, clock regions, I/O pads, cells location and net connectivity.

In the image, six rectangular regions are visible, labeled as $XnYn$, which are characterized by having their own clock resources to ensure zero skew clock across the device. These clock regions can also be identified in the bitstream, making them a vital resource for implementing a targeted fault injection campaign. For this reason, the two Pblocks have been placed in different regions.

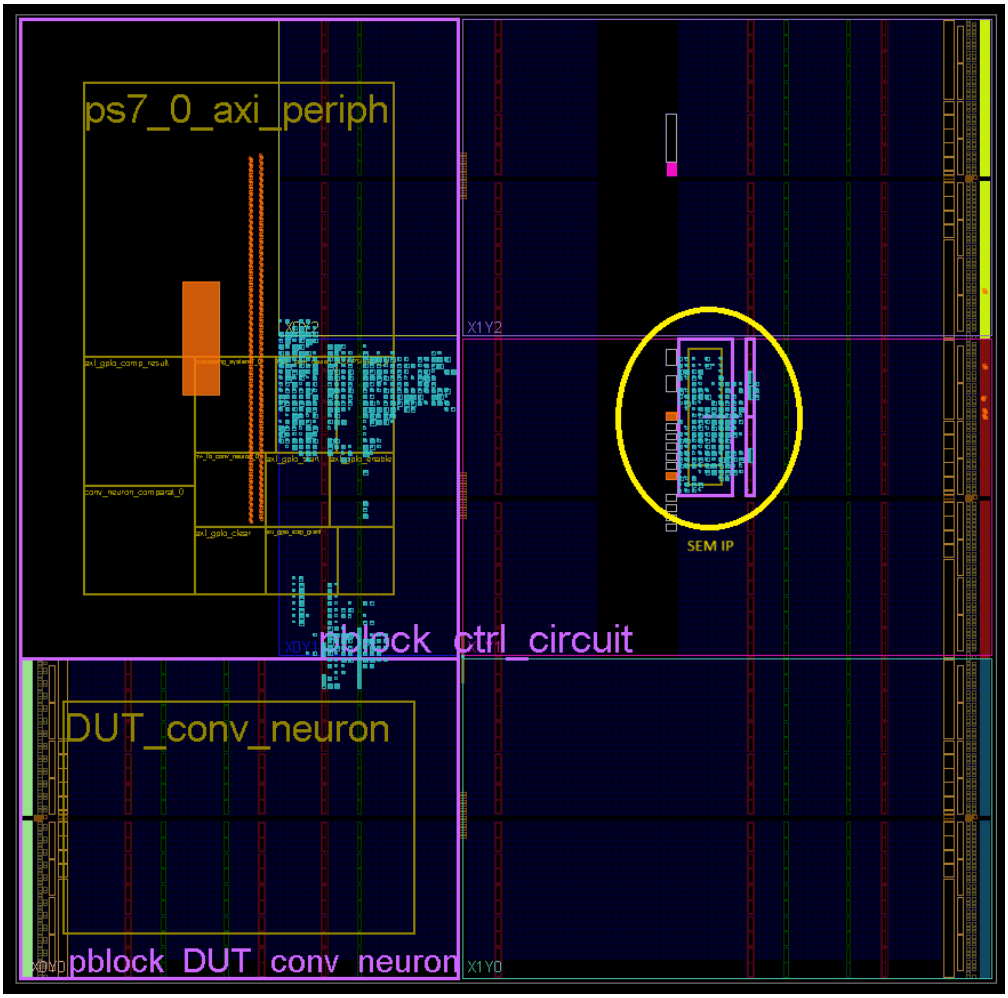

Figure 3.23: Zynq 7020 in Device Window with Pblocks

The *pblock_DUT_conv_neuron* is allocated in the X0Y0 region, while the *pblock_ctrl_circuit* occupies the X0Y1 and X0Y2 regions, ensuring that all the necessary resources are available. In the X1Y1 region is visible a third Pblock which is automatically generated by Vivado whenever the SEM IP is instantiated.

To further reduce the risk of injecting errors into locations other than those of the DUT, two properties of the Pblocks have been used [33]:

- CONTAIN_ROUTING: to prevent signals inside the Pblock from being routed outside the Pblock
- EXCLUDE_PLACEMENT: to enforce exclusive resource usage within the Pblock's defined area for its assigned logic

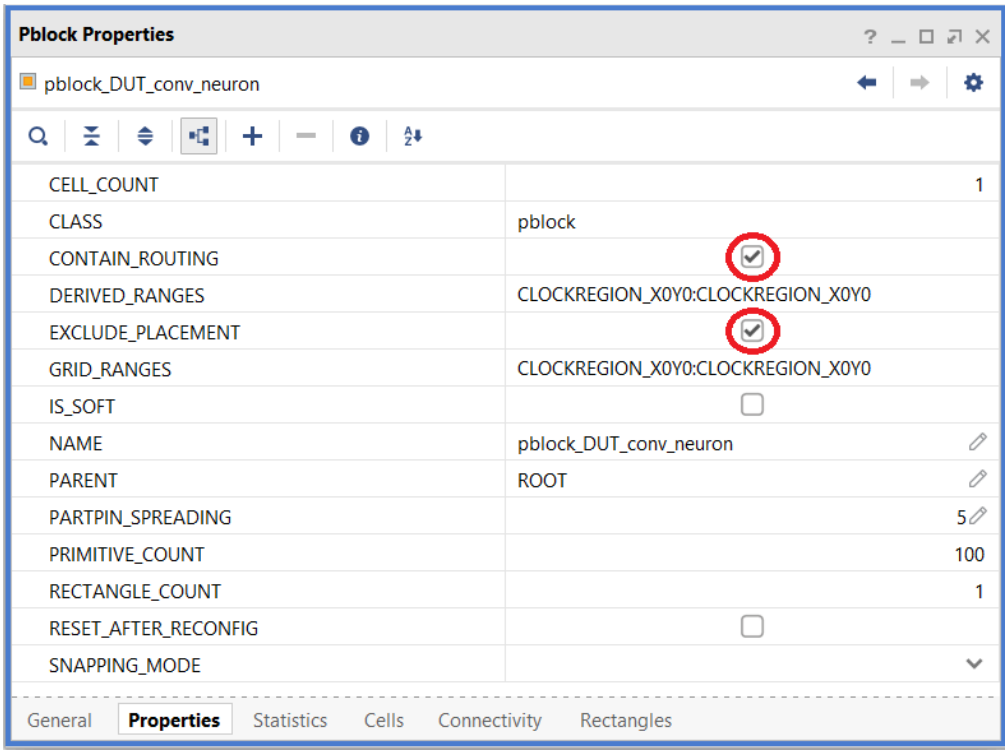

Figure 3.24: Pblock Properties Tab in Vivado

At this point, the implementation can be run. If no violations are reported, the design is ready to be implemented on the FPGA.

3.3.3 Bitstream Generation

The bitstream file contains all the information about place and route and it is used to configure the PL. During the bitstream generation other useful files can be generated by Vivado.
For this thesis, the additional files needed to properly use the SEM IP and run the reliability analysis are:

- **Raw Bit File (.rbt)**: it contains the same information of the binary bitstream file (.bit) but in ASCII format [34].
- **EBC File (.ebc)**: ASCII text file used as a reference model. It contains the same content seen by the SEM controller during the SEU readback of the FPGA Configuration Memory. However, it is important to note that this file does not contain all the data used to program the device. Indeed, SEM IP cannot scan Block Memory and so no BRAM contents frames are present in the EBC file (they are in the .rbt file). If Correction by Replace is enabled, this file is used by the controller to correct corrupted frames.
- **EBD File (.ebd)**: ASCII text file that shows which bits of the SEU readback are marked as essential. The EBD file is used to mask the EBC one: a '1' in the EBD file corresponds to an essential bit in the EBC file and so in the FPGA Configuration Memory. If Error Classification is enabled, this file is used to classify a bit error as critical or not.

Essential Bits are those bits that have an association with the circuitry of the design. So, if an essential bit changes, the design circuity changes, and the function of the design may fail.
EBD file is also used to perform error injection campaigns [35]. Indeed, since only a small percentage of the CRAM bits are essentials, it would be a waste of time injecting errors to bits that will not never lead to a function failure. From EBD file is possible to compute the Linear Frame Address (LFA), used in the SEM IP injection command (see section 3.3.5).

To generate the essential bits files, a bitstream property must be specified [36] in the constraints file (.xdc) of the Vivado project as follow:

set_property bitstream.seu.essentialbits yes [current_design].

Note that this property is valid only if SEM IP is instantiated. When set, in the runme.log file it is reported how many essential bits are presents in the design (figure 3.25). The runme.log and all the bitstream files can be found in implementation folder of the Vivado Project.

```
Writing bitstream ./BD_16bitRe_compare_wrapper.ebc...
Creating essential bits data...
This design has 350527 essential bits out of 25697632 total (1.36%).
Creating bitstream...
Writing bitstream ./BD_16bitRe_compare_wrapper.ebd...
```

Figure 3.25: Essential Bits in runme.log file

Now the hardware design is ready to be exported into the Xilinx Support Archive (XSA) file, a proprietary format, enabling its use with the Vitis platform for application development.

