POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Study and development of fault tolerant operating systems for aerospace applications

Supervisors

Prof. Luca STERPONE

Ing. Daniele RIZZIERI

Ing. Sarah AZIMI

Candidate

Salvatore Gabriele LA GRECA

July 2022

Summary

In the last few years, the number of missions devoted to the exploration of the universe has increased. Predictions show 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 ones including CubeSats. Therefore, the number of electronic devices and the job complexity assigned to them is increasing as well.

Electronic devices must be tailored to work in a reliable way. Whatever is the purpose of a spacecraft, from the smallest one to a complete rover exploring another planet. Particularly, in a complex environment like space, where there are many disturbances such as diverse temperature variations or radiations. The latter is one of the most common causes of failure in spacecrafts and greatest enemy of electronic components. Thus, a system needs to be as dependable as possible. The dependability of a system is mainly affected by aspects like reliability, availability and safety, especially for space applications.

Nowadays, FPGAs is increasingly being used in aerospace applications due to their flexibility. The flexibility given by this kind of hardware is a key aspect in the success of a mission because of their high costs, high duration and high complexity. As an example, the Mars Perseverance Rover is almost based on FPGAs. In this rover, one FPGA can be found in the automatic entry unit. This unit is responsible for the automatic entry, descent and landing on Mars. Once the rover is landed, this unit would be useless and would become a dead hardware. However, it is based on a FPGA hardware so it has been reprogrammed by NASA engineers from Earth to handle computer vision tasks.

Consequently, this thesis aims to develop some techniques to create FPGA designs tolerant to "Single Event Upset" faults (that are very common, especially in FPGAs). Taking this into consideration, the proposed solution aims to detect faults caused by SEUs in the Xilinx Microblaze CPU by using a custom peripheral. The custom peripheral has been developed in order to be fault-tolerant itself thanks to a Triple Module Redundancy design.

Finally, when a fault is detected, a partial reconfiguration of the FPGA is triggered. This action will upload a partial bitstream only in a subportion of the FPGA, aiming to reconfigure only the CPU area of the design and to restore the original behaviour. This partial reconfiguration allows to achieve a faster down-time, and consequently a higher availability of the system. This process is entirely managed by the DFX (Dynamic Function Exchange) Controller IP. The DFX Controller loads the configuration file from the memory and sends it to the configuration port of the FPGA.

Moreover, a custom script has been developed providing to designers and developers an easy and most automatized way to convert an existing Xilinx design into a design that supports the partial reconfiguration of the Microblaze.

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

TMR

Triple Module Redundancy

SEE

Single Event Effect

SOC

System On Chip

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 ones including cubesats.

Due to this increase in the number of missions, the overall number of electronic devices on board has increased, and the job complexity assigned to those devices has increased as well. Nowadays, electronic components 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 components.

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 (in particular a Xilinx Microblaze soft-core, that will be explained in details later on) 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 components, 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

General Background

Before going further in the implemented solutions, it is 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.

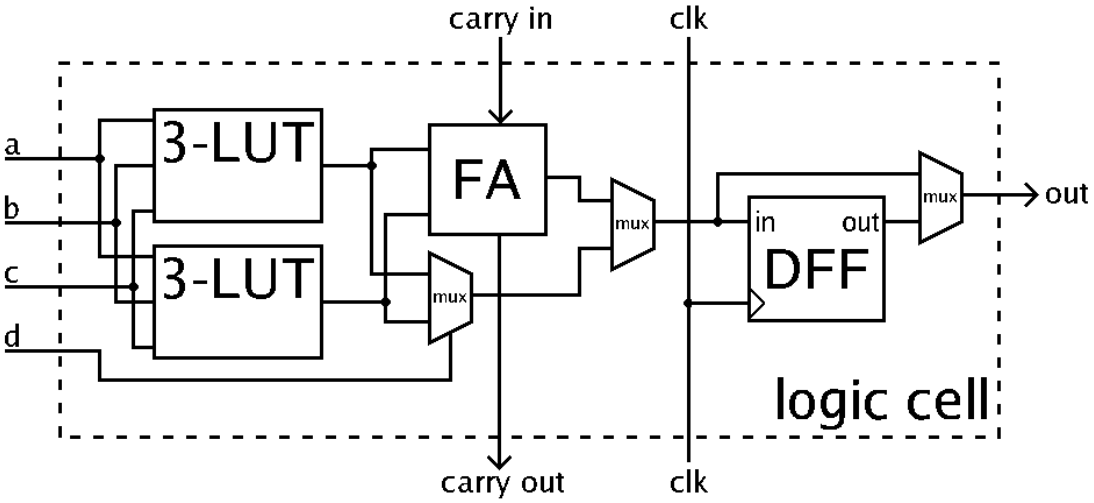


Figure 2.1: Simplified schematic of a FPGA cell

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 is 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 is 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

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.

They are designed using the same HDLs Languages as the FPGAs, but the similarities stop there. Once the description is complete, specific ASIC softwares are used to synthesize and implement onto a technology library. While the corresponding technology library in FPGAs is simpler (made of LEs and routing elements), on ASICs it is a lot more complex. A typical ASIC technology library consists of a set of basic logic gates (like 2 input NAND, 3 input OR, 2 input FA, etc.) provided by the manufacturer that will manufactur the chip. Once a HDL description is mapped on top of the ASIC library, the so called *gate-level netlist* is sent to the manufacturer. Here, ad-hoc technicians will start to work on this

netlist, doing the *route* & *place* of the netlist and as output of this process, a set of masks will be generated. The masks are used to *print* the circuit in the silicon. On top of all this process, tests engineers must prepare a set of tests that in order to test the correct functionalities of the circuit during the various stages of the manufacturing process, until the end of the process itself.

