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	Candidate
Prof. Luca STERPONE	Salvatore Gabriele LA GRECA
Ing. Daniele RIZZIERI	
Ing. Sarah AZIMI	

July 2022

Summary

In the last few years, the number of missions devoted to universe exploration has increased. Predictions show that the amount of missions in the current decade is expected to be almost three times the amount of missions in the previous one, without considering low-cost and low-weight missions, like the ones including CubeSats. Therefore, the total quantity of electronic devices and the job complexity assigned to them is increasing as well.

Electronic devices must be tailored to work reliably, whatever the purpose of a spacecraft, from the smallest one to a complete rover exploring another planet. Particularly, this concept holds 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 the 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 are increasingly being used in aerospace applications due to their flexibility: this is a key aspect of the success of missions because of their high costs, high duration and high complexity. As an example, the Mars Perseverance Rover is almost based on FPGAs. As an example, in the rover's architecture, an 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, it would be useless and would become dead hardware. However, it is based on FPGA hardware so it has been reprogrammed by NASA engineers from Earth to handle computer vision tasks.

Consequently, this thesis aims to study and develop some techniques to mitigate errors induced in soft-cores by "Single Event Upset" faults, which are very common, especially in FPGAs. This area of interest is particularly crucial because complex software, like operating systems, running on top of this hardware, that may be faulty, can create uncoverable and dangerous situations.

Before going deep into the argument, the thesis starts by explaining what an FPGA is, its differences from ASICs and the reason why the space industry is moving towards this technology. Furthermore, Chapter 2 introduces some concepts about radiations on electronic devices, how are they classified and what effects they can cause on a system. In addition to that, Chapter 3 introduces, with a great level of detail, all the tools and techniques used. The main purpose behind this chapter is to give to the reader a deeper knowledge of the arguments treated in this thesis work and to be able to recreate the proposed solution in the future, in almost a straightforward way.

After that, the reader should be capable of understanding all the concepts that are mentioned in the main chapter of this thesis. Indeed, Chapter 4 analyzes different solutions that have been taken into consideration. The advantages and disadvantages of each solution are also discussed. Without entering into many details, 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 consists of a partial scrubbing of the configuration memory of the FGPA, in the area where the MicroBlaze physically is. This is achieved by the usage of a partial bitstream, to restore the original behavior. The partial reconfiguration allows for achieving a faster downtime, and consequently a higher availability of the system. This process is entirely managed by the DFX (Dynamic Function Exchange) Controller IP, offered by Xilinx. The DFX Controller loads the configuration file from the memory and sends it to the configuration port of the FPGA.

Moreover, a custom workflow has been developed to allow partial reconfiguration of a MicroBlaze in an older version of Vivado. Additionally, a custom script has been developed based on this workflow, thus providing 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.

To conclude, Chapter 5 and Chapter 6 are devoted to the analysis of the results coming from this work and the analysis of the possible implications, applications and future work that can be done addressing further research interests.

Acknowledgements

This thesis work would not be possible without the support of many people. I would like to thank my supervisors for their support and help in the development of this thesis.

Thanks to my partner, Heidi G., for constantly listening to me rant and talk about strange things over and over, and for the sacrifices you have made and shared with me in order to help me pursue this Master's Degree. I also want to express my deep appreciation for Carol who taught me the importance of wit, sound sleep, and playfulness and for her cuddly, fidelity and love.

Thanks to my parents, my father Gioacchino, my mother Maria Carmela and my sister Alice Carlotta, for your endless support, both economically as well as for believing in me with regular encouragement in every step to continue in my path.

I also would like to thank all my respected teachers and colleagues in the Control and Computer Engineering department.

Without all of you, I could have never reached this point in my life.

"Life is a journey, and every journey eventually leads to home."
Crestfallen Saulden, Dark Souls II

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XII

1 Introduction 1
1.1 Thesis Motivation . . . 3

2 General Background 4
2.1 Hardware Technology . . . 4
2.1.1 FPGA Architecture . . . 4
2.1.2 FPGAs vs. ASICs . . . 6
2.1.3 FPGA or ASIC in Aerospace Applications? . . . 7
2.2 Radiations . . . 9
2.2.1 Radiation sources . . . 9
2.2.2 Radiation problems on Earth: the Super Mario 64 glitch . . . 10
2.2.3 Types of radiation . . . 10
2.2.4 Single Event Effects . . . 11

3 Thesis Background 16
3.1 PYNQ-Z2 Development Board . . . 16
3.2 Xilinx soft-core: the MicroBlaze . . . 18
3.3 Xilinx FPGA Standard Design Flow . . . 19
3.3.1 Steps towards the Bitstream Generation . . . 20
3.3.2 Fundamentals of the Xilinx's Bitstream structure . . . 22
3.3.3 Software Development . . . 26
3.4 Fault Injection Tool . . . 27
3.5 Integrated FPGA Debugger . . . 29

4 Analysis and hardening of an FPGA Design with a MicroBlaze 31
4.1 How SEUs affect the MicroBlaze? . . . 32
4.2 Strategies and adopted solutions . . . 38
4.3 Development of a watchdog . . . 40
4.3.1 What is a watchdog? . . . 40
4.3.2 How to implement a watchdog? . . . 41
4.3.3 How to harden the watchdog? . . . 45
4.3.4 Integration of the watchdog in the design . . . 50
4.4 Design with Partial Reconfiguration . . . 56
4.4.1 Vivado Design Flow for Dynamic Function Exchange . . . 57
4.4.2 DFX with MicroBlaze in Vivado 2021.1 . . . 60
4.4.3 Xilinx DFX Controller . . . 62
4.5 Integration of the watchdog and the DFX . . . 63
4.5.1 Partial bitstream storage . . . 63
4.5.2 How to enable the ICAP port . . . 65
4.5.3 ICAP instantiation . . . 66
4.5.4 Connection of the Watchdog and the DFX Controller . . . 68
4.5.5 DFX Decoupler: what is it? . . . 69
4.6 From a manual workflow to a fully automated one . . . 70
4.6.1 The automatation script . . . 71
4.6.2 Script for partial bitstream to C header generation . . . 74

5 Experimental Analysis 76
5.1 Fault Injection Environment . . . 76
5.1.1 Watchdog Inhibition . . . 79
5.2 Experimental Results . . . 80

6 Conclusions 85
6.1 Future Work . . . 86

A Watchdog FSM - VHDL Code 87

B Watchdog - C drivers 90

C Fault Injection - SDE output with correction 92

Bibliography 94

List of Tables

3.1 ZYNQ 7020 SoC Memory Map 18
3.2 7 Series Configuration Packet: Type 1 Header OP Field 25
3.3 7 Series Configuration Registers 25

4.1 SEU consequences in SRAM-based FPGAs [32] 32
4.2 Fault injection result for the basic MicroBlaze design 37
4.3 Detailed explanation of the states of the FSM 44
4.4 Voter truth table. The red cells indicate the faulty output. 46
4.5 Output signal matrix for the no-tmr version. 48
4.6 Comparison between full bitstream and partial bitstreams sizes. . . 60
4.7 ICAPE2 Interface description. 66

List of Figures

2.1 Simplified schematic of an FPGA cell 5
2.2 The intrisic BJTs in the CMOS Technology that can cause a Latchup. Deepon, CC BY-SA 3.0, via Wikimedia Commons 12
2.3 Example of a Single Event Upset in a memory element. 13
2.4 Simple SRAM Cell layout. Inductiveload, Public domain, via Wikimedia Commons. 13
2.5 Example of error-detection circuitry in SRAM. 14

3.1 Schematic of the PYNQ-Z2 Development Board 16
3.2 Schematic of ZYNQ 7020 SoC 17
3.3 [23]Overview of a Microblaze SoftCore 19
3.4 Example of Block Design 20
3.5 Basic scheme of a fault injection tool [24] 27
3.6 Example of a ILA IP instantiation in a Block Design. 30
3.7 ILA's debugging GUI in Vivado. The ILA is waiting for the trigger, and the trigger is set waiting to have a certain signal equal to 1. . . 30

4.1 Schematic of a basilar MicroBlaze design 33
4.2 Resulting hierarchy of the MicroBlaze design 34
4.3 Resulting floorplan of the MicroBlaze design, with the PBLOCK on the top side. Microblaze cells are highlighted in red 36
4.4 High level scheme of the fault tolerant design. 39
4.5 Timing diagram of a very basic watchdog. 41
4.6 Timing diagram of a more sophisticated watchdog. 41
4.7 Possible digital circuit implementation of a watchdog. 42
4.8 Timing diagram of a more sophisticated watchdog. 44
4.9 Triple Modular Redundancy (TMR) scheme. 45
4.10 1-bit voter circuit scheme. 46
4.11 Basic TMR scheme vs. full TMR circuit scheme. 47
4.12 Input signal matrix for the no-tmr version. 48
4.13 Interfaces of the TMR Watchdog. 49

4.14 Conceptual representation of the final watchdog IP. 50
4.15 Instantiation of the Watchdog IP. 53
4.16 Watchdog IP configuration wizard. 54
4.17 Example of partial reconfiguration modules. 56
4.18 Example of modules grouped under a partial reconfiguration area. . 57
4.19 Partition definition with two reconfigurable modules, each one with its own wrapper and .xci file. 59
4.20 Flow used to generate a fully working Reconfigurable MicroBlaze design. 62
4.21 Basic scheme of the DFX Controller flow. 62
4.22 ICAP Interface. 66
4.23 ICAP Interface as seen by an ILA core. 68
4.24 Enhanced majority voter design. 68
4.25 DFX Decoupler scheme. 70
4.26 Decoupling of a Reconfigurable MicroBlaze region. 70
4.27 Scheme of the script flow. 71

5.1 Hardware schematic to count the number of times the watchdog time outs. 77
5.2 Hardware schematic to inhibit the watchdog using a physical switch. 79
5.3 Chart representing the executed Fault Injection campaigns. 81
5.4 Charts showing the times the reconfiguration is triggered (or not) and how many times it solved the issue (or not). 82
5.5 UART loopback schematic. 83
5.6 Chart representing the repeated Fault Injection campaigns with the new firmware. 84
5.7 Charts showing the times the reconfiguration is triggered (or not) and how many times it solved the issue (or not). Second run with the new firmware. 84

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 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.

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 deciding autonomously the trajectory to follow or applying some complex algorithms to the data collected before sending them back to the ground.

Whatever the purpose of a spacecraft, from the smallest one to a complete rover exploring another planet, electronic devices must be tailored to work reliably, even in a complex and harsh environment like 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 with a real-world example, a piece of history. On September 22, 2021, the ESA's INTEGRAL spacecraft autonomously entered into emergency safe mode [2]. INTEGRAL is a space telescope for observing gamma rays, and it was launched into Earth's orbit in 2002. Something catastrophic was happening for the mission 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 uncontrollably.

This episode created a lot of problems for the mission itself, and the team of

engineers responsible for the INTEGRAL spacecraft had to deal with it: because the spacecraft was spinning, data from the spacecraft were 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-year-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 radiation. Charged, ionized particles, from the Van Allen belt, caused an 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 radiation affecting the onboard electronics. From this example, we can understand how crucial is fault-tolerant analysis during all the stages of development of a new space component, in order to produce a dependable system. The concept of a 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 (usually it coincides with the period 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 radiation-tolerant components are used. They are usual referred as *radiation-hard* or *rad-hard* for simplicity. In most of the cases, *COTS* (Commercial Off-The-Shelf) hardware [3] is used, which is hardware meant to be used in a generic environment, and on top of that logical mitigation techniques [4] 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 in terms of costs.

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 space. In particular, the main goal is to investigate the outcome that can occur when SEU faults affect the CPU (in particular a Xilinx Microblaze soft-core, which will be explained in more detail 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.

Consequently, the goal is to study and develop some techniques to mitigate errors induced in soft-cores by Single Event Upset faults, which are very common, especially in FPGAs. This area of interest is particularly crucial because complex software, like real-time operating systems (for instance, FreeRTOS), running on top of this hardware, that may be faulty, can create uncoverable and dangerous situations [5].

The hardware model on which the techniques are based 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 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 sophisticated [6] hardware, 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. Obviously, FPGAs meant to be used in space need to undergo a lot of tests [7], for example in facilities where ultra-high heavy-ion test beams are used to see how the FPGA reacts to real radiation effects.

Chapter 2

General Background

Before going further with the problem analysis, it is better to introduce a few background concepts. In particular how FPGAs work, what kind of radiations exists and how FPGAs are affected by them.

2.1 Hardware Technology

2.1.1 FPGA Architecture

Field Programmable Gate Arrays *FPGAs* 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 functionalities. It is hardware that can be programmed to perform specific tasks.

It differs from a CPU, which is an already designed hardware. A CPU does 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 as the process of writing a series of instructions that the CPU will eventually execute. This is done by exploiting Programming Languages. An FPGA, instead, is like LEGO bricks. Each LEGO brick does not have any function or purpose alone, but when 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 must be assembled to perform the specific task we want. The description is done by exploiting Hardware Description Languages (HDL) like VHDL or Verilog.

The basic FPGA design consists 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 helps in achieving the goal of the User's Application in the overall design.

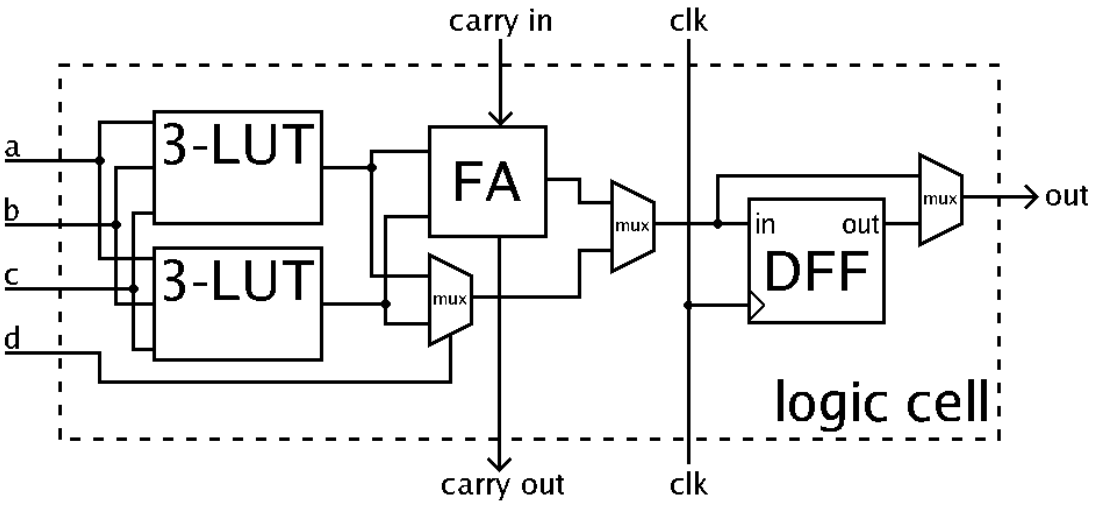


Figure 2.1: Simplified schematic of an FPGA cell

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. Those are commonly used functions embedded in the circuit. They reduce the overall area required and give those functions increased speed compared to building them from logical primitives (because are implemented in silicon, and built out of transistor instead of LUTs, enabling 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 completes the design (i.e. the description of the FPGA is written using some HDL language), the design needs to be mapped on top of the FPGA's hardware resources. This is done using the Vendor's specific software and it is in charge of deciding which FPGA's Logic Element (LE) is assigned to which subpart of the description and how each LE is configured. Then, all the LEs needs to be connected among themselves and the I/O pads, and this is done by routing algorithms that decide the best way to connect them. Once all the implementation

steps are done, a configuration file is generated that will eventually be 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 functional blocks (i.e. the ones that can be used by the user to implement the application). Those memory elements can be thought 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 them 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. When the FPGA is powered off, the configuration memory content is lost because the SRAM is a volatile memory. Hence, it requires an external memory where the bitstream can be retrieved to re-program it. The SRAM approach is based on the Complementary Metal-Oxide-Semiconductor (CMOS) technology.