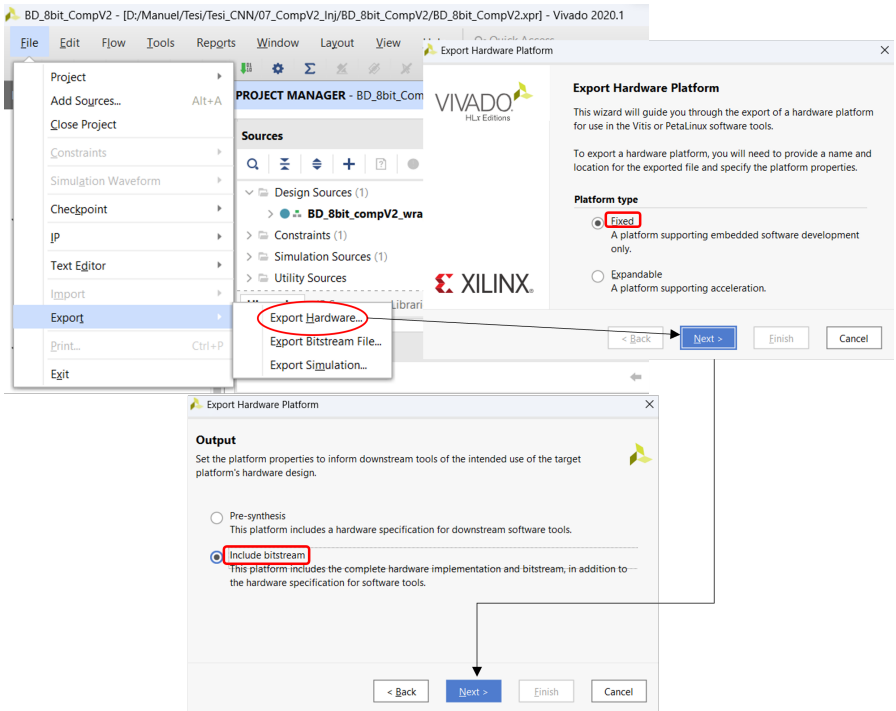

Figure 3.26: Generating XSA File in Vivado

3.3.4 Application Development for PS

At this point, the XSA file is used by Vitis configure its environment to develop the application. The following simple steps must be followed:

- From the main screen of Xilinx Vitis, click on **File** > **New** > **Application Project**.
- The window shown in Figure 3.27 opens. Select the *"Create a new platform from hardware (XSA)* tab, then click **Browse** to choose the newly generated XSA file.
- In the next screen, choose a name for the project, then click **Next** twice and finally press **Finish**.

Now, the Vitis environment is ready and the application can be developed

Thanks to the example projects that that can be selected during configuration phase (highly recommended), the access to practically all the C libraries necessary for the proper functioning of the device is granted.

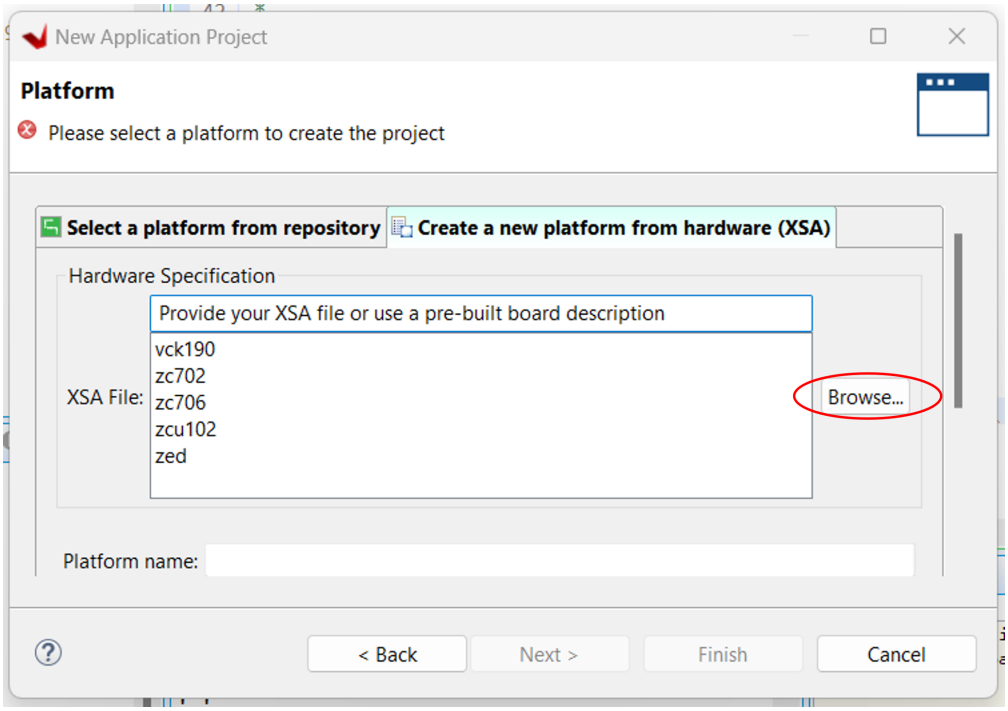

Figure 3.27: Importing XSA in Vitis

3.3.5 Error Injection Campaign

Automation is a key factor in simplifying and accelerating the reliability analysis of our DUT. As introduced in section [3.2.3], our setup involves the entire test being carried out by the host PC (Master), which interfaces with the PS (Slave) of our FPGA. This section describes the flowchart of the two applications developed for this Thesis, effectively illustrating how each fault injection campaign was performed.

3.3.5.1 Overview

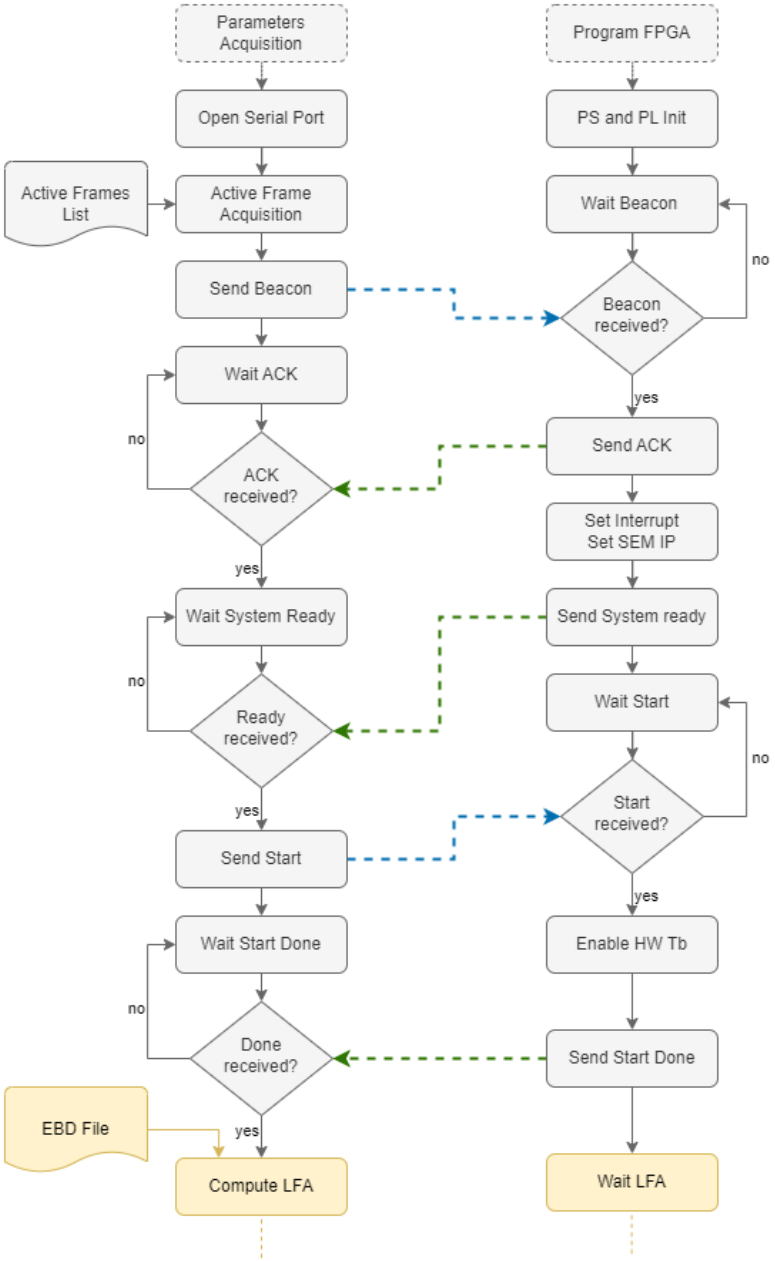

Figure 3.28: Host PC (on the left) and PS (on the right) Flowchart, Configuration Phase

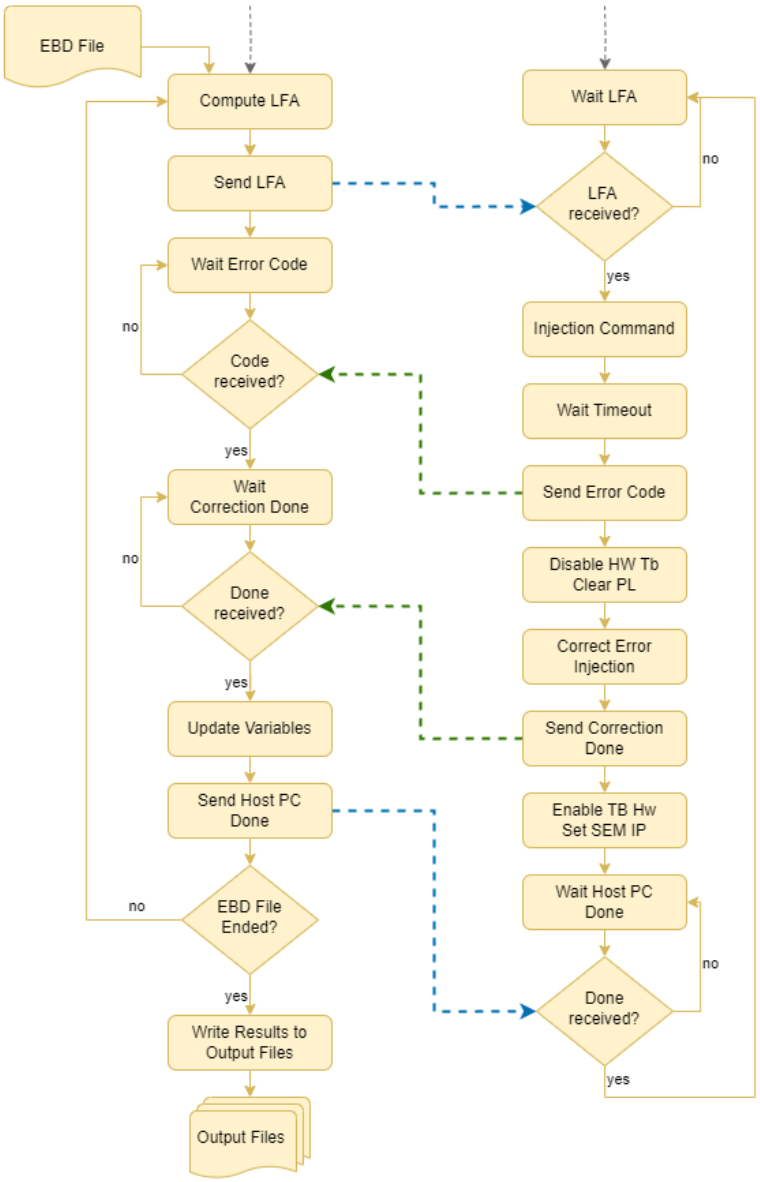

Figure 3.29: Host PC (on the left) and PS (on the right) Flowchart, Injection Phase