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.

To recap:

- ASICs circuits are faster, less power-hungry than FPGAs.
- ASICs are more complex to design and implement (hence more expensive) than FPGAs.
- FPGAs are more flexibile than ASICs.

2.1.3 FPGA or ASIC in Aerospace Applications?

In the aerospace industry, we are witnessing a turnaround in the last years regarding the hardware technology. FPGAs are typically much less radiation hardened than ASICs, so they are more prone to SEUs as well as lower total ionizing dose tolerance, but there are techniques to reduce these deficiencies. However, FPGAs are used on a lot more missions nowadays than 15 years ago, for all the reasons that make FPGAs a better choice than ASICs.

As an example, Mars Exploration Rovers were something like 90% ASICS. The last JPL's Martian Rover, Perserverance, is a very complex system and it is a very challenging design from the engineering point of view: it has multiple sensors and cameras to collect as much data as possible and, due to the volume of live data being recorded and the long data transmission time from Mars to Earth, a powerful processing system is essential. Early Mars rovers were basing their workload mainly on CPUs and ASICs as the processing units, while nowadays FPGAs are taking on much of the workload, like in Perseverance.

There are different reasons behind this choice. The first one is the flexibility given by their re-programmability: because of the different stages a mission is made of, some parts of the system could be useful only in some of those stages (maybe intermediate ones) and they will never be used again. This is a waste of resources:

FPGAs can be a great help in this aspect and Perserverance rover is an example. It utilizes an almost decade-old FPGA technology (Xilinx Virtex-5, introduced in May 2006 on 65 nm technology) as one of the main processing units. This unit is responsible for rover entry, descent and landing on Mars. Once the rover is landed, this unit would be useless and would become a *dead hardware*. However, it is based on a FPGA hardware so it has been reprogrammed by NASA engineers from Earth to handle computer vision tasks.

Other units on Perseverance such as radars, cameras, UHF transceivers, radar, and X-ray (used to identify chemicals) are controlled using Xilinx's FPGAs. Another interesting point is that Perserverance uses machine learning algorithm running on FPGAs, and they are so well optimized that it is achieving higher performance levels (about 18 times) than Curiosity rover (landed on Mars in 2012 and still active).

Another advantage of using FPGAs is the faster time-to-space. There are different points that help in achieving this advantage. Not only the development on FPGA is faster than on ASICs (cost of design, development and fabrication of an ASIC are not present), but the most important thing is that there are many and many changes in the processing unit is architecture during project's development phase. There is usually a very stricted launch window for the mission that can be missed, and FPGAs help in two ways mainly:

- Physically changing or adding more to a space system is a real challenge. The installation itself is not that difficult, but the system has to be recertified, proving that the it is still dependable. Furthermore, FPGAs simplify this greatly: the only thing to prove is that the FPGA chip is safe to fly with. Once this is done, the overall number of different parts to be certificied is reduced. Second, a change of the bitstream or of the software running on a *soft-core* take a lot less time to certify.

- Software and Hardware development can be done in parallel. This is a great advantage for the software development team, because a first iteration of the hardware can be prepared and ready to use be used by the software team faster and the software team can start to work on the software itself.

FPGAs are not only helpful during the development phase, but even during the operational phase. Missions are prepared to last a relatively long time, but usually the quality of the work is so high that they last much longer. Examples are Mars rovers: Opportunity landed on the Red Planet in 2003 and it was ended by a martian dust storm in 2018, so it lasted for 15 years. Curiosity in 2012 and in 2022 is still active. This is a so long period that, speaking again about

re-programmibility, the processing system architecture may require changes to let the mission continue working. In fact, different things can go wrong in a decade and having a full reconfigurable system (from remote in particular) is a must, giving ground engineers a lot more possibilities to fix the system or to add/remove components.

On the radiation tolerant side, vendors offer radiation-tolerant FPGAs. On top of that, it is possible to apply some logic changes to the design like TMR (Triple Module Redundacy) to a portion of the design or even to the entire design. Basically, it consists in triplicating the design and add a voter at the outputs. If a radiation error occurs, it will theoretically affect only one module so there will be two different results from the three modules (two correct and one wrong caused by the radiation). The voter will select the correct result (that is the majority). This is an example of making a design more robust to radiation.

2.2 Radiations

We are going to understand better why radiation effects regarding electronic devices are one of the primary concerns for the aerospace industry.

2.2.1 Radiation sources

Where does the radiation originate from? Unfortunately, the Universe and in particular the Solar System are full of radiations. The natural space radiation environment can damage electronic devices in different ways, ranging from a degradation in performances to a complete functional failure. More and more a space system goes deeper in the space, less and less it is protected by the Earth's atmosphere.

Close to the Earth, there are two three sources of radiation: the Van Allen Belts, the Sun and the Cosmos itself. Van Allen Belts are zones of energetic charged particles, that are generated for example by the Sun, and captured by the Earth's magnetosphere. By trapping those charged particles, the magnetic field deflects them and protects the atmosphere from destruction. The two Earth's main belts extends from an altitude of 640 km to 58.000 km, in which radiation levels vary. Between the two belts, the *inner* and the *outer* there is a zone called *safe zone* where the level of radiation is pretty low. Spacecrafts travelling beyond the LEO (Low Earth Orbit) go through the two belts, and beyond the belts they face additional hazards from cosmic rays and solar particle events (coronal mass ejections and solar flares).