Consequently, FPGAs are alternatives to hard-core CPUs. This means that a CPU can be implemented out of logic primitives on an FPGA and is defined as *soft-core*, alongside the hardware that is used to implement the application like peripherals, memory and other components. Modern FPGAs support *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 on top of 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 Full Adder (FA), etc.) provided by the manufacturer that assembles the chip. Once an 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 an output of this process, a set of masks is generated. The masks are used to *print* the circuit on the silicon. On top of all this process, test engineers must prepare a set of tests in order to verify the correct functionalities of the circuit during the various stages of the manufacturing process, until the end of the process itself.

This allows the implementation of entire microprocessors, memories (including ROM, RAM, EEPROM and flash) and other large components 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, which can run into millions of dollars.

To recap:

- ASICs circuits are faster, and less power-hungry than FPGAs.
- ASICs are more complex to design and implement (hence more expensive) than FPGAs.
- FPGAs are more flexible 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 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, Perseverance, 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 give great help in this aspect and Perseverance rover is an example. It utilizes an almost two decades 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 landed, this unit would be useless becoming a *dead hardware*. However, it is based on 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-rays (used to identify chemicals) are controlled using Xilinx's FPGAs. Another interesting point is that Perseverance uses machine learning algorithms running on FPGAs, and they are so well optimized that it is achieving higher performance levels (about 18 times) than the Curiosity rover (which landed on Mars in 2012 and is still active).

Another advantage of using FPGAs is the faster time-to-space. Different points help in achieving this advantage. Not only the development on FPGAs 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 architecture during the project development phase. There is usually a very restricted 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 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 certified is reduced. Second, a bitstream or software change takes 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 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 landed in 2012 and it is still active in 2022. 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. Different things can go wrong in a decade and having a fully 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 Redundancy) to a portion of the design or even to the entire design. Basically, it consists in triplicating the design and adding 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 degradation in performance to a complete functional failure. More and more a space system goes deeper in space, and less and less it is protected by the Earth's atmosphere.

Close to the Earth, there are 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 extend 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 traveling 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 designed 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 devices. 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 reward was offered to anyone 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 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 added value 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 cumulative damage (i.e. passage of many charged particles in different moments in time, and each particle has a relatively 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 high energy) then it can be *destructive* or *non-destructive*, and they are usually referred to 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 an 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 pass 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 within 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 a functional error, for example, a wrong value in the memory from *0xe* to *0xf*).

Among 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 an n-p-n-p structure (corresponding to an NPN and a PNP transistor stuck next to each other), a CMOS Latchup structure is created. If one of these two transistors 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 as long as some current flows through them. This phenomenon will increase the current draw and can bring to the destruction 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 exist that may not appear until later.

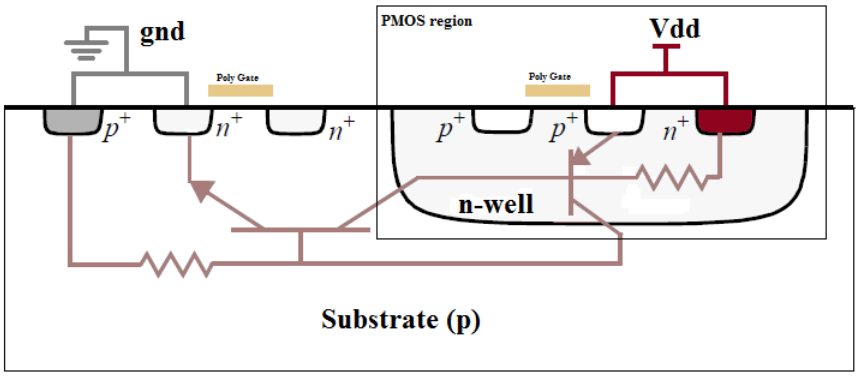


Figure 2.2: The intrisic BJTs in the CMOS Technology that can cause a Latchup. Deepon, CC BY-SA 3.0, via Wikimedia Commons

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 the destruction of the device.

Single Event Gate Rupture

SEGR is the destructive rupture of the gate oxide (or any dielectric layer in a transistor). The effects can be observed in power MOSFETs as an increase in the 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 radiation arrives. In more complex circuits, it can cause other malfunctions like resets and memory values modification.

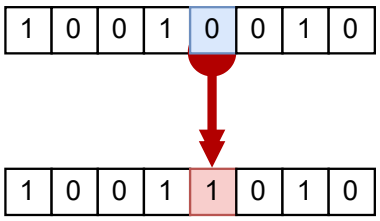


Figure 2.3: Example of a Single Event Upset in a memory element.

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

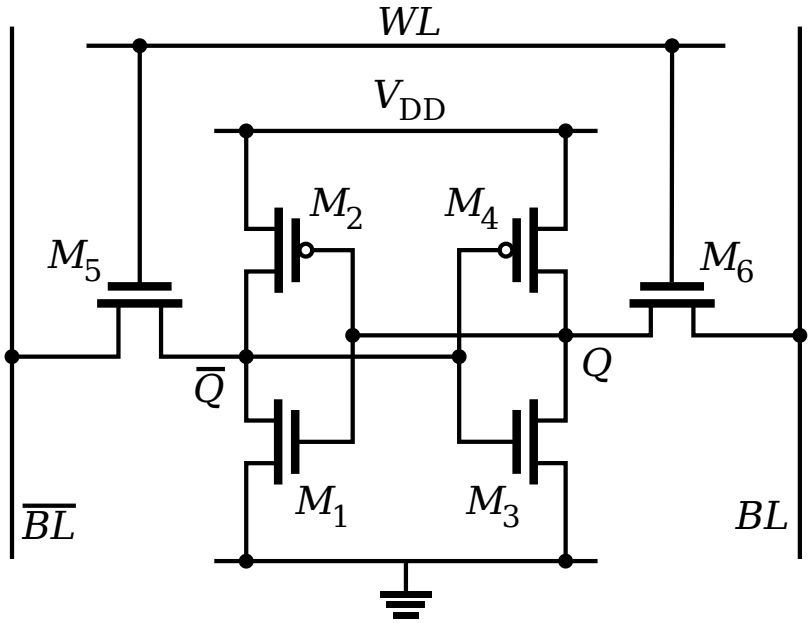


Figure 2.4: Simple SRAM Cell layout. Inductiveload, Public domain, via Wikimedia Commons.

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 radiation strikes one of those transistors, can happen that the M3's gate voltage goes low, causing a flip of the configuration and thus a flip of the stored bit.

As explained in Section 2.1.1, most FPGAs' memory configurations are based on SRAM technology. A fault that occurs in a configuration SRAM of an FPGA can lead to completely disastrous failures compared to traditional SRAM used purely for memory storage. This is because upsets have no effect until an address pointing to a word affected by an upset is read out of SRAM.

Luckily, error detection and correction circuits can be used to detect and correct the fault, without causing a failure in the undergoing operations. Those are based on the usage of redundant bits for each word. As an example, Hamming codes to detect and correct single-bit error, SEC-DED (Single Error Correction - Double Error Detection) to correct a single error and detect one or double errors, or SEC-DED-DAEC [18] (Single Error Correction-Double Error Detection-Double Adjacent Error Correction) to correct adjacent errors in multiple words. An example of error detection circuitry is shown in Figure 2.5.

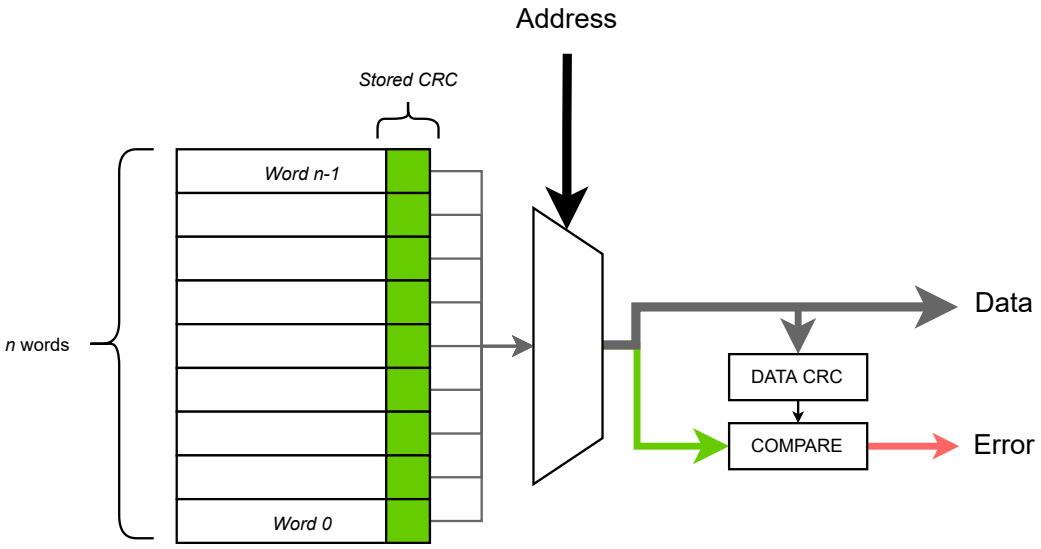


Figure 2.5: Example of error-detection circuitry in SRAM.

In an FPGA, the configuration SRAM is not utilized the same way as traditional SRAM. Indeed, direct (logic) connections from the configuration to the user logic exist. If an upset occurs in a used configuration bit, then upset occurs in logic.

Because of this difference in the SRAM usage (not dealing with data words anymore but every bit is meaningful at any moment), traditional SRAM detection and corrections schemes can't be applied to FPGAs anymore. If a bitflip occurs, the FPGA configuration itself is modified, leading to a malfunction of a module or a routing modification.

The actual technology trend see scaling down to smaller sizes [19], 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, which 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 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.

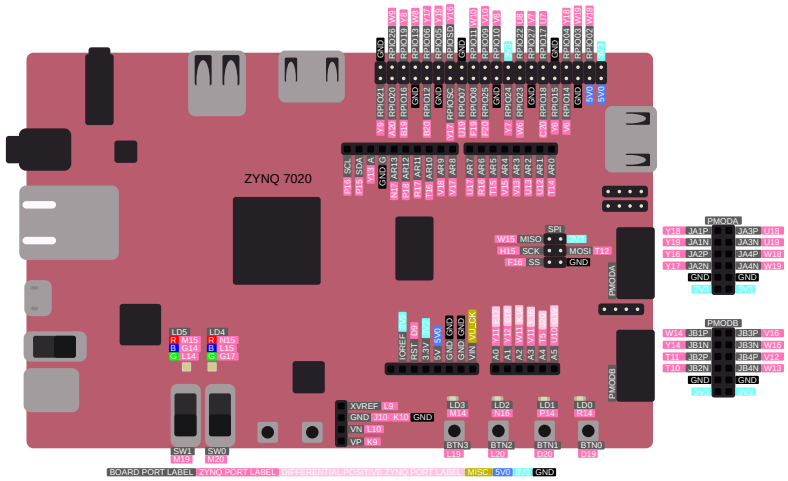


Figure 3.1: Schematic of the PYNQ-Z2 Development Board

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 processors, 512 KB L2 Cache, 256 KB On-Chip Memory and other modules like FPUs, Flash Controller, DRAM Controller, GPIOs and so on.

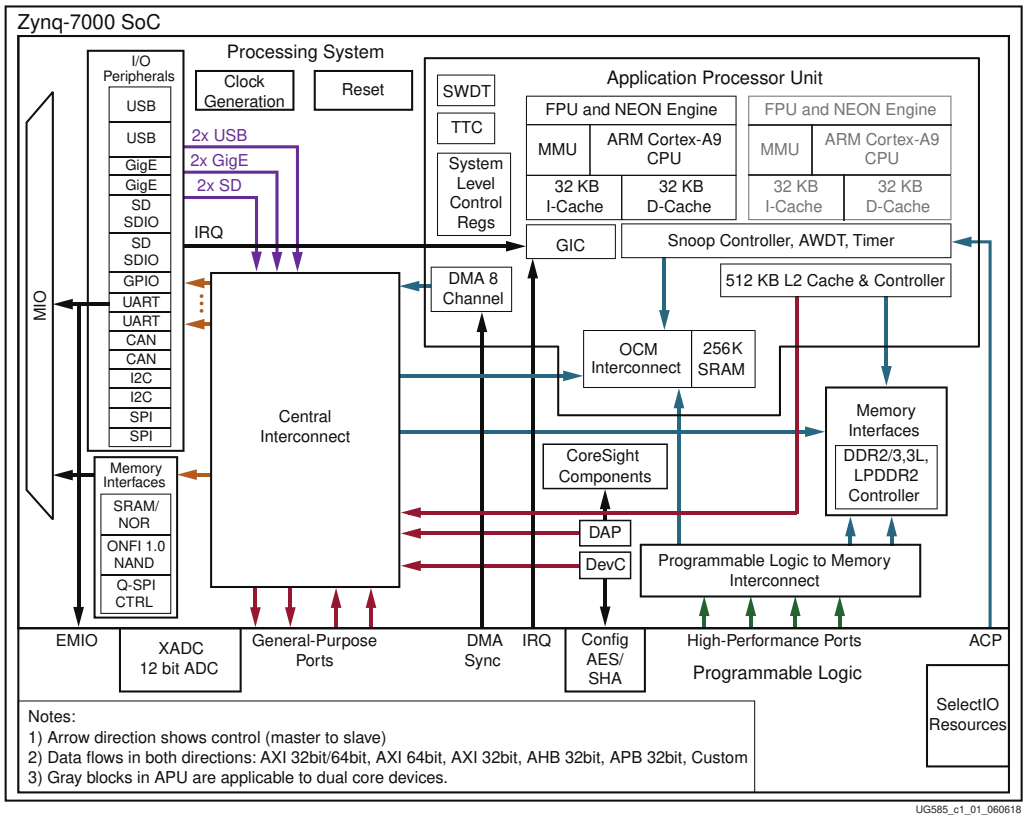


Figure 3.2: Schematic of ZYNQ 7020 SoC

A schematic is shown in Figure 3.2. The second part is the PL, which consists in an 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 High Performance and/or General Purpose AXI Ports. This enables

the usage, for example, of the DDR3 RAM and the On-Chip Memory (OCM) from the PL. The board can be programmed through a JTAG interface, which allows uploading 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 soft-core: the MicroBlaze

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.

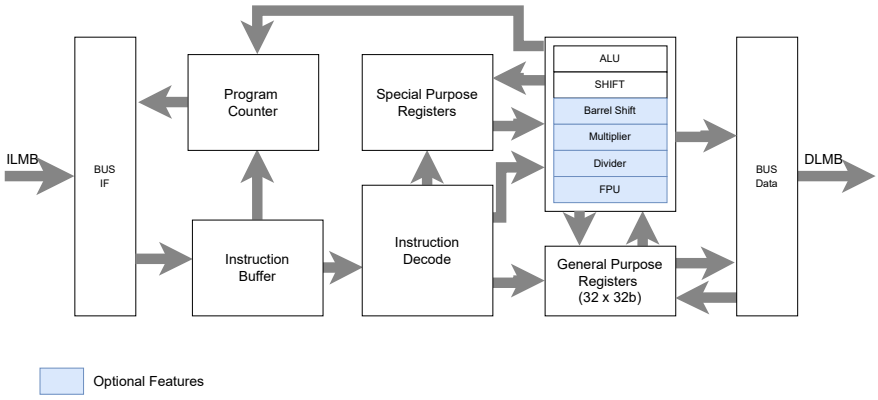


Figure 3.3: [23]Overview of a Microblaze SoftCore

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. Examples 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 the 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 bitstream 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, which contains 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 in a single environment.

One of the key features of the Vivado Design Suite is the choice given to the user to perform the design flow through the Graphical User Interface (GUI) or by TCL commands. The GUI, known 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 employing 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 allows 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 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 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.