In figures 3.28 and 3.29, you can observe the flowchart of the two applications, with the host PC on the left and the PS on the right. As easily observable, we opted for a handshake-like approach, where operations are synchronized through the exchange of "go" and "done" signals between the two programs. This approach was chosen for its simplicity of implementation. Indeed, by leveraging blocking

functions like *scanf()* and *getchar()*, it is possible to synchronize the entire process correctly.
Data exchange occurs through serial communication, using the Micro-USB port of the TUL PYNQ-Z2 evaluation board, connected to one of the two available UARTs of the PS.

The entire flow can be divided into two main phases::

- **Configuration Phase:** in this phase, simple configuration operations for the host PC and initialization for both the PS and PL take place.

- **Injection Phase:** it can be considered the core of this Thesis. In this phase, thanks to the SEM IP, errors are injected into the CRAM for all essential bits of the DUT, emulating every possible SEU. Meanwhile, the Checker compares the signals of the DUT and the Golden Model, and in case of a mismatch, the corresponding error code is sent to the PC, saved, and ultimately reported to the results files.

Following is a detailed description of the two flows.

3.3.5.2 Host PC Flowchart

The host PC is responsible for orchestrating the entire algorithm, generating addresses for fault injection to be sent to the SEM IP, and saving the obtained results. In this section, we will detail all the operations carried out by the Python script running on PC.

Parameters Acquisition: Upon launching the script, it prompts the user to enter some general parameters via the terminal, such as the location of the EBD file, serial port data, and the names of the output files.

Open Serial Port: As can be inferred, in this step the port for communication is opened. For this operation to succeed, the FPGA must already be configured and connected to the PC.

Active Frames Acquisition: In 7 Series and Zynq-7000 devices, CRAM is composed by frames and each frame contains 101 words of 32 bits each. An Active Frame refers to a memory frame where at least one bit is equal to "1", indicating that a portion of the RTL has been configured in that point.The developed methodology involves injecting errors only into the DUT to minimize analysis time. As mentioned earlier [3.3.2], the DUT is confined to Clock Region X0Y0, which corresponds to several frames in the CRAM. Knowing which of these frames are actually used is therefore crucial.

By comparing the EBD file with the Raw Bitstream File, this information can be derived. A second small Python script performs this analysis and saves the active frames of interest into a file, which is then acquired in this step by the Host PC application.

Send Beacon and Wait ACK: This marks the first handshake of the entire process. The PS is waiting for the PC, which sends a signal to proceed and waits for the response of the device.

Wait System Ready: After the first handshake, the PC waits for the PS to have configured and set up the entire PL correctly.

Send Start and Wait Done: With everything configured, the PC signals the PS to start the RTL, enabling the Input Generator IP, then waits for done signal.

Compute LFA: At this point, the RTL loaded onto the FPGA is active: the Input Generator stimulates both the DUT and Golden Model, and the Checker compares certain signals of them. Only the SEM IP is in Idle, waiting for the command to perform fault injection. This command is sent to the block by the PS, which needs to know which bit of the CRAM has to be flipped. The calculation of the address is left to the Host PC. The advantage of using the LFA is that it can be easily computed by parsing the EBD file.
The EBD file reflects Configuration Memory organization, and it is composed by an informational header followed by some number of lines and each line has 32

characters that can be either a '0' (non-essential) or a '1' (essential). The numerical lines represent the 32-bits words of a frame in the Configuration Memory. The LSB of a word is on the far right of the line. Here below the first lines of EBD file:

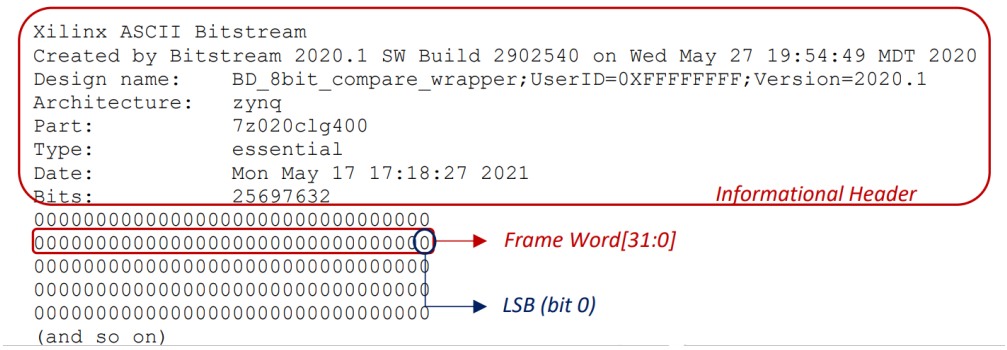

Figure 3.30: First Lines of the EBD File

So, in the EBD file, every group of 101 lines hold the essential bits data for a configuration frame. The first group is a pad frame, so not valid for error injection [37] As illustrated in section [3.2.6.6], the complete error injection address is composed by three part:

- **Linear Address (LA):** it corresponds to the frame number. The first frame after the initial pad one stands for LA=0. The next 101 words group is for LA=1 and so on. It ranges from 0 to MAX, where MAX depends on the device (see Product Guide for details [29]). Only the Active Frame on Clock Region X0Y0 are considered in the EBD file, the others are ignored.
- **Word Offset (WD):** it ranges from 0 to 100 and it represents a word inside a frame.
- **Bit Offset (BT):** it ranges from 0 to 31 and it represents a bit inside a word.

From the EBD file, supposing that line zero is the first data line and that word zero is the first line of each frame, LA and WD can be computed as:

- LA = line_number/101 (only the integer part).
- WD = line_number%101

When LA, WB and BT are computed, the injection command can be generated as illustrated in Figure 3.19.

Send LFA: The Injection Command in the LFA format is sent to the PS.

Wait Error Code: The PC waits for the result of the fault injection. As shown in 4.2.1 several output codes are generated by the Checker.

Wait Correction Done: In this step, the PC waits for the PS, through the SEM IP, to correct the just-inserted error and restore the entire PL to its initial conditions.

Update Variables and Send PC Done: The variables containing the results of the various injections are updated. A "done" signal is then sent to the PS to indicate that an injection cycle is complete, and a new one could start shortly.

Write Results: The steps from *Compute LFA* to *"Send Host PC Done"* are repeated until DUT Essential Bits are available in EBD file. Upon completion, the fault injection campaign stops, and the results are written to the output files, ready for analysis.

3.3.5.3 PS Flowchart

In this section, the flow followed by the PS is detailed.

Program FPGA and PS/PL Init: After configuring the FPGA and loading the application developed with Vitis, as described in [3.3.4], the code starts and goes through a system init phase where the PS and PL are configured correctly, using C-functions and parameters defined in the Xilinx Libraries. Once the initialization is complete, our code begins, which for now, involves keeping the *icap_grant* signal at zero (not granted) and keeping all blocks of the RTL under reset.
The FPGA must be connected to PC and configured before launching the Host application.

Wait Beacon and Send ACK: The PS is ready to proceed and waits for the PC. An ACK is sent back.

Set Interrupt and SEM IP: The interrupt system for the PS is initialized. The Error Code in updated via interrupt, to be sure nothing is missed out. Then, the PCAP_PR bit is cleared, and the *icap_grant* signal is asserted (granted), as illustrated in section [3.2.6.3]. The SEM IP is then moved to idle state, awaiting further instructions (code snippet 3.1).

Send Ready and Wait Start: The PS is now ready and awaits the start signal from the PC.

Enable HW Tb: Once the start signal is received, the PS enables the Checker and the Input Generator. The entire RTL is now active, with the DUT processing the samples received from the HW Tb. The Checker interrupt is enabled.

Send Start Done and Wait LFA: The PS informs the PC that it is ready to receive the address for error injection.

Inject Error: After receiving the LFA, the PS sends the Error Injection Command to the SEM IP, as shown in code snippet 3.2.

Wait Timeout and Send Error Code: Before checking the result of the injection, the PS waits for WAIT_TIME defined as 3000 clock cycles. After the timeout, two scenarios may have occurred:

- **No Error:** if the Checker has not detected any mismatch, it is assumed that the essential bit affected by the injection is not critical for the functionality of the DUT. In this case, no interrupt has occurred, and the error code sent to PC is zero.
- **Error Occurred:** if the Checker has detected a mismatch, it means that the injection has affected a critical bit. In this case, an interrupt has occurred, updating the error code variable with the corresponding value, which is then sent to the PC.

Listing 3.3: PS code to send Error Code to PC after Timeout

```
usleep(WAIT_TIME);
if(compInterruptStatus==0){
    //No Interrupt
    tmpValue=0;
    xil_printf("%u\r\n", tmpValue);
}
else if(compInterruptStatus==1){
    //Interrupt
    xil_printf("%u\r\n", errorCode);
}
```

Disable HW Tb and Clear PL: The injection can be considered complete, thus the Input Generator is disabled and the PL is cleared.

Correct Error Injection and Send Done: Before moving on to the next injection, it is necessary to restore the CRAM to its original state. To do this, the functionalities with which the SEM IP was configured for this Thesis are once again used.
The PS sends the "Enter Observation Command" (Figure 3.18) to the SEM Controller granting to the IP the capability to locate and correct the error in the CRAM. After correction, a done is sent to the Host PC.

