UNIVERSITY

LEVEL's Degree in COURSE

LEVEL's Degree Thesis

Study and development of fault tolerant operating systems for aerospace applications

Supervisors

Prof. Luca STERPONE

Prof. NAME SURNAME

Prof. NAME SURNAME

Candidate

Salvatore Gabriele LA GRECA

MONTH YEAR

Summary

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Acknowledgements

ACKNOWLEDGMENTS

"HI"
Goofy, Google by Google

Table of Contents

List of Tables

List of Figures

Acronyms

SEU
Single Event Upset

COTS
Commercial Off-The-Shelf

FPA
Field Programmable Gate Array

ASIC
Application Specific Integrated Circuit

CLB
Configurable Logic Block

LAB
Logic Array Block

LUT
Look-up Table

HDL
Hardware Description Language

CPU
Central Processing Unit

DSP
Digital Signal Processing

CMOS

Complementary Metal-Oxide Semiconductor

Chapter 1

Introduction

In the last past years, the number of missions devoted to the exploration of the universe has increased. Predictions shows that the number of missions in the current decade is expected to be almost three times the number of missions in the previous decade, without considering low-cost and low-weight missions like the one including cubesats.

Due to this increase in the number of missions, the number of electronic devices on board has increased, and the job complexity assigned to those devices has increased as well. Nowadays, electronic instruments are used not only for navigation purposes, but also for the analysis and manipulation of data. The most advanced spacecrafts are capable of decide autonomously the trajectory to follow, or to apply some complex algorithms to the data collected before sending them back to the ground.

Whatever is the purpose of a spacecraft, from the smallest one to a complete rover exploring another planet, electronic devices must be tailored to work in a realiable way, even in a complex environment like the space, where there are many disturbances like big temperature variations or radiations, one of the most common causes of failure in the spacecraft and greatest enemy of electronic instruments.

To understand better the problem, we can start from a real-world example, a piece of history. On September 22, 2021, the ESA's INTEGRAL spacecraft autonomously entered into emergency safe mode. INTEGRAL is a space telescope for observing gamma rays, and it was launched into Earth orbit in 2002 Something catastrophic was happening for the missions itself: one of the spacecraft's three reaction wheels had switched off without warnings. This caused a ripple effect that brought the satellite to begin to rotate incontrolably.

This episode created a lot of problems for the spacecraft, and the team of engineers responsible for the INTEGRAL spacecraft had to deal with it: due to the

fact that the spacecraft was spinning, data from the spacecraft were only reaching ground control in a difficult way, and the batteries were quickly discharging because of the missing orientation of the solar panels towards the Sun. ESA was going to lose a 19-years old space telescope.

With only a few hours of energy left to save the mission, the Integral Flight Control Team, together with Flight Dynamics and Ground Station Teams started working on a solution, and with quick thinking and ingenious ideas, they found the cause of the problem and rescued the spacecraft. The root of the problem was radiations. Charged, ionised particles, from the Van Allen belt, caused a SEU in the control system of the spacecraft, deciding erroneously to shut down the reaction wheel.

This story is an example of the problems that can happen during space missions due to radiations affecting the on board electronics. From this example, we can understand how crucial is the fault tolerant analysis during all the stages of development of a new space component, in order to produce a dependable system. The concept of dependable system is a complex one, and in space missions there are mainly three factors that can affect the dependability of a system:

- *Reliability*: the probability of a system to work as expected, continuously, in a given period of time (usually it coincides with the period of time of the mission itself).

- *Availability*: the probability of a system to work as expected at a generic moment in time, in the future.

- *Safety*: the ability of a system to work in a given environment, without any risk of serious damage.

With the increasing need for protection against unwanted effects caused by radiations, since the first interplanetary mission in the 60s with the Mariner 2 mission, there have been an increasing number of studies and techniques developed to deal with the problem. At the hardware level, there are *hardware mitigation techniques*, where usage of radiation-tolerant hardware components are used and hardware created with those components is called *radiation-hard* or *rad-hard* for simplicity. In most of the cases, *COTS* (Commercial Off-The-Shelf) hardware is used, which is a hardware meant to be used in a generic environment, and on top of that logical mitigation techniques are used to protect the system from the effects of radiation. The latter solution is easier to implement, and it is more efficient than the former one.