2.2.2 Radiation problems on Earth: the Super Mario 64 glitch

Here on Earth, electronic devices are often not shielded or design to tolerate radiations. Usually, only safety-critical systems undergo the same kind of radiation-tolerant techniques as the ones used in the space system, like Aviation and Nuclear Power Plants, for instance.

Even if there is a big magnetosphere protecting the planet's surface, some charged particles still escape and travel until they reach the ground and some everyday device. In 2013, a player was challenging another player in Nintendo's Super Mario 64 game. Suddenly, Mario was teleported into the air, saving crucial time and providing an incredible advantage in the game. The glitch caused the attention of a lot of players, and a $1000 reweard was offered to anynone who could replicate the glitch. Users tried in vain to recreate the scenario, but no-one was able to emulate that giant leap. In the end, after eight years, users concluded that the glitch was not replicable because it was caused by a charged particle coming from the outer space that caused a bit-flip in the value that defines the player's height.

Another curious case was the one related to the electronic voting machine in Belgium in 2003. A bit-flip here caused an adding of 4096 extra votes to a candidate. The error was only detected because there were more preferential votes than the candidate's own list, which is impossible in the voting system. The official explanation was "the spontaneous creation of a bit at the position 13 in the memory of the computer". it is not a coincidence that the value added was exactly 4096, in hexadecimal `0x1000`, that is 2^{12}.

2.2.3 Types of radiation

The most common way to classify radiations is based on their effects on electronic devices. If the effect is the result of a cumulative damage (i.e. passage of many charged particles in different moments in time, and each particle has a relative low energy) then it can be a *total ionizing dose* or a *displacement ionizing dose*. If the effect is the result of a single charged particle (with a high energy) then it can be *destructive* or *non-destructive*, and they are usually referred as SEE (Single Event Effects).

Total ionizing dose

Most electronic devices are based on MOS transistors, forming the basis for digital logic. The common way to use those transistors is as *electronic switches*: there are

two isolated contacts, the source and the drain (i.e. the switch is off, no current). When a positive charge is applied to the gate (in the case of a NMOS transistor), electrons (that are negative charges) are allowed to pass from the two isolated contacts (i.e. the switch is on).

When ionizing radiations passes through the device, electrons are moved away from the material leaving "holes" of missing charge, acting as positive charge carriers. These holes can find their way to the gate oxide and become trapped: this phenomenon is called *total ionizing dose*. The effect of this phenomenon is the same as applying some positive voltage to the gate. With enough accumulated charges, the effect is to have the transistor always on, or better, in the *stuck-on state*.

Displacement ionizing dose

Another form of cumulative damage is the *displacement ionizing dose*. This is the effect of a single charged particle passing through the device. What happens is that an atom is displaced from the material, modifying the crystal structure of the material itself. These microscopic effects create traps and recombination centers, eventually leading to the modification of the free flow of the current. This will ultimately impact the device's performance.

2.2.4 Single Event Effects

When a single high-energy charged particle passes through the device, it can cause a *destructive* or *non-destructive* effect. The particle creates a momentary change of charge in the device, creating an unexpected current that can affect the device in various ways. Some effects may be completely destructive, while others may degrade performance to the point that the device doesn't work anymore in the limits required by the circuit or the system itself. Other effects cause the device to momentarily work in a wrong way, causing a functional failure (so it is not destructive from the point of view of the device but can cause an functional error, for example a wrong value in the memory from *0xe* to *0xf*).

Within the destructive effect, the most common are Single Event Latchup (SEL), Single Event Burnout (SEB) and Single Event Gate Rupture (SEGR).

Single Event Latchup

In CMOS technology, there are a lot of intrinsic BJT (Bipolar Junction Transistor). When a special arrangement of PMOS and NMOS transistors is used, resulting in

a n-p-n-p structures (corresponding to a NPN and a PNP transistor stucked next to each other), a CMOS Latchup structure is created. If one of these two transistor is activated (accidentally by a high-energy charged particle), the other one will be activated too, creating a feedback loop. They will both keep each other activated for a long as some current flows through them. This phenomenon will increase the current draw and can bring to the destrupution of the device. Usually, the only way to correct this situation is to make a *power cycle*, so completely shutting down the device and then restarting it. However, latent damage may exists that may not appear until later.

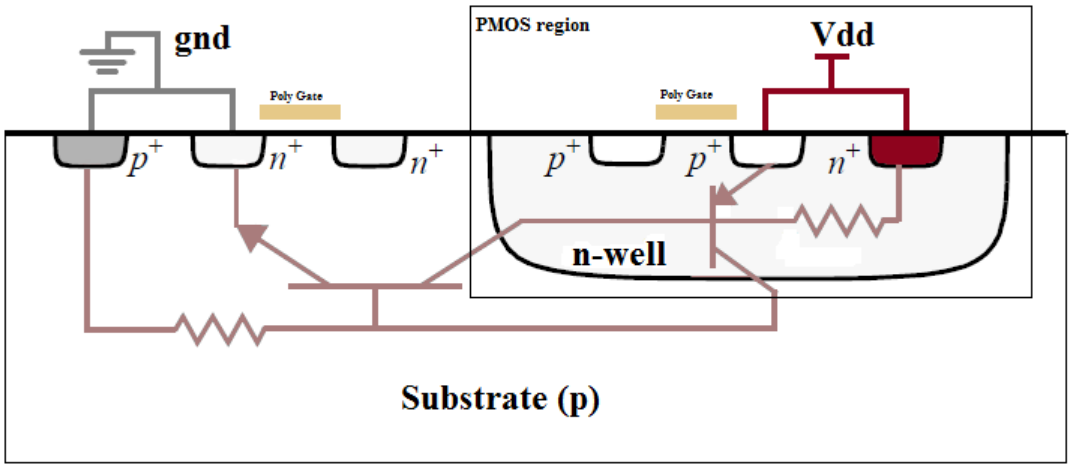


Figure 2.2: The intrisic BJTs in the CMOS Technology that can cause a Latchup. Deepon, CC BY-SA 3.0, via Wikimedia Commons

Single Event Burnout

Can happen when an incident particle initiates an avalanche charge multiplication effect. This leads to an increasing current, leading to a thermal runaway of the device, causing local melting or ejection of molten material in a small-scale explosion. Obviously, the result is a complete destruction of the device.