Enable HW Tb and Set SEM IP: The RTL is re-enabled, and the Input Generator is started again.

Wait Host PC Done: After the PC has completed its processing for the just-concluded injection, it informs the PS to return to the "Wait LFA" state.

3.3.6 Results Analysis and Redesing

In this phase, the results obtained from the fault injection campaign are analyzed. The first test campaign is always performed on the original DUT, without improvements against SEUs. This approach allows identifying the most critical parts of the

block, enabling targeted interventions to minimize the potential resources usage, a critical aspect for FPGA.

After the first campaign, a new RTL is developed, where solutions such as TMR are applied to reduce the faults recorded in the just-concluded campaign. In Chapters 4 and 5, detailed analyses of the results are presented, along with explanations of the strategies used to counteract the faults.

3.3.7 Repetition of the steps from 1 to 6 until the results satisfy reliability requirements and other desired specs

After the redesign, all the steps seen so far must be repeated to carry out a new reliability analysis.

From the second campaign onwards, efforts are made to implement increasingly better solutions until achieving results that meet the specified reliability requirements for the circuit under examination.
Having multiple FPGAs or a FPGA with more hardware resources also allows multiple campaigns to be conducted in parallel, significantly reducing test time.

Chapter 4

Single Neuron

In this chapter, the RTL of the ZFNet Input Layer Convolutional Neuron under study in this Thesis, its mitigated version, and the results of the reliability analyses conducted will be detailed.

4.1 Block Description

A synthesizable version of the neuron under study is made available by Alpha Data through an open-source license. Among the versions of the neuron provided, the basic version without architectural optimizations has been selected. This neuron is used to implement the input layers of the ZFNet, also made available with synthesizable code by Alpha Data. RTL codes and Documentation are available on the Alpha Data website [6].

The Figure 4.1, taken from the CNN Library documentation, illustrates the neuron's architecture.

As observable, it is based on a common Multiply-Accumulate hardware with a ReLU (Rectifier Linear Unit) [38] on the output. The ReLU block implements an activation function defined as follow:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

This function can be disabled through a generic parameter in the VHDL code. Both versions have been analyzed in this Thesis.

The Input Stream, in our setup, is generated by the Input Generator and the data sent are the so called *features*.

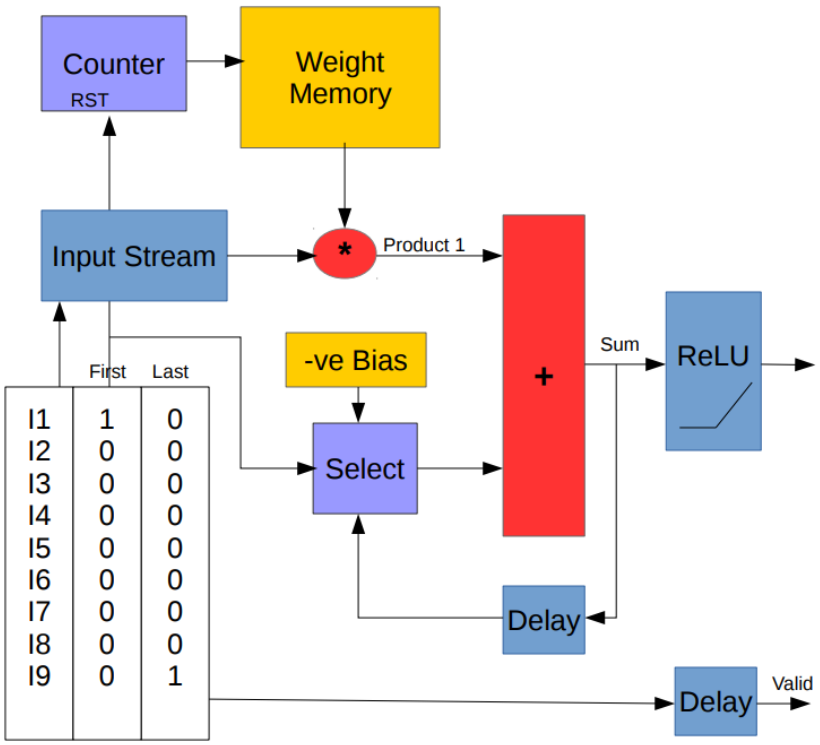

Figure 4.1: A possible ZFnet Convolutional Neuron architecture, by Alpha Data

4.2 Reliability Analysis

In this section, the results of different test campaigns are presented, and a possible architecture more tolerant to SEUs is also explained. Before that, the implemented monitoring system for the neuron is detailed to understand correctly the results illustrated.

4.2.1 Monitoring System for Single Neuron

As already extensively explained, during the fault injection campaign, the DUT and Golden Model are continuously compared by the Checker. This section illustrates the points in the neuron where checks are performed and how the Error Code is derived.

In the following figure, the same architecture from the previous section is shown with additional details. The internal pipeline registers are introduced and there are six called *"checkpoints"* marked by red crosses.

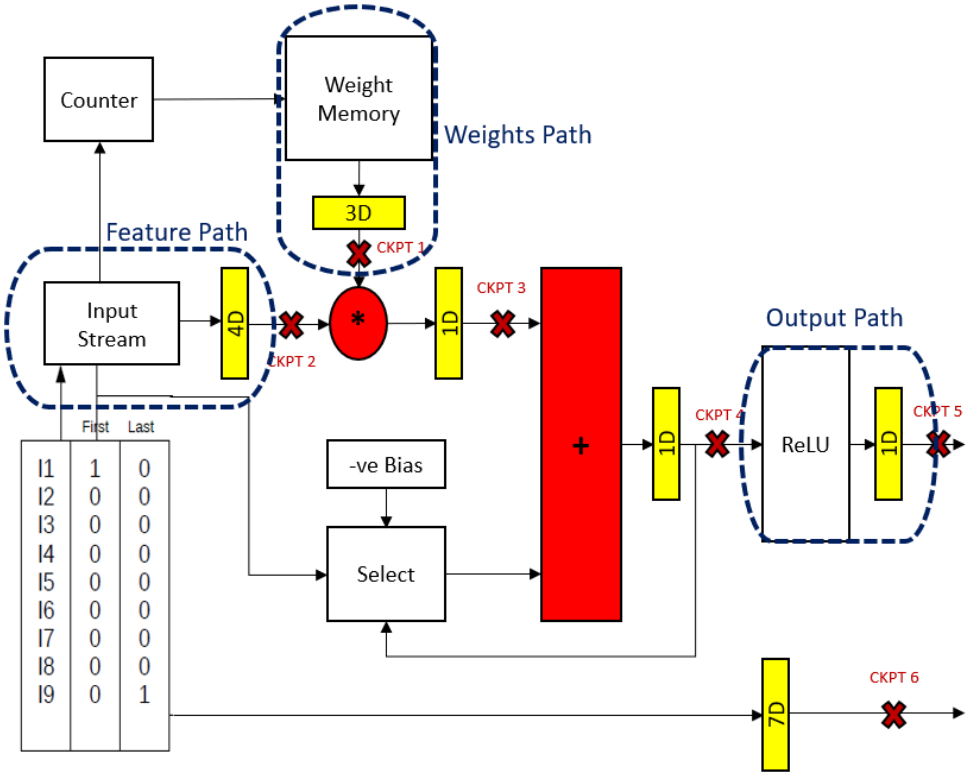

Figure 4.2: Neuron architecture with registers and checkpoints

The checkpoints, as can be inferred, are the signals from the DUT that are compared with the Golden Model. By monitoring internal signals as well, it is possible to understand more precisely where a mismatch has occurred and whether it has propagated to the output, allowing for a better characterization of the neuron.

In the following sections, to better interpret the collected results, the checkpoints are also referred to as follows:

- checkpoint 1 = Weight.
- checkpoint 2 = Feature.
- checkpoint 3 = Product.
- checkpoint 4 = Sum.
- checkpoint 5 = Output.
- checkpoint 6 = Output Valid.

The Checker, paying attention to the different internal delays due to registers, receives six signals from the DUT and six from the Golden Model, compares the corresponding pairs, and generates a 6-bit output signal, the so called Error Code, where each bit corresponds to the result of one of the six comparisons.

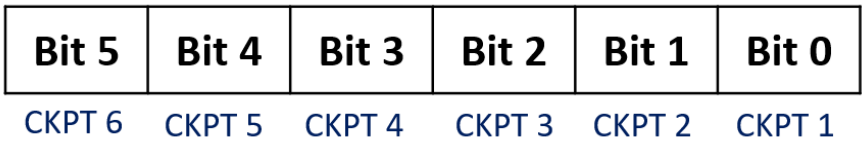

Figure 4.3: Error Code structure

As shown in Figure 4.3, the bit 0 of the Error Code contains the result of the comparison for checkpoint 1, bit 1 for checkpoint 2, and so on. If the single bit is equal to "1", then a mismatch has been detected for that checkpoint. It is a simple but effective code. For example, assuming an error code of "011101," it is immediately evident that the fault originated at the output of the weight memory and propagated to the output of the neuron.

4.2.2 Results of the first Fault Injection campaign

Following the methodology described in section 3.3, the tolerance to SEUs of the original version of the neuron was characterized in the first place. In this section the results of the analysis are illustrated.

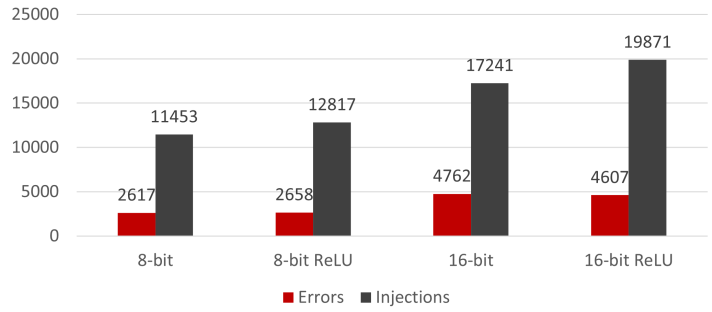

Figure 4.4: Total errors over total injection for Single Neuron

As visible in Figure 4.4, four different implementations of the neuron have been characterized:

- **8-bit:** implementation with 8-bit data size and ReLU disabled.
- **8-bit ReLU:** implementation with 8-bit data size and ReLU enabled.
- **16-bit:** implementation with 16-bit data size and ReLU disabled.
- **16-bit ReLU:** implementation with 16-bit data size and ReLU enabled.