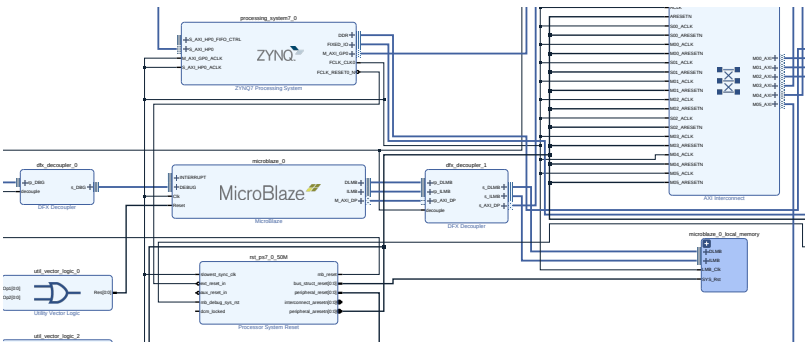


Figure 3.4: Example of Block Design

Moreover, the Block Design Tool allows the user to define the memory mapping of the AXI peripherals concerning 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 separated 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 validating the design, as 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 testbench. The testbench is usually an 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 is possible to assign some constraints. Those constraints, defined in an XDC file, regard 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 an 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 contains IPs, these are synthesized first. The user can decide the Synthesis approach adopted by the tool, for example, if the synthesis must follow a timing optimization strategy or an area optimization approach.

The next step is Implementation. The Implementation is the final step, where the gate netlist, produced as an 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 behavior where the block can be mapped. The choice may be driven by the need to minimize or balance the wiring across the 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 fulfill 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 among them as needed.

Once the design has been implemented, the final step of the bitstream generation can be performed. The default generated bitstream 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 bitstream.
- *.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 an FPGA, the .bit file is the default format generated by Vivado.

Bitstream Header

The header contains some information like the design name, build date, and FPGA target name. Those pieces of data are 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 (i.e. *7z020clg400*). The *c* and *d* fields 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 the 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 processor may steal the PCAP back (and stop the ICAP) at any time. This choice has been made in order to ensure that the ARM TrustZone remains in control of the security of the system all the time. ICAP is a potential backdoor, and would compromise security if the processor could not prevent and regulate its use.

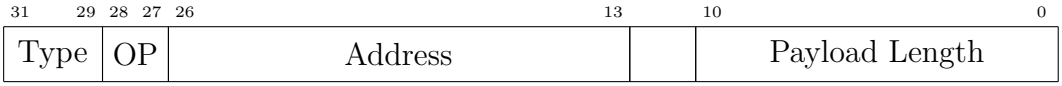
The raw bitstream in Xilinx's 7 series FPGA consists of three sections:

- 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 establish 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 shown one. Type 2 packets are used when the payload length exceeds the 11 bits available in a type 1 packet. Type 0 should exist, 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, which usually 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 do 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.
STAT	00111	Fixed	The Status Register (STAT) indicates the value of numerous global signals in the FPGA.

Table 3.3: 7 Series Configuration Registers

3.3.3 Software Development

Xilinx provides some tools to help software developers to develop applications for hard ARM cores (such as in the ZYNQ7020) or soft-cores such as Microblaze (one or multiple instances of the core). The most important one is Xilinx Software Command-Line Tool (XSCT). XSCT is a tool that allows developers to easily manage the FPGA via a command-line interface and to write scripts, based on TCL, to automate some steps.

XSCT allows mainly two things: creating and managing projects and accessing the FGPA's JTAG interface to program the FPGA itself or to debug applications. For what concerns the JTAG access, XSCT offers functions like the upload of a bitstream to program the FPGA, upload of .ELF executable files to be run by a specific core (it can be either an ARM core or a Microblaze), the ability to control a core (start, stop, reset, etc.), the ability to read and write registers and access the memory space of a core and ultimately to debug an application.

A more high-level tool is available, called Vitis. Vitis is a software development environment for FPGA development. It is a tool that allows developers, through an eclipse-based environment, to easily manage projects and applications. Vitis is a wrapper around XSCT.

A Xilinx Software project is made of a platform project. A platform project is a description of the hardware architecture on which the software will run. To create a platform project, the starting point is to extract a hardware description from a hardware design. The hardware description contains information like the available cores, the memory space for each core, the available peripherals (and relative software drivers, if not standalone) and the bitstream to program the FPGA. Vivado can generate such a description only for Block Design-based projects, via the `export_hardware` TCL command or via the GUI itself, which produces the *.xsa* file.

Vitis, or XSCT, take the .xsa file as input to create a platform project. Once it is created, it is possible to create a system project. A system project is a software project that contains multiple applications, each one based on a specific core. An application project can be standalone (so a bare-metal firmware), FreeRTOS based or petaLinux based (if available).

Once a project is created, software developers can start writing their code (usually in C) and customize the Board-Support Package (BSP) that allows changing basic things like the *stdout* and *stdin* used peripherals to print or read some text, respectively.

3.4 Fault Injection Tool

Fault injection is a widely used technique for fault tolerance evaluation. A common architecture for this kind of tools is presented in Figure 3.5:

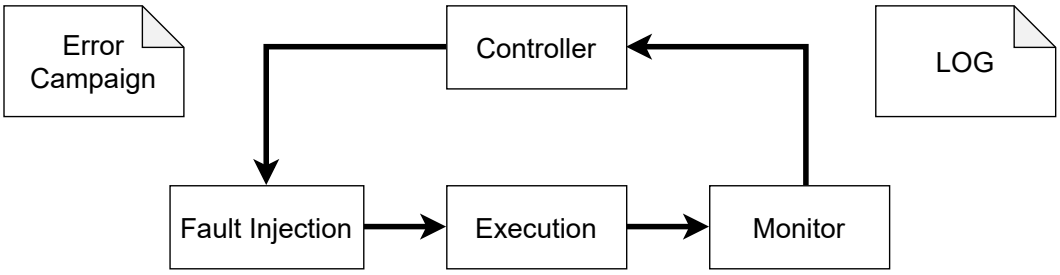


Figure 3.5: Basic scheme of a fault injection tool [24]

In the general scheme, the following elements are present:

- *Controller*: it is the main application and the orchestrator of all the other components of the tools.
- *Fault Injection*: it is the module responsible for injecting faults into the system, according to a specific error campaign.
- *Execution*: the output of the fault injection is executed.
- *Monitor*: the behavior of the system is monitored and logs are generated.

A widely accepted classification of the different injection strategies is summarized as follows:

Hardware-Based fault injection consists of the generation of physical errors into the integrated circuits. They can be divided into fault injections with contact and fault injections without contact. The one with contact consists in perturbating the integrated circuits via perturbation introduced at the pins (for example a rapid and minimal change of the power supply voltage) while in the case without contact there is an external source that produces physical phenomena such as heavy-ion radiations that induce faults in the integrated circuits.

Software-Based fault injection consists of the generation of software errors. They can be artificially inserted into a software system, both at compile time or at run time. The ones at compile-time [25] are inserted into the source code of the software or at the assembly level after the compilation of the original

source code itself. The ones at run-time are inserted through a trigger (for example a timeout or a software trap) that executes the fault injection module, altering the behavior of the software.

Simulation-Based fault injection simulation-based fault injection is a technique that allows to simulate of the system and to inject faults into it. The faults are injected from within the simulation environment, and it can be done by modifying directly the high-level description of the design with a faulty model or by using built-in commands in the simulator that force error in the simulation of the design, not in the hardware description of the hardware itself. Example of simulation-based fault injection systems are SST [26] and VERIFY [27].

Emulation-Based fault injection emulation-based fault injection is a technique consisting of a real implementation in an FPGA. For these platforms, the development board is connected to a PC that acts as a Controller by defining the fault injection campaign, controls the execution of the faulty design and monitors the behavior of the system under test.

The Fault Injection tool used is a kind of Emulation-Based fault injection [28]. Its goal is to create a faulty bitstream starting from the base one. Consequently, the faulty bitstream simulates a fault by forcing a random bit-flip, as a SEUs does. The tool offers the possibility to target a specific portion of the FPGA, in order to test the fault tolerance of a subpart of the design, instead of the whole one. This allows designers to execute a targeted fault campaign of the design under consideration.

Thanks to tools like PyXEL [29] it is possible to obtain a visual low-level representation of the bitstream. This allows an understanding of how and where the design's modules are mapped into the bitstream and consequently how each bit in the bitstream is connected to the design. Hence, by forcing some modules to stay in a specific portion of the FPGA, using some constraints as explained in chapter 3.3.1, it is possible to understand the position of the modules in the bitstream.

As a result, it is finally possible to have a targeted fault campaign. The Controller, part of the tool, can start producing n-faulty bitstreams, where n is given as input by the user. Once all the faulty bitstreams are generated, the Controller starts the Execution part.

As an important note, because of the bit-flips introduced into the bitstream, is necessary to remove any CRC checksum that may be present, that is checked during the upload in the FPGA, as explained in chapter 3.3.1. To overcome this

problem, Vivado must be instructed to generate a bitstream without CRC. It can be done with the following TCL commands:

```
open_run impl_1
set_property BITSTREAM.GENERAL.CRC DISABLE [get_designs impl_1]
write_bitstream
```

The first execution only is related to the golden bitstream and the Monitor will capture the golden result, i.e. the correct and expected result. The result is intended as the *stdout* output of a testbench firmware, over the UART. The firmware can implement, for example, a simple algorithm like matrix multiplication or bit count. Obviously, the more the algorithm is varied, in terms of used instructions and hardware, the more the chance to detect an injected fault, because of the higher probability to stimulate that fault.

All the remaining Execution outputs are compared against the golden, thus each run is classified as correct or faulty. When the campaign ends, a summary is produced, as shown in the following example, extracted from a very small fault injection campaign:

```
Total injected bitflips = 17

    --- FUNCTIONAL ANALYSIS ---
Correct results     -> 14 [82.35%]
Faulty results(SDE) -> 0  [00.00%]
MicroBlaze halted   -> 3  [17.65%]
Total exceptions    -> 0  [00.00%]
```

3.5 Integrated FPGA Debugger

A powerful tool offered by Xilinx is the Integrated Logic Analyzer (ILA) IP. It is a logic analyzer core that can be used to monitor the internal signals of a design. The ILA core includes many advanced features of modern logic analyzers, including Boolean trigger equations, and edge transition triggers. Thus, it is possible to debug in real-time the internal behavior of the design directly by Vivado, without the need for external tools.

To bebug a port, the ILA core must be instantiated, for example in a Block Design, and connected to the signals to debug. The configuration wizard allows to easily monitor an entire AXI bus or simple signals.

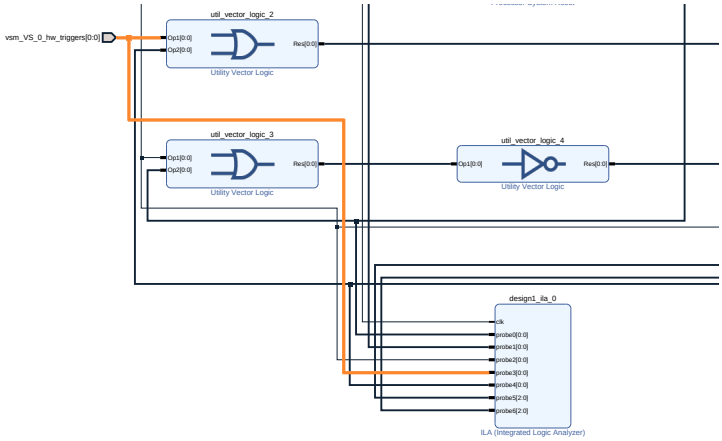


Figure 3.6: Example of a ILA IP instantiation in a Block Design.

When the designer is satisfied with the number of signals to monitor, it is possible to continue with the normal Vivado Design Flow until the Implementation and Write Bitstream phases. The implementation requires a bit more effort due to the creation of the *dbg_hub* core, which is the intermediary between the ILAs inserted into the design (where the signals to debug are connected) and the JTAG interface used to access and debug the design.

To access the Debug functionalities, open Vivado's Hardware Manager and connect to the Hardware Server. If the FPGA is already configured, Vivado automatically opens the GUI to access and visualize all the signals and set various triggers. The GUI is similar to the one used to simulate the design:

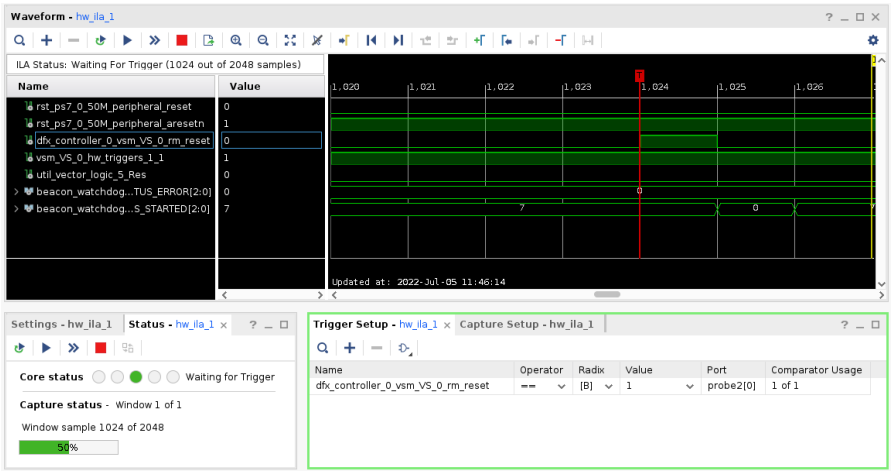


Figure 3.7: ILA's debugging GUI in Vivado. The ILA is waiting for the trigger, and the trigger is set waiting to have a certain signal equal to 1.

Chapter 4

Analysis and hardening of an FPGA Design with a MicroBlaze

An SRAM-based FPGA is sensitive to SEUs, as explained in Chapter 2. To better understand the effects of radiation on the FPGA, it is better to see a design as an abstraction of two layers. The two main layers are:

- *Application layer*: includes the logic and memory elements as described by the user.
- *Configuration layer*: includes the logic and memory elements that are used to implement physically the user's design in the FPGA.

From the logical point of view, a particle causing an SEU can affect one of the two layers, producing different consequences:

- SEUs in the Application Layer manifest as transient errors that could affect the stored data or the state of the user logic memory elements such as BRAMs or Flip-Flops.
- SEUs affecting the Configuration Layer manifest as persistent errors, that could be reverted using a reconfiguration process.

The first ones are transients because they are in the user logic and are directly controlled by the user. Because of that, they may be detected or corrected, it depends on how the logic has been designed. The seconds are persistent because they directly affect how the bottom hardware layer works: from the point of view of the user, it is like a real hardware fault that cannot be corrected.

Persistent errors can have two main consequences:

- They can change a routing element connection or can completely disconnect internal wires.
- They can change the behavior of a LUT.

SEUs in the configuration layer are the most common type of errors in SRAM-based FPGAs because the application layer virtually uses less area than the configuration layer. A summary of the different causes of SEUs is presented in the following table:

Layer		Element	SEU Consequence
Configuration	Routing	Muxes	Wrong input selection, open net, wrongly driven or left open
		PIP	Wrong connection or disconnection between nets
		Buffers	Output net wrongly driven or left open
	Logic	LUT	Wrong function inputs and outputs
		Control Bits	Wrong function inputs and outputs
		Tie Offs	Wrong function initialization
Application		RAM Blocks	Wrong application data
		CLB Flip-Flops	Wrong application data or state

Table 4.1: SEU consequences in SRAM-based FPGAs [32]

The following analyzes are focused on SEUs affecting the configuration layer, as they are the most common type of errors in SRAM-based FPGAs. However, the proposed techniques allow designers to detect and correct SEUs in the application layer, too.

