POLITECNICO DI TORINO

Master Degree Course in Mechatronic Engineering

Master Degree Thesis

Software-Based Radiation Effects Analysis on AP-SoC Embedded Processor

Supervisors

Prof. Luca Sterpone

Ph.D. Sarah Azimi

Candidate

Daniele Rizzieri

Academic Year 2020/2021

Summary

Fault tolerance analysis is a crucial and fundamental phase in any development process that has the aim to produce a dependable system and throughout the decades, each field of science improved its knowledge about all the possible threats that can affect a system during operation: space exploration field is one of the most active and focused on these studies. This research has been developed as part of the mission HERA by ESA, due to launch in 2024, and has the main goal of investigating what are the outcomes that can occur when SEU and SEMU faults affect several parts of a well-known system under test.

For this purpose, a well-known System-On-Chip architecture has been considered to analyze the effects of radiations striking the hardware during the execution of an application. Furthermore, through the use of two *ex novo* fault injection platforms, observations on the criticality level of the various parts of the SoC have been deduced, along with other key observations coming from the obtained data. The first developed tool-set addresses fault injections in the main memory of the computation system (Random Access Memory, RAM), while the second one aims at the injection of faults in the registers of the CPU. In both cases the goal is to simulate radiation related faults at runtime, during the execution of an application. To perform a complete analysis, several detailed fault models have been chosen, each related to specific scenarios.

To perform the radiation effects analysis a subset of applications from the MiBench benchmark suite has been chosen as test input. It has been decided to classify each injection outcome by means of process exit status, crash data (core

dump) and by the obtained functional results.

One of the main results is the observation of the segmentation fault as the most common malfunction, throughout all the fault models and among all the benchmark applications. In particular, it has been observed that if the main memory control logic gets affected and the process fails, the probability that a segmentation fault occurred are much bigger than the ones related to other malfunctions (i.e. Floating-Point Unit exception, Illegal Instruction and others). Another key point is represented by the Silent Data Error (SDE) analysis. In fact, it has been observed that we have a considerably higher probability to have an SDE when an alteration of the values in the registers occurs. On the contrary, the probability to obtain a hang process is much higher when the main memory portion related to it gets corrupted, while it is nearly null when registers get altered during execution.

General Structure

In chapter 1 an introduction about the topic of dependability is given, along with the motivations that justify the relevance of this thesis work.

Chapter 2 is dedicated to the state-of-the-art survey: here a comparison with other available works has been done, to highlight the processes and the choices persecuted by other authors, underlying why there has been the need to develop two new fault injection platforms from scratch.

In chapter 3 one can find a brief introduction about all the context around this thesis work, with particular attention to the radiation effects theory and the technology that has been involved.

In chapter 4 an in-depth description of the developed fault injection platform is given. Here, all the choices that have been made throughout the process are explained and motivated, along with the choices of which output data have been collected and analyzed.

In chapter 5 the experimental results and their thorough analysis are reported, discussing graphs and charts explaining the obtained data.

Chapter 6 and 7 are devoted to conclusions coming from this work and the analysis of the possible implications, applications and future work that is planned or that can be done addressing further research interest.

Acknowledgements

Ringraziamenti

Grazie a Renata e Stefano, pilastri portanti che mi hanno retto e supportato con pazienza e amore in tutti questi anni, come solo due meravigliosi genitori come loro avrebbero potuto fare.

Grazie a Lucia, che mi ha supportato, sopportato e mi ha accompagnato da vicino e da distante in questa importante parte della mia vita.

Grazie a Stefano, Guglielmo, Riccardo e Marco per esserci sempre stati, nonostante tutto. Grazie per aver contribuito a rendere questi anni dentro e fuori dall'università una piacevole e soddisfacente avventura.

Grazie a Silvia per la nostra amicizia, i preziosi consigli e per aver letto e corretto l'inglese migliorabile di questa tesi.

Contents

List of Figures

Abbreviations

BRAM Block Ram Memories.

COTS Commercial Off The Shelf.

ECSS European Cooperation for Space Standardization.

ESA European Space Agency.

ESCC European Space Components Coordination.

FPGA Field Programmable Gate Array.

IC Integrated Circuit.

LET Linear Energy Transfer.

LUT Look-Up Tables.

MBU Multiple Bit Upset.

PL Programmable Logic.

PS Processing System.

RHBD Rad-Hard-By-Design.

SEE Single Event Effects.

SEFI Single Event Functional Interrupt.

SEGR Single Event Gate Rupture.

SEL Single Event Latch-Up.

SEU Single Event Upset.

SoC System-On-Chip.

Chapter 1

Introduction

This thesis begins with a story. It is November 8th, 2011 and it's launch day for Phobos-Grunt , the first Mars sample return mission by Roscosmos, the Russian Federation space agency. The mission has the main objective of visiting Phobos, Mars major natural satellite, and delivering back to Earth samples of its soil, after analyzing them *in situ*. The main payload of the mission is the Chinese orbiter Yinghuo-1, to be delivered in orbit around Mars to observe the external environment of the planet. Other science goals were to study the Martian radiation, plasma and dust environments and to search for possible past or present life, by means of what are called "biosignatures". With a project cost of 1.5 billion rubles (€36.5 million) and a mission funding of 2.4 billion rubles (€59 million), the Russian Mars exploration journey was expected to last from 2011 to 2014, when the satellite was planned to return to Earth.

Two hours and a half after launch, the main propulsion unit was expected to insert the spacecraft in an intermediary orbit, prior to departure towards the Martian system. Contact with Phobos-Grunt was lost right after the engineers discovered that it never reached its target orbit, remaining instead in its parking orbit, near Earth. Later, the European Space Agency tried to communicate with the satellite and after several unsuccessful attempts ESA decided to end the effort to try to make contact. Without propulsion, Phobos-Grunt has been slowly descending towards

Earth atmosphere and on the 15th of January 2012 the spacecraft was destroyed during re-entry, spreading its fragments into the Pacific Ocean.

After several in-depth investigations, on 1st February 2011 the head of Roscosmos announced that a burst of cosmic radiation could have hit the satellite, causing a reboot of several parts of the system, forcing it into a standby mode.

The mission had been later criticized by scientists, engineers and experts all around the space exploration community, claiming lack of pre-flight testing, lack of funding and use of parts that were not adequately prepared for the hazardous space environment.

This story can really help us understand why fault tolerance analysis is a crucial and fundamental phase in any development process aiming to produce a dependable system. When talking about dependability it is important to understand what are the key qualities that a dependable system must have. *Availability* is one of them: it refers to the probability that a system will work as expected at a generic time in future. A different concept is expressed by the term *reliability* which refers instead to the probability that the system will work as expected repeatedly and in a consistent manner over time. Another key quality of a dependable system, finally, is *safety*: it indicates the ability of a system to either work correctly or to interrupt its operations without causing serious damages or failure.

In order to achieve a good level of dependability, one must consider several types of analysis and testing to be used across the entire development of the project, in order to completely understand what are its flaws and its most sensitive parts. Starting from the requirements specification, through all the steps to deployment, after or during each phase it is necessary to conduct different types of test in order to assure that the obtained result is the expected one, even under unusual conditions. This has to be done considering several possible fault modes that can occur during operation, maximizing the so-called *fault coverage*, that is the number of faults that can be identified by the considered test cases, among all the possible faults.

Throughout the decades, each field of science improved its knowledge about all the possible threats that can affect a system during operation: space exploration field is one of the most active and focused on these studies. This is due to the fact that, most of the times, space missions put together the effort of thousands of people, with fundings that can go up to several billions of euros. Furthermore, in case of human spaceflights, astronauts' lives are on the line and safety concerns are even more crucial.

In particular, since the first interplanetary missions in the 1960s, there have been growing concerns about space radiation effects on humans and on the exposed instruments of spacecrafts and satellites.

With the need for countermeasures against these unwanted effects, many radiation hardening techniques have been developed. These mitigation methods can be put in action either via specific physical modifications or through the adoption of ad-hoc logical structures. Regarding hardware mitigation techniques, they can be applied during the design phase or after the implementation. If the component has been designed and implemented with the purpose of being robust against radiations, then it is called "radiation-hard" - *rad-hard* for simplicity- or "rad-hard-by-design" (RHBD). In other cases, the hardening countermeasures are applied on a Commercial Off The Shelf (COTS) component and most of the times it is easier to apply a logical mitigation method with respect to an hardware solution.

The faults that can occur in an electronic device under the effects of highly energetic particles can be summarized in the so-called Single Events Effects (SEE). They can be further divided into several categories, according to the nature of the fault:

- **Non-destructive faults (*soft-errors*)**:
  - **Single Event Transient (SET)**: voltage glitches with a duration that can go up to some nanoseconds, which may possibly lead to permanent errors.
  - **Single Event Upset (SEU)**: error in the operation of the device caused by a free charge created by heavy particle ionization. In a flip-flop based circuit (e.g., a memory, a register) this results in a so-called *bit-flip*: the

inversion of the value of a stored bit, from 0 to 1 or vice versa. If the inversion affects more than one bit, then we say that a Single Event Multiple Upset (SEMU) or a Multiple Cell Upset (MCU) occurred.

  - **Single Event Functional Interrupt (SEFI)**: abrupt interruption of normal device functions, requiring a power reset to resume normal operations.

- **Destructive faults (*hard-errors*)**:

  - **Single Event Latch-up (SEL)**: abrupt short-circuit between the power supply rails of a MOSFET transistor, compromising the operation of the whole surrounding circuitry and possibly damaging the device itself.

  - **Single Event Gate Rupture (SEGR)**: permanent modification in the conductive paths of a transistor, leading to non-nominal behavior of the device itself.

1.1 Motivation

This thesis work places itself in the field of SEE analysis, and has the main goal of investigating what are the outcomes that can occur when SEU and SEMU faults affect several parts of a well-known system under test.

This type of analysis can be seen as a preliminary and preparatory phase preceding the radiation test. This latter has to be conducted in a very specific facility where the system would be subject to heavy particles beams over a certain period of time. Through this type of preliminary analysis, the effectiveness and relevance of the radiation test can be maximized, since one can predict with a good level of approximation what are the expected outcomes of the test and, if possible, apply some mitigation techniques prior to the test itself, on both hardware and software. In this way it is also possible to check whether the hardening technique is enhancing the system fault tolerance or not, possibly avoiding the need of repeating the radiation test and thus saving a lot of budget, also in terms of time resources.

This research has been developed as part of the HERA mission by ESA, which is due to launch in 2024. This mission has the main objective of reaching a binary asteroid system, Didymos, and observe and study the composition of the secondary body of this system, Dimorphos, which will be previously impacted by the DART spacecraft, developed in the homonymous mission by NASA. Among the payloads of the European mission, there will be two Cubesat satellites that will orbit around Dimorphos to study its morphology and composition.

Figure 1.1. Hera Mission Logo

The hardware model that has been taken into account for conducting the described analysis is the flight computer of one of the Cubesat satellites, which incorporates a System-On-Chip as the main computational unit. By analyzing the obtained results, it'll be possible to predict what is the probability that a certain error will occur during the operation of the system in a real scenario. This is a crucial information, because it can be used to design (or re-design) the system in order to tolerate these faults, possibly integrating some error handling hardware or software.

Chapter 2

State of the art

When approaching the testing phase of an integrated circuit, there are many different parts that can be addressed. The careful choosing of the logical and physical parts to be tested is a fundamental step to achieve a fault coverage that is compliant with the specified requirements. According to choice there are several different testing techniques that can be adopted or, in some cases, *must* be adopted. For example, when working in space exploration fields, one must apply the standards proposed by the related agencies, like the European Cooperation for Space Standardization (ECSS) and the European Space Components Coordination (ESCC). Another example of standards to has to be strictly observed is the ISO26262, related to the automotive field.

To verify that a device complies with the imposed radiation hardness requirements, there are several approaches that can be followed. Starting from the most straightforward, there is the radiation test. It basically consists of putting the DUT under a radiation beam for a predefined amount of time, under controlled environment condition (e.g., vacuum, controlled temperature). During this operation, the DUT undergo several functional tests and the outcomes are collected for further investigations about how the device worked under radiations influence. An example of a radiation test experimental setup is described in [1], where the results of a radiation test are used to validate and study the effectiveness of a new Single

Event Transient (SET) sensitiveness analysis method on nanometric flash-based technology. Other descriptions and studies of radiation test results can be found in [2], where results are used to validate a software-based Single Event Upset (SEU) emulation method.

The main drawback of radiation testing is that it is a very expensive operation, by means of both time and budget resources. It requires a very specific technical facility to be conducted and several months of preparation to be efficiently performed. For these reasons, there are several alternative approaches that can be good compromises, avoiding major drawbacks of the radiation test. These techniques can also be used to properly prepare the radiation test and make it even more effective and valuable.

One of these testing "alternatives" is surely the simulation. When simulating, we're recreating the conditions we'd have in an operational environment, with a good grade of accuracy and with enormously lower costs with respect to radiation-testing. In [3] we can find an example of a new 3D simulation-based approach that has been compared with radiation-test results. Another example of a simulation environment can be found in [4].