Single Event Gate Rupture

SEGR is the destructive rupture of a gate oxide (or any dielectric layer in a transistor). The effects can be observed in power MOSFETs with an increase of current flow when turned on, or in digital circuits with stuck bits.

Single Event Upset

This is the most common non-destructive effect. As known as *bit-flip*, it is caused by a particle that forces a digital signal to an opposite value momentarily. It can lean in a temporary modification of the digital output in a combinatory circuit, and the modified value can be memorized in a flip-flop or any other memory element if sampled at the same time a radiation arrives. In more complex circuits, it can cause other malfunctions like resets and memory values modifications.

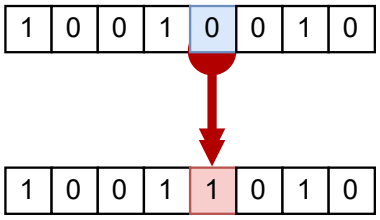


Figure 2.3: Example of a Single Event Upset in a memory element.

What is shown in Figure 2.3 can for example happen in a SRAM memory. Each cell is made of a cross-coupled transistors. Each side couple are connected forming an inverter (NOT logic function), and the output of the inverter is connected to the gates of the second couple.

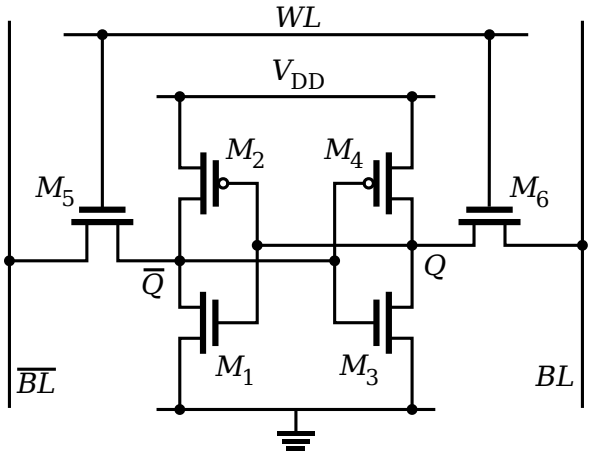


Figure 2.4: Simple SRAM Cell layout. Inductiveload, Public domain, via Wikimedia Commons.

In Figure 2.4, a simple layout is proposed. In order to have a logic 0 as output ($BL = 0$), M3 is active (thus M4 is not active). So M2 is active (thus M3 is not active). If a radiation strikes one of those transistor, can happen that the M3's

gate voltage goes low, causing a flip of the configuration thus a flip of the stored bit.

As explained in Section 2.1.1, most FPGAs' memory configuration are based on SRAM technology. If a bitflip occurs, the FPGA configuration itself is modified, leading to a malfunction of a module or to a routing modification.

The actual technology trend see a scaling down to smaller sizes, trying to pack more transistors in less area. This scaling affects how radiations modify the behavior of the devices. Those devices are generally less affected by cumulative damage, it means that total ionizing dose or displacement damages are less likely to occur due to the smaller area of each transistor so less area where charges can accumulate or material displacement. On the other hand, Single Event Effects are more likely to occur, because a single particle can hit more than one transistor, causing a more complex damage like multiple bit-flips at once.

Chapter 3

Thesis Background

This chapter is about the background of the thesis, in order to understand better further chapters and as a help and reference to reproduce the results of this thesis in the future.

3.1 PYNQ-Z2 Development Board

The *PYNQ-Z2* is a development board designed for the Xilinx University Program. It is equipped with a Xilinx ZYNQ 7020 SoC (XC7Z020-1CLG400C), 512 MB of DDR3 RAM and 16 MB of QSPI Flash Storage. The board provides a clock reference thanks to a crystal oscillator with a frequency of 50 MHz. The reference clock is used by the PS and can be provided to the PL too.

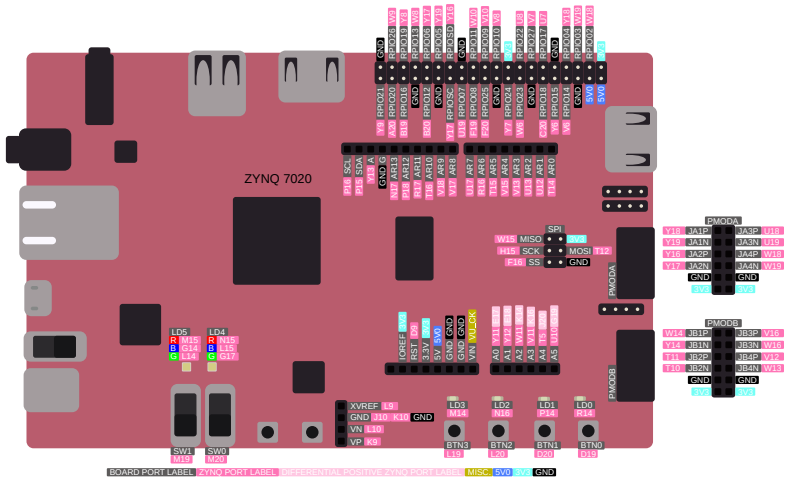


Figure 3.1: Schematic of the PYNQ-Z2 Development Board

The SoC is made of two subparts: a Processing System (PS) and a Programmable Logic (PL). The PS is the main part of the SoC, containing two 650 MHz ARM Cortex-A9 processor, 512 KB L2 Cache, 256 KB On-Chip Memory and other modules like FPUs, Flash Controller, DRAM Controller, GPIOs and so on.