The number of injections corresponds to the total number of essential bits for that version. As expected, a higher number of used resources correspond to an increase in essential bits.
The number of errors corresponds to all the mismatches detected by the Checker, including those in the internal checkpoints that did not propagate to the output.

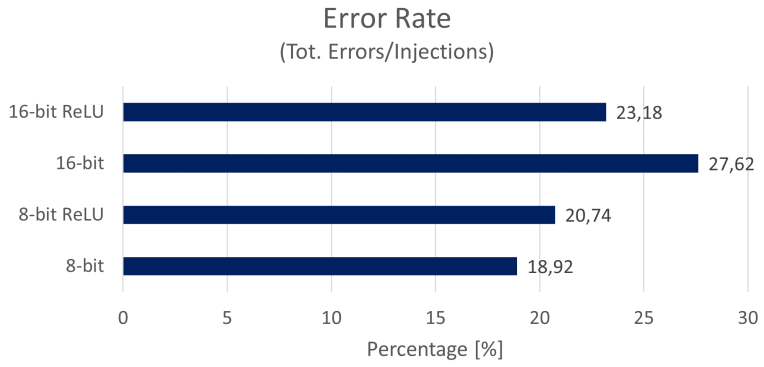

Figure 4.5: Error rate for Single Neuron

As observable from the error rate, even though the ReLU block implies a higher number of essential bits, it helps mitigate the effects of SEUs.

Examining the errors in detail, the following histograms are derived:

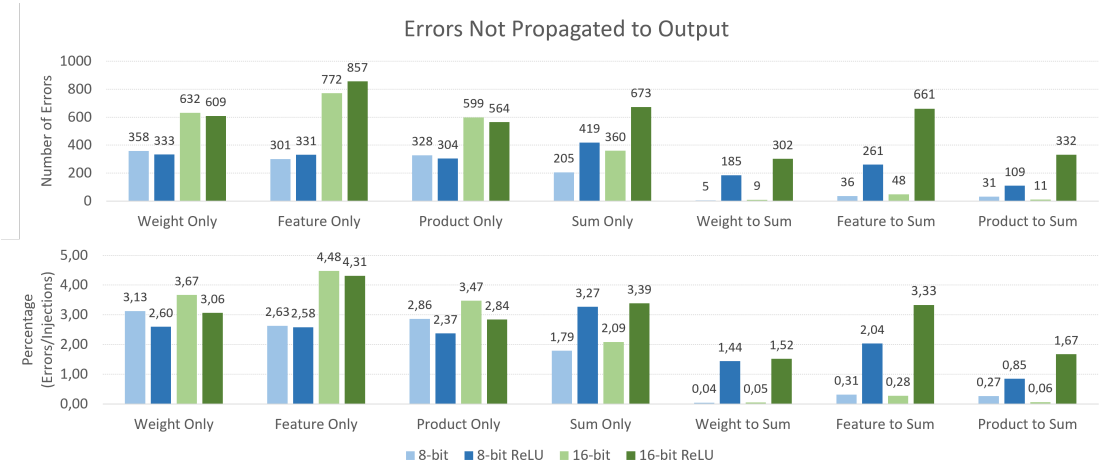

Figure 4.6: Errors not propagated to output - Single Neuron

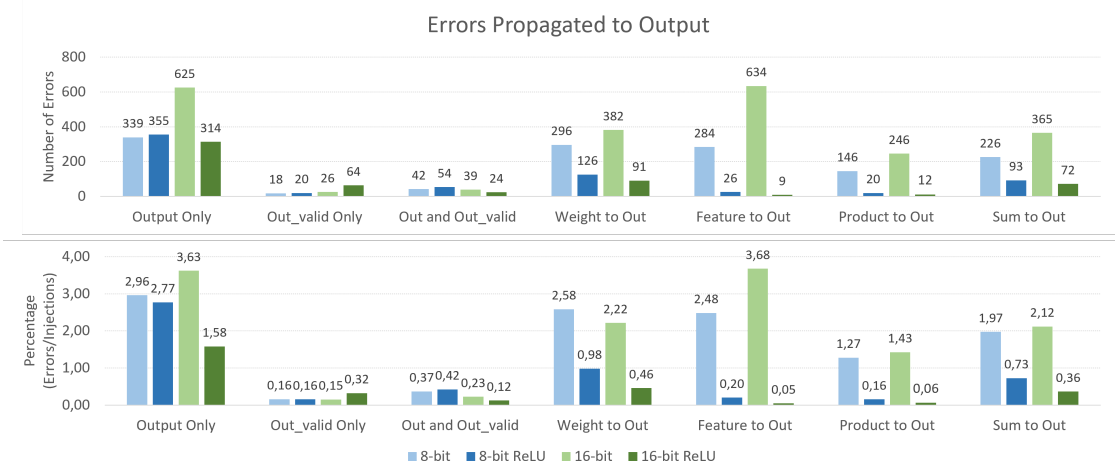

Figure 4.7: Errors propagated to output - Single Neuron

Only a subset of the detected faults is effectively propagated to the outputs, thus affecting the functionality of the neuron.
The following histogram illustrates the error rate calculated for these kind of faults.

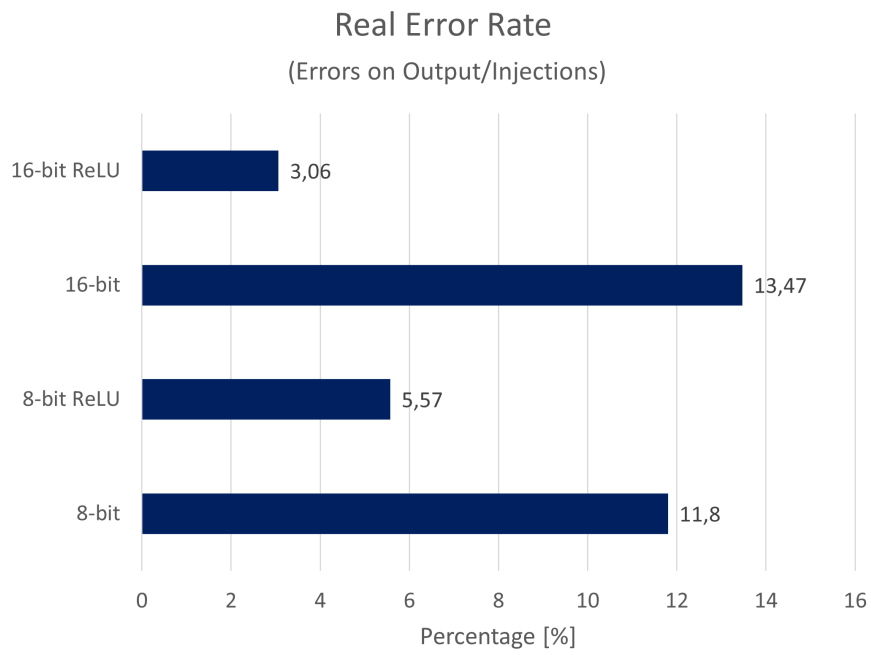

Figure 4.8: Rate of errors on output over injection for Single Neuron

4.2.3 Design of a more Fault Tolerant Neuron

The most evident result of the analysis is that the ReLU function, as expected, nullifies many of the internal faults. For this reason, the redesign was carried out considering the results of versions with ReLU function disabled.

The number of resources available in FPGAs, especially BRAMs and DSPs, represents a significant limitation. For this reason, a targeted TMR mitigation was explored, triplicating only specific parts of the circuit.

As shown in Figure 4.9, the proposed architecture involves triplicating the input Feature pipe, the counter that generates addresses for the Weight Memory, the output Weight pipe from memory, and the output registers. Weight Memory, Multiplier and Adder were not triplicated, even though they are susceptible to SEUs, to avoid increasing the number of used BRAMs and DSPs resources.
The voting blocks are implemented with a single three-input voter. The voter on the outputs is external to the DUT and it is implemented in the Checker. This is

because, once integrated into the entire ZFNet, the outputs of one layer of neurons become the inputs of the next layer and would thus use the voter on the input features.

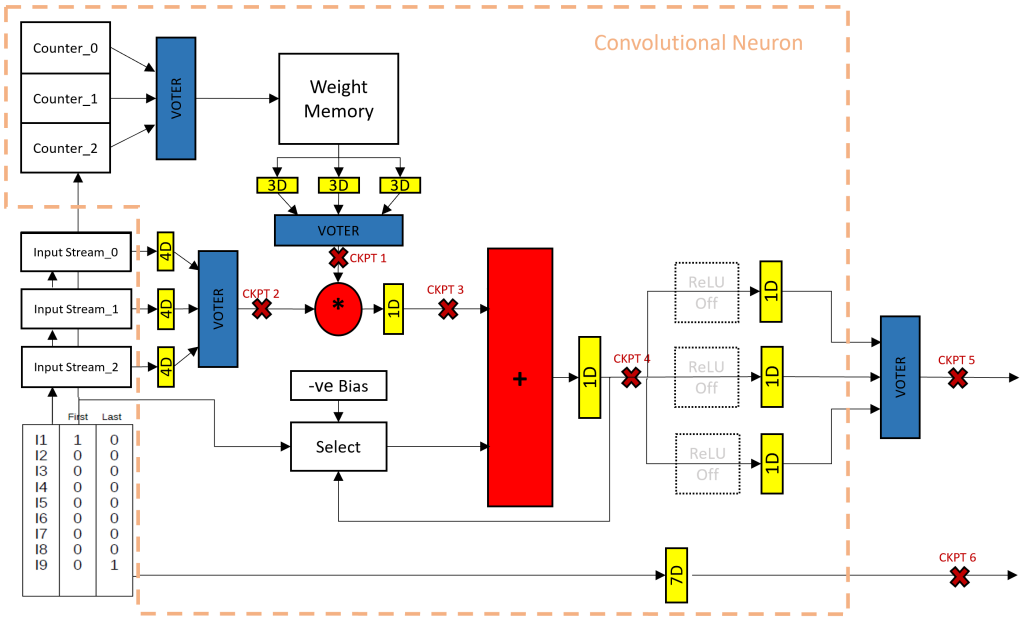

Figure 4.9: Single Neuron architecture with targeted TMR

4.2.4 Results of Fault Injection campaign on mitigated Single Neuron

The campaign was conducted on the 8-bit version and all essential bits were injected.