Another alternative approach is the one chosen for this work and it is related to fault emulation. Please note that there is a crucial difference between simulation and emulation, since this latter pretends to recreate the *effects* of the operational environment (e.g. heavy-ions particles). This means emulating the various faults that can appear because of external circumstances such as radiation threats, extreme thermal conditions and others, specific to the fields of operation. In case of radiation environment, among all the different faults that can occur (SEE), the most frequent one is the Single Event Upset (SEU), also called *bit-flip*: it is the alteration of a single bit in a memory or in a register, changing from 0 to 1 or vice versa. The emulation of bitflips can be done in many ways. One could, for example, simulate the hardware platform under test and access the bitflip target in a completely software-based manner. This is the case discussed in [5]. Another approach for bitflip injection can be found in [6] and [2], where upsets are induced by means of specific executable code, triggered by interrupts at predetermined instants: this technique is also referred to as Code Emulated Upsets (CEU). Furthermore, as can be seen in [7], a hybrid technique can consist of compiling the source code of the application in an intermediate language and "inject" here the bitflips through the use of ad-hoc instructions. Previous works about CEU approaches can be found in [8] and [9].

All of the above-cited works have different application cases which most of the time strongly depends on the target hardware and in some cases also on the software to be tested itself. As an example, one can look at the Zero Overhead Fault Injection Tool (ZOFI)[10], a fault injection platform that is *time-based*: the bitflips are injected at a random time during the execution of the application and, as described in [10], this makes the platform not suitable for programs with a short execution time.

Another software-based fault injection platform that intervenes at a random time during the execution has been developed by CAROL Reliability Group in Universidade Federal do Rio Grande do Sul (UFRGS).[11][12] This fault injection tool is called *carol-fi*: it supports several fault models and permits to inject a bitflip in a randomly chosen variable during the execution of an application under test. Since it relies on time to decide when to inject the fault, it has the same limits described for ZOFI[10]: for short running programs the time resolution of the injector could be not sufficiently small, leading to the failure of the injections.

Continuing the state of the art analysis of software-based fault injection tools, another example can be found in ESIFT (Efficient System for Error Injection) by Tian et al.[13] In this latter, basing on the application under test a list of possible fault injection points is derived. Then, by using GDB (GNU Debugger) features, the program is run and according to the selected fault model the execution of the application gets corrupted in different ways, ranging from alteration of data contained in variables to the corruption of registers and other memory locations.

In [14], the PROPANE injection tool is described and here a different approach has been pursued. Within PROPANE, the fault injector module aims at stopping the execution when a certain trigger is reached. Then, a fault-free portion of the executable code gets substituted with a faulty one and the execution gets resumed. Finally, the execution outcomes and the propagation of the error are observed by means of predefined variables and their contents.

When the scope of the test is the evaluation the robustness of a certain program against some kind of fault, a key concern are the fault locations. This aspect is crucial by means of testing time and several fault collapsing techniques can be applied, based on equivalence classes that can involve several points of view, from the software level to the hardware implication of each fault model. This can be particularly useful if a high fault coverage (close to exhaustive testing scenarios) is needed. In [15], Hari et al. describe a tool to evaluate the resiliency of an application against transient faults. The tool exploits advanced fault pruning techniques based on the analysis of the machine code to reduce testing effort and maximize the efficiency and the detection level of the conducted fault injection campaigns.

Instead of targeting the application under test directly, a different approach has been followed by Horstmann et al., as described in [16]. The main proposed idea is to monitor the whole system using the running operating system and then inject faults in locations that include OS specific resources: these are hardware counters, sensor values and others.

The work presented in this thesis follows the approach of SEE emulation through two different software-based injection platforms addressing microprocessor registers and system main memory. Several different fault models have been considered and the developed platform permits an in-depth evaluation of the outcomes of the tests, also allowing the repetition of the exact same injections over time. Furthermore, the proposed platforms allow us to keep track of which injections generated certain misbehaviors and to investigate the failure causes in order to study the propagation of the faults inside the system.

Chapter 3

Background

3.1 Radiation effects

When an electronic circuit is operating in the harsh space environment, it is subject to the presence of highly energetic particles. When an energetic ion strikes a region of an integrated circuit (IC), its physical composition can be altered temporarily or in a permanent way, causing effects on its operations. In this section I will summarize the main concepts of radiation effects on semiconductor circuits.

Back in the 60s, the first paper regarding the occurrence of SEE was not studying the space environment threats, instead it was foreseeing the future occurrence of radiation-induced faults due to the terrestrial cosmic rays: as one can imagine, in fact, hazardous radiation environments can also be found here on Earth (e.g., cosmic rays not filtered by the atmosphere, artificial and natural radioactive environments) and not just in outer space. [17]

3.1.1 Radiation characteristics

First of all, what is a *radiation*? A radiation is a group of energetic particles that exchange energy with other particles when colliding and interacting with them. A radiation can have different particle composition, depending on its nature. Cosmic rays, mesons and alpha particles are the main types. The first ones refer to hydrogen, helium and other atoms nuclei, traveling at fraction of light speed towards our solar system from outer space and generated by explosions of stars millions of years ago. Among the cosmic rays there are also highly energetic electrons and protons: these latter are the main contribution of the solar wind, which is not constant and fluctuates in intensity in different periods of the year. Mesons can be produced by the collision of cosmic rays with the terrestrial atmosphere, therefore they carry much lower energy. Despite this, they can still produce unwanted effects on irradiated silicon devices. Finally, alpha particles are created in two ways: the first one is the decay process of radioactive elements and the second one is the interaction of other particles inside the silicon structure of IC, when an highly energetic particle travels through it. [18]

Flux and Fluence

The radiation particles *flux* is defined as the number of particles that pass through a unitary area section of the target material during a unitary time and it is measured in [#particles] $\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$. Starting from the flux, one can compute the *fluence*, which is the integral of the flux over a certain amount of time. It is measured in [#particles] cm^{-2}.

Cross Section

Another very important concept related to radiation effects is the *cross section* of a device. A possible definition: it is the area measure of the surface of the device such that if a particle strikes that area, then there will be a fault. As an areal quantity, it is usually measured in cm^2. Since usually not all the surface area of a device is sensitive to radiations, more precise definitions are needed, often related to specific fault models. For example, one could define the SEU cross section of a sea-of-gates array as an integer number of gates over the total.

Linear Energy Transfer (LET)

When a particle strikes a device and make its way through the material, it rapidly loses energy as it goes in-depth. This is mainly due to the interactions with other nuclei, that can lead the to the generation of secondary energetic particles. An important measure that is crucial in the characterization of SEE is the so-called Linear Energy Transfer (LET): it is a function of the particle energy and the material density and it is defined as the amount of energy lost by the particle per unitary path length ($\frac{\mathrm{MeV}}{\mathrm{cm}}$) traveled by the particle itself, normalized by the density of the material ($\frac{\mathrm{mg}}{\mathrm{cm}^3}$): therefore, it is measured in $\frac{\mathrm{MeV\,cm}^2}{\mathrm{mg}}$ [19] [18].

3.1.2 Physical effects

There are several physical effects that can take place at the silicon level of the device when radiation hits occur and these can lead to observable errors and permanent damaging of different parts of the IC. In particular, there are three main physical effects that has to be considered.

Funneling effect

The *funneling* effect is the most common one and it occurs when a heavy ion break through the junction and depletion layers of a semiconductor device, creating an ionization track and distorting the equipotential surfaces of the silicon structure. This distortion corresponds to the creation of two opposite direction currents that lead to an abrupt modification of the charge density in the area. This burst is extinguished after a short amount of time and its length depends on the LET of the particles and other factors.[20][18] An explanatory figure regarding this phenomenon can be found at 3.1.

Displacement effect

Another relevant physical effect is the so-called *displacement effect*. It occurs when a colliding particle displaces atoms in the silicon lattice, creating interstitial spaces

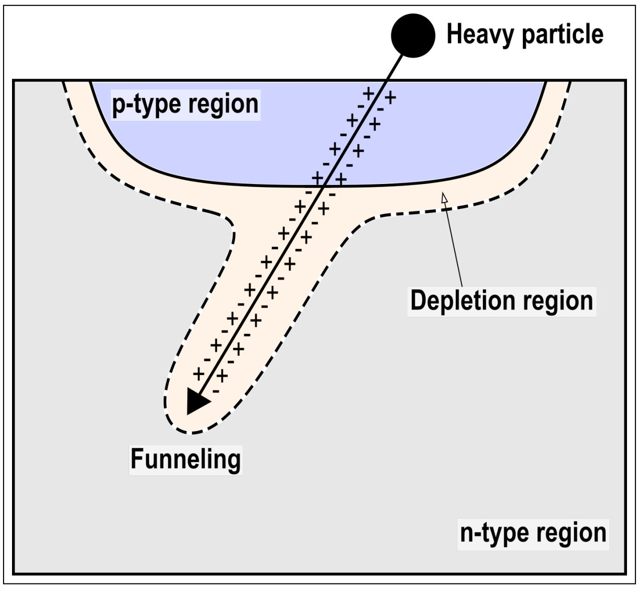

Figure 3.1. Funneling caused by a heavy particle hitting a p-n junction.

that can lead to changes in the electrical functionality of the IC. The amount of energy that the radiation nuclei lose during the lattice displacement is called Non-Ionizing Energy Loss (NIEL) but also *displacement damage equivalent dose.* It is measured in $\frac{\mathrm{MeV}}{\mathrm{g}}$.[19]

Charge accumulation effect

By the combination of the previous two effects, a third one can be identified and it is the charge accumulation. During operation time, several unwanted conditions can occur and the effects of both funneling and lattice displacement can lead to the accumulation of charge in certain locations, with the consequent electrical faults that result in an erroneous operation by the device. The accumulated charge can be measured in C.[21][19]

3.1.3 Single Event Effects (SEE) overview

The fault models that can be identified as a result of one or more of the previously discussed effects can be summarized in two main categories: destructive and non-destructive faults. The destructive faults, also called *hard errors*, are the ones that permanently damage the device itself while for non-destructive faults, also called *soft errors*, the unwanted effects disappear after a certain time or after a power cycle of the interested parts. Hard and soft errors can then be further divided into different categories, as explained in the following paragraphs: the soft errors are divided in SEU, SET and SEFI, while the main hard error types are SEL and SEGR.[18]

Single Event Upset (SEU)

Also referred to as bit-flip, the SEU are the observable result of a charge modification, due to funneling or displacement. In a transistor based memory element (e.g., a latch, a register or a memory cell), a particle could modify the stored charge and force the information node to this new altered state. If the modification affects more than one bit, then we say that a Multiple Bit Upset (MBU) occurred.

Single Event Transient (SET)

As a result of the funneling effect, abrupt currents can be generated in the silicon lattice. This flow of charge can create voltage *glitches* at the output of transistors which length can go from picoseconds to nanoseconds, depending mainly on the LET of the particle in the material. These abrupt changes in the voltage levels can be interpreted as bit transitions of the type 0-1-0 or 1-0-1, thus possibly creating faults that will be propagated in the system and will affect its functionality.[22]

Single Event Functional Interrupt (SEFI)

It is an interruption of the functionalities of the system, recoverable only after a power cycle of the whole device or after performing a hard reset.

Single Event Latch-up (SEL)

We have a Single Event Latch-Up when there is a modification of the silicon structure which lead to a modification in the current flows. In particular we could have an abrupt increase of current passing through some parts of the device, that can result in a permanent damage of the device itself if the power is not cut off in a short time.

Single Event Gate Rupture (SEGR)

When a highly energetic particle strikes a transistor, its gate insulator could be permanently damaged. This event is referred to as Single Event Gate Rupture and it can bring the interested silicon region to a high temperature condition due to newly created high conduction paths. If nothing shuts off the device, this condition could lead to the semiconductor material melting.

3.2 Technology background

In the increasing need for low power consumption and with the continuous evolution towards miniaturization of computing system, a natural evolution can be found in the so-called System-On-Chip (SoC). This latter is an integrated circuit which includes a main Processing System (PS) and several other components such as various interfaces, clock signal generators, special-purpose controllers and integrated memory chips.[23] To further reduce the amount of physical chips and space required by the IC, another main part that may be present in a SoC is the Programmable Logic (PL) which consists of a Field Programmable Gate Array (FPGA) that can be specifically programmed to implement a vast variety of sequential and combinatorial systems, through the use of Look-Up Tables (LUT) and special-purpose chips such as Block RAM memories (BRAM) and others. The various components of a SoC are interconnected through data BUS that can be either intellectual propriety of the company producing the SoC or standardized by protocols like the Advanced Microcontroller Bus Architecture (AMBA), introduced by ARM in 1995.[24]

3.2.1 Zynq-7000 All Programmable SoC family

Introduced in 2011 by Xilinx, the Zynq-7000 All Programmable System On Chip (AP-SoC) is a family of embedded system which incorporates both a processing system and a programmable logic.