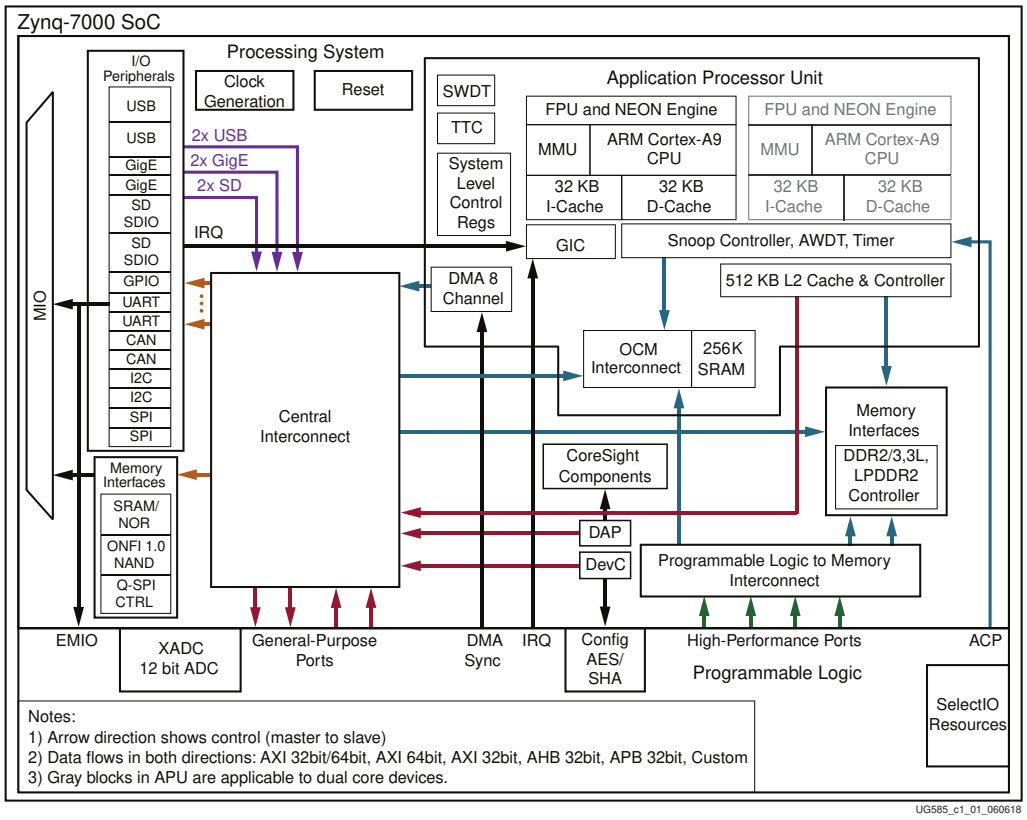


Figure 3.2: Schematic of ZYNQ 7020 SoC

A schematic is shown in Figure 3.2. The second part is the PL, which consists in a FPGA with the following characteristics:

- 13,300 logic slices, each with four 6-input LUTs and 8 flipflops
- 630 KB block RAM (BRAM)
- 220 DSP slices
- On-chip Xilinx analog-to-digital converter (XADC)

The PL can access the Processing System's memory space, as shown in Table 3.1, through a High Performance and/or General Purpose AXI Ports. This enables

the usage, for example, of the DDR3 RAM and of the On-Chip memory (OCM) from the PL. The board can be programmed through a JTAG interface, which allows to upload a firmware to be executed from the PS or to program the PL via a bitstream. Moreover, it provides a virtual UART interface that can be used as input/output both for the PS and the PL.

Memory Mapping		
Address Start	Address End	Device
0x00000000	0x3FFFFFFF	DDR & OCM
0x40000000	0xBFFFFFFF	PL
0xC0000000	0xDFFFFFFF	Reserved
0xE0000000	0xE02FFFFF	Memory mapped devices
0xE0300000	0xE0FFFFFF	Reserved
0xE1000000	0xE3FFFFFF	NAND, NOR
0xE4000000	0xE5FFFFFF	SRAM
0xE6000000	0xF7FFFFFF	Reserved
0xF8000000	0xF8FFFFFF	AMBA APB Peripherals
0xF9000000	0xFBFFFFFF	Reserved
0xFC000000	0xFDFFFFFF	Linear QSPI - XIP
0xFE000000	0xFFEFFFFF	Reserved
0xFFF00000	0xFFFFFFFF	OCM

Table 3.1: ZYNQ 7020 SoC Memory Map

3.2 Xilinx's Microblaze soft-core

The Microblaze is a soft-core (or soft-microprocessor) designed for Xilinx's FPGAs. Introduced in 2002, it is based on a RISC architecture, with an ISA (Instruction Set Architecture) similar to the DLX architecture. It is a pipelined processor and, with few exceptions, the MicroBlaze can issue a new instruction every cycle, maintaining single-cycle throughput under most circumstances.