4.1 How SEUs affect the MicroBlaze?

As anticipated in the previous sections, the object of interest of this thesis work is the analysis of the MicroBlaze behavior when affected by SEUs and how those effects can be mitigated by constructing a series of ad-hoc hardening techniques.

First, in order to understand how the MicroBlaze reacts to SEUs affecting itself, a series of fault injection campaigns must be executed. The idea is to start with a very minimal hardware design that includes a MicroBlaze and a set of minimal

peripherals. Thanks to the Block Design tool, the preparation of the design is very simple and straightforward, and the result is shown in Figure 4.1:

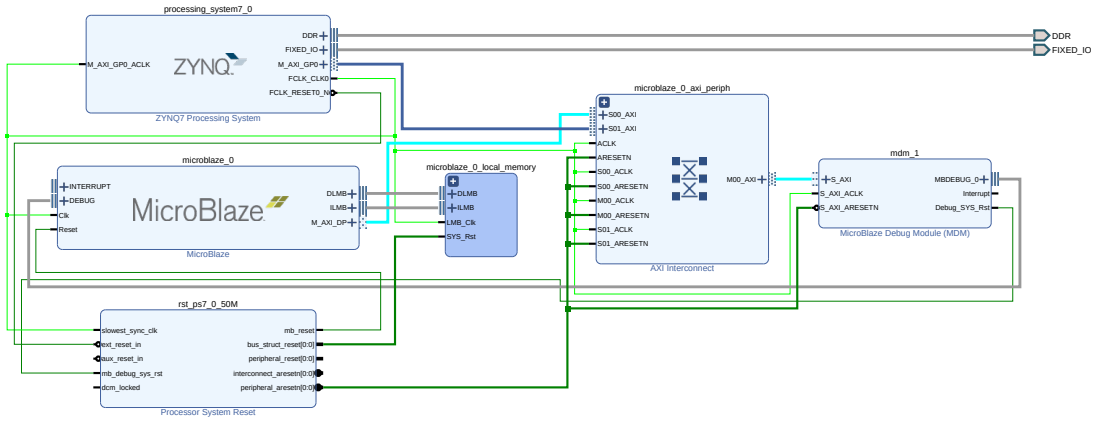


Figure 4.1: Schematic of a basilar MicroBlaze design

In the schematic, the following blocks are present:

- *ZYNQ7 Processing System*: it represents the ZYNQ7020's Processing System (PS) from the point of view of the Programmable Logic (PL), as explained in Chapter 3.1. It can offer a wide range of functionalities to the PL. For the moment, it is only used as a clock source and as a reset source. Via the ZYNQ7 PS block's configuration wizard, it is possible to configure a Phase-Locked Loop (PLL) in order to generate a clock for the PL with a specific frequency. For now, the PLL is configured to generate a clock for the PL with a frequency of 50 MHz, the same as the reference one given by the PYNQ-Z2 board.

- *Processor System Reset*: it is a soft IP that provides a mechanism to handle the reset conditions for a given system. The core handles numerous reset conditions at the input and generates appropriate resets at the output. For this simple design, the PSR can handle reset requests both from the PS and from the *Debug Core*. It generates an active-high reset signal for the MicroBlaze core and for the Local Memory. Moreover, it generates an active-low reset signal for the AXI peripherals.

- *MicroBlaze*: it represents the MicroBlaze instance under test. It has as inputs some debug signals from the Debug Core, the clock and the reset signal coming from the PSR. It offers as outputs the two memory buses for the Local Memory, one for the data memory (DLMB) and one for the instruction memory (ILMB). The last output is the AXI bus, which is used to access the peripherals through a *AXI Interconnect*.

- *Local Memory*: it is a sub-design (automatically generated by Vivado) that interfaces some BRAMs (a special kind of memory offered by the FPGA) with the Local Memory Bus (LMB). Block RAMs (or BRAMs) means Block Random Access Memory. Block RAMs are used for storing large amounts of data inside FPGAs.
- *AXI Interconnect*: it is a sub-design (automatically generated by Vivado). As the name suggests, it is used to connect one or more AXI memory-mapped master devices to one or more memory-mapped slave devices.
- *MicroBlaze Debug Module*: it is the Debug Core, and its main job is to enable JTAG-based software debugging of the MicroBlaze core. Moreover, it includes a configurable UART via an AXI interface. The UART's *RX* and *TX* signals are transmitted over the device JTAG port and can be accessed via the XSCT tool. With this setup, XSCT offers designers the possibility to interact with the MicroBlaze core via a UART and to control the MicroBlaze core (status, registers, and software debug in general).

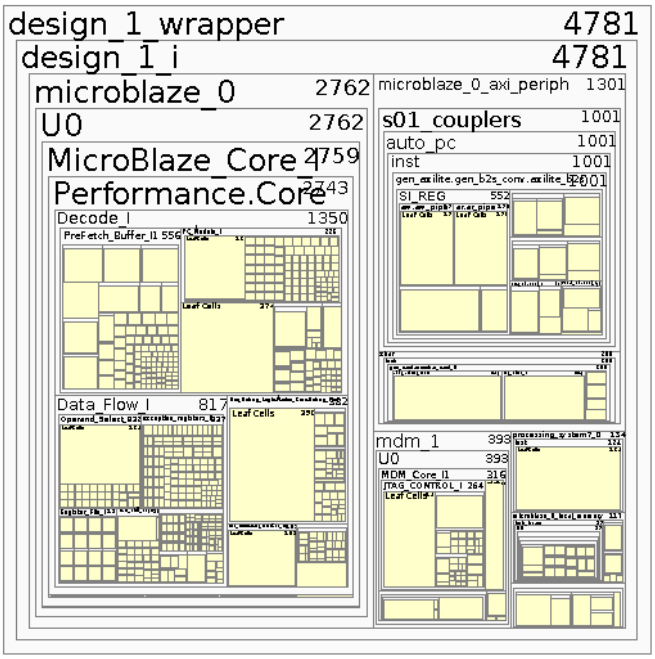
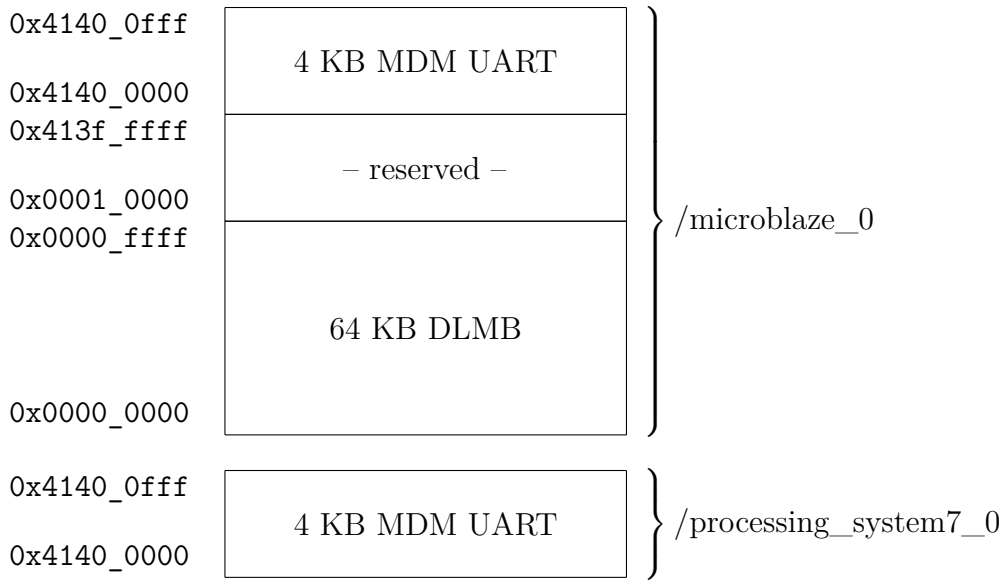


Figure 4.2: Resulting hierarchy of the MicroBlaze design

By looking at the schematic, there are two AXI masters connected to the AXI Interconnect. Hence, there are two different memory address spaces, and they are configured as follows:



Finally, the design definition is ready and it can be synthesized and implemented. Because the aim of this design is to be analyzed by injecting faults, the MicroBlaze has been constrained to be placed in a specific portion of the FPGA, as explained in Chapter 3.4. This is possible with Vivado by defining a PBLOCK.

A PBLOCK is a collection of cells, grouped in one rectangular area or region that specifies the device resources contained by the PBLOCK. PBLOCKs are used during floorplanning. A design floorplan is broadly defined as a set of physical constraints used to control how the logic is placed into the FPGA. A good floorplan can help reduce routing congestion and improve the quality of timing results. On the other hand, a bad floorplan can reduce performances as well as unmet constraints if the required placement is unfeasible.

As an example, the above design is implemented with the following constraints:

```
create_pblock pblock_1
add_cells_to_pblock [get_pblocks pblock_1] [
    get_cells -quiet [
        list design_1_i/microblaze_0
    ]
]

resize_pblock [get_pblocks pblock_1] -add {
    SLICE_X54Y102:SLICE_X67Y148
}

set_property IS_SOFT 0 [get_pblocks pblock_1]
```

In the above constraints, a PBLOCK called *pblock_1* is first defined. Then all the cells belonging to the Microblaze instance (*design_1_i/microblaze_0*) are added to the PBLOCK, and finally, the PBLOCK is resized. The resize operation is used to define the physical resources that are included in the PBLOCK. As the final operation, the PBLOCK is marked as a *not soft PBLOCK*, which means that each MicroBlaze cell must be placed obligatorily in that specific PBLOCK, and so it is a hard constraint.

Once the constraints are ready, the design can be synthesized and implemented. The resulting floorplan is shown in the following figure:

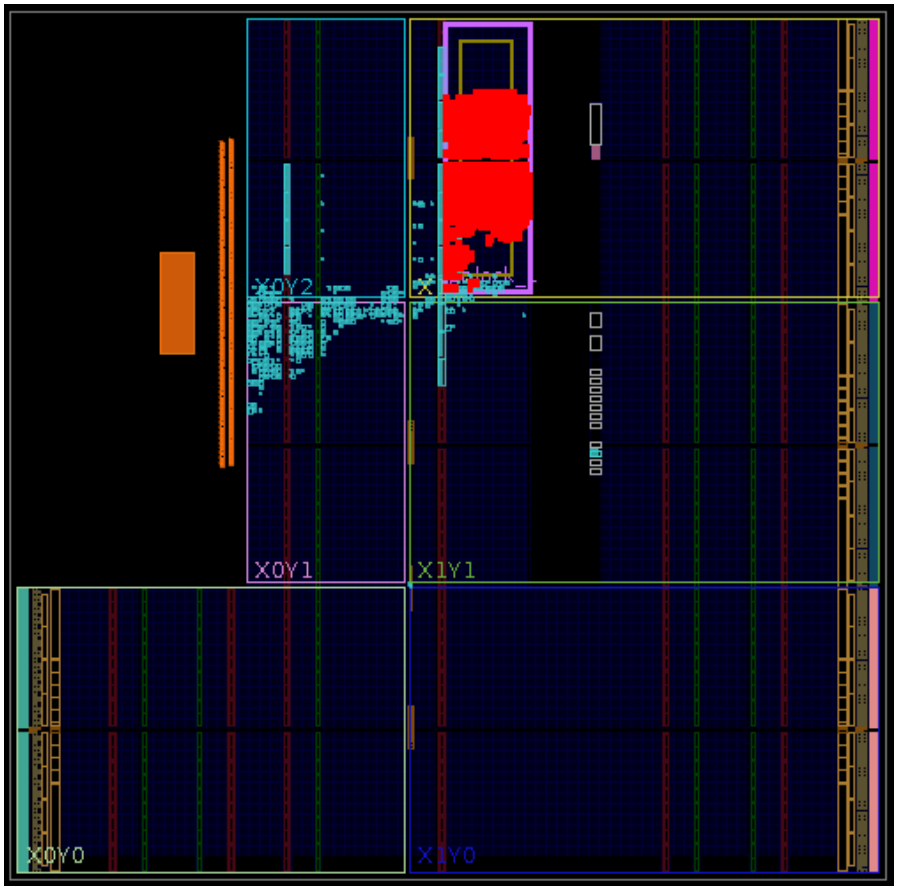


Figure 4.3: Resulting floorplan of the MicroBlaze design, with the PBLOCK on the top side. Microblaze cells are highlighted in red

Finally, it is possible to launch the fault injection tool by providing the bitstream with no CRC and a .elf file to be executed by the MicroBlaze at each run. The resulting fault injection campaign is shown in the following:

Functional Analysis		
	Total	*Percentage*
Correct results	9618	77.94 %
Faulty results (SDE)	131	1.06 %
MicroBlaze halted	2359	19.12 %
Raised exceptions	233	1.89 %
Total injected bitflips	12341	100.00 %

Exceptions		
	Total	*Percentage*
`XEXC_ID_FSL`	2	0.02 %
`XEXC_ID_UNALIGNED_ACCESS`	65	0.53 %
`XEXC_ID_ILLEGAL_OPCODE`	107	0.87 %
`XEXC_ID_M_AXI_I_EXCEPTION_or_XEXC_ID_IPLB_EXCEPTION`	0	0.0 %
`XEXC_ID_M_AXI_D_EXCEPTION_or_XEXC_ID_DPLB_EXCEPTION`	55	0.45 %
`XEXC_ID_DIV_BY_ZERO`	3	0.02 %
`XEXC_ID_STACK_VIOLATION_or_XEXC_ID_MMU`	0	0.0 %
`XEXC_ID_FPU`	0	0.0 %

Table 4.2: Fault injection result for the basic MicroBlaze design

From the above results, it is possible to see that in the majority of the cases, the MicroBlaze correctly executes its job. This is because the executed firmware possibly stimulates only a sub-part of the core. However, there are a lot of faulty cases ($> 21\%$), and they are divided into three categories:

- *SDE*: the MicroBlaze executes the entire code until the end (the end condition is detected when the MicroBlaze prints a specific string, like `DONE_1 DONE_1 DONE_1`). However, the output differs in some measures from the golden one.
- *Halted*: the MicroBlaze is halted, meaning that the end condition is not detected.
- *Exception*: the MicroBlaze raised an exception.

Even tho SDEs and exceptions can be directly detected by the firmware running on the core, it is not possible to detect the halted state. In theory, would be possible to detect and try to correct the FPGA configuration. Nonetheless, trying to do so would lead to two main problems:

1. Halt conditions are the majority of the cases ($> 19\%$ or about 86% among the faulty conditions). In this case, the firmware is almost not able to run and the fault would not be detected.

2. During SDEs or exceptions, the MicroBlaze runs until the end of the code, but it is not guaranteed that all the instructions are executed or are executed correctly.

Thus, a different approach is needed to detect and correct the fault.

4.2 Strategies and adopted solutions

Because of the problems raised in the previous section, different approaches need to be evaluated. This thesis work is focused on SEUs affecting the configuration layer of the FPGA, as they are more likely to occur. The adopted strategy can detect and correct those errors, previously defined as persistent errors, and may correct errors in the application layer too, under the condition that the overall design (hardware or software) is engineered in such a way to detect them.

To overcome those persistent errors, there are some techniques able to exploit the particular reconfigurable capabilities of the FPGAs. The following are some of the techniques taken into account:

Data Scrubbing is a general technique based on the concept of a virtual background task that periodically checks memory content for errors, then corrects detected errors using redundant data in the form of different checksums or copies of data. It is useful to correct and prevent (accumulation) errors in the information stored in memory. In FPGAs, scrubbing can be used to mitigate both persistent errors in SRAM cells (i.e., the configuration memory) and transient errors in user-memory elements such as BRAMs. To perform configuration memory scrubbing, the configuration memory data must be read sequentially from the start to the end and compared to the original configuration bitstream or an error check code such as a cyclic redundancy check (CRC). Scrubbing can be performed (both check and correction) without interrupting the device's functionalities. In aerospace applications, scrubbing is a common technique to mitigate the effects of SEUs. However, there are a few aspects to overcome like often the scrub operation must be performed, because of the very limited area and power constraints. Hence, scrubbing alone is a weak mitigation strategy without any other technique applied [33]. This is mainly because if a configuration bit is hit while the circuit is active, the error propagates in the design and can lead to a failure and the scrubber has no time to fix the error. An example of good design would be having a scrubber joined by a design with a triple modular redundancy check. The TMR design can detect and mask the error by itself, without causing a failure and meanwhile, the scrubber can be notified of the error and fix it. In this sense, scrubbing can be useful against error accumulation too.