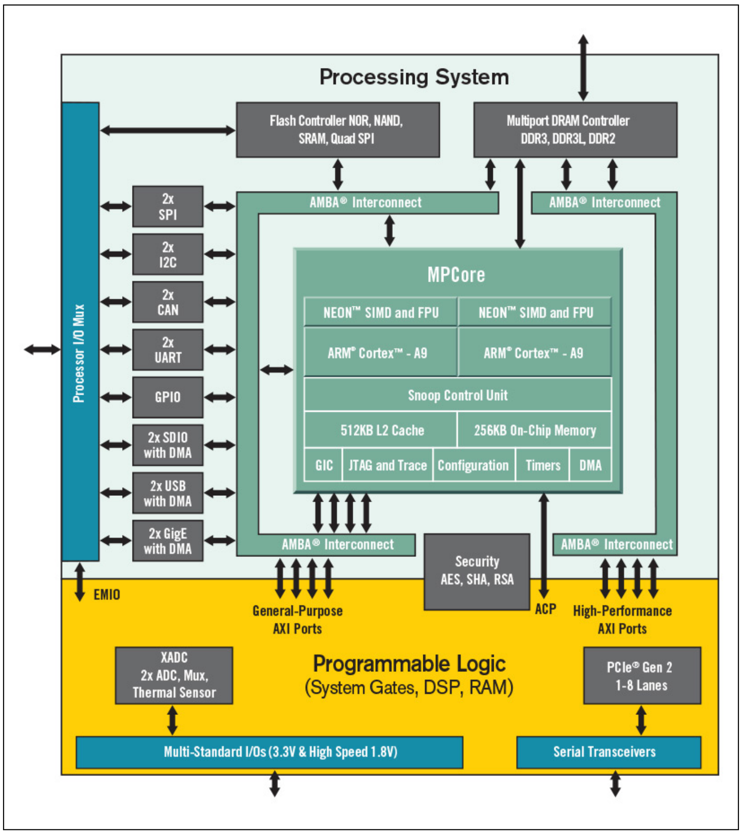

Figure 3.2. Zynq-7000 AP-SoC Block Diagram[25]

As can be seen in the general block diagram of this SoC (Fig.3.2), in the PS we can find a dual core ARM Cortex A9 processor, with a processing speed of up to 1 GHz. This main component permits the SoC to support the execution of full operating systems such as Linux and it is based on the ARMv7-A architecture, which in turn is based on the 32-bit RISC model. A crucial element of this architecture - which is built around a 8 stage pipeline processing - is the Neon engine. Neon is a technology by ARM that allows the PS to do a large number of integer or floating point operations at the same time. The key concept behind Neon is the Single Instruction Multiple Data (SIMD) logic, which consists of applying the same operator to multiple input data, instead of conducting repeated operations one at a time: this results in a tremendous cut in the execution time of these computations, giving massive benefits in applications such as media processing, voice and image recognition, computer vision and deep learning [26]. ARMv7-A architecture embeds also a Memory Management Unit (MMU) module that permits virtual memory management, as required by modern operating systems, and supports two different standard instruction set: the 32-bit based one and the so-called Thumb-2 set, based on 16-bit instructions. For both sets, most of the instructions support conditional execution. Furthermore, special-purpose instructions for the Neon Engine and Vector Floating Point (VFP) are also present.[27]

The available core registers are 17, each containing 32 bits: 13 general-purpose registers, 3 special-purpose registers - Stack Pointer (SP), Link Register (LR) and Program Counter (PC) - and the Current Program Status Register (CPSR) containing information about the current state of the core and its operating mode. Adding to these, among the accessible registers we also find the Neon registers, used for SIMD operations. [27]

Among the blocks surrounding the processor, on the Zynq-7000 SoCs there is a 512KB level 2 cache and a 256KB On-Chip Memory. This last can be particularly useful during the translation of logical addresses. Inside the processor we also find two level 1 32KB cache memories for storing data and instructions.[28]

The interconnection between the PS and the PL consists of AMBA based multiple BUS which are also connecting the processor with the other PS internal blocks. In particular, PS-PL communication is compliant with the AXI standard. AXI stands for Advanced eXtensible Interface and it is part of the ARM AMBA standard: it consists of a high-speed, multiple-master multiple-slave interface for on-chip communication.[29]

The processing system, in addition to a DRAM controller for DDR memories interface, also provides other external interfaces to communicate with the surrounding embedding system. These are General-Purpose Input-Output (GPIO) ports but also protocol specific interfaces such as CAN, SPI, I2C and UART.

Inside the Zynq-7000 SoC family we can find different configurations of PL, according the purpose and the performance level and the power consumption required by the SoC. The FPGAs provided by Xilinx for this SoC integrate up to 444K programmable logic cells, 2020 Digital Signal Processing (DSP) slices and up to 26MB of BRAM, divided in 36KB blocks.[30]

The use of this SoC is widely spread in industrial and mission-critical scenarios. Examples can be found in industrial automation, computer vision, medical application, high resolution media processing, signal transmission but also robotics and space applications.[31][32][33]

3.2.2 PYNQ-Z2 Development Board

A very popular development board which embeds a Zynq-7000 AP-SoC is the PYNQ-Z2 Board by TUL. In fact, this system integrates a Zynq-7020 SoC as the main processing unit, with a PL composed of 13,300 logic slices, 630KB of BRAM and 220 DSP slices.[34] Among the main features of this tool we find an integrated 512MB DDR3 RAM memory, a MicroSD slot for operating system installation and booting, plenty of GPIO pins and several I/O interfaces for both data and video, like Ethernet, HDMI, USB, UART and JTAG.[35] A more detailed overview of the PYNQ-Z2 board can be found in Figure 3.3.

Another important aspect of the board is the capability to run complete operating systems like Linux: in particular, Xilinx provides a default operating system installation running Ubuntu 18.04, based on version 5.4 of Linux Kernel. Regarding the PL, through software instruments like Vivado Design Suite by Xilinx it is possible to implement FPGA systems and accelerators to be loaded onto the PYNQ board, using the concept of Overlay object. In this way it is possible to access these hardware IP blocks using the operating system, in particular, through Python and PYNQ specific libraries.[37]

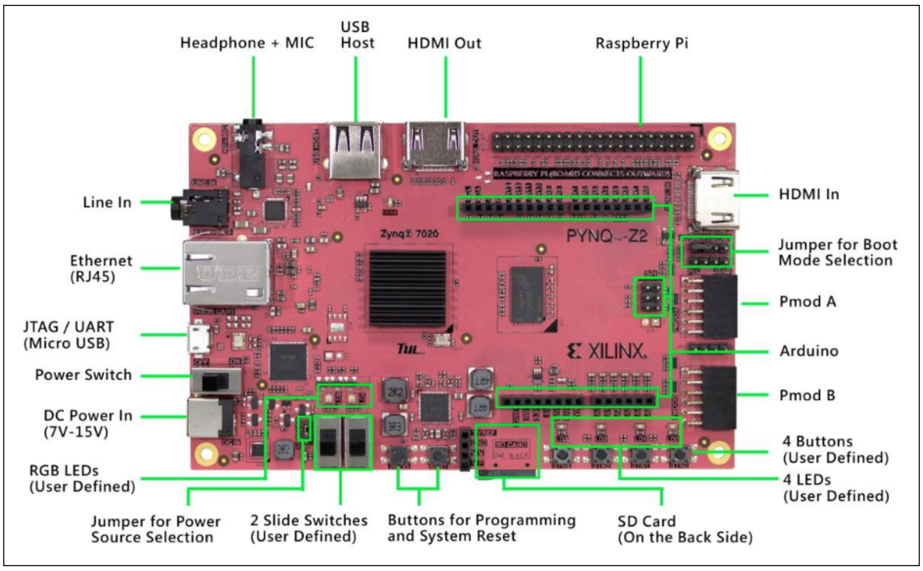

Figure 3.3. PYNQ-Z2 Board Overview [36]

3.2.3 Linux Processes and Signals

A key construct that is at the base of every operating system is the concept of *process*. When an application gets executed by a PS, it is referred to as a process and has various information and resources associated with it: virtual memory portions, files, timers and communication sockets are some examples. Each process is identified by a Process ID (PID) which is univocal within the OS session.[38] In Linux-based operating systems, all the processes are created starting from a parent process called *init*, which has PID equal to 1 and gets started as first process by the OS. During its execution a process can enter several states, each of them characterized by some characteristics. The main states in which an application process can be are depicted in Fig.3.4

As can be seen in the above figure, after a process gets created and a PID is assigned to the execution, it enters the *running* state. From here a process could be stopped by an interrupt, causing it to go in *ready* state. In this latter there are all the processes that have all the resources they need for their execution and are waiting to be executed. Depending on the scheduling policies, the process will be

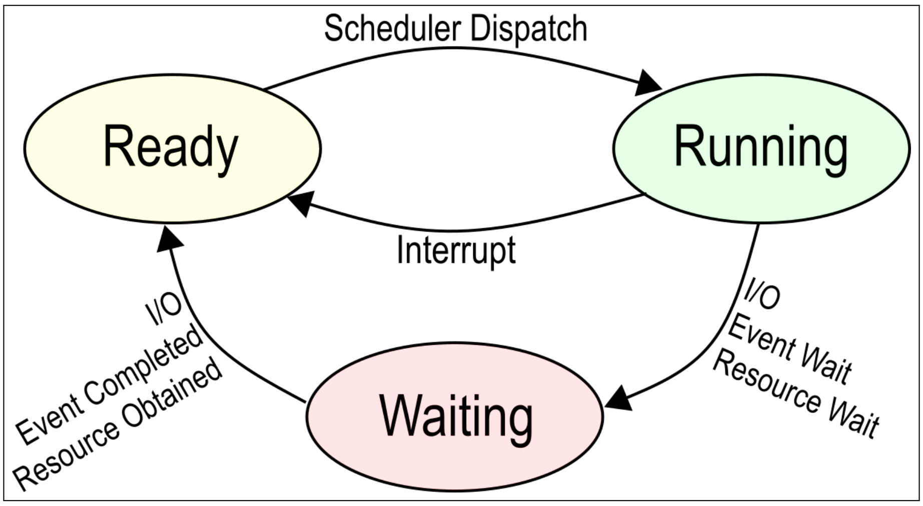

Figure 3.4. Main Process States in Linux-Based OS [38]

subsequently put back in the running state to continue its work. Another condition that is reachable for a running process is the *waiting* state: here the execution instance waits for any resource or event that is needed for continuing its flow and when it obtains them, it is then put back in *ready* queue.[39]

From the running state, there are two main ways for a process to terminate its execution. Firstly, it can simply finish its operations without the occurrence of any error or exception, so that the program simply returns from its main function. Alternatively, an application could go through a premature termination and this can happen either if an exception - hardware or software - is raised by the OS or if the process receives a specific indication to do so by another process. In both cases, the instruments used to force the abrupt termination of processes are the so-called *signals*. In Linux-based operating systems, there are 31 signals used for Inter-Process Communication (IPC), each of them identified with a different *signal number*.[40] According to the specific meaning of a signal, there are certain compulsory operations that a process must go through when receiving it. In Fig.3.1 there are some of the possible signals with the relative numbers and the operations to be conducted by the receiving process. These latter are: *terminate*, *ignore*, *coredump*, *stop* and *continue*. With *stop* and *continue* indications, the process execution is stopped and resumed, respectively. When the default action is *coredump*, the execution terminates and the present core situation - by means of registers and other data - is saved to a specific format file called *core dump*: this file can be used later by the user to investigate the cause of the termination. If a signal is instead associated with the default action *terminate* the receiving process terminates its operations and, finally, with the *ignore* action the signal is ignored by the process, which continue its execution flow.[40] Other than these default actions, a process can be instructed by the application developer to handle the different signals with specific functions, called *signal handlers*, and act following predefined directives. If a process is terminated by a default action, then its exit status returned to the parent process is equal to the number of signal that caused its termination. [41]

Num.	Name	Default Action	Description
-11	SIGSEGV	coredump	Segmentation fault: the process tried to access a memory address which is out of is memory mapping: i.e., a bad address is present as an instruction operand or as content of special registers such as PC, SP and others.
-4	SIGILL	coredump	Illegal Instruction: the process tried to execute an illegal, malformed or privileged instruction. This can happen if an OPCODE of an instruction gets corrupted.
-7	SIGBUS	terminate	Bus Error: it occurs when the process tries to access a memory location that the CPU cannot physically address, e.g. non-existent address, unaligned access, paging errors. This signal can be also generated due to an hardware fault.
-6	SIGABRT	coredump	Abort: the process handled internally some bad behavior and raised itself this exception using some specific instruction (e.g.*svc* instruction), which tells the process to terminate.
-5	SIGTRAP	coredump	Trace/breakpoint trap: it occurs when the executed instruction causes the processor to enter Debug state, e.g. *bkpt* instruction.
-8	SIGFPE	coredump	Arithmetic operation error: it occurs when the process tries to execute an erroneous arithmetic operation (e.g. division-by-zero). This can happen if one of the operand is corrupted.
-9	SIGKILL	terminate	Kill signal: it can be sent from a process to another to tell the receiving process to terminate the execution.
-15	SIGTERM	terminate	Termination signal: the receiving process immediately terminates its execution.