The Microblaze has an interface to the AXI Interconnect, used to connect to other peripherals and memories. It has a dedicated bus, LMB Bus, for access to local-memory (FPGA's BRAMs): this can be used both for Instruction (ILMB) and Data (DLMB) storage.

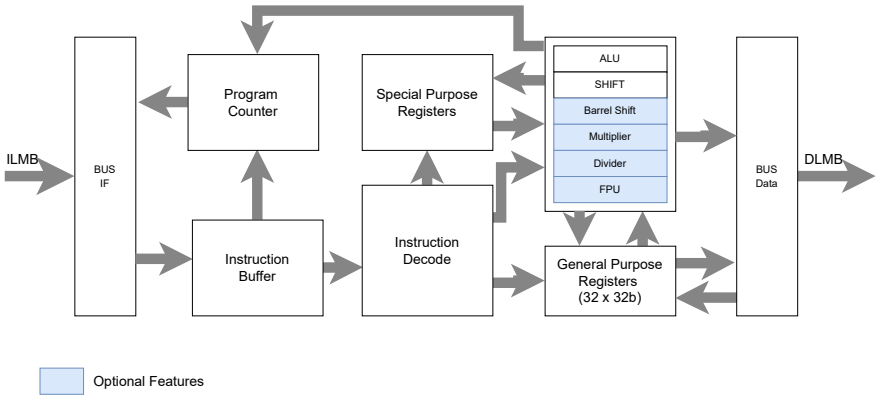


Figure 3.3: [3]Overview of a Microblaze SoftCore

A general overview of the Microblaze architecture is shown in Figure 3.3. Because it is meant for FPGAs, and FPGAs are flexible by construction, a Microblaze instance can be personalized in many ways to fit the user's needs. Example of configurations are the cache size (or the cache can be enabled or disabled at all), pipeline depth (3-stage, 5-stage, or 8-stage) and bus-interfaces. There are some presets, like the area-optimized one which uses a 3-stage pipeline and sacrifices clock frequency for reduced logic area. The performance-optimized preset expands the execution pipeline to 5 stages. One of the most important configuration is related to the supported ISA: key processor instructions which are rarely used but more expensive to implement in hardware can be selectively added/removed (e.g. multiply, divide, and floating point operations).

3.3 Xilinx FPGA Standard Design Flow

Xilinx offers a software suite for Xilinx's FPGAs. The provided software suite is *Vivado Design Suite*, and this thesis has been developed using version 2021.1. The suite supports designers in all the steps of the design process, from the initial HDL design to the final FPGA bistream generation. At each stage of the design flow, the design can perform analysis and verification, by performing logical simulations of the design, estimation of power consumption, constraints definition, I/O and clock planning, design rule checks (DRC) and modification of implementation results.

Together with the HDL description, Vivado offers an IP catalog. IP stands for Intellectual Property, and each IP is an already developed and tested design ready to be integrated into the user's own design. An example of IP offered by default is the Microblaze IP, that containts the Microblaze core. The Vivado's Catalog is a comprehensive list of all the IP offered by different repositories: Xilinx's IP, IP

obtained from third parties, and end-user designs targeted for reuse as IP into a single environment.

One of the key features of the Vivado Design Suite is the choiche given to the user to perform the design flow by means of the Graphical User Interface (GUI) or by TCL commands. The GUI, known as as *Vivado Integrated Design Environment* (IDE), allows the user to follow the evolution of the design visually from the HDL and IP instantiation up to its implementation on physical resources. The TCL commands allow the user to control the design flow by means of scripts. The interesting thing is that each action performed by the user in the GUI corresponds to an exact TCL command that can be seen from the TCL Console available in the IDE. This allow the user to understand what is the TCL command for that specific action and to script the design flow easily.

3.3.1 Steps towards the Bitstream Generation

The starting point of the design design flow is the description of the system. The description can be made of a set of HDL files (Vivado supports Verilog, VHDL and SystemVerilog), a set of design constraints (XDXC) and a set of IP instantiations.

Thus, a design can be a combination of IPs and hand-written HDL code or it can be a full IP-centric design, where the user instantiates IPs he/she wants to use and interconnects them (usually via AXI Interface but also via other interfaces or custom interfaces, it depends on the IP). For the IP-centric design flow, Vivado offers the *Block Design* tool, which allows the user to visually instantiate e move and connect IPs visually, where each IP corresponds to a block, and to connect them by drawing connections similar to a schematic or using connection automation features provided with a set of DRCs (to ensure proper IP configuration and connectivity), as shown in Figure 3.4.

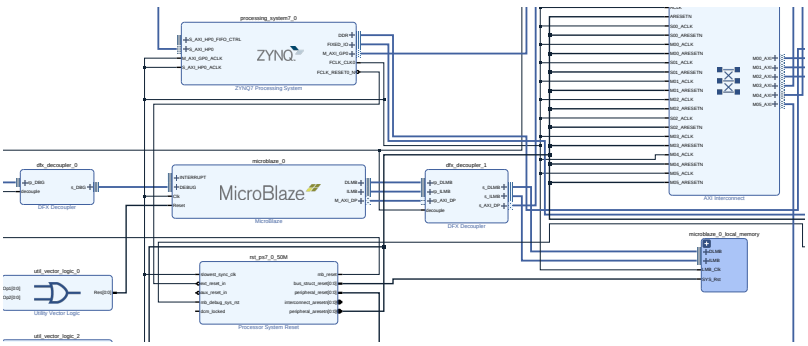


Figure 3.4: Example of Block Design

Moreover, the Block Design Tool allows the user to define the memory mapping of the AXI peripherals with respect to the AXI masters. In the example above there are two masters , the ZYNQ7 Processing System and a Microblaze, and both of them are connected to the same AXI Interconnect IP. All the other peripherals are connected to the same AXI Interconnect IP, so in the end there are two memory address spaces (one for each master) and each master will be able to access all the peripherals as the other master. The Block Design tool then allows to validate the design, the memory map correctness, and will package the design into a single design source.