	Original Neuron	**Mitigated Neuron**
Essential Bits	11453	21167
Total Errors (Rate)	2167 (18,92%)	1619 (7,65%)
Errors on Output (Rate)	1351 (11,79%)	1047 (4,95%)

Table 4.1: Fault Injection results - Original and Mitigated Neuron

Examining the faults propagated to the outputs, the following results are obtained:

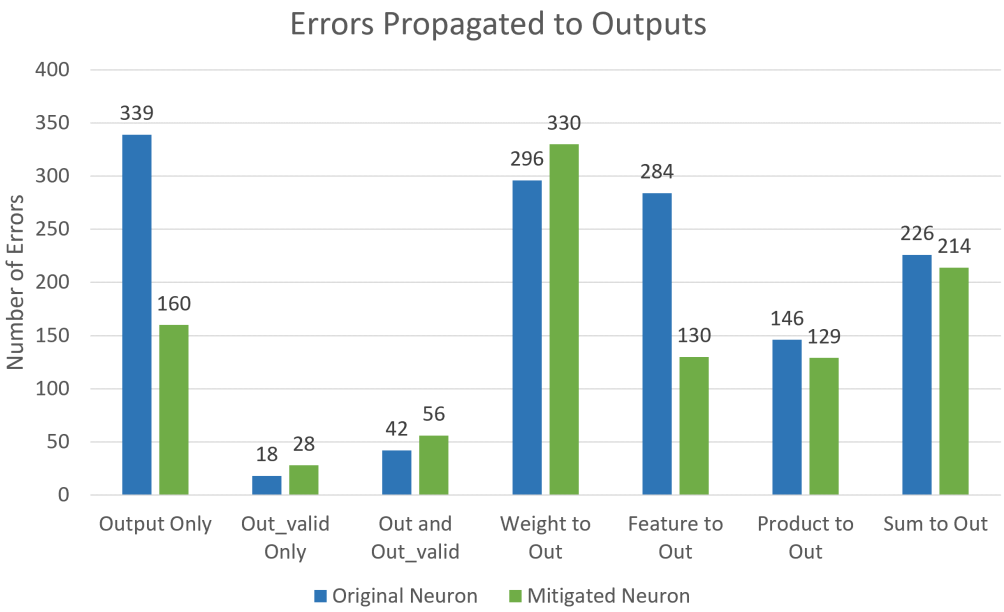

Figure 4.10: Errors propagated to output - Original and Mitigated Single Neuron

As expected, groups Product to Output, Sum to Output and Valid are practically unchanged. Output only and Feature to Output groups show a significant reduction.
On the contrary, Weight to Output group shows an increase. It is possible that to reduce them, the entire memory needs to be triplicated but it might be disadvantageous on an FPGA, considering resource usage.

As shown in Table 4.2, referred to Zynq-7020 hardware, the targeted TMR approach allowed achieving the previously presented results at the cost of only a few additional resources. This is in contrast to a general TMR, which would have required 3 Block RAMs (RAMB18) and 6 DSPs, making the implementation of a complete ZFNet on an FPGA challenging.

		Original Neuron		Mitigated Neuron	
Site Type	**Available**	**Used**	**% Util**	**Used**	**% Util**
Slice LUTs	10000	10	0,10	46	0,46
Slice Registers	20000	79	0,40	137	0,69
Slice	2500	18	0,72	31	1,24
Unique Control Sets	2500	4	0,16	4	0,16
Block RAM Tile	30	0,50	1,67	0,50	1,67
RAMB36/FIFO	30	0	0,00	0	0,00
RAMB18	60	1	1,67	1	1,67
DSPs	60	2	3,33	2	3,33

Table 4.2: Original and Mitigated Neuron resources usage

Chapter 5

ZFNet Input Layer

In this chapter, the results of the analyses performed on the input layers of the ZFNet Convolutional Neural Network are reported.
Two fault injection campaigns were performed: in the first one, the layers use the original neurons, while in the second one, these neurons are replaced by the modified ones with targeted TMR.

5.1 Block Description

In this section the two architectures under test are introduced.

5.1.1 ZFNet Input Layer with original Neurons

Figure 5.1 illustrates the structure of each layer.

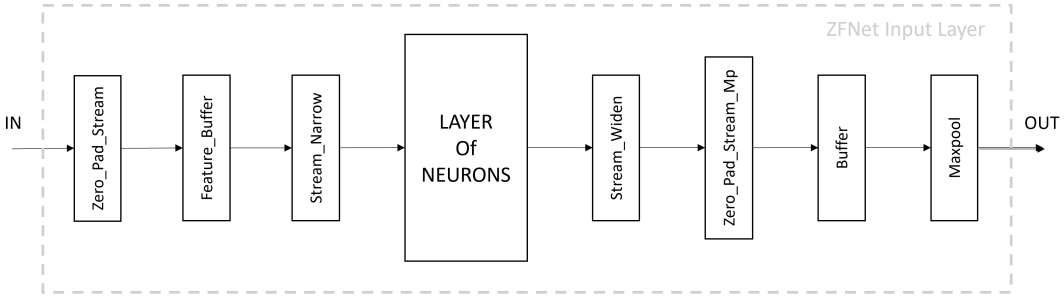

Figure 5.1: ZFNet input layer structure

For this analysis, the number of neurons in layer 0 has been reduced to 8 (instead of 96), while that of layer 1 has been increased to 22 (instead of 256).
Despite this reduction, it is interesting to note that it was still necessary to use two clock regions (X0Y0 and X1Y0) to implement the RTL correctly, providing further support for the choice of targeted TMR.

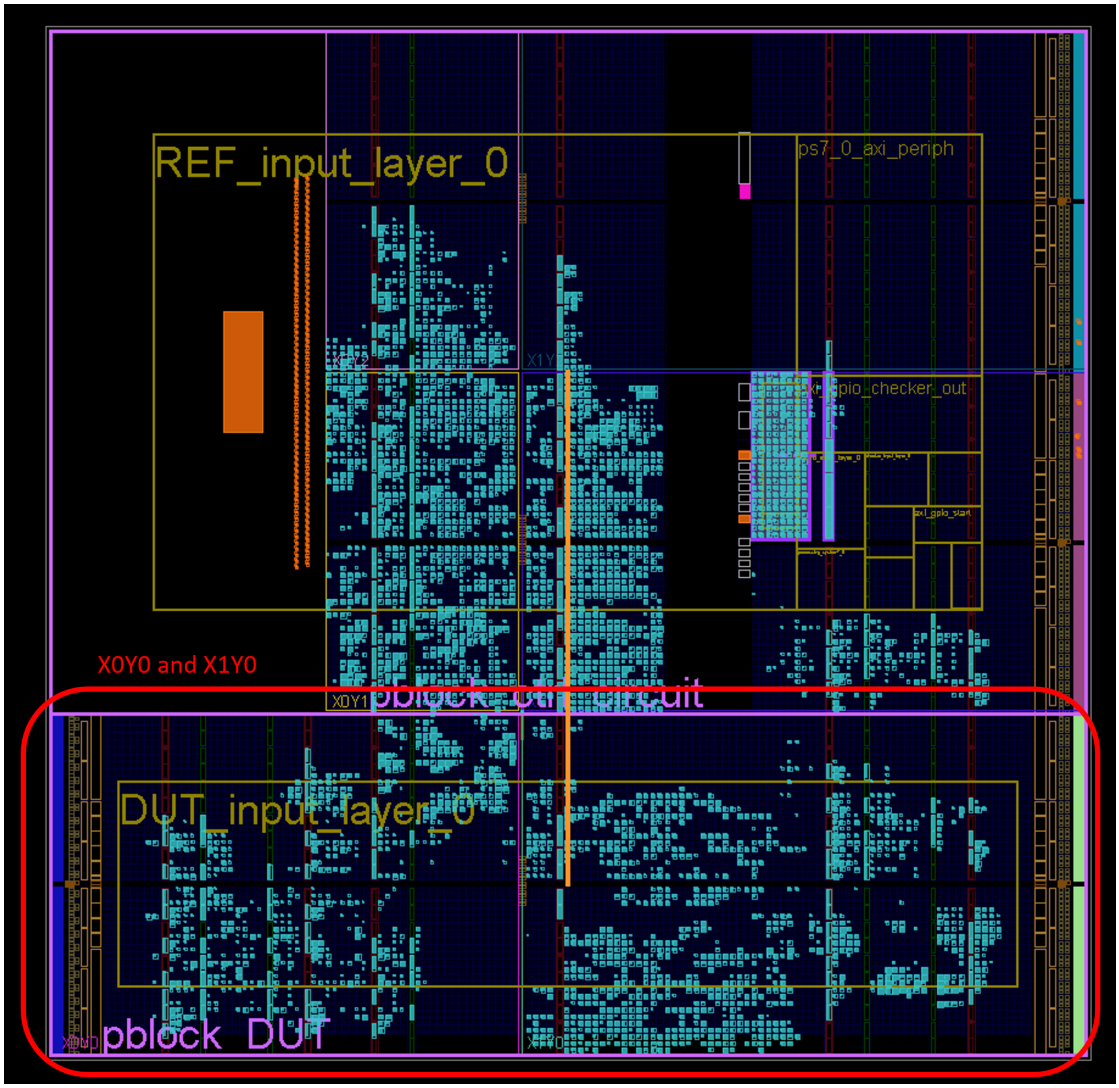

Figure 5.2: ZFNet input layers physical implementation

5.1.2 ZFNet Input Layer with modified Neurons

Figure 5.3 illustrates the structure of the modified input layer.