Dynamic partial reconfiguration [34] allows run-time reconfiguration without application layer interruption. This technique cannot detect errors by itself, so it must be combined with other error detection techniques such as those based on redundancy. These correction techniques take advantage of the subdivision of the configuration memory into frames, which contain information related to the configuration of specific parts of the design. While these techniques allow increasing the protection capability against radiation effects, they introduce several penalties to the design, particularly in terms of performance. In literature, some techniques are proposed, like innovative placement algorithms able to improve the running frequency up to 44% by reducing the interconnection delays between resources [35].

The chosen strategy is to use the Dynamic Partial Reconfiguration technique. For this thesis work, only the MicroBlaze area is configured as dynamic reconfigurable. As said, this is useful only to fix errors in the configuration area related to the FPGA, but something that detects the error is needed. Hence, a self-made watchdog is developed to detect the error and trigger the reconfiguration. The reconfiguration is handled by a dedicated controller. A high-level scheme of the design is presented in the following figure:

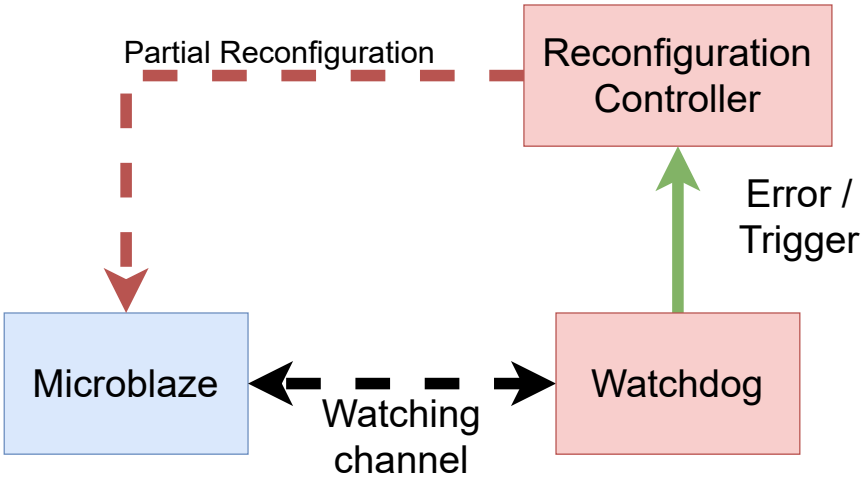


Figure 4.4: High level scheme of the fault tolerant design.

In figure 4.4, the following parts are highlighted:

MicroBlaze it is intended both as the instance of the MicroBlaze itself and as the physical area of the FPGA where the MicroBlaze is mapped and configured as dynamic reconfigurable.

Watchdog its job is to continuously check the MicroBlaze's status via a *watching*

channel (that is, a channel that is used to monitor the MicroBlaze's status) and if the MicroBlaze is halted, it triggers the reconfiguration via a dedicated signal *Error/Trigger*.

Reconfiguration Controller it is the controller that handles the partial reconfiguration when it is signaled to do so by the watchdog. It is responsible for the reconfiguration of the MicroBlaze's area.

4.3 Development of a watchdog

Among the modules to be added to the design in order to achieve a fault-tolerant design, the watchdog is the most critical one. If it fails in detecting errors, the overall design doesn't result protected from SEUs affecting the MicroBlaze. This is because the watchdog would not able to detect the error and trigger the reconfiguration.

4.3.1 What is a watchdog?

In computer systems, a watchdog is essentially a timer (that may be hardware or software) that is used to detect and recover from computer malfunctions. Watchdog timers are widely used in computers to facilitate the automatic correction of temporary hardware faults. Can be thought of as a down-count timer. When the timer elapses, it generates a timeout signal.

During normal operation, the computer regularly restarts the watchdog timer to prevent it from timing out. If due to a hardware fault or program error, the computer fails to restart the watchdog, the timer will elapse generating a timeout signal. The timeout signal is used to initiate corrective actions. The act of restarting the watchdog timer is usually called *kicking*.

Both generally speaking or strictly related to this case of study, a watchdog timer provides automatic detection of catastrophic malfunctions that prevent the computer from kicking it. However, there are often less severe types of faults that do not interfere with the kicking operation but still require watchdog oversight. In the specific case, can be for example a fault affecting the Arithmetic Logic Unit (ALU) of the MicroBlaze or the AXI interface towards peripherals. To support these, the system should be designed so that the watchdog timer is not kicked anymore in these less-severe faults. This can be done by writing some software routines that can self-test the CPU and its functionalities. The CPU will kick the watchdog only if all tests have passed.

4.3.2 How to implement a watchdog?

Once understood the principles of a watchdog, it is possible to implement it and tailor its functionalities to the needs of having a fault-tolerant system on FPGA. From a high-level perspective, the watchdog is at heart a timer. This means that it must have some form of timing, so it needs a clock and reset signals. Moreover, the timer must restart every time the kicking action is performed. If the timer elapses, it generates a timeout signal. The following is a timing diagram of this basic watchdog:

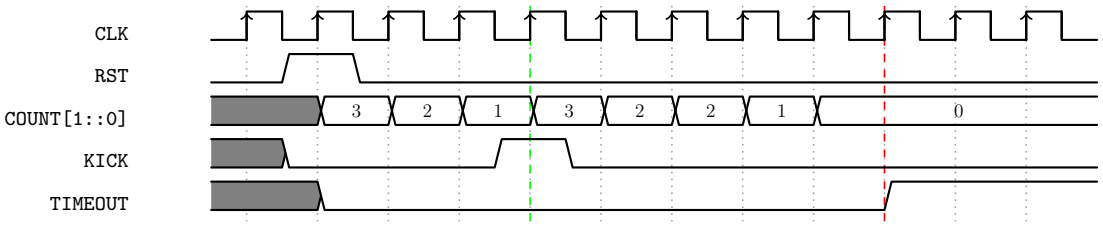


Figure 4.5: Timing diagram of a very basic watchdog.

In figure 4.5, the system initially is in an unknown state. Once the reset signal *RST* arrives, synchronously the watchdog is reset and starts counting down. When the timer reaches 1, luckily a *KICK* signal arrives (green line), signaling that the CPU is correctly working, and the timer is restarted from the initial value of 3. 4 clock cycles later, the timer reaches 0, and the *TIMEOUT* signal is generated (red line) because no *KICK* signal has arrived.

A more sophisticated implementation of a watchdog could be based on a up-count timer, an input data containing the maximum number of clock cycles the timer can count before it generates a timeout signal, and a start signal that is used to start the timer. The following is a timing diagram of this implementation:

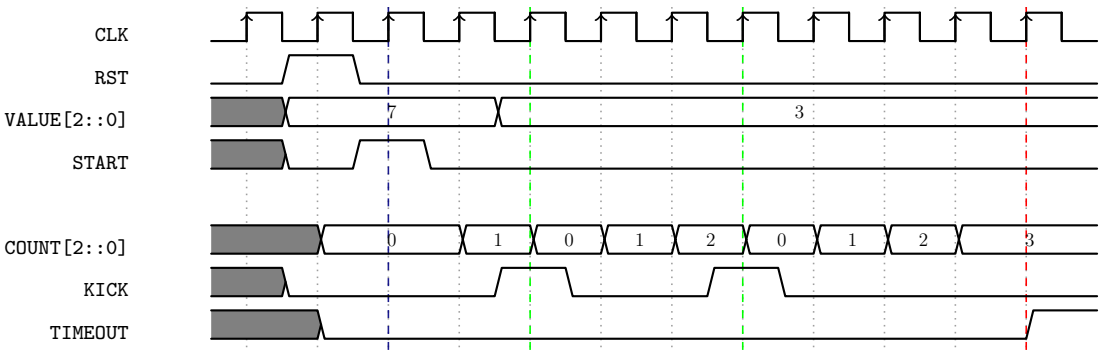


Figure 4.6: Timing diagram of a more sophisticated watchdog.

In the figure above, we have a different working mechanism compared to the previous one. After the reset signal *RST*, the timer is reset to its initial value of 0 and stays in this state until the *START* signal is received. At this point, the input value *VALUE* is set at 7 from the external: 7 is the final value of the timer, if this value is reached, the timer expires. Once the *START* signal is received (blue line), the timer starts counting up. The next clock cycle sees a change in the maximum value, from 7 down to 3. The *KICK* signal is generated two times (green line), signaling that the CPU is correctly working, and the timer is restarted from 0. At a certain point in time, the timer reaches 3 but the *KICK* signal is not arriving, thus at the next clock cycle, the *TIMEOUT* signal is generated (red line).

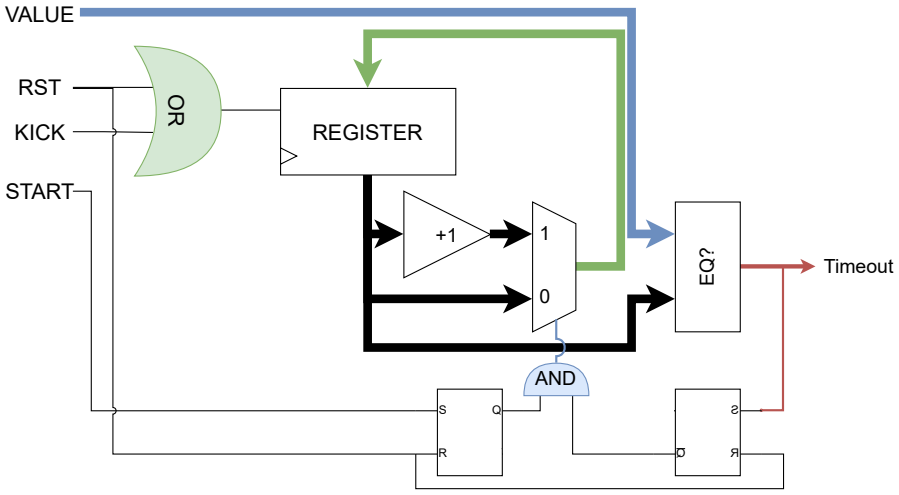
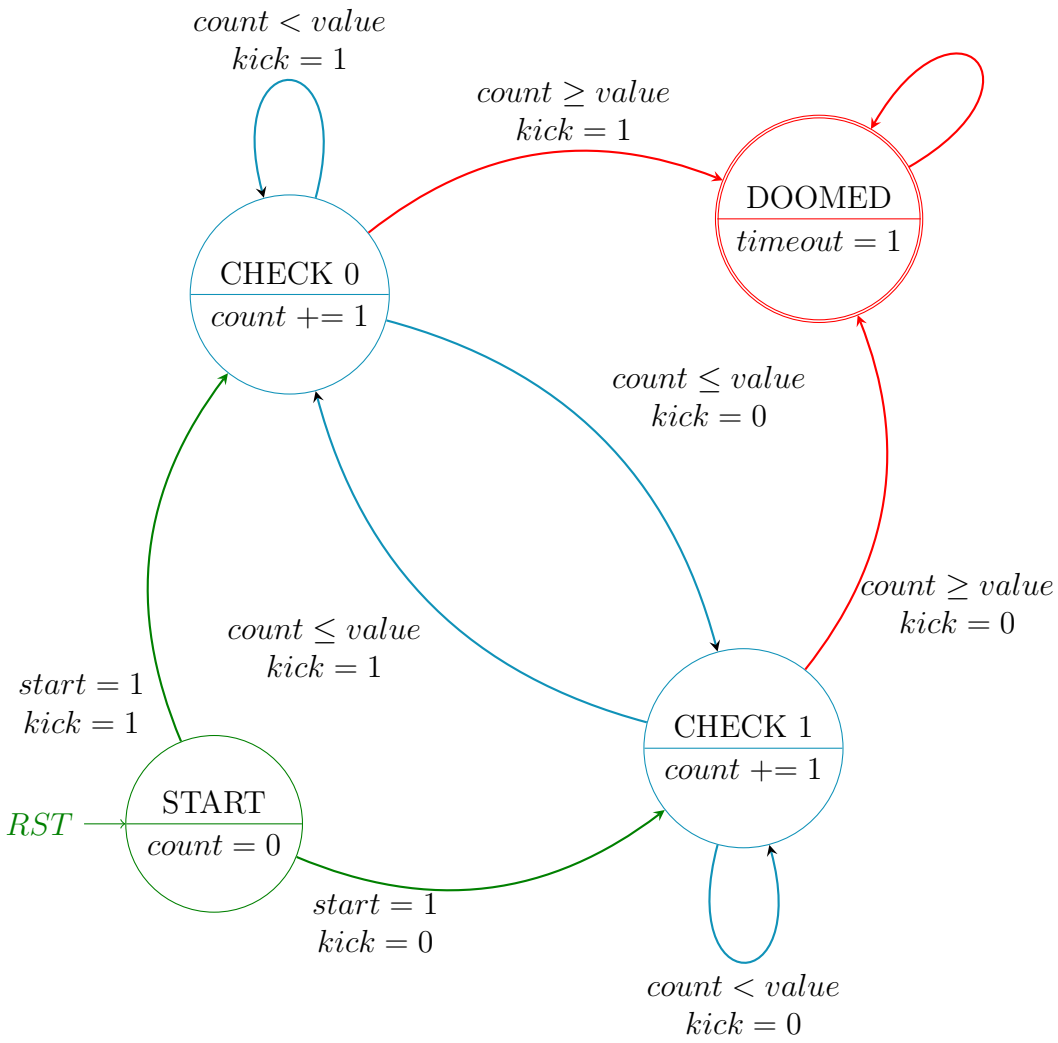


Figure 4.7: Possible digital circuit implementation of a watchdog.

This implementation can be used as it is and can achieve good results. A possible circuit implementation, even with some timing differences, can be seen in Figure 4.7. The problem is that the *KICK* signal represents a single point of failure in terms of fault tolerance. If the CPU is hit by an SEU, it can leave the signal stuck in a high state, and the watchdog will continuously reset the timer, maskin the real CPU's status.

To overcome this problem, a more sophisticated signaling mechanism is used. To keep things simple, the act of kicking is done at every $H \rightarrow L$ or $L \rightarrow H$ transition of the *KICK* signal. This way, if the signal is stuck in a certain state, the watchdog will expire correctly. The final circuit implementation is based on a Finite State Machine, that is, a set of states that are interconnected by transitions. It is faster to implement when things get more complicated, and it is easier to understand than the previous diagram. The following is a diagram of the Finite State Machine that implements the final version of the watchdog:



Formally speaking, this FSM is a Moore machine. In the theory of computation, it means that the current output values are determined only by the FSM current state [36]. Inputs only affect the next state, and state transitions may happen only at each rising edge of the clock signal.