1.1 Thesis Motivation

The main motivation for the development of this thesis is to develop some techniques to deal with the problem of radiation in the space. In particular, the main goal is investigating the outcome that can occur when a SEU faults affects the CPU of a system (like the navigation system of a spacecraft), and how to deal with them by applying some innovative ideas to enhance the system's robustness and so the global fault tolerance of the system.

The hardware model on which the techniques are developed is the FPGA. FPGAs are used on a lot more space missions nowadays than in the past, for all the reasons that make FPGAs better than ASICs, mainly due to their flexibility. Because of the complexity of space missions, flexibility is a key factor in the success of a mission, both during the development and during the operational phases.

For this thesis, the usage of FPGAs has one big advantage, among other things: randomly generated SEU faults can be injected easily without using any strange and sophisticated instruments, a PC is enough. This is crucial in the study of radiation effects: it's possible to develop a systematic way to inject faults, and they can be repeated over time in order to be able to study the effect of the same SEU with different solutions.

Chapter 2

Background

Before going further in the implemented solutions, it's better to introduce a few background concepts. In particular, concepts about how FPGAs works, what kind of radiations exists and how FPGAs are affected by them.

2.1 Hardware Technology

2.1.1 FPGA Architecture

FPGAs (Field Programmable Gate Arrays) are used in a wide range of applications, from signal processing to machine learning applications. In particular, it is an integrated circuit designed to be general purpose: after manufacturing, it has no funcionalities. It is a hardware that can be programmed to perform specific tasks.

It differs from a CPU. A CPU is an already designed hardware that is designed to do only one thing in a very optimized way: execute code, from a pre-defined Instruction Set. In this case, the action of *programming* is referred to the process of writing a series of instructions that the CPU will eventually execute. This is done by exploiting Programming Languges. A FPGA, instead, is like LEGO bricks. Each LEGO brick alone does not have any function or purpose, but when assembled (so put together with other bricks), it can be used to perform a specific task. Here, the action of *programming* is referred to the process of writing a *description* on how all the bricks will be assembled to perform the specific task we want. The description is done exploiting Hardware Description Languages (HDL) like VHDL or Verilog.

The basic FPGA design conists of I/O pads (to connect with the outside world), a set of routing channels and a set of LEGO bricks. A LEGO brick in the FPGA is a logic block (and depending on the vendor, it can be called CLB or LAB) that

can be programmed to perform a very specific task that in the overall design helps in achieving the goal of the User's Application.