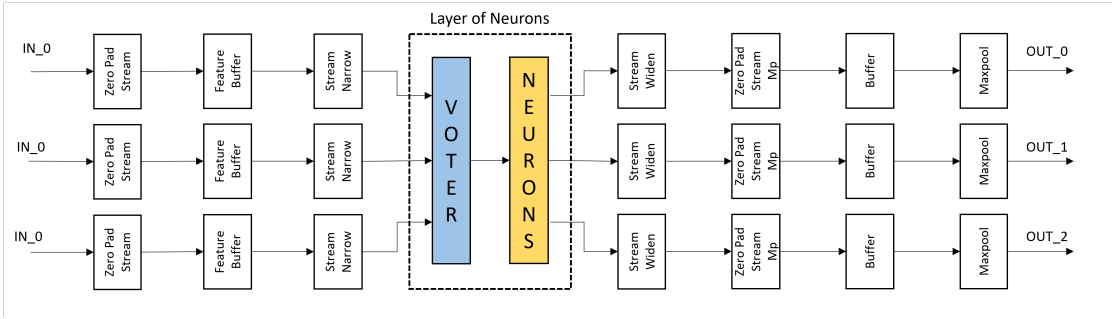

Figure 5.3: ZFNet input layer structure with modified Neurons

Since the inputs and outputs of the modified neuron are triplicated, each layer will also see its inputs and outputs triplicated. This results in a significant increase in area, as shown in the Table 5.1, referred to Zynq-7020 Programmable Logic.

		Original Layers		**Modified Layers**	
Site Type	**Available**	**Used**	**% Util**	**Used**	**% Util**
Slice LUTs	22800	3955	17,35	12016	52.70
LUT as Logic	22800	3937	17,27	11615	50.94
LUT as Memory	7400	18	0,24	401	5.42
Slice Registers	45600	4625	10,14	13552	29.72
F7 Muxes	11400	204	1,79	612	5.37
F8 Muxes	5700	98	1,72	294	5.16
Slice	5700	1620	28,42	3946	69.23
SLICEL	3850	1059	27,51	2626	68.21
SLICEM	1850	561	30,32	1320	71.35
Unique Control Sets	5700	137	2,40	334	5.86
Block RAM Tile	60	29	48,33	57	95.00
RAMB36/FIFO	60	0	0,00	0	0.00
RAMB18	120	58	48,33	114	95.00
DSPs	100	30	30,00	30	30.00

Table 5.1: Original and Modified Input Layer resources usage

While the overall resource utilization has increased, the number of DSPs has

remained unchanged.

5.2 Reliability Analysis

Before detailing the results obtained during the reliability analysis, it is essential to note that monitoring occurs differently compared to the Single Neuron. For the layers, there are no internal checks: only the outputs of the DUT and the Reference Model are compared, as shown in Figure 5.4

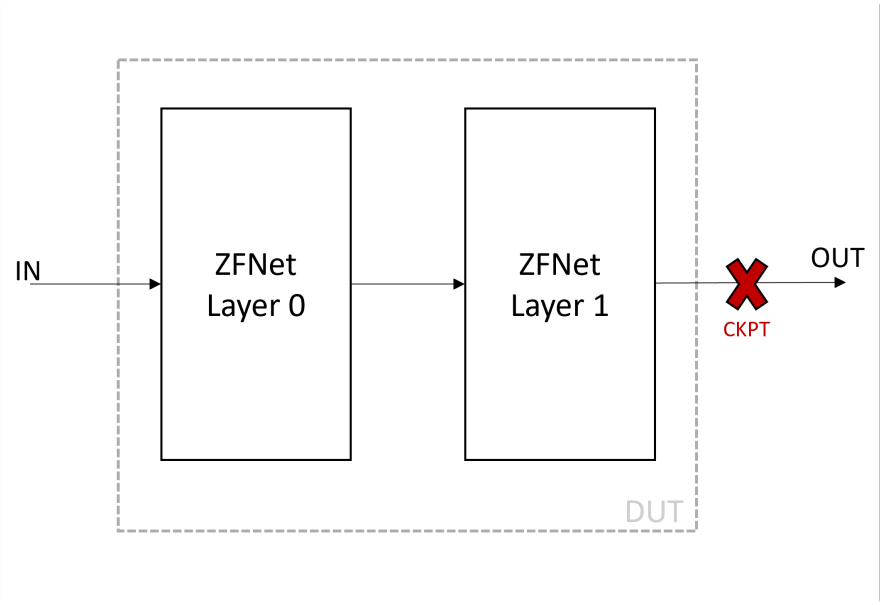

Figure 5.4: DUT structure

5.2.1 Results of Fault Injection campaigns

Given the substantial size of the two examined blocks, the number of essential bits has also increased significantly. To manage test times, the decision was made to test only a subset of these essential bits, randomly selected by the test program. Considering the SEUs rate for a Low Earth Orbit, approximately 93 events per day[39], and recognizing that our analysis targets only essential bits, 500.000 errors were injected into the CRAM to thoroughly test the block. The results obtained are reported in Table 5.2.

	Original	**Modified**
Total Essential Bits	915.325	2.653.015
Total Injections	500.000	500.000
Total Errors	1.199	952

Table 5.2: Fault Injection results - Original and Mitigated Input Layers

The data shows that the layers are already quite tolerant to SEUs even without modification. However, evaluating the error rate on the total injected bits, the Original has a rate of 0,23% , while the Modified one has a rate of 0,19%, resulting in a higher reliability.

Chapter 6

Conclusion

In this Thesis, an environment for studying the radiation effects on Xilinx Zynq-7020, but applicable to the entire range of Xilinx SRAM-based FPGAs, has been presented. The environment exploits the capabilities of the LogiCORE IP Soft Error Mitigation (SEM) Controller provided by Xilinx, capable of precisely injecting faults into the configuration memory of the PL, detecting and correcting existing faults, and granting the possibility to identify critical bits of the implemented circuits through Essential Bits technology. The entire system was set up to analyze the radiation effects on the components of a hardware version of the ZFNet Convolutional Neural Network, provided by Alpha Data.

Given the hardware complexity of the system under examination, a methodology was studied and proposed to simplify and accelerate the possibility of identifying SEUs-tolerant solutions for neural networks. The redundancy within neural networks was exploited, focusing on analyzing not the entire system but individual neurons, the core of the entire network. After a careful study of the SEM IP, the development of supporting RTL, and the creation of two software applications, one to be run on the ARM Cortex-A9 processor of the Zynq-7020 and one to be run PC to autonomously manage the entire operation, a fault injection campaign was performed to all the critical bits of the single neuron, thereby identifying all critical points of the device.

Following this analysis, a new version of the neuron was studied and proposed in an attempt to make it more robust to SEUs while not significantly increasing

the demand for logic resources, a limitation of FPGAs. The choice was made to attempt mitigation through targeted Triple Modular Redundancy, thus triplicating only certain internal parts and not the entire block. The new architecture was then validated through a new fault injection campaign, proving more robust than the original version.

The modified neuron was then tested within the first two layers of the ZFNet. Two fault injection campaigns were conducted, one with the original neurons and the second with the modified neurons. After 500.000 injections, the analyses revealed that the examined system is already very tolerant, with an error rate of 0.23%, while the modified version achieved an error rate of 0.19%, demonstrating the potential validity of the pursued approach.

6.1 Future Works

The entire work presented in this Thesis, while proving to be valid, serves as only a starting point and provides numerous insights for further developments and improvements. There are various areas where advancements and refinements to the proposed approach are possible.

Firstly, concerning the testing environment, a crucial step for the future would be the increase in hardware resources. Given the size of the ZFNet network, implementing a multi-platform system, where the design is distributed among different working FPGAs, could allow for testing and characterizing the neural network under more realistic conditions. Although the complexity of implementing the fault injection system would increase, the benefits could be significant.

Another potential avenue for improvement involves optimizing the resources used by neurons. Exploring the use of resource-optimized versions, for example, by reducing throughput but sharing available resources, could be considered. Alpha Data already provides optimized versions of neurons in its CNN Library, and adopting such versions could positively impact the overall system performance.

Despite the positive results obtained with TMR on individual neurons, the analysis of the two layers in Chapter 5 showed minimal differences between the two solutions. Therefore, it may be essential to explore new alternatives or continue refining targeted TMR. For instance, adopting a targeted TMR that, along with

resource sharing, allows for the triplication of specific elements such as the Multiplier, Adder or directly the Weight RAM, could represent a further step forward in the quest for more efficient and SEU-tolerant solutions.

Bibliography

[1] *What is Machine Learning and How Does It Work? In-Depth Guide — techtarget.com*. [Accessed 28-11-2023]. URL: https://www.techtarget.com/searchenterpriseai/definition/machine-learning-ML (cit. on p. 1).

[2] Sanchez Ernesto Gavarini Gabriele Ruospo Annachiara. «Evaluation and mitigation of faults affecting Swin Transformers». In: *29th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS 2023)*. 2023, pp. 1–7. DOI: 10.1109/IOLTS59296.2023.10224882 (cit. on p. 2).

[3] Arden Dertat. *Applied Deep Learning - Part 4: Convolutional Neural Networks — towardsdatascience.com*. [Accessed 28-11-2023]. URL: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2 (cit. on p. 2).

[4] Stefano Silvestrini and Michèle Lavagna. «Deep Learning and Artificial Neural Networks for Spacecraft Dynamics, Navigation and Control». In: *Drones* 6.10 (2022). ISSN: 2504-446X. DOI: 10.3390/drones6100270. URL: https://www.mdpi.com/2504-446X/6/10/270 (cit. on p. 2).