Table 3.1. Common Linux OS Signals

Chapter 4

Developed Fault Injection Platforms

In this chapter I'm going to discuss the followed methodology and the developed tools. The main objective has been to create a SEE analysis environment monitoring and controlling the behaviour of a well-known system, namely an ARM Cortex-A9 microprocessor. This latter has been chosen as hardware of interest for reasons related to the HERA mission, to which this research is a small contribution. The analysis has been conducted by emulating the effects of SEU on the Static RAM of the Processing System (PS) and on the core internal registers during the execution of an application SW. For this purpose, two different software platforms have been developed and several fault models have been considered. Both of the developed environments are implemented using python3 and run on the target ARM microprocessor through a Linux-based Operating System (OS), including a set of automatic tools to study the outcomes of the tests and investigate the possible causes of failure.

4.1 Fault models

One of the fundamental steps that has been faced is the selection of the fault models to be considered. According to the fault model of interest, a different testing approach has been developed. The addressed fault models are listed here below:

- SEU in Main Memory
- MBU in Main Memory
- "Clear Content" Fault in Main Memory
- SEU in CPU Registers
- MBU in CPU Registers
- "Clear Content" Fault in CPU Registers
- "Preset Content" Fault in CPU Registers

4.1.1 Single Event Upset (SEU) in Main Memory

When a computing system is exposed to heavy particles hazardous environment, each of its components is potentially vulnerable to unwanted malfunctionings. Among the different parts of the system that could go through upsets there is surely the system main memory. The considered system hardware model - namely PYNQ-Z2 Development Board - integrates a 512MB DDR3 RAM as main memory and that is the location where all the executable binaries are loaded right before their execution by the PS.

In a real case scenario, it could happen that a certain cell of the random access memory gets hit by a heavy particle, leading to the upset of an information bit through a Single Event Upset (SEU). Possible effects coming from this fault involve mainly the execution of the application that contains the corrupted bit, resulting in abrupt terminations and non-nominal behaviours in general. A further discussion could be done regarding the presence in the system of an Error Correction Code

(ECC): this refers to a whole class of techniques that permits the detection and correction of certain kinds of errors in memory, such as single bit or multiple bits alteration. The integration of these techniques obviously implies a certain amount of computational effort and, most of the times, a non-negligible physical space occupation overhead in the silicon implementation. These aspects make the adoption of these error correction instruments a key variable and not an obvious choice when designing a computing system, depending also on budget and power consumption project limits.

Nevertheless, depending on the implementation and integration of an ECC controller, a non-negligible amount of upsets could still escape the detection/correction, and this is an additional reason supporting the decision to consider this fault model.

4.1.2 Multiple Bit Upset (MBU) in Main Memory

During a heavy ions strike, the hitting particle traveling through the silicon structure is able to generate secondary energetic particles, as a result from the impacts with other nuclei. These secondary ions could still be highly energetic and this can lead to further unwanted effects in the nearby of the interested silicon region. From a practical point of view, when talking about memory devices this can results in alterations that involve more than one memory element, leading to the upset of multiple bits of information. As presented in [42][43], several multi-bit upset configurations can be observed depending on the energy and type of particles and also on the hitting bodies angle of incidence. Some examples of MBU clusters can be found in Fig.4.1.

Also in this case, a further discussion can be made about the implications deriving from the presence or absence of an ECC system. In fact, depending on the type and on the implementation of an error detection system, several multi-bit upsets configuration could be either detected or not. This is also due to the fact that, when using ECC techniques, multiple errors in memory are more difficult to be detected and corrected with respect to single errors.

It has been decided to consider clusters of two consecutive bits being upset. This

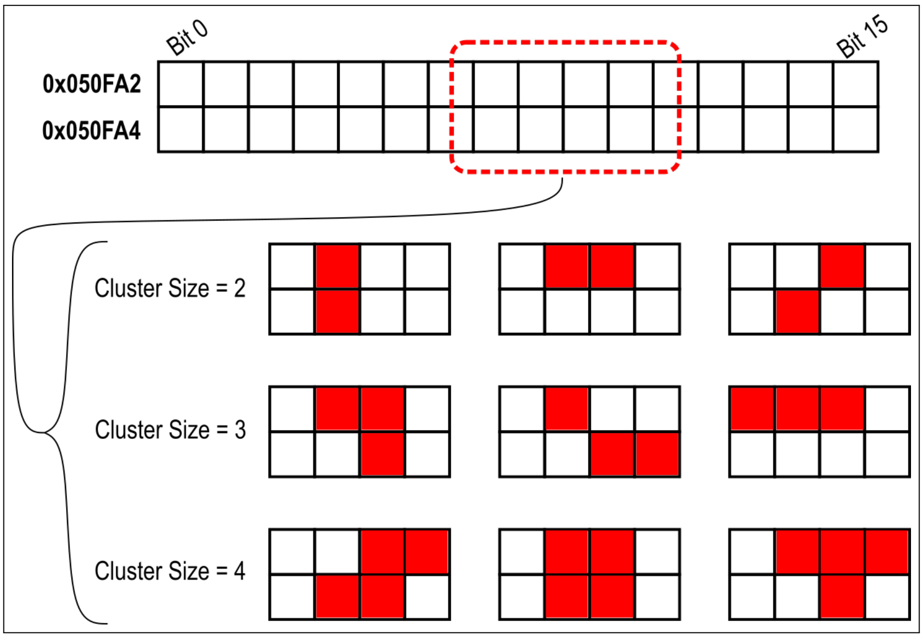

Figure 4.1. Possible MBU cluster configurations, as observed in [42]

specific fault configuration has been chosen among several possible ones, as shown in Fig.4.1 also thinking about the higher occurrence probability with respect to other configurations.[42]

4.1.3 "Clear Content" Fault in Memory

Among the elements that compose a memory device, one of the most critical part is the control logic. This circuitry has the fundamental role of managing the memory device ad all the other parts surrounding it, controlling several signals and decoding the instructions coming from the PS and other components of a computing system. When this part of the memory system gets corrupted either by a SEU or a MBU, the possible outcomes can be very different and can lead to very differentiated scenarios. Among these latter we could find the situation where instead of reading the correct value of a memory element (e.g., a memory word), a value equal to zero is read.

Other possible elements which failure can lead to this type of fault are the Memory Management Unit (MMU) and the clock signal of the system itself. Regarding the first of the two, it is a dedicated logic system, external to the memory, which role is to manage the translation between virtual addresses - computed and processed by the PS - and physical addresses. It could happen that, if a SEU or MBU occurs in the MMU, a certain virtual address gets translated into a wrong physical address, possibly leading to the fault model described in this section.

Another possible cause for this type of fault can be found in a fundamental element of the overall computing system: the clock signal. In particular, during the strike of Single Event Transients (SET) an unwanted voltage burst - also called clock glitch - could appear in this very signal and the MMU or the control logic of the SRAM could misinterpret a rising or a falling edge of the synchronization signal, leading to the transmission of an erroneous data value. [44][45] Concerning this fault model, the most vulnerable parts are the MMU, the clock signal and the control logic of the different memory units. An explanatory drawing about the considered fault locations can be found in Fig.4.2.

4.1.4 Single Event Upset (SEU) in CPU Registers

Crucial elements of a processing system are the ones that permit fast calculations and efficient data manipulation: these are the CPU registers. In modern micro-architecture, the registers are usually implemented as fast Static RAM (SRAM) blocks, having dedicated input and output ports in order to achieve minimum reading and writing latency. Another possible way to implement CPU registers resides in the use of Flip Flops composing logical structures suitable for this type of data storing and manipulation.

In a processing unit there could be different types of registers, according to the specific functions they carry out. The PS registers can be divided into two main domains: *user-space* and processor or *architecture-space*. The first category contains all the registers that are accessible by the user through the operating system, using specific software like debuggers, architecture-specific applications or by developing

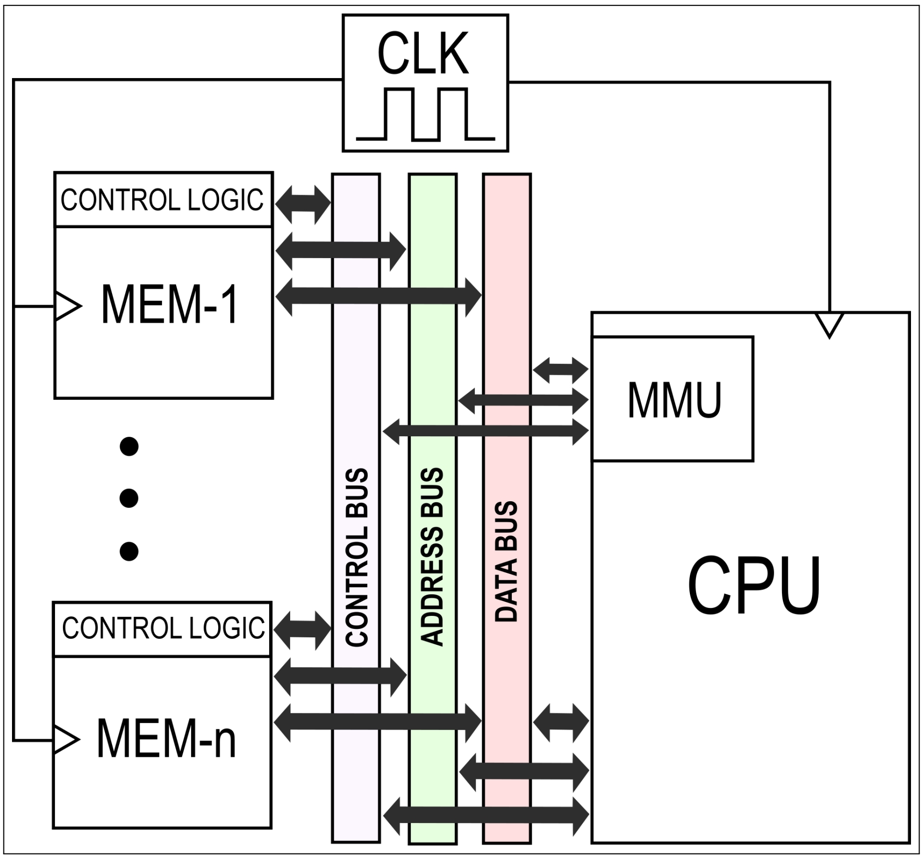

Figure 4.2. Memory "Clear Content" main fault locations

software using low level programming languages, such as Assembly.[46]

The user-accessible registers are often divided into two main categories, according to the function they carry out: there can be general-purpose and special-purpose registers. The first ones are used for manipulating data, storing operands, results of instructions but also memory addresses and other types of data. The special-purpose registers are instead used to store more specific information, such as instructions, memory instruction addresses, control flags and others. Among the most common special-purpose registers we find the Program Counter (PC), the Stack Pointer (SP), the Link Register (LR) but also state registers for keeping track of the operating modes of the CPU and for specific modules like the Floating Point Unit (FPU).[46]

An example of user-space registers can be found in Fig.4.3 referring to the ARMv7 micro-architecture, which has been the target of the conducted tests.[27]

The registers that lies in the architecture-space are instead visible and accessible only by the CPU itself during certain operating modes and, sometimes, by the OS kernel: these are control registers, state registers and hardware-specific registers, manipulated and accessed to conduct operations that most of the times are invisible to the user.

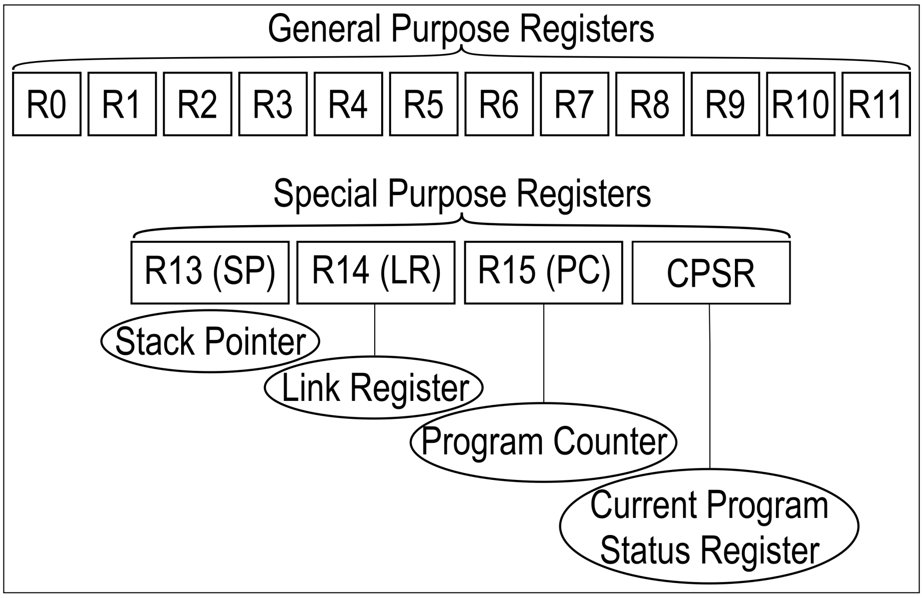

Figure 4.3. ARMv7 Architecture User Registers [27]

When an SEU occurs in these memory locations, the effects can be widely spread across multiple scenarios. For example, if a single bit of the PC gets corrupted, this could lead to the execution of an erroneous instruction. If the Stack Pointer register get modified, the resulting condition could induce a wrong return point for a function being executed. These two unwanted behaviours are obviously correlated with subsequent problems in terms of security and system stability.