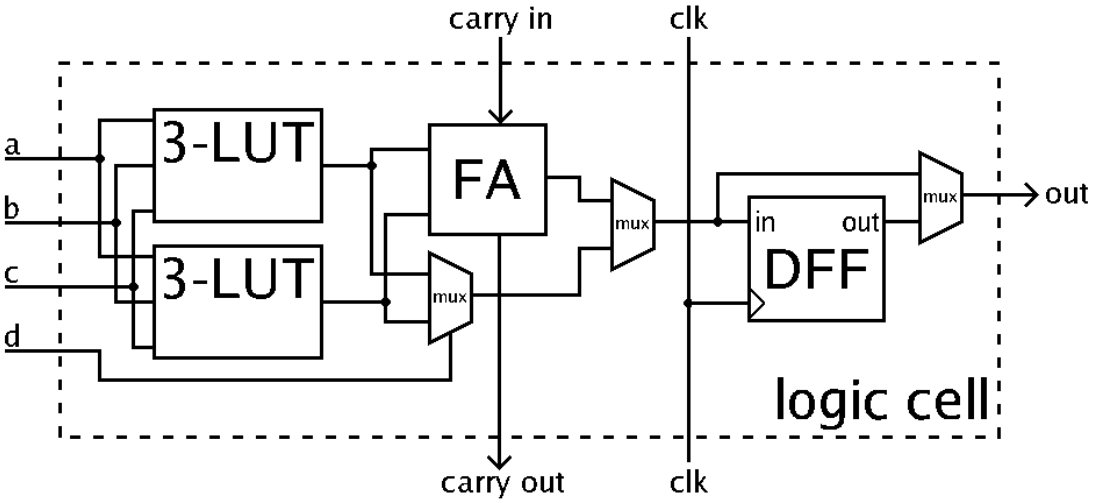

Figure 2.1: Simplified schematic of a FPGA cell

A basic logic block consists of a few Logic Elements. As shown in figure 2.1, a Logic Elements is made of LUTs, a Full-Adder (FA), a D-Type Flip Flop and a bunch of multiplexers. This particular architecture can work in two modes: *normal* mode and *arithmetic* mode. Thanks to the Flip Flop, FPGAs can implement operations where some kind of memory is required.

Modern FPGAs are very complex and expand upon the above capabilities to include other functionalities in silicon. Having these common functions embedded in the circuit reduces the area required and gives those functions increased speed compared to building them from logical primitives (because are implemented in-silicon, built out of transistor instead of LUTs, so they have ASICs-level performance). Examples of these include multipliers, generic DSP blocks, embedded processors, high speed I/O logic (like PCI/PCI-Express controllers, DRAM Controllers and so on and so forth) and embedded memories.

Once the User's Application is designed (i.e. the description of the FPGA is written), the design needs to be mapped onto the FPGA's hardware resources. This is done using the Vendor's specific software and it's in charge of deciding which FPGA's LE is assigned to which subpart of the description and how each LE is configured. Then, all the LEs needs to be connected between themselfs and the I/O pads, and this is done by routing algorithms that decides the best way to connect them. Once all the implementation steps are done, a configuration file is generated that will eventually used to program the FPGA and is called *bitstream*.

All the programmable bits (like the content of the LUTs, some multiplexers selection signals or the routing details) are stored in the FPGA in memory elements that are outside the FPGA's funcional blocks (i.e. the one that can be used by the user to implement the application). Those memory elements can be though of as a big array of bits, or a *shift register*. It's the *configuration memory*: it stores the configuration bits of the entire FPGA and is loaded with the bitstream when the FPGA itself is programmed. Most FPGAs rely on an SRAM-based approach to be programmed: this allows to be in-system programmable (so the FPGA chip can be programmed without unmounting it from the board and from the system itself) and re-programmable (can be programmed as many times we want), but require external boot devices. Because the SRAM is a volatile memory, when the FPGA is powered off, the configuration memory content is lost. An external memory where the bitstream can be retrieved is required in order to re-program it. The SRAM approach is based on CMOS.

Consequently, FPGAs are alternatives to hard-core CPUs. This means that on a FPGA a CPU can be implemented out of logic primitives (called *soft-core*), alongside with the hardware that is used to implement the application like peripherals, memory and other components. Modern FPGAs supports *at runtime programming*, this lead to the idea of *reconfigurable systems*, where for example a CPU can be reconfigured in order to enable/disable some of its functionalities to suit the task at hand. The concept of *reconfigurable systems* is also used in another manner and will be explained further in the next chapters.

2.1.2 FPGAs vs. ASICs in Aerospace Applications

An *ASIC* (application-specific integrated circuit) is an integrated circuit chip customized for a particular use. ASIC chips are typically fabricated using metal-oxide-semiconductor (MOS) technology. Thanks to the miniaturization of the MOS-based transistors and the improvement in the design tools, the maximum complexity (and hence functionality) possible in an ASIC has grown from 5000 logic gates to over 100 million.

This allows to implement entire microprocessors, memories (including ROM, RAM, EEPROM and flash) and other large component in a single chip. Usually, for lower production volumes, FPGAs may be more cost-effective than an ASIC design. This is due to the non-recurring engineering (NRE) cost of an ASIC, that can run into millions of dollars.