The idea behind the logic of this FSM is that at the reset, the FSM waits for the START signal. Once it is detected, it goes into a loop between two complementary states. They are implemented in such a way to be able to detect signal transitions. The FSM stays in this loop until the count reaches the input value, and if this happens and the kick signal still doesn't toggle, the FSM goes into a final state asserting the timeout signal. The FSM will stay in this state until it is reset again. If the input timeout value changes, it is captured only when a transition is detected. The following is a description of the states in the FSM:

State	Output	Brief Description
START	COUNT = 0 TIMEOUT = 0 STARTED = 0	This is the initial state after the reset of the machine. Here the count stays at 0, waiting for start = 1. When the start signal is asserted, it goes to CHECK 1 or CHECK 0, depending on the current kick value (because of the transition detection, if kick = 0, the watchdog waits for kick = 1, $L \rightarrow H$ transition, and vice versa).
CHECK 0	COUNT += 1 TIMEOUT = 0 STARTED = 1	Watchdog started and waiting for the kick signal to go low. When the kick signal goes low, the FSM goes to CHECK 1, detecting the $H \rightarrow L$ transition. While the transition is not detected, the count keeps increasing at each clock tick. When the count reaches the input value, the watchdog goes to DOOMED.
CHECK 1		Watchdog started and waiting for the kick signal to go high. When the kick signal goes high, the FSM goes to CHECK 0, detecting the $L \rightarrow H$ transition. While the transition is not detected, the count keeps increasing at each clock tick. When the count reaches the input value, the watchdog goes to DOOMED.
DOOMED	COUNT += 0 TIMEOUT = 1 STARTED = 1	The watchdog is expired. The timeout signal is asserted and the FSM waits indefinitely until a reset arrives.

Table 4.3: Detailed explanation of the states of the FSM

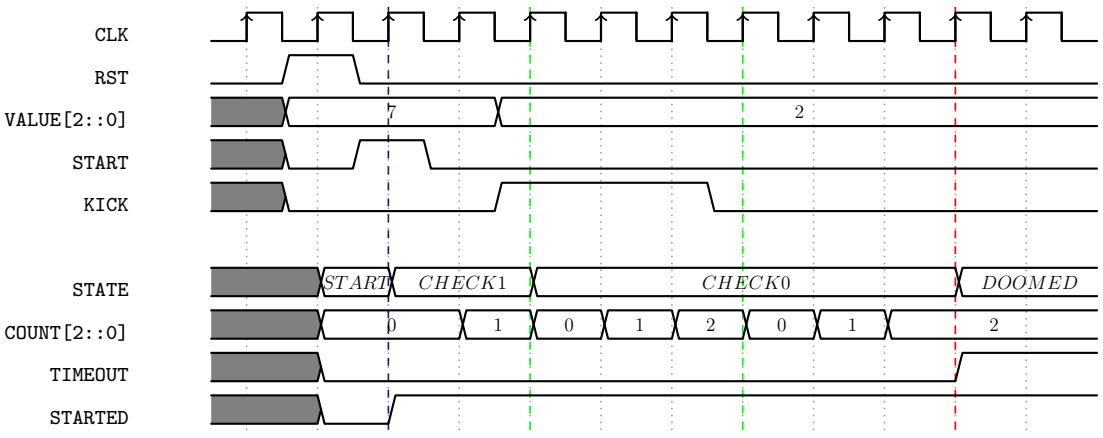


Figure 4.8: Timing diagram of a more sophisticated watchdog.

The FSM has been implemented as a High-Level State Machine (HLSM). An HLSM is a natural extension of an FSM, used to capture more complex behaviors, including multi-bit data inputs and outputs rather than just single bits, local storage and supports arithmetic operations like adds and comparisons, rather than just basic boolean operations. It has been implemented in VHDL and the code is available in Appendix A. In figure 4.8, the timing diagram of the final watchdog behavior is presented.

4.3.3 How to harden the watchdog?

Once the watchdog is implemented, it needs to be hardened. This is needed to prevent the watchdog from being inhibited by an SEU affecting it. The most basic type of fault tolerance technique is the Triple Modular Redundancy (TMR). It basically consists in having three different modules that, given the same input generate the same output in normal conditions. The three outputs are given as input to a voter circuit.

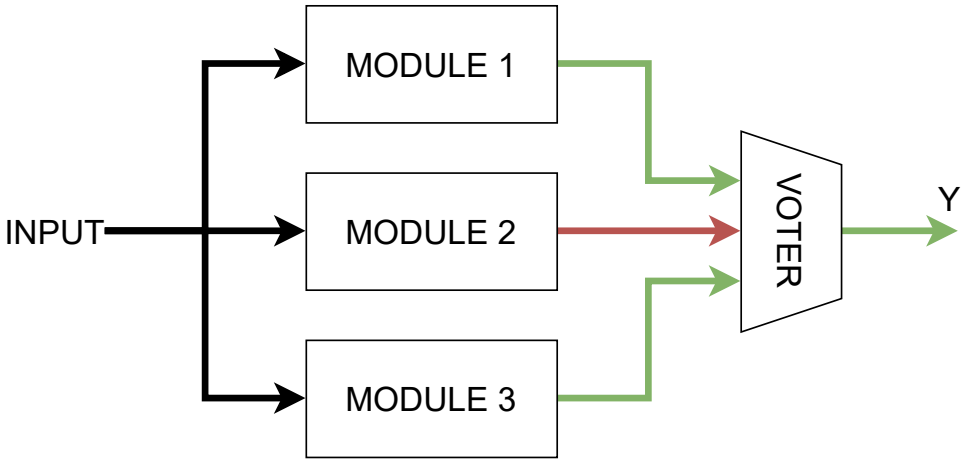


Figure 4.9: Triple Modular Redundancy (TMR) scheme.

The three modules can be identical or have a different implementation, but they must all generate the same output given the same input.

The voter circuit is a majority-voting system, which means that if all the three outputs are the same, the voter circuit gives as output one of the three inputs. If one of the three outputs is different (because one of the three modules is faulty), the voter circuit is capable of detecting the difference and outputs the non-faulty result. Thus, the fault is masked and it is not propagated to the rest of the system. The truth table of a 1-bit voter circuit is the following:

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 4.4: Voter truth table. The red cells indicate the faulty output.

It can be implemented as a circuit in the following way:

$$Y = \text{majority}(A, B, C) = AB + AC + BC = \overline{\overline{AB} + \overline{AC} + \overline{BC}} = \overline{\overline{AB} \cdot \overline{AC} \cdot \overline{BC}} \tag{4.1}$$

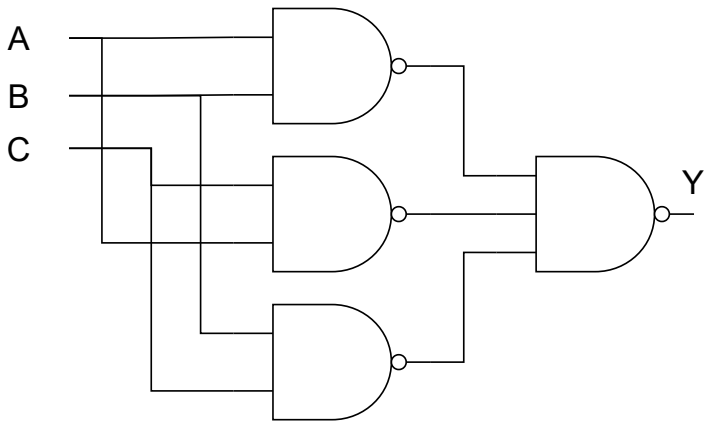


Figure 4.10: 1-bit voter circuit scheme.

The majority gate itself could fail, representing a single point of failure. This can be protected by applying triple redundancy to the voters themselves. Hence, three voters are used, one for each copy of the next stage of TMR logic, as shown in the following figure:

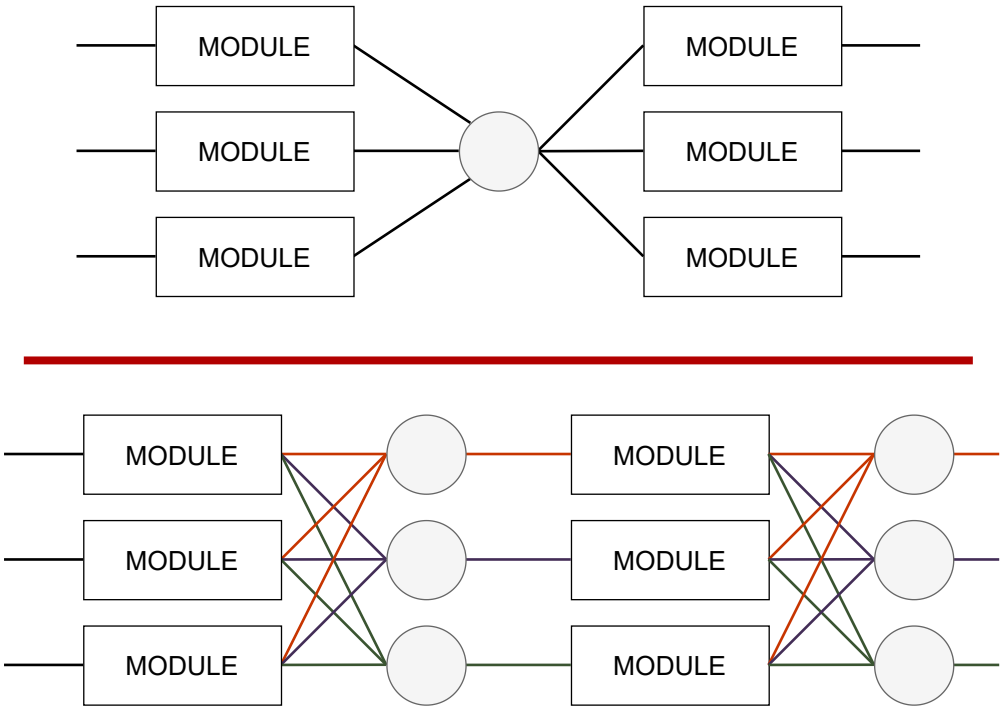


Figure 4.11: Basic TMR scheme vs. full TMR circuit scheme.

Even tho the second scheme doesn't present any single point of failure, most systems stick with the simplest scheme. This is because the majority of gates are much less complex than the systems that they guard against, so they are much more reliable. In those cases, by using some reliability calculations, it is possible to find the minimum voter realiability required to have a fully functional TMR scheme.

However, because this section is about how to harden a watchdog implemented in FPGA, even interconnections can be affected by faults due to SEUs affecting the route part of the configuration layer. Consequently, a full TMR scheme is the most reliable way to protect the watchdog.

Hence, the idea is to instantiate the watchdog component three different times and vote each input and output with three different voters. The overall design is implemented in Verilog. To simplify the code, two matrices of signals have been created: one for the no-tmr version and one for the tmr version. As an example, because of the full TMR scheme, there are three different *START* signals (at this point of the design, the driver of these signals is not yet specified). The idea is to access these 3 different instantiations of signals by using indexes and not directly the signals themselves with different names (for example *START_0*, *START_1*

and *START_2*) to not create a confusional code and to be able to automatize the code generation.

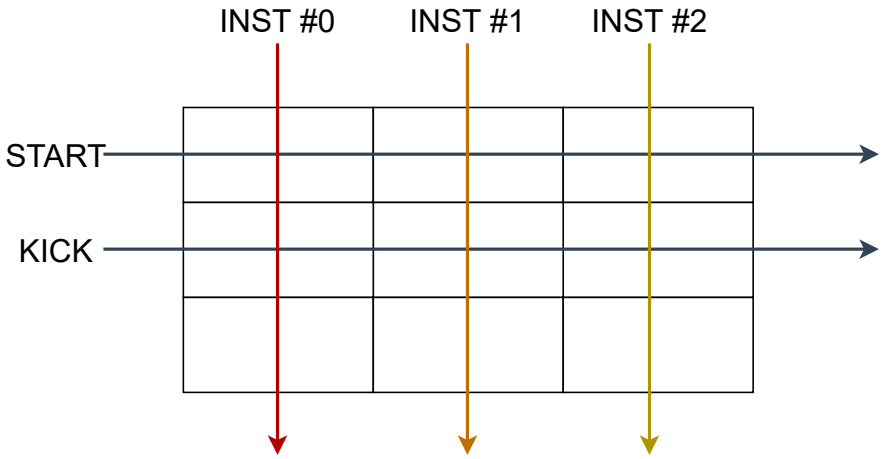


Figure 4.12: Input signal matrix for the no-tmr version.

The above scheme can be read as follows:

- The START (KICK) signal instance 0 is at row 0 (1), column 0.
- The START (KICK) signal instance 1 is at row 0 (1), column 1.
- The START (KICK) signal instance 2 is at row 0 (1), column 2.

The same concept applies to the output signals of the voters. Three voters create three different output signals and each voter has as input the same triplet of signals as the other ones. Hence, the tmr matrix works as the no-tmr one. The following table shows an example, supposing the no-tmr matrix is configured as the one shown in Figure 4.12 for what concerns the *START* signal:

Voter #	A	B	C	Y
1	`NOTMR[0][0]`	`NOTMR[0][1]`	`NOTMR[0][2]`	`TMR[0][0]`
2	`NOTMR[0][0]`	`NOTMR[0][1]`	`NOTMR[0][2]`	`TMR[0][1]`
3	`NOTMR[0][0]`	`NOTMR[0][1]`	`NOTMR[0][2]`	`TMR[0][2]`

Table 4.5: Output signal matrix for the no-tmr version.

The main problem with this solution is that all the three voters are performing exactly the same things on the same inputs and producing the same outputs. Ideally, this is the aim of the design, but once it is synthesized, everything is collapsed into

a single voter and a single watchdog. This happens because the synthesizer tries to optimize the design by reusing and sharing the logic between similar components. A situation that must be absolutely avoided, because the aim is exactly to have three different copies of everything: signals, voters and watchdogs.

To overcome this problem, the synthesizer must be notified about what it can optimize and what it can not. The following is an example of instantiation of the two matrices of signals and voters in Verilog, telling the synthesizer that they must not be optimized out:

```
(* dont_touch = "true" *) wire notmr[3:0][2:0];
(* dont_touch = "true" *) wire tmr[3:0][2:0];

generate
  genvar jdx;
  for (jdx = 0; jdx < 3; jdx = jdx + 1) begin
      for (idx = 0; idx < 4; idx = idx + 1) begin
          (* dont_touch = "true" *) voter_bus #(
              .NBITS(1)
          ) voter_ith (
              .DATA_IN0(notmr[idx][0]),
              .DATA_IN1(notmr[idx][1]),
              .DATA_IN2(notmr[idx][2]),
              .DATA_OUT(tmr[idx][jdx])
          );
      end
  end
endgenerate
```

Finally, the design is implemented. All the inputs that go to the watchdog are taken from the tmr matrix, and all the outputs generated by the watchdogs (for example the three timeout signals) go to the no-tmr matrix. The latter are automatically voted because of the previous instantiation of the voter components, as shown above. The final outputs of the overall TMR Watchdog are connected to the respective TMR version of the signals.

Figure 4.13: Interfaces of the TMR Watchdog.

4.3.4 Integration of the watchdog in the design

The watchdog is finally created but it is still not easily usable. The goal is to have a watchdog that can be easily inserted into any design. To achieve this, the watchdog can be packaged as an IP to allow the user to easily integrate it with the Block Design tool. Vivado offers an easy-to-use IP creation wizard, through which a design can be packed and distributed, together with a C library for high-level interaction with the hardware via software drivers.

Unfortunately, this is not enough. The idea is to connect the watchdog to the MicroBlaze, in this way it would be able to control it (for example the kicking action) and check its status. The problem is that the watchdog uses discrete signals as an interface and there is no direct way to access these signals from the CPU. There are two possible ways to overcome this situation:

- The MicroBlaze controls the watchdog via a GPIO peripheral. This is the most simple way to do it, but it is not very flexible. The various GPIO channels are connected to the watchdog signals and the CPU can control the GPIO channels via the AXI interface.
- The MicroBlaze direct interfaces with the watchdog through a dedicated AXI interface. This is the most flexible way to do it but increases the implementation complexity of the watchdog IP.

The second option is the most convenient because there are fewer actors in the overall fault-tolerance chain that can be affected by faults. Luckily, the Vivado IP creation wizard provides an automatic way to create an AXI wrapper on top of which the watchdog can be instantiated and the different signals connected to the various registers. Hence, the watchdog can be controlled and monitored by the MicroBlaze via those, as shown in the following.