Just like the special-purpose registers, also the general-purpose registers are vulnerable to this type of fault. In fact, if an operand stored in a register gets abruptly altered this can result in an illegal instruction exception or maybe a wrong addressing offset can be produced, leading to a paging exception or a segmentation fault.

With this fault model I addressed all the above described situations and, with a specific fault injection tool, all the ARMv7 architecture user registers will be randomly injected with single event upsets, evaluating and characterizing all the possible outcomes.

4.1.5 Multiple Bit Upset (MBU) in CPU Registers

Just like it happens for system main memory, due to the implementation nature of the registers it could happen that as a result of a single highly energetic ion strike multiple memory cells get altered simultaneously. This could depend on different factors like the incidence angle of the particle and its energetic level. In fact, based on these (and other) characteristics, the particle traveling through the silicon structure could generate secondary particles that will collide with the surrounding molecules, interacting with them and possibly alterating other memory elements (i.e., other information bits).

With this fault model, I considered the situation where two in-line adjacent bits gets corrupted and inverted at the same time. This specific situation has been chosen also thinking about the occurrence probability of this type of MBU cluster with respect to other configurations, as shown in Fig.4.1.[42]

4.1.6 "Clear Content" and "Preset Content" in CPU Registers

Depending on the implementation choice for the registers, one could have several different types of control logic. When an SRAM implementation type is involved, the control logic circuitry - similarly to what happens with system main memory elements - is responsible for monitoring and driving the control and data signals of the registers. If instead the CPU registers are implemented through the use of Flip Flops (FF), also in this case some kind of control system is required to properly interact with the registers. In both cases, there are several situations that could occur in the case of a Single Event Transient (SET) involving some part of the control logic. In particular, the here-defined "Clear Content" and "Preset Content" refer to malfunctionings that could occur as a result of this eventuality.

The Clear Content fault model addresses the situation where, during the execution of an application, the whole content of a register gets corrupted and all the information bits get forced to 0. On the contrary, the Preset Content fault model considers the unwanted situation where all the bits contained in a register get forcefully put to 1.

There are several possible fault conditions that can result in these described malfunctionings. Starting from the SRAM register implementation, we could have a Clear Content or a Preset Content fault if some part of the related control logic gets involved in a SET: this can cause misinterpreting of several control signals that can result in the described faulty situations. If instead the registers are implemented through the use of FF, there could be specific signals directly manipulating the whole content of the register, namely the CLEAR and the PRESET control signals. Depending on the implementation at silicon level, if a SET affects one of these two signals or some other part of the surrounding control logic then it could happen that one of the two situations described above (all-0s and all-1s)occur.

A typical configuration for three common types of asynchronous FF - namely Set/Reset, D and J/K - is reported in Fig.4.4.

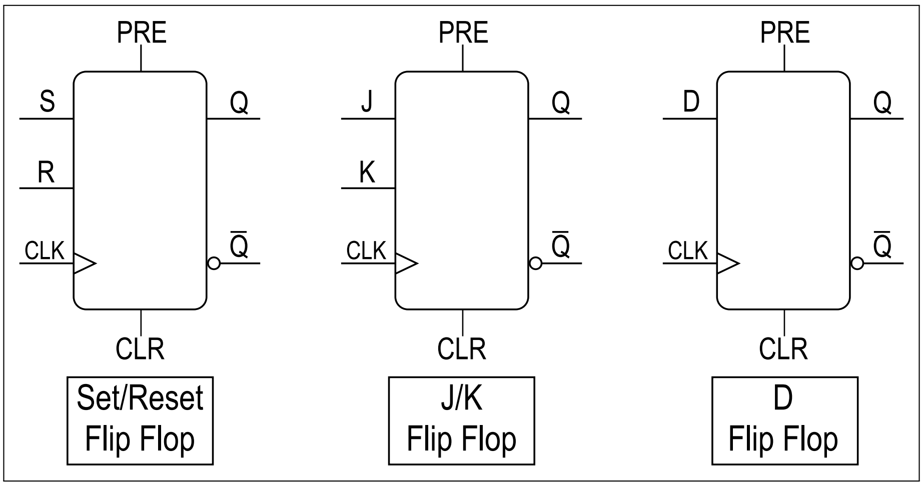

Figure 4.4. Common configurations for asynchronous Flip Flop

4.2 Fault Injection Platforms

To achieve a complete characterization of all the described fault models, two radically different injection approaches have been studied and developed to address separately memory and registers injections. Both require an executable application SW as minimal input data. An optimal input data is represented by the source code of the SW under test: starting from this, the executable file is then compiled integrating debug symbols from the source code and therefore providing extra information to the platform, which will exploit this data to provide even more in-depth report data.

Without loss of generality, without compromising the test outcomes and for sake of simplicity and test speed, both the platforms are developed using python3 and they have been directly run on the target PS. This is possible using an OS, namely Ubuntu 18.04, based on version 5.4 of Linux kernel.

Before describing the two approaches, it is important to highlight that basing on the addressed fault model, the injection procedures can slightly differ: for example, in the case of CLEAR - line or register - fault model there is no need of choosing a specific bit to be flipped, since the entire value stored in the register or the memory word will be altered by the platform. Nevertheless, the general approach of each developed injection tool remains the same as it is described in the following sections.

4.2.1 MBIP: Memory Bitflip Injection Platform

This first approach is based on the emulation of SEU in memory through the alteration of the executable file prior its execution. In this way, the binary code to be loaded in system SRAM by the PS is a corrupted version of the original fault-free compiled application SW. As a result, the final situation is the same as if a SEU occurred in system memory during execution.

As can be seen in Fig.4.5 the platform operational flow can be divided in these main phases:

1. Execution of golden application file
2. Creation of faulty executable files
3. Execution of faulty executable files
4. Exit code/return status collection and classification
5. Application results collection and classification

In the first phase, the platform runs the original executable file - possibly compiling it first from source code - and the fault-free application result is stored as golden reference. Secondly, the developed tool takes the original executable file, chooses a random bit from its binary content and inverts its value. To permit the repetition of the same injections over different test sessions, the tester is also allowed to provide a specific randomization seed in the platform input parameters. The obtained altered file is saved, and this injection process is repeated as many times as the tester requested, through a dedicated input parameter. The next phase consists of the execution of all the corrupted SW binary files and this is done by exploiting multiprocessing features provided by python3. The exit code and the return status of these sub-processes are then collected and classified by the main process and, finally, the faulty applications results are compared with the golden reference produced by the fault-free SW file. More in-depth information about output classification can be found in Section 4.3.

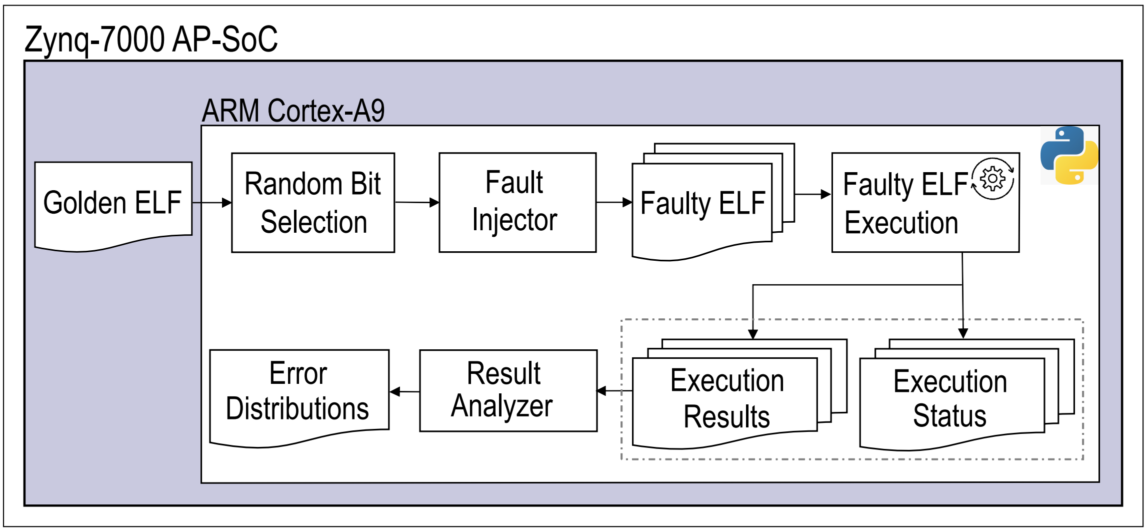

Figure 4.5. Memory Bitflip Injection Platform (MBIP) Workflow

4.2.2 RBIP: Register Bitflip Injection Platform

This second approach is closer to a real scenario and resorts to the alteration of CPU register values during the very phase of SW execution, as it'd happen in an operational condition.

The PS under test is based on the ARMv-7 architecture and since this is a totally software-based approach the addressed registers are all the "software accessible" ones.[27] More precisely, they are: the Floating Point Status and Control Register (FPCR), the 64 NEON technology registers [26], the general-purpose registers (R1 to R13) and three special-purpose registers, namely Program Counter (PC), Link Register (LR) and Stack Pointer (SP).

The operating principle of RBIP is described in Fig.4.6 and can be summarized in the pseudo-code reported in Fig.4.7.

As a first stage of the algorithm, as done in MBIP, the fault-free application is run, and the obtained golden result is saved for future functional analysis.

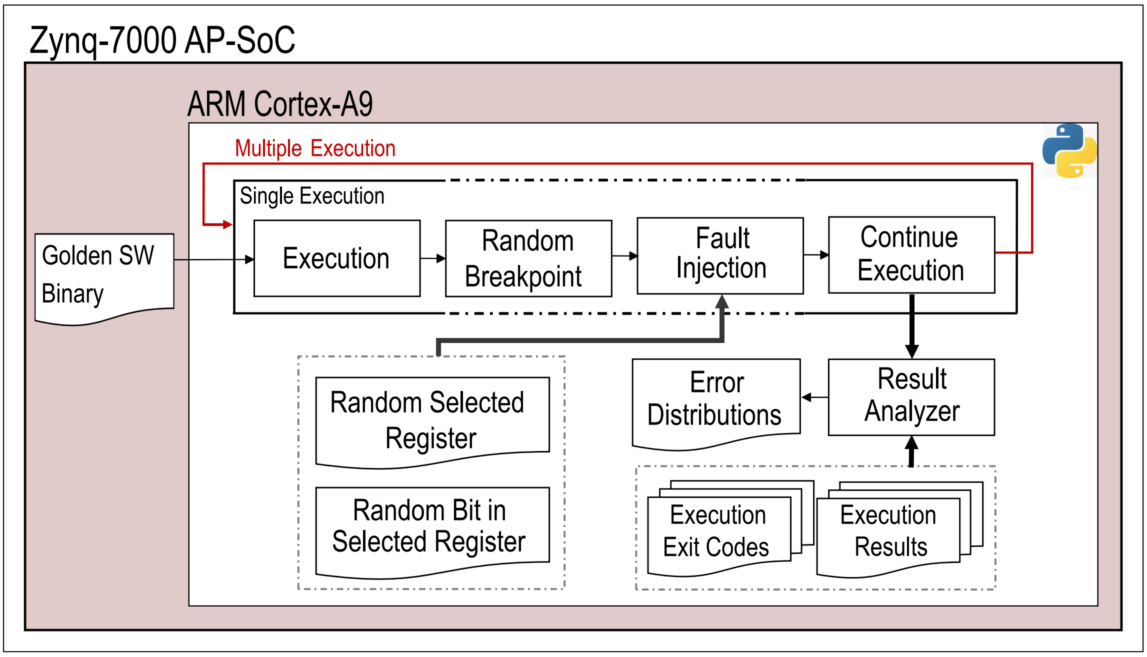

Figure 4.6. Register Bitflip Injection Platform (RBIP) Workflow

The randomization of the injection parameters - which are breakpoint, target register and bit to be flipped - is carried out as an "atomic" operation before all the injections. This is done for sake of repeatability: in fact, like in MBIP, the tester can provide a specific randomization seed to repeat the same injections over time. The breakpoint parameter is composed of two parts: firstly, a random number of source code line is selected, from zero to the total Lines Of Code (LOC) count of the application. Then, a random number between 0 and 1000 is chosen: this is the number of machine instructions (i.e., Assembly code) that will be executed starting from the line of code selected as first parameter. In this way, it is possible to achieve a good level of randomization in the breakpoint selection, comparable with a time-based approach which is instead not suitable for applications with a short execution time.[10] Once the chosen breakpoint is reached the injection takes place.