1. what asics are 2. how they differs from fpgas 3. why fpga are better in aerospace applications (flexibility, needs to certificate only 1 piece of chip and the hardware can change during the development of the system, ability to reprogram

from remote during operational phase) or the fact that software and hardware development can be done in parallel instead of sequentially so faster development and test and implementation time

2.2 Radiations

Chapter 3

Hello

[Hi 1, Goofy]
$\mathrm{kg\,s^{-1}}$

Figure 3.1: Hi

3.1 Extremely long name with manual linebreak which otherwise would not fit the page

1. A
2. B
3. C

Figure 3.2: HI

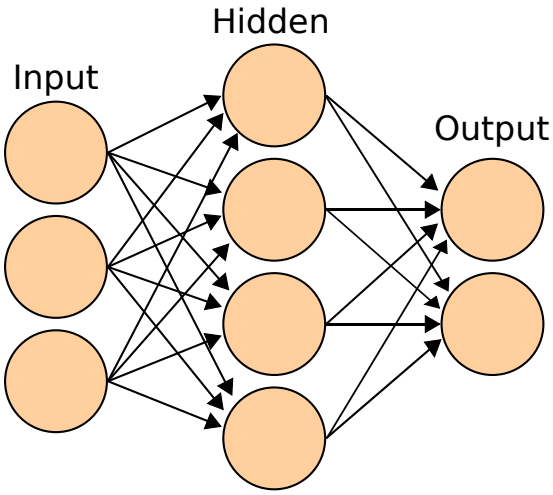

Figure 3.3: SVG

ReLU	$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$
Softmax	$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} i = 1, ..., J$
tanh	$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$

Table 3.1: Examples of activation functions, operating either element-wise or vector-wise, depending on the function

$$output = f_{activation}\left(\sum_{\#neurons} input_i + bias\right) \tag{3.1}$$

- A
- B
- C

Algorithm 1 Adam optimizer algorithm. All operations are element-wise, even powers. Good values for the constants are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. ϵ is needed to guarantee numerical stability.

procedure ADAM($\alpha, \beta_1, \beta_2, f, \theta_0$)
  ▷ α is the stepsize
  ▷ $\beta_1, \beta_2 \in [0, 1)$ are the exponential decay rates for the moment estimates
  ▷ $f(\theta)$ is the objective function to optimize
  ▷ θ_0 is the initial vector of parameters which will be optimized
  ▷ Initialization
  $m_0 \leftarrow 0$ ▷ First moment estimate vector set to 0
  $v_0 \leftarrow 0$ ▷ Second moment estimate vector set to 0
  $t \leftarrow 0$ ▷ Timestep set to 0
  ▷ Execution
  while θ_t not converged **do**
    $t \leftarrow t + 1$ ▷ Update timestep
    ▷ Gradients are computed w.r.t the parameters to optimize
    ▷ using the value of the objective function
    ▷ at the previous timestep
    $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
    ▷ Update of first-moment and second-moment estimates using
    ▷ previous value and new gradients, biased
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
    ▷ Bias-correction of estimates
    $\hat{m}_t \leftarrow \dfrac{m_t}{1 - \beta_1^t}$
    $\hat{v}_t \leftarrow \dfrac{v_t}{1 - \beta_2^t}$
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \dfrac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ ▷ Update parameters
  end while
  return θ_t ▷ Optimized parameters are returned
end procedure

MSE / L2 Loss / Quadratic Loss	$\frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{N}$
(Binary) Cross Entropy (average reduction on higher dimensions)	$\frac{\sum_{i=1}^{N} \sum_{j=1}^{C} \hat{y}_i \log (y_{i,j})}{N}$
Categorical Cross Entropy (sum reduction on higher dimensions)	$-\sum_{i=1}^{N} \hat{y}_i + \log \left(\sum_{i=1}^{N} \sum_{j=1}^{C} y_{i,j} \right)$

Table 3.2: y is the output of the network, N is the batch size multiplied by the number of outputs (e.g. pixels), C is the number of classes and $\hat{y}$ is the correct output.

Appendix A

Galileo

```
    import os
    os.system("echo 1")
```

$\mathcal{O}\left(n \log n\right)$
`numpy`

Bibliography

[1] S. Zhang, C. Zhu, J. K. O. Sin, and P. K. T. Mok. «A Novel Ultrathin Elevated Channel Low-temperature Poly-Si TFT». In: 20 (Nov. 1999), pp. 569–571 (cit. on p. 8).