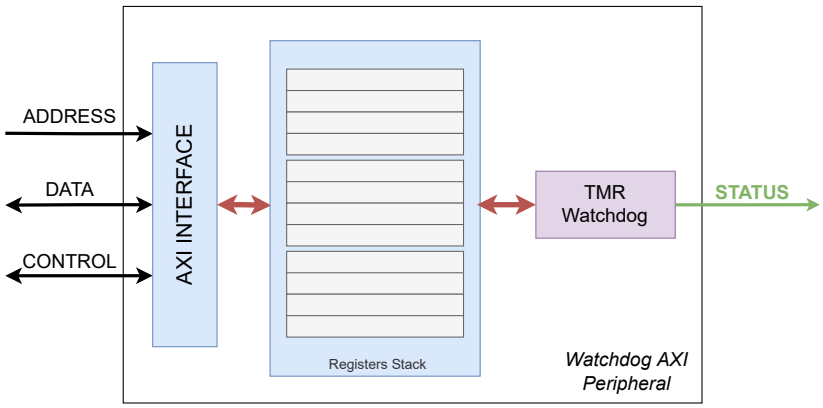
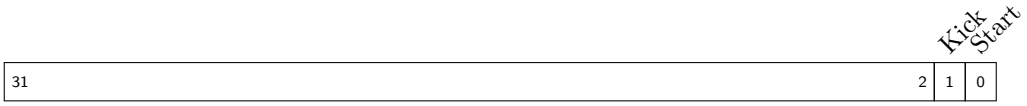


Figure 4.14: Conceptual representation of the final watchdog IP.

By design choice, there are 4 registers, one for each of the 3 watchdog instances. The second nibble in the lowest byte of the address is the index of the corresponding watchdog. They follow the Full TMR design as explained in the previous section, internally coded with no-tmr and tmr matrices. Read-only fields are taken from the no-tmr matrix, in this way the CPU or any other AXI master can check if there are faulty modules.

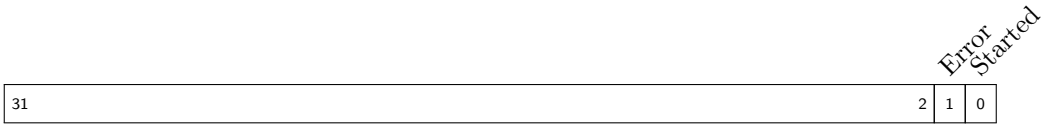
The following is a description of how the register space is divided and seen by AXI masters:

Register 4.1: Control Register (0x00 - 0x10 - 0x20)



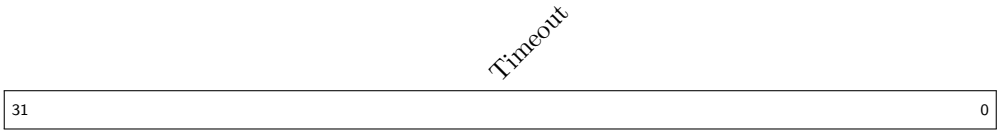
Start	Setting this bit activates the Watchdog. If already activated, this bit has no effects.
Kick	The kick bit is directly connected to the kick signal of the watchdog. The MicroBlaze is in charge of writing the right value in this field (i.e. do the right transitions).

Register 4.2: Status Register (0x04 - 0x14 - 0x24)



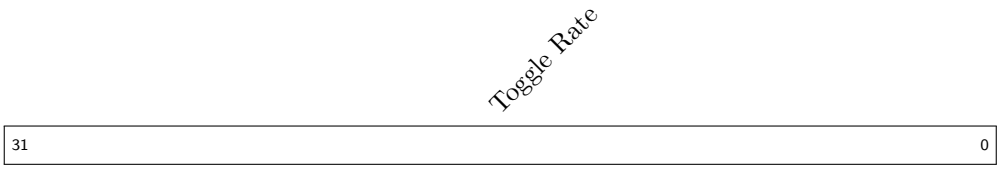
Started	The bit is set if the watchdog is started.
Error	The bit is set if the watchdog timed out.

Register 4.3: Timeout Register (0x08 - 0x18 - 0x28)



Timeout	32 bit value that represents the number of cycles required before the watchdog times out.

Register 4.4: Toggle Rate Register (0x0C - 0x1C - 0x2C)



Toggle Rate	32 bit value that represents the actual toggling rate of the watchdog, in clock cycles. The CPU can use it to estimate, for example, if there are differences between the expected toggling rate and the real one.

To ease the job of programmers, the Watchdog IP offers a driver with a set of functions to control the watchdog. It is based on a newly defined data type that represents the Watchdog register memory space conveniently, by using C structs, unions and bitfields constructs. The full description is shown in Appendix B.

```
typedef struct {
    union {
        u32 *baseAddress;
        watchdog_module_t *module;
        struct {
            watchdog_module_t module0;
            watchdog_module_t module1;
            watchdog_module_t module2;
        } *modules;
    };
} GBcnCtrl;
```

Once the Watchdog IP repository han been added to the list of IP repositories in Vivado ($Vivado \rightarrow IP \rightarrow Repository \rightarrow +$), it can be instantiated in the design and connected to a AXI Interconnect to make it accessible to masters, like the MicroBlaze, via an intermediate AXI Interconnect:

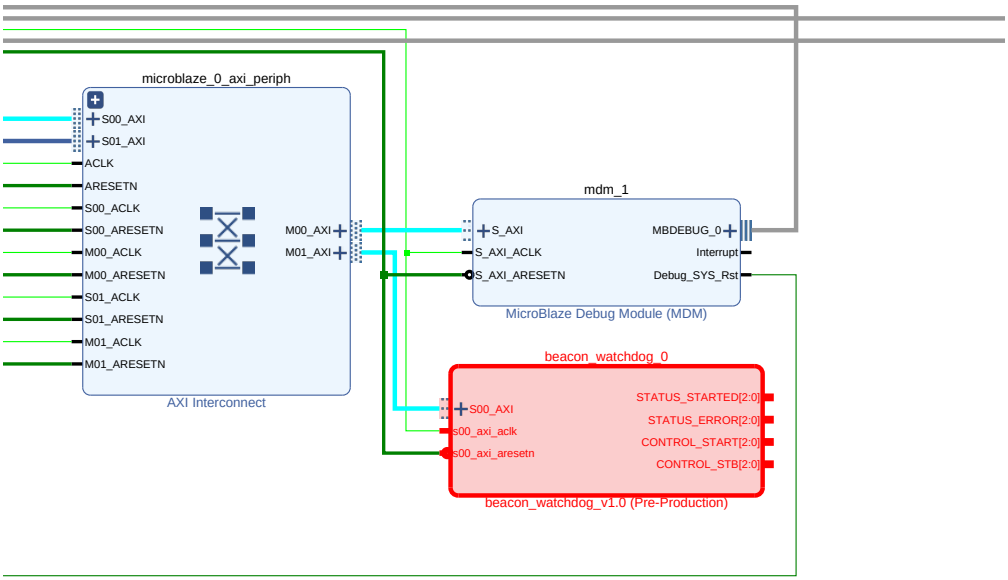


Figure 4.15: Instantiation of the Watchdog IP.

As conceptually shown in Figure 4.14, the IP presents two interfaces:

- *S00_AXI*: The AXI interface is used to communicate with the watchdog from an AXI master. It includes the s00_clk and s00_rstn signals.

- *Custom Interface*: The custom interface is meant to expose some signals towards other peripherals that may or may not need it. Each of those signals is a vector of 3 sub-signal, one for each watchdog module. In particular, the *CONTROL_START* and *CONTROL_STB* are directly connected to the register's bits Start and Kick without any TMR transformation. The *STATUS_STARTED* and *STATUS_ERROR* are instead the TMR version of the watchdog's started and timeout signals, respectively.

The IP can be easily customized via a custom IP wizard or via TCL commands. The following is the GUI wizard:

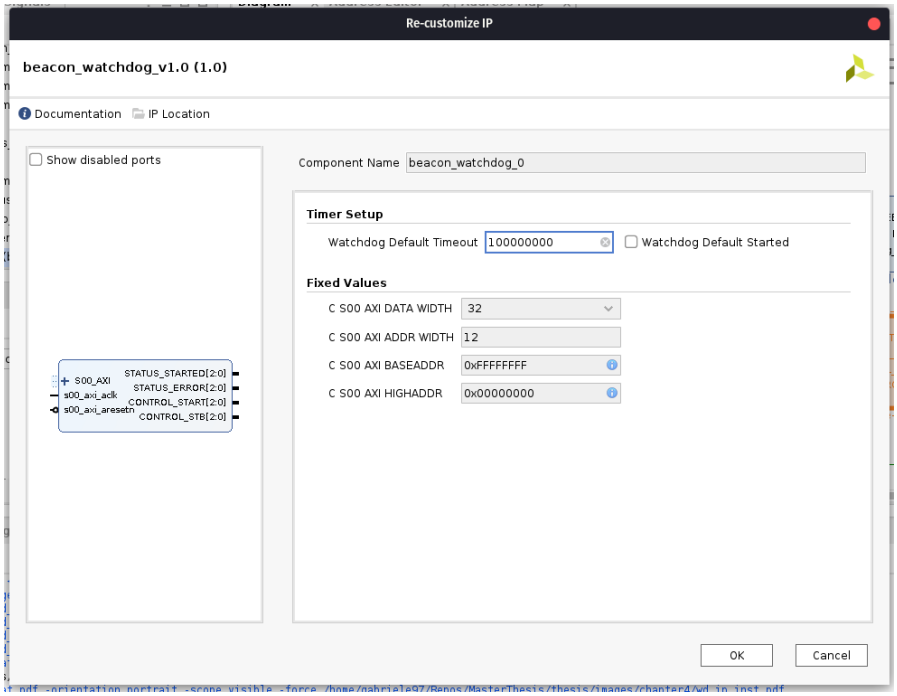


Figure 4.16: Watchdog IP configuration wizard.

It is possible to set a custom initial value for the timeout register and the default start condition of the watchdog: if it is checked, at each reset the watchdog is immediately started without waiting for the *Start* bit to be set. This can be useful for different purposes, but in particular for fault injection testing.

Finally, when the IP is instantiated in a Block Design and the final .xsa file is generated, watchdog drivers are automatically inserted and ready to be used with Vitis. Thus, it is able to generate a Board Support Package (BSP) that contains and defines these new drivers.

Follows an example of the usage of the drivers to control the watchdog. It demonstrates the usability of the final design, with a set of very high-level functions. The example demonstrates a possible way to organize the kicking act across the code to cover not only CPU hangs but also other faults (computational errors, self-test code, etc.).

```c
#include "beacon_watchdog.h"

/* header file for the hardware parameters,
like peripheral addresses */
#include "xparameters.h"
int main() {
  GBcnCtrl hBcn; //Handle to the watchdog
  uint32_t value;

  value = XPAR_BEACON_WATCHDOG_0_S00_AXI_BASEADDR;
  GBcnCtrl_Initialize(&hBcn, value);

  // Timeout set to 2 seconds (double of the watchdog's clk freq)
  GBcnCtrl_SetTimeoutValue(&hBcn, XPAR_CPU_CORE_CLOCK_FREQ_HZ<<1);
  GBcnCtrl_Start(&hBcn);

  value = hBcn.modules->module0.DATAREG;
  printf("Module 0 Timeout Value: %d\r\n", value);

  value = hBcn.modules->module1.DATAREG;
  printf("Module 1 Timeout Value: %d\r\n", value);

  value = hBcn.modules->module2.DATAREG;
  printf("Module 2 Timeout Value: %d\r\n", value);

  while(1) {
    checksum ^= (res += 2);

    // Example of self test check on checksum
    if(checksum & 0x3 == 0x0) {
      // FAULT!! Stop kicking, hardware needs to be reset!
      while(1); // Waiting for timeout expiration
    }

    // Kicking (automatic transition detection)
    GBcnCtrl_Toggle(&hBcn);
  }

}
```

4.4 Design with Partial Reconfiguration

Partial reconfiguration is a way to change or update an FPGA design without the need to re-program the whole FPGA. Hence, it is possible to reprogram only a portion of the design without interrupting the rest of the design. In Xilinx's world, this is called *Dynamic Function Exchange* (DFX).

There are different reasons why people do this. The main reason is *area*: perhaps the FPGA is already full but more functions need to be pushed to the FPGA, and a way to achieve this is to partially reconfigure the design. Hence, the whole system is analyzed and blocks that are not used at the same time are grouped together. Each of the identified groups is assigned to a piece of FPGA fabric and that portion is marked as reconfigurable.

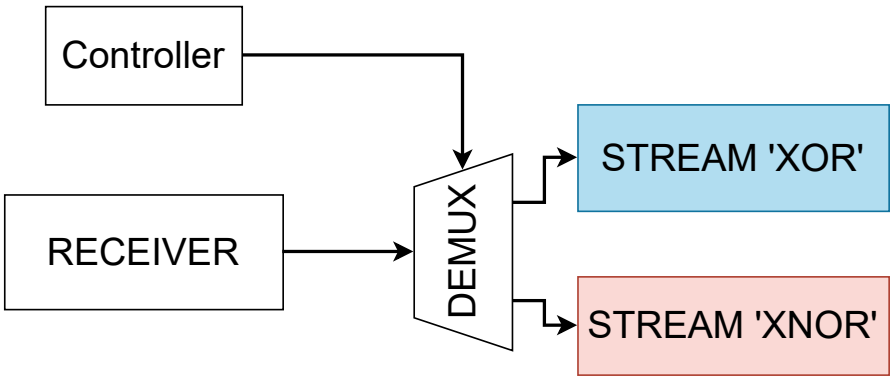


Figure 4.17: Example of partial reconfiguration modules.

In the above figure, an example is shown. There is a receiver module that receives data from some source (it can be an Ethernet interface for instance). The system needs to compute a checksum of the received data, but not always in the same manner. Hence, two modules compute the checksum in two different ways. The whole design as it is does not fit in the FPGA, but luckily the two modules are not used at the same time so they can be grouped. The group is then marked as reconfigurable and the controller, before requesting to compute a checksum to the needed module, reconfigures the area with the module that the controller designed as the one needed to compute the checksum. In this way, the system continues to work in other sub-parts of the design, and meanwhile, the partial reconfiguration and the checksum computation are performed. The final design is the following:

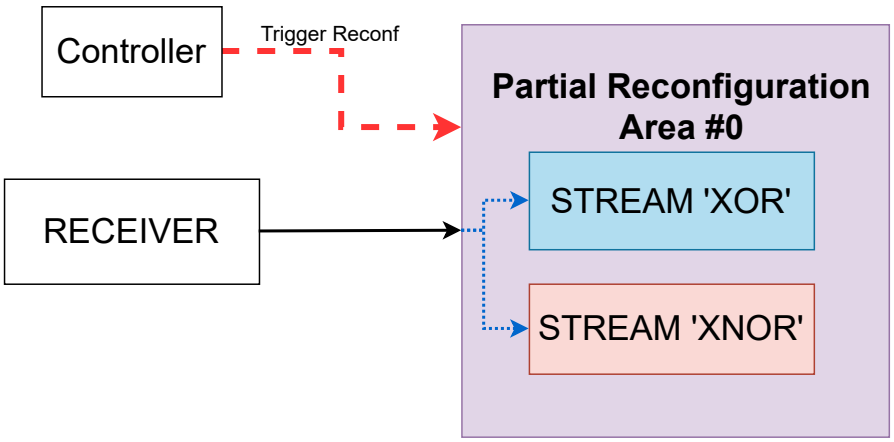


Figure 4.18: Example of modules grouped under a partial reconfiguration area.

Furthermore, partial reconfiguration is not limited only to functionality exchange. It can be used to implement a *partial scrubbing* of the design, for example when a fault affecting a reconfigurable area is detected. This is the case under study for what concerns this thesis work, as shown in Figure 4.4. This allows to reconfigure the area without interrupting the rest of the design and mainly it is faster, reducing the down-time of the system, thus improving the overall availability. Moreover, the bit-flip causing the fault is definitely removed thanks to its reconfiguration.