The bitflip injection phase is carried out through the use of GDB, an open-source debugger that uses low level system calls like *ptrace* to inspect and alter the

```
golden_SW_execution
SAVE(golden_SW_result)
LOOP(num_of_injections){
        random_register ← random_select(register_list)
        random_breakpoint ← breakpoint_selection()
        random_bit ← random_int(1..32)
}
LOOP(num_of_injections){
        GDB{
                SW_start
                SW_stop(random_breakpoint)
                BITFLIP(random_register, random_bit)
                SW_continue
                SAVE(exit_code)
        }
        SAVE(SW_result)
}
execution_analysis(exit_codes)
functional_analysis(SW_results, golden_SW_result)
```

Figure 4.7. Register Bitflip Injection Platform (RBIP) Pseudo-code

value of core resources and to control the flow of the monitored application.[47] In particular, it is responsible for starting the program, stopping it at the randomly chosen breakpoint, inject the bitflip in the target location and then let the program continue the execution. All these operations are done automatically without user interaction and this is possible using a python extension API, which can be integrated in GDB during its installation.

The platform terminates its operations with the execution analysis and the functional analysis: these processes are analogous to the ones carried out in MBIP and have the main purposes of analyzing the exit-code and compare the obtained results with the golden one, respectively.

4.3 Classification Criteria

In this section we'll describe what are the different data that have been collected, classified and analyzed by the platforms in order to produce a human readable report at the end of test operations.

First, it is useful to underline that the selected classification criteria are shared by both platforms and also among all the considered fault models: this permits a rapid and clear comparison of the obtained results, as described in the subsection devoted to Experimental Analysis. An example of a test report - which format is in common with both tools and the different fault models - is reported later in this section, in Fig.4.8.

The most important collected data are the exit codes of the injected executions. In fact, in Linux based OS, each process can terminate its execution in basically two ways. Without any error, the exit code of the process is 0 and this indicates that the program executed normally, and no fatal exception occurred. If instead the exit code is different from 0, this indicates that the SW produced some kind of exception that couldn't be masked or safely handled by the OS. When the OS receives an exception signal from the microprocessor, an exception handling function is called. Then, after further investigations conducted by the OS about the nature of the malfunction, a process signal is sent to the process being run. This latter is instructed to either handle the signal and continue the execution or avoid handling the signal and perform the signal default action, which can be the termination of the process itself. These different behaviours are based on the specific meanings of the signals but also on the application that is being run.[40]

In Linux systems, if not otherwise instructed by the SW developer, the exit-code of a process that prematurely finishes its execution is equal to the signal number received by the process itself. By collecting and classifying these exit codes, at the end of each test session we achieve to have a clear picture of how many processes crashed and, most importantly, what are the most common causes of these abrupt terminations.

The other main element that has been collected for each process is the application functional result. Regardless of the nature of the SW output - it could be a string or a number - all the results are stored into different binary files. After the exit-code classification, a functional analysis is conducted and the result files coming from the faulty executions are compared with the golden data, obtained from the fault-free application run. Also in this case, useful information can be extracted: in particular, it is possible to deduce how many of the processes produced a faulty output data among all the obtained ones. Furthermore, by combining this deduction with the results of the exit code analysis, the platform is able to compute how many times the SW terminated without errors and still produced an incorrect functional result: this situation is also referred to as Silent Data Error (SDE) and it is usually a big concern in functional testing.

```
###TEST_PARAMETERS###
prog_name = basicmath
injection_num= 1000
rand_seed= 1618995355
###EXECUTION_ANALYSIS###
distinct_exit_codes= 4
|0|SUCCESS| 988
|-11|SIGSEGV| 10
|-4|SIGILL| 1
|-6|SIGABRT| 1
terminated_deadlock= 0
###FUNCTIONAL_ANALYSIS###
no_file= 0
correct_results= 978
faulty_results= 22
sde= 10
```

Figure 4.8. Register Bitflip Injection Platform (RBIP) Pseudo-code

4.3.1 Crash Analysis

When investigating the effectiveness of a mitigation method or studying more in-depth what are the causes of a specific crash, it is often useful to analyze the core-dump files. These latter are produced by any process that is terminated by a signal having *coredump* as default action.[40] A process core-dump file contains all the information that are available related to the state of the processor at the time of the process abrupt termination, i.e. the crash. Contains data such as the memory mapping of the process, the registers state, the file mapping of the process and others: these data are all useful to comprehend the cause of the crash and, possibly, what are other subsequent implications due to a specific registers state, reached because of the crash.

To exploit this precious information, both the developed tools look for the core-dump file in the file system, for each of the crashed processes. The location of the core-dump files is not fixed and depends on the operating system and on the installed software packages. In this case, a specific command has been exploited to retrieve the core-dump file, i.e. *coredumpctl*[48].

Once this process-specific file is obtained, the developed tool uses GDB to analyze it and report some pre-selected data. These are:

- **Memory Mapping**. It is crucial to see which portions of memory were associated to the process and which not, permitting to understand whether an application tried to access a memory area that wasn't designated to it or not
- **Crash Instruction**. This is the instruction that caused the crash and it is another key point to be studied, because through this data it is possible to analyze, for example, which instructions are more likely to produce a crash and why;
- **Registers State**. To better understand the state of the system at the time of the crash, here is another fundamental information to be studied. In fact, it allows a more in-depth analysis of the processor, permitting to see the values in every register and to detect possible corrupted or illegal values for the

instruction that would have been executed.

An example of crash analysis report can be seen here below in Fig.4.9.

```
----------INJ_NUM=48----EXITCODE=-11(SIGSEGV)----------
Memory mapping at runtime:
Address   Kbytes Mode  Offset   Device    Mapping
00010000     316 r-x-- 00000000 0b3:00002 bitcnts_inj48
0006e000      12 rw--- 0004e000 0b3:00002 bitcnts_inj48
00071000     136 rw--- 00000000 000:00000   [ anon ]
b6fff000       4 r-x-- 00000000 000:00000   [ anon ]
befdf000     132 rw--- 00000000 000:00000   [ stack ]
ffff0000       4 r-x-- 00000000 000:00000   [ anon ]
mapped: 604K    writeable/private: 280K    shared: 0K

Crash Instruction: 0x0001c316 <+182>: str r1, [r3, #8]

Registers state:
r0             0xe      14
r1             0x0      0
r2             0x40000000       1073741824
r3             0x493e0  300000
r4             0x0      0
r5             0x0      0
r6             0xbefff3d8       3204445144
r7             0xbefff370       3204445040
r8             0x6f010  454672
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x107    263
sp             0xbefff370       0xbefff370
lr             0x1c62f  116271
pc             0x1c316  0x1c316 <sysmalloc+182>
cpsr0x800f0030  -2146500560
fpscr0x0        0
d0{u8={0x6e,0x63,0x68,...f32={0xffffffff,0x0},f64=0x0}
d1{u8={0x0,0x0,0x0,0x0,...f32={0x0,0x0},f64=0x0}
d2{u8={0x0,0x0,0x0,0x0,...f32={0x0,0x0},f64=0x0}
d3{u8={0x0,0x0,0x0,0x0,...f32={0x0,0x0},f64=0x0}
...
d29{u8={0x0,0x0,0x0,...f32={0x0,0x0},f64=0x0}
d30{u8={0x0,0x0,0x0,...f32={0x0,0x0},f64=0x0}
d31{u8={0x0,0x0,0x0,...f32={0x0,0x0},f64=0x0}
```

Figure 4.9. Example of crash report

Chapter 5

Experimental Analysis

In the first part of this chapter I'm going to describe the environment that has been set up for the tests, by means of chosen benchmark applications and the reasons behind the related choices. The second part of this chapter will instead illustrate in detail the obtained experimental results, highlighting some interesting considerations.

5.1 Benchmark Setup

The chosen SW under test has been taken from the MiBench benchmark suite, developed by the University of Michigan.[49] This collection contains several types of application and has the main purpose of testing the performances of the hosting computer. It is divided into different main categories, each one related to the fields of use of the contained programs. The chosen benchmark programs are *basicmath*, *bitcount* and *FFT*. This choice was made for sake of execution easiness and more importantly, for the close correlation with a space mission case scenario, where most of the computation effort is more likely devoted to mathematical calculations and signal processing operations. Nevertheless, the dynamic instruction distribution of the three SW varies significantly, as can be seen in FIg.5.1.

basicmath

This program carries out basic mathematical operations that usually do not have a dedicated part of hardware: these are cubic functions solving, angles conversion and other tasks. All these calculations can be useful, for example, to compute kinematic and dynamic equations of terrestrial or space systems, as well as for their autonomous and automatic control. The output of this program is a set of messages related to the results of all the different mathematical operations performed during the execution.

bitcount

Bit manipulation capabilities are a key part of a computing system and this benchmark tests this very kind of tasks. In this application several counting algorithms are used to check the number of bits in an array of integer, whose length can be imposed by input parameters. The produced countings are then returned as textual output of the application.

FFT

As can be deduced from the name of the algorithm, this benchmark program compute the Fast Fourier Transform (FFT) of an input signal composed of pseudo-random sinusoidal components, having variable amplitudes and frequencies. Almost certainly, in every system that must transmit and receive signals an FFT utility is present and used and that is the reason why this application has been selected for testing. The output of this program is composed of real and imaginary part of the frequency components of the input signal.

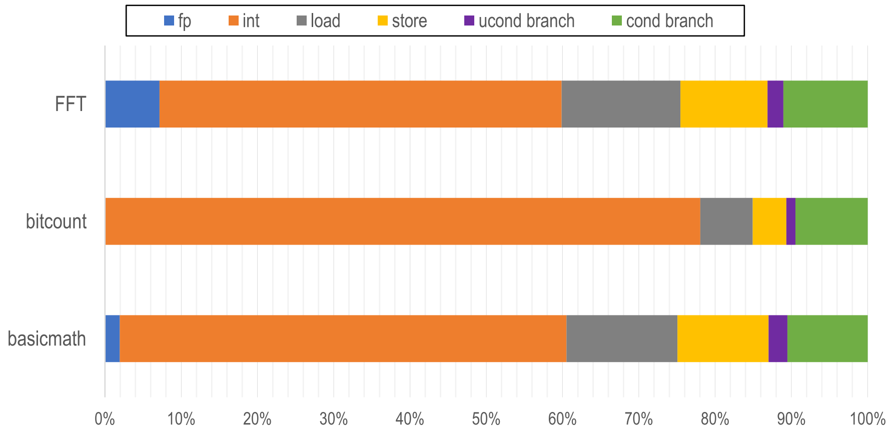

Figure 5.1. MiBench Suite: Selected SW Instructions Distribution

5.2 Experimental Results

In this section the main obtained results will be presented and discussed, putting emphasis on what has been reasonable and predictable and what has been unexpected and more interesting. First of all, it is important to understand how the fault-injection campaigns have been conducted, in order to better comprehend the results and their importance. An important question that I faced while starting these campaigns was: how many injections are enough? To take this decision, several campaigns with increasing number of injections have been conducted and the results were varying until around 10'000 injections. Above this limit, the obtained results were stable around certain values. Trying also with different fault models, the behaviour was pretty much the same, so it has been decided to choose 10'000 as minimum number of injections for each of the campaign.

Each fault-injection campaign has been characterized by two factors: the benchmark program under test and the considered fault model. Making some calculation: having 3 benchmark programs and 7 fault models to be addressed, during the experimental phase a total of 210'000 injections have been carried out. Using the developed platforms the test time could vary a lot. The two main reasons for this are the execution time of the benchmark programs under test - which subsequently depends on the input dataset - and the nature of the two fault-injection methods itself. In fact, when using the memory injection platform, 10'000 injections took roughly 10 minutes on average, but this time goes up to 3 hours in the case of injections in registers resources.

5.2.1 Error Rate Analysis

To tackle the analysis of the massive amount of obtained data it has been decided to first detect the *total error rate*. This value has been computed as the percentage of processes that unsuccessfully terminated their execution. In this category fall the processes that correctly terminated the execution but failed to produce a correct result but also the ones that have been interrupted by an OS signal and prematurely terminated their execution. This last situation is also usually referred to as *crash*. The total error rates related to the memory injection platform are reported in Table 5.1.

Application	**Total Error Rate** [%]		
	SEU	MBU	Clear Content
basicmath	6.08	6.59	6.89
bitcount	1.65	1.68	1.38
FFT	3.94	4.01	4.32

Table 5.1. Total Error Rate - Memory Fault Models

Looking at the values in the table above, the total error rates appear reasonable, increasing as the intrusiveness of the fault type increases. Going more in detail about this analysis, in Fig.5.2 are reported the distributions of the unsuccessful processes for the different memory fault models. Here we can see that, when considering the main memory, these distributions vary a lot, depending on the related benchmark application.