[5] Jeffrey S. George. «An overview of radiation effects in electronics». In: *AIP Conference Proceedings* 2160.1 (Oct. 2019), p. 060002. ISSN: 0094-243X. DOI: 10.1063/1.5127719. eprint: https://pubs.aip.org/aip/acp/article-pdf/doi/10.1063/1.5127719/14196223/060002_1_online.pdf. URL: https://doi.org/10.1063/1.5127719 (cit. on pp. 2, 5).

[6] *White Papers and Application Notes - Alpha Data — alpha-data.com*. https://www.alpha-data.com/resources/white-papers-and-application-notes/. [Accessed 28-11-2023] (cit. on pp. 3, 57).

[7] Andreas. *Single Event Effects - The Achilles heel of modern aerospace electronics — engineeringpilot.com*. [Accessed 28-11-2023]. URL: https://www.engineeringpilot.com/post/single-event-effects-the-achilles-heel-of-modern-aerospace-electronics (cit. on pp. 3, 6, 10).

[8] Timothy R. Oldham. «Basic Mechanisms of TID and DDD Response in MOS and Bipolar Microelectronics». In: *2011 NSREC Short Course Paper* (2011). URL: https://nepp.nasa.gov/pages/pubs.cfm (cit. on pp. 4, 5).

[9] Claude Leroy and Pier Giorgio Rancoita. «Particle interaction and displacement damage in silicon devices operated in radiation environments». In: *Reports on Progress in Physics* 70 (Mar. 2007), p. 493. DOI: 10.1088/0034-4885/70/4/R01 (cit. on p. 5).

[10] Yu Xie, Tingting Qiao, Yizhuang Xie, and He Chen. «Soft error mitigation and recovery of SRAM-based FPGAs using brain-inspired hybrid-grained scrubbing mechanism». In: *Frontiers in Computational Neuroscience* 17 (2023). ISSN: 1662-5188. DOI: 10.3389/fncom.2023.1268374. URL: https://www.frontiersin.org/articles/10.3389/fncom.2023.1268374 (cit. on p. 5).

[11] Luca Sterpone Niccolò Battezzati and Massimo Violante. *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Springer New York, 2011. DOI: 10.1007/978-1-4419-7595-9 (cit. on p. 5).

[12] J. Miller, L. Taylor, C. Zeitlin, L. Heilbronn, S. Guetersloh, M. DiGiuseppe, Y. Iwata, and T. Murakami. «Lunar soil as shielding against space radiation». In: *Radiation Measurements* 44.2 (2009), pp. 163–167. ISSN: 1350-4487. DOI: https://doi.org/10.1016/j.radmeas.2009.01.010. URL: https://www.sciencedirect.com/science/article/pii/S1350448709000122 (cit. on p. 8).

[13] R.C. Baumann. «Radiation-induced soft errors in advanced semiconductor technologies». In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316. DOI: 10.1109/TDMR.2005.853449 (cit. on p. 8).

[14] Hainan Liu Feitao Qi Tao Liu et al. «Comparison Study of Bulk and SOI CMOS Technologies based Rad-hard ADCs in Space». In: *ESA's AMICSA 6th International Workshop on Analogue and Mixed-Signal Integrated Circuits*

for Space Applications (2016). URL: https://indico.esa.int/event/102/contributions/54/attachments/48/56/Comparison_Study_of_Bulk_and_SOI_CMOS_Technologies_based_Rad-hard_ADCs_in_Space_v1.pdf (cit. on p. 8).

[15] *Error correcting codes. Brilliant.org.* [Accessed 28-11-2023]. URL: https://brilliant.org/wiki/error-correcting-codes/#:~:text=An%20error%20correcting%20code%20(ECC,ECCs%20defend%20against%20data%20corruption. (cit. on p. 9).

[16] R. E. Lyons and W. Vanderkulk. «The Use of Triple-Modular Redundancy to Improve Computer Reliability». In: *IBM Journal of Research and Development* 6.2 (1962), pp. 200–209. DOI: 10.1147/rd.62.0200 (cit. on pp. 9, 10).

[17] Laurent Lesage, Boris Mejías, and Marc Lobelle. «A software based approach to eliminate all SEU effects from mission critical programs». In: *2011 12th European Conference on Radiation and Its Effects on Components and Systems*. 2011, pp. 467–472. DOI: 10.1109/RADECS.2011.6131353 (cit. on p. 9).

[18] Ahmad Sheikh and Aiman El-Maleh. «Double Modular Redundancy (DMR) Based Fault Tolerance Technique for Combinational Circuits». In: *Journal of Circuits, Systems and Computers* 27 (Oct. 2017), p. 1850097. DOI: 10.1142/S0218126618500974 (cit. on p. 10).

[19] James Wetzel et al. «Beam Test Results of the RADiCAL—A Radiation Hard Innovative EM Calorimeter». In: *IEEE Transactions on Nuclear Science* 70.7 (July 2023), pp. 1296–1300. ISSN: 1558-1578. DOI: 10.1109/tns.2023.3268590. URL: http://dx.doi.org/10.1109/TNS.2023.3268590 (cit. on p. 11).

[20] Mario García Valderas, Marta Portela García, Celia López, and Luis Entrena. «Extensive SEU impact analysis of a PIC microprocessor for selective hardening». In: *2009 European Conference on Radiation and Its Effects on Components and Systems*. 2009, pp. 333–336. DOI: 10.1109/RADECS.2009.5994670 (cit. on p. 11).

[21] O. Ruano, Francisco García-Herrero, Luis Aranda, A. Sanchez-Macian, Laura Rodríguez, and Juan Antonio Maestro. «Fault Injection Emulation for Systems in FPGAs: Tools, Techniques and Methodology, a Tutorial». In: *Sensors* 21 (Feb. 2021), p. 1392. DOI: 10.3390/s21041392 (cit. on p. 11).

[22] *XUP PYNQ-Z2 — xilinx.com*. [Accessed 28-11-2023]. URL: https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html#overview (cit. on p. 14).

[23] *Zynq 7000 SoC — xilinx.com*. [Accessed 28-11-2023]. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (cit. on pp. 14, 15).

[24] *AXI Reference Guide (UG1037) — docs.xilinx.com*. [Accessed 28-11-2023]. URL: https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide (cit. on pp. 14, 39).

[25] *Vivado Design Suite User Guide (UG893) — docs.xilinx.com*. [Accessed 28-11-2023]. URL: https://docs.xilinx.com/r/2021.1-English/ug893-vivado-ide/Results-Windows-Area (cit. on pp. 16, 42).

[26] *Vitis Unified Software Platform Documentation (UG1400) — docs.xilinx.com*. [Accessed 28-11-2023]. URL: https://docs.xilinx.com/v/u/2020.1-English/ug1400-vitis-embedded (cit. on p. 17).

[27] *Vitis HLS — xilinx.com*. [Accessed 28-11-2023]. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html (cit. on p. 17).

[28] *Soft Error Mitigation (SEM) Core — xilinx.com*. [Accessed 28-11-2023]. URL: https://www.xilinx.com/products/intellectual-property/sem.html#overview (cit. on p. 22).

[29] *SEM Ccontroller Product Guide (PG036) — docs.xilinx.com*. [Accessed 28-11-2023]. URL: https://docs.xilinx.com/r/en-US/pg036_sem/ (cit. on pp. 22, 23, 25, 29, 30, 35, 52).

[30] Nicola Tisat. *Progettazione e sviluppo di un'architettura tollerante alle radiazioni basata su FPGA riconfigurabile dinamicamente*. Tesi di Laurea Magistrale. 2018 (cit. on pp. 30, 31).

[31] *Processor System Reset Module v5.0 Product Guide (PG164) — docs.xilinx.com*. [Accessed 28-11-2023]. URL: https://docs.xilinx.com/v/u/en-US/pg164-proc-sys-reset (cit. on p. 39).

[32] *AXI GPIO v2.0 Product Guide (PG144) — docs.xilinx.com*. [Accessed 28-11-2023]. URL: https://docs.xilinx.com/v/u/en-US/pg144-axi-gpio (cit. on p. 40).

[33] *Vivado Design Suite Properties Reference Guide (UG912) — docs.xilinx.com*. [Accessed 28-11-2023]. URL: https://docs.xilinx.com/r/en-US/ug912-vivado-properties/PATH_MODE (cit. on pp. 41, 43).

[34] *Vivado Design Suite Tcl Command Reference Guide (UG835)— docs.xilinx.com*. [Accessed 28-11-2023]. URL: https://docs.xilinx.com/r/en-US/ug835-vivado-tcl-commands/write_bitstream (cit. on p. 44).

[35] Óscar Ruano, Francisco García-Herrero, Luis Alberto Aranda, Alfonso Sánchez-Macián, Laura Rodriguez, and Juan Antonio Maestro. «Fault Injection Emulation for Systems in FPGAs: Tools, Techniques and Methodology, a Tutorial». In: *Sensors* 21.4 (2021). ISSN: 1424-8220. DOI: 10.3390/s21041392. URL: https://www.mdpi.com/1424-8220/21/4/1392 (cit. on p. 44).

[36] *AR41197 - What is the difference between the EBC and the EBD file generated by the BitGen essential bits command? — support.xilinx.com*. [Accessed 28-11-2023]. URL: https://support.xilinx.com/s/article/41197?language=en_US (cit. on p. 44).

[37] *AR67337 - How to use the SEM IP error report to look up bit error locations using essential bit data in an EBD file? — support.xilinx.com*. [Accessed 28-11-2023]. URL: https://support.xilinx.com/s/article/67337?language=en_US (cit. on p. 52).

[38] Dominik Stursa and Petr Dolezel. «Comparison of ReLU and linear saturated activation functions in neural network for universal approximation». In: *2019 22nd International Conference on Process Control (PC19)*. 2019, pp. 146–151. DOI: 10.1109/PC.2019.8815057 (cit. on p. 57).

[39] A.L. Vampola, M. Lauriente, D.C. Wilkinson, J. Allen, and F. Albin. «Single Event Upsets correlated with environment». In: *IEEE Transactions on Nuclear Science* 41.6 (1994), pp. 2383–2388. DOI: 10.1109/23.340591 (cit. on p. 70).