Now that the design is defined, the user can proceed with a logic simulation or with the Synthesis of the design. Of course, in order to test the design, the user needs to write its own testbench. The testbench is usually a HDL file where the DUT (Device-Under-Test, that is the module the user wants to simulate) is instantiated and proper stimuli are applied. The simulator is then able to run simulate and let the user see all the waveforms.

Before going ahead with the Synthesis step, it possible to assign some constraints. Those constraints, defined in a XDC file, regards for example the PIN assignment (it is possible to assign a *port* of the design to a physical pin of the FPGA) or the placement of some modules in a particular region of the FPGA.

Once the constraints are defined, the Synthesis can be performed. The Synthesis is the process of transforming a HDL description into a gate-level representation. The output is a netlist of the whole design. Vivado performs the Synthesis in a bottom-up approach, that is, the lower modules are synthesized first, and then the higher modules are synthesized. If the design containts IPs, these are synthesized first. The user can decide the Synthesis approach adopted by the tool, as for example if the synthesis must follow a timing optimization strategy or an area optimization approach.

The next step is the Implementation. The Implementation is the final step, where the gate netlist, produced as output of the synthesis step, is mapped to the FPGA specific resources and the design is routed. The implementation step is the most complex one and is made of different steps:

- *Design Optimization*: the netlist is optimized to reduce the number of required resources and to fit the target FPGA device.
- *Placement*: each block required by the design is mapped into a physical resource of the FPGA. There are many resources available with the same behaviour where the block can be mapped. The choiche may be driven by the need to minimize or to balance the wiring accross to FPGA and/or to

minimize the circuit delay (i.e. maximize the speed). The placement tries to follow the constraints defined in the XDC file. If it is not possible to fullfill the constraints, the placement will fail and the user will be notified.

- *Post-Placement Physical Optimization*: the placement is further optimized.
- *Route Design*: the design is routed, meaning that the physical resources are connected to each other as needed.

Once the design has been implemented, the final step of the bitstream generation can be performed. The default generated bitsteram is a binary bitstream (.bit), that can be used to program the FPGA. However, the user can also generate bitstreams in different formats.

3.3.2 Fundamentals of the Xilinx's Bitstream structure

The bitstream is a file that is usually given as input to some tools that programs the FPGA, via some defined interface. Because of the different tools and interfaces used for different scenarios, the bitstream format is not always the same. The most common formats are:

- *.bit*: a binary file that contains initially a header, followed by the raw bitsteam.
- *.rbt*: same structure as .bit, but it is ASCII encoded, meaning that the header is human-readable and the raw bitstream is written as literal '0' and '1' characters for each bit.
- *.bin*: a binary file that contains only the raw bitstream.
- *.mcs*: a file that can be used to program a PROM (includes addresses and checksum info).

Even tho the .bin file contains all the necessary data for programming a FPGA, the .bit file is the default format generated by Vivado.

Bitstream Header

The header containts some informations like the design name, build date, target name and are totally ignored by the FPGA. The main reason for this format to exist is that the header is required by tools like Vivado, to better analyze it before starting the programming.

The hex dump of a .bit file header looks like the following:

```
00000000:  00090ff0 0ff00ff0  0ff00000 0161002a |.............a.*|
00000010:  64657369 676e5f31  3b557365 7249443d |design_1;UserID=|
00000020:  30584646 46464646  46463b56 65727369 |0XFFFFFFFF;Versi|
00000030:  6f6e3d32 3032312e  31006200 0c377a30 |on=2021.1.b..7z0|
00000040:  3230636c 67343030  0063000b 32303232 |20clg400.c..2022|
00000050:  2f30362f 31360064  00093133 3a34363a |/06/16.d..13:46:|
00000060:  30340065 003dbafc  ffffffff ffffffff |04.e.=..........|
00000070:  ffffffff ffffffff  ffffffff ffffffff |................|
00000080:  ffffffff ffffffff  000000bb 11220044 |.............".D|
00000090:  ffffffff ffffffff  aa995566 20000000 |..........Uf ...|
```

There are several fields in the header, each one is indicapted by a letter (*a*, *b*, *c*, *d*, *e*). The first one containts the design name *design_1*, the UserID and the Vivado version used to generate the bitsteam. The second one contains the FPGA part on which the bitstream has been generated for *7z020clg400*. The *c* and *d* are the date and time, respectively. The *e* field contains some additional information. Each letter is followed by the length of the field (including a trailing 0x00). After the header, there are few bytes that are used only to add some padding (`0xffffffff`) to the bitstream.

Raw Bitstream

Here the configuration logic starts its job. The configuration logic is part of the FPGA that can be accessed via a configuration port, and acts as a State Machine. Each value written in the bitstream is like a command to the configuration logic, that may or may not change the state machine's state.

In ZYNQ system, that are mainly two configuration ports: the *ICAP* and the *PCAP*. Both are used to program the FPGA, but the first one can be used only by the hard-cores in the SoC, while the second one can be used by the FPGA to program itself. The ICAP and PCAP are mutually exclusive, so only one of them can be used at a time. They are connected with a 2:1 mux, and the selection pin is connected to a bit in one of the configuration registers of the ARM cores.