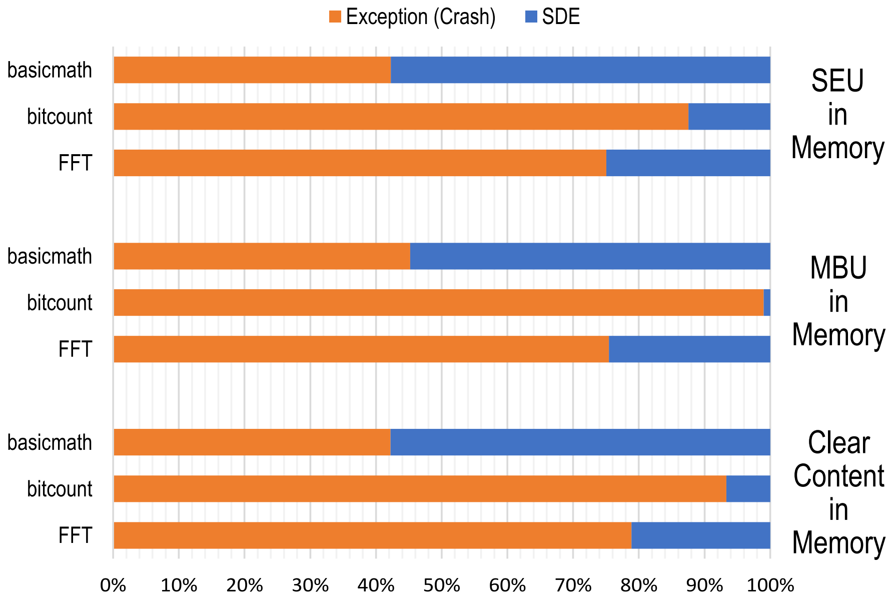

Figure 5.2. Error Rate Distribution - Memory Fault Models

The same considerations has been made also for the register fault models. In particular, in Table 5.2 and Fig.5.3 one can find the total error rate and the failure distribution between SDE and *crashes*, respectively, for the register resources fault injection campaigns that have been conducted.

Starting to analyze Table 5.2, there are several points that are worth to be highlighted. As a first observation, here the average total error rate is bigger with respect to the injections in main memory. This denotes that the misbehaviors due to a fault affecting a register are in general more severe and less tolerable: the probability that this will result in an unsuccessful execution is higher. Another key observation is related to the *Clear Content* and *Preset Content* fault models: it is clear that for these two types of fault the total error rates depend a lot on the application that is being run. As a result, one can observe the massive error rate (61.55%) related to the Clear Content fault model when executing *bitcount*, denoting the predictable vulnerability of the counting algorithms against this type of fault. A further point of interest can be seen in Fig.5.3, where it is clear that the highest probability of having an SDE is related to the Clear Content fault model.

Application	Total Error Rate [%]			
	SEU	MBU	Clear Content	Preset Content
basicmath	7.76	8.1	8.43	9.53
bitcount	4.16	4.5	61.55	1.33
FFT	6.11	11.01	6.49	8.2

Table 5.2. Total Error Rate - Registers Fault Models

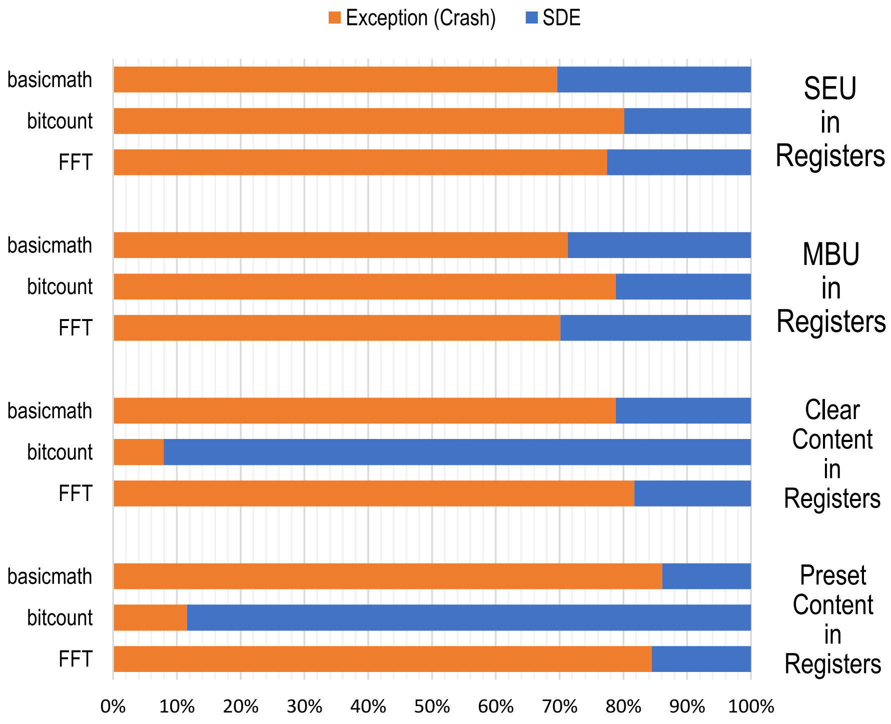

Figure 5.3. Error Rate Distribution - Registers Fault Models

5.2.2 Exceptions Analysis

The main goal of this subsection is to analyze in-depth the distribution of the exceptions that occurred for the processes that prematurely terminated their execution. This analysis is important for many reasons. Firstly, if we understand which is the most common exception for a certain fault model, then we can better assess the criticality of the malfunction and possibly focus the mitigation effort towards certain components of the application with respect to others. This permits to increase the fault tolerance together with the comprehension of which could be the recovery options in case of failure.

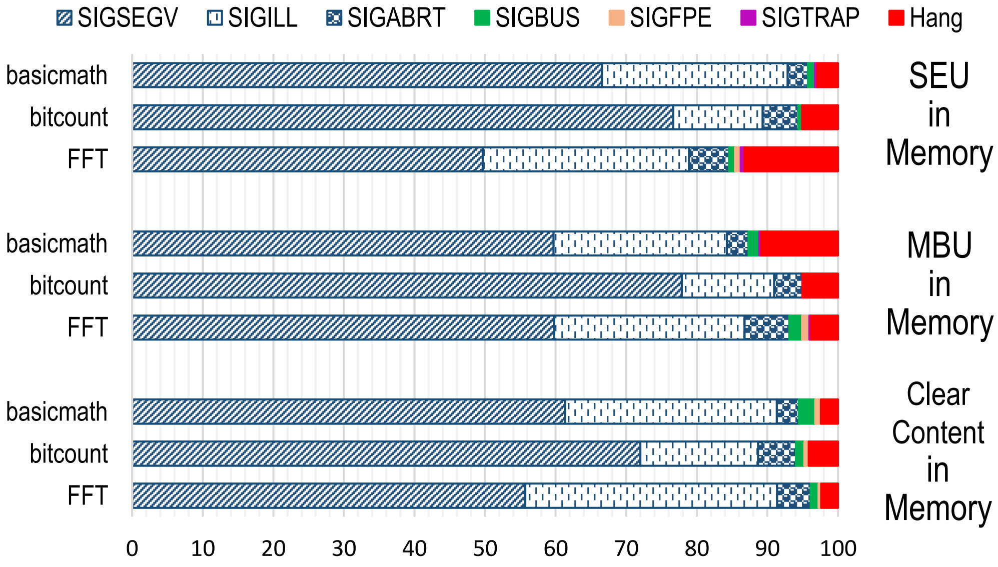

Figure 5.4. Exception Distribution - Memory Fault Models

In this section three important charts are reported: these are Fig.5.4, Fig.5.5 and Fig.5.6. The first two charts are presenting basically the same information that is presented in the third one but this latter proposes a different and interesting "point of view". In Fig.5.4 and Fig.5.5, in fact, the exception distribution has been reported for each considered fault model, so that it is possible to comprehend which are the most common exception for each of fault type. One of the most interesting aspects about these charts is the clear predominance of the *segmentation fault* as

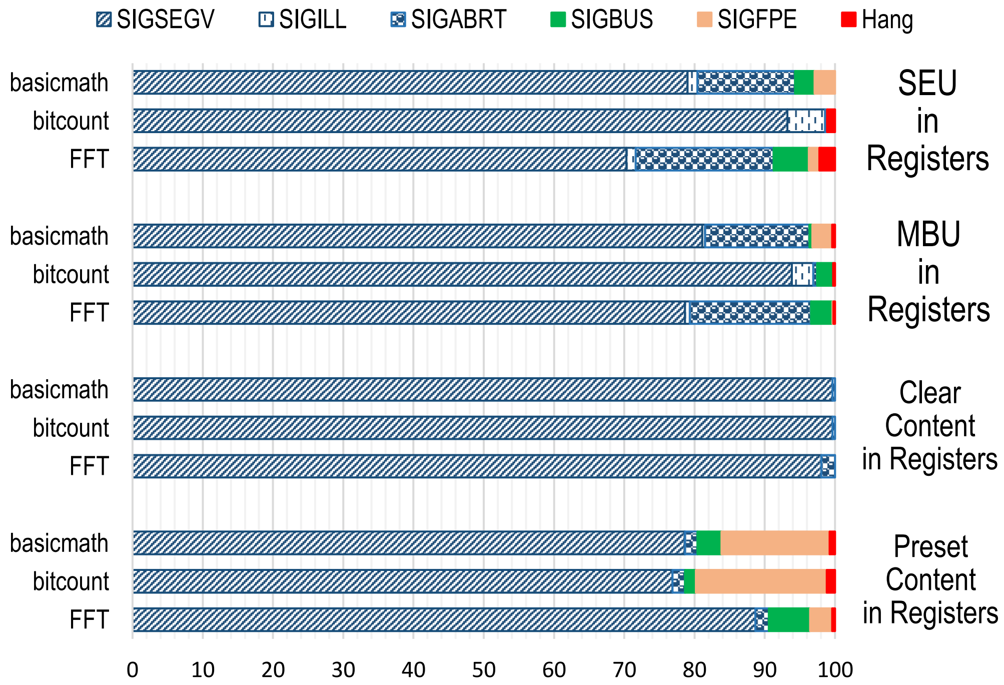

Figure 5.5. Exception Distribution - Registers Fault Models

the most common error with respect to the others. In particular, when considering the "Clear Content" fault affecting a register resource, this error appears almost every time a process terminates due to this malfunction.

In Fig.5.6 instead, a different point of view is presented. Here we can see, for each error type, which fault type is the most probable cause. In this way we're able, for example, to maximize the efficiency of studies which objective would be to find the cause of certain errors. As an operational example, let's imagine a computing system operating in an harsh radiation environment. Our application, running on that system, suffered several critical failures due to *illegal instruction* (SIGILL) exceptions: in this case, looking at a chart like Fig.5.6 we can immediately see that the most common cause for this exception is related to the Clear Content fault affecting the main memory. As a result, it is likely that one of the part of the system that has to be tested - and possibly radiation hardened - will be the main memory control logic.

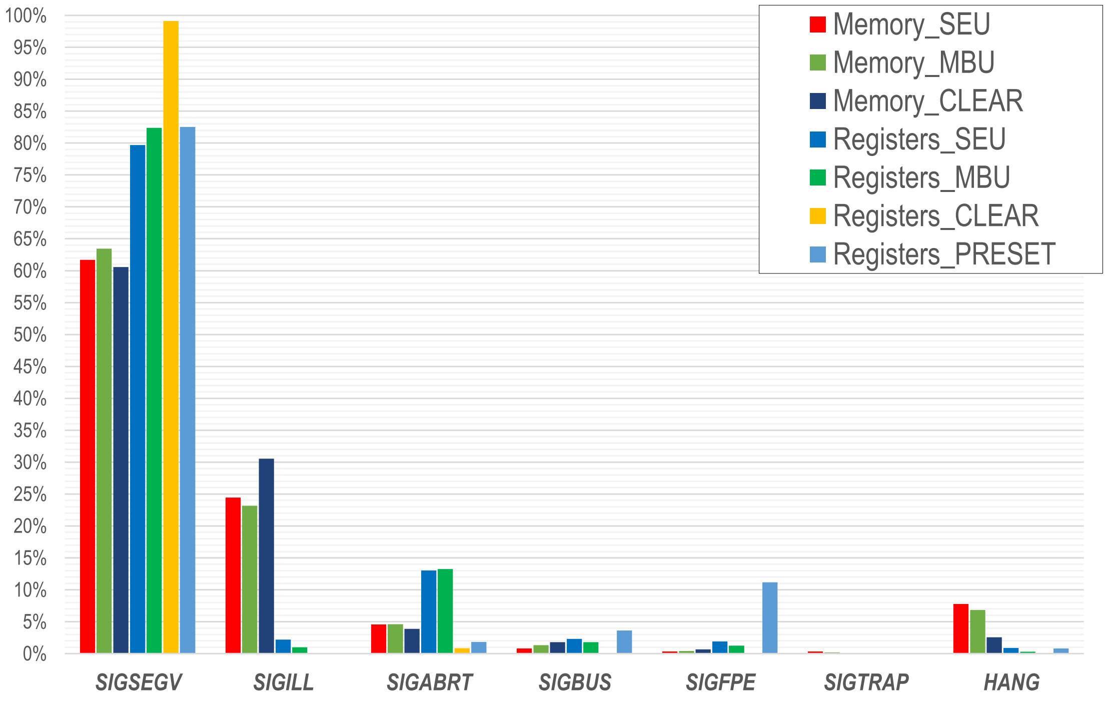

Figure 5.6. Fault Models Contribution - Comparison

Another key point that is worth to be highlighted is related to the Register Preset Content fault model: this type of malfunction is the one with the highest rate of Floating Point Unit (FPU) exceptions with 11.2% out of the total number of interruptions, while the other fault models follow with significantly lower rates, equal or below 2%.

Finally, a third interesting result concerns the number of hang processes. The fault model that most often is responsible for this condition is the one related to SEU in system memory, leading to a hang process roughly once every ten crashed executions. Other fault models follow with rates around 6% and lower.

Chapter 6

Conclusions

In this thesis, a radiation effects analysis environment have been developed, able to emulate different SEE faults on both the memory and register resources of a well-known AP-SoC, namely Zynq-7020, while an application SW is executing on the embedded ARM Cortex-A9 microprocessor. To achieve this, two *ex novo* fault injection platforms have been developed and tested. These tools exploit standard Linux Operating System features - like multi-processing management - and architecture specific characteristics together with the GNU Debugger (GDB).

Each of the presented tools have been designed around the system under test. Nevertheless, these platforms have been developed in a portable fashion: with few modifications the two injection environments are easily adaptable also to other systems and architectures. For this reason the development of the tools is still ongoing and will continue with the related work that is already planned for the future, when the environment will be used to investigate other systems using different architectures and related benchmark programs.

After the development, the fault injection campaigns took place. A secondary objective of these campaign was to verify each of the platform features and correct them in case of errors or inconsistencies. Several corrections have been applied to make the output data more readable and useful for further analysis. Moreover, as a main objective, an in-deep evaluation of the fault injection campaign data has been performed, in order to investigate the effects of the injected faults on the system and to study the causes that provoked the applications abrupt terminations. To do so, one key feature that has been developed is the *crash analyzer*: this subtool permits to analyze the *coredump* data collected by the Linux OS for each of the processes that prematurely terminated the execution. In this way, a complete characterization of each terminated process is possible, with in-detail information about the microprocessor core condition at the time of the crash: this capability could be (and will be) exploited for future works.

6.1 Future Work

These new platforms appeared to be useful also for testing and validating new innovative mitigation techniques, permitting to comprehend which types of faults can be mitigated and which not, without the need of a radiation test conducted in a dedicated facility. Regarding this type of test, another main application of the developed tools is the preparation that has to be done *prior* the radiation test itself. In fact, one of the objectives for the future is to use an improved version of these software platforms to investigate the behavior of a system before testing it under the effect of a real highly energetic particles beam. This will permit to predict which parts of the system are likely to be more vulnerable than others and which instead are probably going to appear as more robust.

Some part of this thesis can be integrated with further research work. For example, to better characterize the vulnerability of a system through the developed instruments, it'd be recommended to consider more benchmark programs. More importantly, it'd be useful to choose the benchmark applications among the ones that would be used in an operational scenario, according to the system under test and its uses. As said, this research has been conducted as minor contribution to the ongoing mission HERA, by ESA: in this context many applications could be eligible for this type of analysis and the obtained results could be useful to estimate the fault tolerance of several key modules of an aerospace system such as, for example, the flight computer of a Cubesat. By combining the obtained data with other simulation results, like mission specific prediction and simulations about Total Ionization Dose (TID), fluence and LET, one could comprehend and assess the failure risks and tackle the questions about how to deal with them.

The possible applications of this research work go beyond the aerospace field. In fact, in these recent years an ever-growing interest is being devoted to the High Performance Computing (HPC) and many studies have been made to investigate the possibility to exploit clusters of computing nodes to create highly efficient and fault tolerant systems, to be used in critical applications where dependability is a key requirement. As a future work it is planned to use the developed fault injection environment in order to study and validate efficient mitigation strategies for embedded computing nodes in HPC applications.

Bibliography

[1] L. Sterpone, Boyang Du, and Sarah Azimi. «Radiation-induced single event transients modeling and testing on nanometric flash-based technologies». In: *Microelectronics Reliability* 55 (Aug. 2015). DOI: 10.1016/j.microrel.2015.07.035.

[2] G.C. Cardarilli et al. «Bit flip injection in processor-based architectures: a case study». In: *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002)*. 2002, pp. 117–127. DOI: 10.1109/OLT.2002.1030194.

[3] L. Sterpone et al. «A 3-D Simulation-Based Approach to Analyze Heavy Ions-Induced SET on Digital Circuits». In: *IEEE Transactions on Nuclear Science* 67.9 (2020), pp. 2034–2041. DOI: 10.1109/TNS.2020.3006997.

[4] Robert A. Reed et al. «Physical Processes and Applications of the Monte Carlo Radiative Energy Deposition (MRED) Code». In: *IEEE Transactions on Nuclear Science* 62.4 (2015), pp. 1441–1461. DOI: 10.1109/TNS.2015.2454446.

[5] Felipe Rosa et al. «A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability». In: *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. 2015, pp. 211–214. DOI: 10.1109/DFT.2015.7315164.

[6] R. Velazco, S. Rezgui, and R. Ecoffet. «Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code Emulating Upsets) injection». In: *IEEE Transactions on Nuclear Science* 47.6 (2000), pp. 2405–2411. DOI: 10.1109/23.903784.

[7] Qining Lu et al. «LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults». In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, pp. 11–16. DOI: 10.1109/QRS.2015.13.

[8] Z. Segall et al. «FIAT - Fault injection based automated testing environment». In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.* 1995, pp. 394–. DOI: 10.1109/FTCSH.1995.532663.

[9] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. «FERRARI: a flexible software-based fault and error injection system». In: *IEEE Transactions on Computers* 44.2 (1995), pp. 248–260. DOI: 10.1109/12.364536.

[10] Vasileios Porpodas. *ZOFI: Zero-Overhead Fault Injection Tool for Fast Transient Fault Coverage Analysis*. 2019. arXiv: 1906.09390 [cs.DC].

[11] Daniel Oliveira et al. «CAROL-FI: An Efficient Fault-Injection Tool for Vulnerability Evaluation of Modern HPC Parallel Accelerators». In: *Proceedings of the Computing Frontiers Conference*. CF'17. Siena, Italy: Association for Computing Machinery, 2017, pp. 295–298. ISBN: 9781450344876. DOI: 10.1145/3075564.3075598. URL: https://doi.org/10.1145/3075564.3075598.

[12] *CAROL-FI GitHub Repository*. https://github.com/UFRGS-CAROL/carol-fi. Accessed: 2021-05-17.

[13] Ninghan Tian, Daniel Saab, and Jacob A. Abraham. «ESIFT: Efficient System for Error Injection». In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*. 2018, pp. 201–206. DOI: 10.1109/IOLTS.2018.8474160.

[14] Martin Hiller, Arshad Jhumka, and Neeraj Suri. «PROPANE: An Environment for Examining the Propagation of Errors in Software». In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '02. Roma, Italy: Association for Computing Machinery, 2002, pp. 81–85. ISBN: 1581135629. DOI: 10.1145/566172.566184. URL: https://doi.org/10.1145/566172.566184.

[15] Siva Kumar Sastry Hari et al. «Relyzer: Application Resiliency Analyzer for Transient Faults». In: *IEEE Micro* 33.3 (2013), pp. 58–66. DOI: 10.1109/MM.2013.30.

[16] Leonardo Passig Horstmann and Antônio Augusto Fröhlich. «A Fault Injection Framework for Real-time Multicore Embedded Systems». In: *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*. 2020, pp. 1–8. DOI: 10.1109/SBESC51047.2020.9277864.

[17] J. T. Wallmark and S. M. Marcus. «Minimum Size and Maximum Packing Density of Nonredundant Semiconductor Devices». In: *Proceedings of the IRE* 50.3 (1962), pp. 286–298. DOI: 10.1109/JRPROC.1962.288321.

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

[19] ECSS Secretariat. *Space Engineering - Methods for the calculation of radiation received and its effects, and a policy for design margins*. Standard ECSS-E-ST-10-12C. Noordwijk, NL: ESA-ESTEC, 2008. URL: https://ecss.nl/standard/ecss-e-st-10-12c-methods-for-the-calculation-of-radiation-received-and-its-effects-and-a-policy-for-design-margins/.

[20] 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.

[21] Matthew J. Gadlage et al. «Digital Device Error Rate Trends in Advanced CMOS Technologies». In: *IEEE Transactions on Nuclear Science* 53.6 (2006), pp. 3466–3471. DOI: 10.1109/TNS.2006.886212.

[22] Nadia Rezzak et al. «SET and SEFI Characterization of the 65 nm SmartFusion2 Flash-Based FPGA under Heavy Ion Irradiation». In: *2015 IEEE Radiation Effects Data Workshop (REDW)*. 2015, pp. 1–4. DOI: 10.1109/REDW.2015.7336733.

[23] K. Mori, H. Yamada, and S. Takizawa. «System on Chip Age». In: *1993 International Symposium on VLSI Technology, Systems, and Applications Proceedings of Technical Papers*. 1993, K15–K20. DOI: 10.1109/VTSA.1993.263614.

[24] D. Flynn. «AMBA: enabling reusable on-chip designs». In: *IEEE Micro* 17.4 (1997), pp. 20–27. DOI: 10.1109/40.612211.

[25] *ZYNQ-7000 SoC Block Diagram*. https://www.xilinx.com/content/dam/xilinx/imgs/block-diagrams/zynq-mp-core-dual.png. Accessed: 2021-04-30.

[26] *NEON Programmer's Guide*. https://developer.arm.com/documentation/den0018/latest/. Accessed: 2021-04-30. 2013.

[27] *ARMv7-A Architecture Reference Manual*. https://developer.arm.com/documentation/ddi0406/latest/. Accessed: 2021-04-30.

[28] *Zynq-7000 SoC Technical Reference Manual*. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. Accessed: 2021-04-30.

[29] *AXI Documentation Webpage*. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Introduction/About-the-AXI-protocol?lang=en. Accessed: 2021-04-30.

[30] *Zynq-7000 SoC Datasheet*. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. Accessed: 2021-04-30.

[31] Erwin Setiawan et al. «Implementation of baseband transmitter design based on QPSK modulation on Zynq-7000 all-programmable System-on-Chip». In: *2017 International Symposium on Electronics and Smart Devices (ISESD)*. 2017, pp. 138–143. DOI: 10.1109/ISESD.2017.8253320.

[32] Adi Candra Swastika and Trio Adiono. «Design and Implementation of Smart Card Interface Device on Zynq-7000 System-on-Chip». In: *2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE)*. 2018, pp. 220–225. DOI: 10.1109/ICITEED.2018.8534777.

[33] Alfredo Jesus Perez-Castillo et al. «Real Time Monitoring of 3 Axis Accelerometer using an FPGA Zynq®-7000 and Embedded Linux through Ethernet». In: *2018 15th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*. 2018, pp. 1–6. DOI: 10.1109/ICEEE.2018.8533999.

[34] *PYNQ-Z2 Board User Manual*. https://dpoauwgwqsy2x.cloudfront.net/Download/PYNQ_Z2_User_Manual_v1.1.pdf. Accessed: 2021-04-30.

[35] *PYNQ-Z2 Product Specification*. https://www.tul.com.tw/images/PYNQ-Z2_PA_v2_pp_20201209_STD.pdf. Accessed: 2021-05-05.

[36] *PYNQ-Z2 Board Overview*. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: 2021-04-30.

[37] *PYNQ-Z2 Overlays - Documentation*. https://pynq.readthedocs.io/en/v2.3/pynq_overlays.html. Accessed: 2021-05-03.

[38] Andrew S Tanenbaum and Todd Austin. *Structured Computer Organization*. 6th ed. Upper Saddle River, NJ: Pearson, 2012.

[39] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating System Concepts*. 10th ed. Wiley, 2018.

[40] Michael Kerrisk. *signal - Linux Programmer's Manual*. https://man7.org/linux/man-pages/man7/signal.7.html. Accessed: 2021-05-12. 2020.

[41] W Richard Stevens and Stephen A Rago. *Advanced programming in the UNIX environment: Paperback edition*. en. 2nd ed. Boston, MA: Addison Wesley, 2005.

[42] Boyang Du et al. «Ultrahigh Energy Heavy Ion Test Beam on Xilinx Kintex-7 SRAM-Based FPGA». In: *IEEE Transactions on Nuclear Science* 66.7 (2019), pp. 1813–1819. DOI: 10.1109/TNS.2019.2915207.

[43] A. Makihara et al. «Analysis of single-ion multiple-bit upset in high-density DRAMs». In: *IEEE Transactions on Nuclear Science* 47.6 (2000), pp. 2400–2404. DOI: 10.1109/23.903783.

[44] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. «An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs». In: *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2011, pp. 105–114. DOI: 10.1109/FDTC.2011.9.

[45] Raul Dario Chipana Quispe. «Single Event Transient Effects in Clock Distribution Networks». PhD thesis. Porto Alegre: Universidade Federal Do Rio Grande Do Sul, Dec. 2014.

[46] William Stallings. *Computer organization and architecture: designing for performance*. Boston: Pearson-Prentice Hall, 2016. ISBN: 0134101618.

[47] *GNU GDB Debugger User Manual*. https://sourceware.org/gdb/current/onlinedocs/gdb/. Accessed: 2021-05-07. 2021.

[48] Michael Kerrisk. *coredumpctl - Linux Programmer's Manual*. https://man7.org/linux/man-pages/man1/coredumpctl.1.html. Accessed: 2021-06-25. 2020.

[49] M.R. Guthaus et al. «MiBench: A free, commercially representative embedded benchmark suite». In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.