At startup, the PCAP is enabled by default, and the ICAP can be enabled if requested. The processeor may steal the PCAP back (and stop the ICAP) at any time. This choice has been made in order to insure that the ARM TrustZone remains in control of the security of the system all the times. ICAP is a potential backdoor, and would compromise security if the processor did not have the ability to prevent and regulate its use.

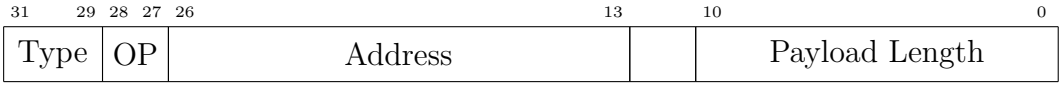
The raw bitstream in a Xilinx's 7 series FPGA consists of three sections:

- Bus Width Auto Detection
- Sync Word
- FPGA Configuration

The bus width auto detection section is a byte pattern inserted at the beginning of every bitstream. The pattern is made of `0x999999bb` and `0x11220044` and they may be surrounded by some padding bytes. The configuration width detection logic always checks the low eight bits, For the x8 bus, the configuration bus width detection logic first finds `0xBB` on the D[0:7] pins, followed by `0x11`. For the x16 bus, the logic first finds `0xBB` on D[0:7] followed by `0x22`. For the x32 bus, the logic first finds `0xBB`, on D[0:7], followed by `0x44`. If the byte after `0xbb` is not correct, the bus width detection logic's state machine is reset, until a valid sequence is found.

When it is found, it switches to the appropriate external bus width state and starts looking for the Sync Word. The sync word is `0xaa995566`. When the sync word is found, the configuration logic switches to the FPGA configuration state, and starts processing configuration packets in the bitstream. Configuration data can be sent both in serial or in parallel mode, where the bus width is fixed thanks to the previous step. Once the Sync Word is detected, the communication mode is fixed and the configuration logic will only work on 32-bit, big-endian words. Thus, the Sync Word is used to esablish a 32-bit alignment, too.

Each configuration packet begins with a one-word header. The header is composed of the following fields:



The content of the header changes according to the *Type* field. The Type 1 header is the showed one. Type 2 packets are used when the payload length exceeds the 11 bits available in a type 1 packet. Type 0 should exists, even if it is not documented.

The *OP* field is used to specify the operation to be performed. The following values are possible:

OP	Description
00	NOP
01	Read
10	Write

Table 3.2: 7 Series Configuration Packet: Type 1 Header OP Field

For NOP operations, that usualy are found as `0x20000000` in the bitstream, the address and payload length are ignored. The address field can be useful in one case: the type 2 packets does not contain any address field, to extend the payload length maximum value. Thus, the configuration logic will use the address field of the previous type 1 packet to determine the address of the type 2 packet. The flow would be a NOP packet with a valid address field followed by a type 2 packet.

Address specified in the configuration packets are mapped to variable-width registers. Some of the registers are:

Register	Address	Length	Description
CRC	00000	Fixed	Automatical updated register: when a packet is received, the configuration logic computer the CRC incrementally and updates the register.
FAR	00001	Fixed	Start address for the next read or write operation for the configuration memory
FDRI	00010	Variable	Register where configuration data are wrote. This is the real content of the configuration memory of the FPGA, the one indicating how the physical cells are used and the interconnections
CMD	00100	Fixed	Used to perform one-shot actions. For example the *RCRC* resets the CRC register or the *START* command begins the startup sequence of the FPGA when the configuration is done.

Table 3.3: 7 Series Configuration Registers

3.3.3 Software Development

3.4 Fault Injection Tool

Chapter 4

Analysis and Hardening of a FPGA Design with a soft core

4.1 How SEUs affect the Microblaze?

Explain here what are the effects of SEUs in the Microblaze.

4.2 Strategies and adopted solutions

Watchdog + DFX because..

4.3 Development of a watchdog

Beacon watchdog here

4.3.1 What is a watchdog?

A watchdog is ..

4.3.2 How to implement a watchdog?

Architecture of the watchdog (FSM)

4.3.3 How to harden the watchdog?

TMR.

4.3.4 Integration of the watchdog in the design

4.4 How to partial reconfigure a design?

4.4.1 What is and how useful is a partial reconfiguration?

4.4.2 Xilinx DFX Controller

4.4.3 Prepare a design for partial reconfiguration

4.4.4 Prepare a design with a Microblaze for partial reconfiguration

4.5 Integration of the watchdog and the DFX

4.5.1 The needed hardware

4.5.2 DFX Decoupler: why?

4.6 A script to automatize the process

Chapter 5

Experimental Analysis

5.1 Fault Injection

5.2 Experimental Results

Chapter 6

Conclusions

6.1 Future Work

Appendix A

Galileo

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

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

Bibliography

[1] European Space Agency. *Three hours to save Integral*. 2021. URL: https://www.esa.int/Enabling_Support/Operations/Three_hours_to_save_Integral.

[2] Jeffrey S. George. «An overview of radiation effects in electronics». In: *AIP Conference Proceedings* 2160.1 (2019), p. 060002. DOI: 10.1063/1.5127719. eprint: https://aip.scitation.org/doi/pdf/10.1063/1.5127719. URL: https://aip.scitation.org/doi/abs/10.1063/1.5127719.

[3] Pieter Anemaet and TV As. «Microprocessor soft-cores: An evaluation of design methods and concepts on FPGAs». In: *part of the Computer Architecture (Special Topics) course ET4078, Department of Computer Engineering* (2003) (cit. on p. 18).