For what concerns Xilinx, the Dynamic Function Exchange is achieved by downloading a *partial bitstream* into the FPGA via a *Configuration Access Port* (CAP) that can be accessed directly from the FPGA itself. The partial bitstream is generated by adding some steps to the standard Vivado Design Flow.

As explained in Section 3.3.2, ZYNQ systems offer two dedicated ports, one for the PS and one for the PL. The PS port is the Processor Configuration Access Port (PCAP) and the PL port is the Internal Configuration Access Port (ICAP). As explained, they are mutually exclusive.

4.4.1 Vivado Design Flow for Dynamic Function Exchange

Unluckily, Vivado versions prior to 2021.2 do not support DFX for Block Designs. Hence, for what concerns modules to be grouped under a partial reconfiguration partition, the designer must move the reconfigurable IP instances outside the Block Design and manually instantiate them in the top-level module.

As a reference, a design with a custom 2-bit counter IP is used. The designer wants to have another counter IP to be used alternatively that is able to count up to

15 (4-bit counter), without using more area. Hence, the two counters are grouped under a partial reconfiguration partition. Each counter is defined as Reconfigurable Module (RM) and within a partition, only one RM at a time can be available.

A reconfigurable partition is seen by other modules as the same component: other modules do not need to care about the reconfigurable module. Hence, a partition is defined by a single HDL wrapper with the same port definitions for all the modules in the partition. Thus, if there are two IPs and both of them have a CLK and Reset port and the third port is different (the first IP has a 2-bit port while the second one has a 4-bit port), a common definition needs to be found. In this simple case, the third port in the partition wrapper will have 4 bits and the IP with the 2-bit port is extended to four by fixing the higher bits at 0.

Through Vivado's IP Catalog, first, a *Binary Counter* IP is instantiated with a 2-bit counter. Then a *Binary Counter* IP is instantiated with a 4-bit counter. Once an IP is instantiated, a .xci file is generated. A .xci file is an XML file that records the values of project options, customization parameters, and port parameters used to create the IP. Once this is done, one wrapper for each IP must be created with the same port definition. The wrapper related to the 2-bit counter is instantiated in the top-level module. This means that the 2-bit counter is designed as the default module and it is inserted in the full bitstream, leading to a normal bitstream and design generated until now. Furthermore, there is a high chance that the reconfigurable modules (RMs) need something from the Block Design part. In that case, the designer must make available to the outside the needed signals to connect the reconfigurable modules, through the usages of Block Design's port definition both for input and output signals.

Once the sources are defined, the Vivado Project must be converted into a DFX-capable Project. This can be done via Tools → Enable Dynamic Function Exchange. This is an irreversible process. It is possible now to define a new partition with a right-click on the main wrapper and select *Create Partition Definition*. Once the partition is created, it is possible to see it inside the *Partition Definitions* tab. Here, a default reconfigurable module is created with the wrapper and the .xci file previously selected. It is possible to create a new reconfigurable module inside the partition where the .xci and wrapper of the second IP must be added (via a manual selection of the files).

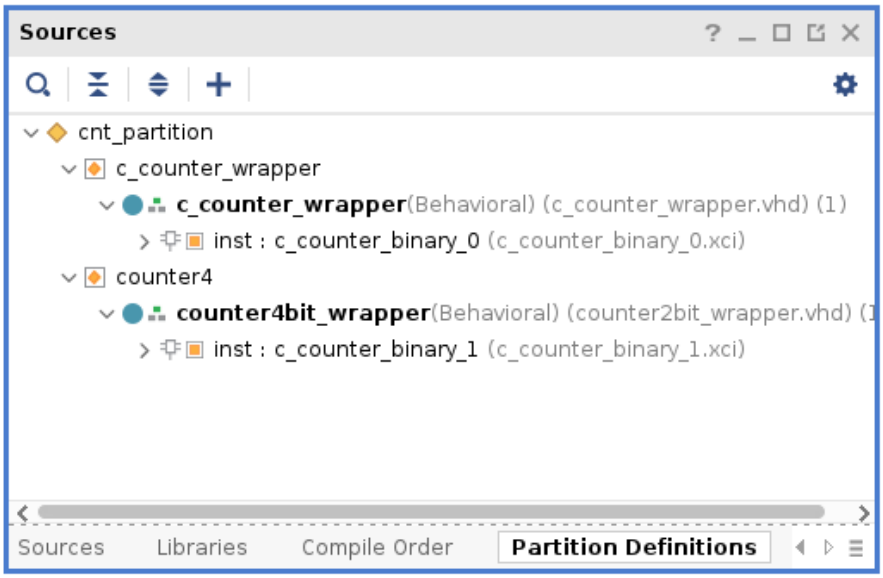


Figure 4.19: Partition definition with two reconfigurable modules, each one with its own wrapper and .xci file.

After the partition definition is completed, it is time to tell Vivado how the possible configurations for the various partitions are defined. Consequently, Vivado will be able to create bitstreams for each of the configurations indicated. To achieve this, on the left sidebar click on *Dynamic Function Exchange Wizard*. For this simple example, there are two configurations: one with the 2-bit counter and the other with the 4-bit counter.

Finally, it is possible to start the Synthesis process. After it has finished, a new set of contrainsts must be defined. In particular, the default wrapper instantiation (the one indicated in the top-level module) must be inserted in a PBLOCK. Once created, the PBLOCK is automatically set as `IS_SOFT = 0`. This represents the area of the FPGA marked as reconfigurable for the partition the instance belongs to.

At this point, everything is defined and ready to be implemented onto the FPGA. If the Flow is executed until the final *write_bitstream* step, the full bitstream is generated (as usual) and one partial bitstream for each of the configurations is generated. In particular, for each configuration a different run is executed, thus for each of them, the implementation and write_bitstreams steps must be performed. Of course, partial bitstreams have a lower size, where the size is logically direct proportional to the area of the PBLOCK:

Bitstream Type	Partition	Size
Full	Whole Design	3.9 MB
Partial	Count 2 Module	640 KB
Partial	Count 4 Module	640 KB

Table 4.6: Comparison between full bitstream and partial bitstreams sizes.

As shown in the table above, the two partial bitstreams have the same size. This is because both of them reconfigure the same PBLOCK, thus the same area. It is possible to test the partial reconfiguration with XSCT using the following script, as an example:

```
connect
target 4 # targets the FPGA

# loads the full bitstream
fpga design_1.bit

# loads the partial bitstream after the full one has been loaded
   previously
fpga -partial design_1_inst_counter4bit_wrapper_partial.bit
```

4.4.2 DFX with MicroBlaze in Vivado 2021.1

In the previous section, an example of Dynamic Function Exchange is presented. As explained, with versions of Vivado older than 2021.1, the DFX flow requires a manual instantiation and management of the various IPs involved in the reconfigurable part. With newer versions, indeed, a new feature called Block Design Containers (BDC) allows users to segment designs into multiple block designs, enabling modular and team-based design flows, including DFX flows.

A BDC can be set as reconfigurable, turning it into a Reconfigurable Partition (RP) and enabling each design source within it to be considered an RM. The DFX Wizard populates each RP with all possible RMs for each RP before defining Configuration and Configuration Runs, similar to the RTL project flow for DFX.

However, using simple IPs or simple HDL designs, the manual flow outside the Block Design tool is feasible. Unfortunately, when a designer wants to partial reconfigure a MicroBlaze, two problems arise, where the second one is a direct consequence of the first:

1. deciding to partial reconfigure a MicroBlaze with the flow described previously becomes immediately unfeasible due to the high number of ports and signals to manually manage outside the Block Design environment.

2. softwares like Vitis are based on the description included in the .xsa file. This file is generated, as explained previously, using Vivado. This description file is exclusively based on the Block Design tool part of the project, because Vivado has no way to understand how the designer is connecting and configuring IPs outside the Block Design. Thus, working with Vitis becomes impossible because Vitis will tell the user that there is no MicroBlaze available in the design.

For what concerns this thesis work, a *hack* is necessary to solve those problems. When the Block Design tool is used, internally Vivado creates a single HDL file (VHDL or Verilog, it depends on the settings of the project itself). This HDL file contains the whole description of the Block Design and all the necessary signal connections. This file can be copy-pasted in a completely different project and set as a top-level module. This is the chosen way to overcome the first problem: avoid manually instantiating the MicroBlaze IPs. This creates a new problem: all the IPs definitions (.xci files) used in the Block Design are not available in the new project, so it is not synthesizable basically. This problem can be easily solved by importing the IPs definitions from the original project to the new one.

This new project is now ready to be configured and make the MicroBlaze a Reconfigurable Module inside its own reconfigurable partition. Hence, it is possible to follow the steps described in the previous section. The only operation to do *manually* is to substitute the MicroBlaze instance with its own wrapper, that can be generated easily. Only a configuration is possible (in the most basic scenario as this one), thus Vivado generates only a partial bitstream together with the full bitstream as usual.

The second problem is still there. From a purely HDL description of the design, Vivado is not able to generate a .xsa description. However, the two projects are practically the same, so the .xsa file originated from the base project can be extracted (it is essentially a .zip file) that contains a .xml description of the peripherals and memories, available CPUs, software drivers and the full bitstream to use to let Vitis program the FPGA. Basically, the bitstream can be substituted with the one generated by the second project (containing the Reconfigurable MicroBlaze) and the game is done.

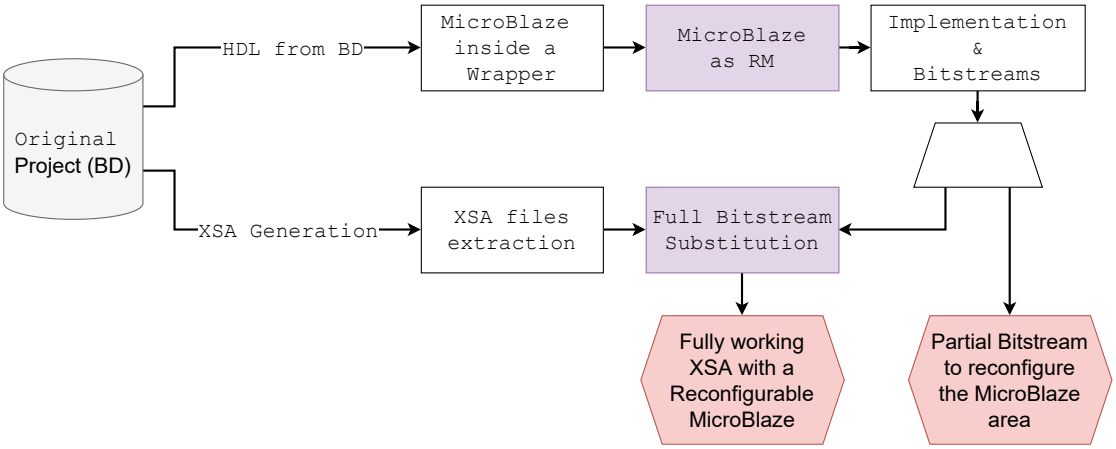


Figure 4.20: Flow used to generate a fully working Reconfigurable MicroBlaze design.

4.4.3 Xilinx DFX Controller

During the previous sections, DFX has been successfully achieved both for simple modules and for more complex ones like a full MicroBlaze. However, there is still no way to perform the partial reconfiguration from within the FPGA itself.

Luckily, Xilinx provides an IP that is able to manage the partial reconfigurations from within the FPGA and ease the process for designers and developers. It is the Xilinx Dynamic Function eXchange Controller (DFX Controller) IP core, that provides management functions for self-controlling partially reconfigurable designs.

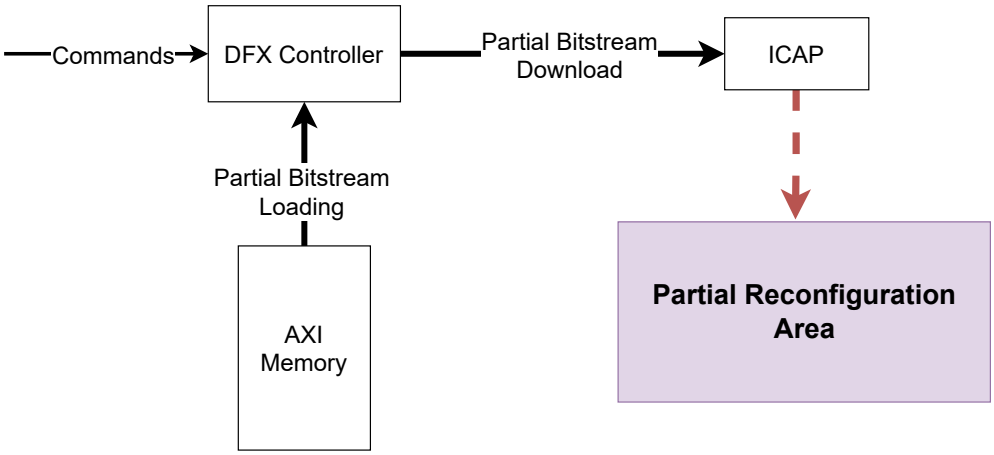


Figure 4.21: Basic scheme of the DFX Controller flow.

As shown in the figure above, the DFX Controller's main job is to fetch the partial bitstream data from an AXI memory peripheral and send it to the ICAP. This action is performed under certain commands (or triggers) sent from another actor in the system.

The DFX Controller can be managed both statically via a GUI Wizard (or TCL) during the IP definition or dynamically from a CPU (like the MicroBlaze) or any other master in the system using a dedicated AXI interface. Internally it is organized as a series of Virtual Sockets. It supports up to 32 Virutal Sockets and each Virtual Socket can manage up to 128 Reconfigurable Modules. A Virtual Socket allows to manage each RM inside it as a whole group: it means that reset signals and other signals are shared between all the RMs inside it.

For what concerns this thesis work, a Virtual Socket with a single Reconfigurable Module is enough to manage the MicroBlaze reconfiguration. For each Reconfigurable Module, the DFX Controller allows defining the memory address where the partial bitstream can be found and the size in bytes. Moreover, it allows to set up the type of reset required. In fact, when a module is reconfigured, it should be reset to be sure that it starts from a clean state.

Last but not least, the DFX Controller allows to trigger a reconfiguration via:

- a software trigger, via DFX Controller's AXI registers.
- a hardware trigger, via a dedicated interface.

4.5 Integration of the watchdog and the DFX

It is finally time to put everything together. The scheme to apply is shown if Figure 4.4, where the watchdog is a simple IP that can be used to trigger the MicroBlaze reconfiguration when an error occurs. The DFX Controller is used to manage the reconfiguration process.

4.5.1 Partial bitstream storage

The first problem to solve is the storage of the partial bitstream. As an example, the generated partial bitstream for the MicroBlaze RM measures 389928 bytes (around 380 KB). A possible storage solution would be the usage of the BRAMs available inside the FPGA and connecting them to the AXI bus via the AXI Interconnect using a Local Memory Controller IP. Unluckily, the used FPGA does not have so many BRAMs available (and a few are used as Instruction and Data memory for the MicroBlaze itself), thus this solution is unfeasible. The proposed solution is to

use the on-board DDR memory, which offers 512 MB of storage easily accessible via the AXI bus by enabling the Slave interface *S_AXI_HPO* in the PS's IP (ZYNQ7 Processing System IP). This interface can be connected to the AXI Interconnect as other slaves, like the peripherals. Once connected, all the masters in the AXI Bus can access the PS's internal address space from `0x00000000` to `0x1FFFFFFF`. As shown in Table 3.1, at address `0x00000000` there is the On-Chip Memory (OCM) where usually the ARM cores executes the code from. To avoid conflicts, only the second half of the DDR is used (so from `0x10000000` to `0x1FFFFFFF`).

To make things easier, the partial bitstream is statically inserted in the .elf file executed by one of the two ARM cores. To achieve this, the partial bitstream is first converted to a C array of 32-bit items, via a script explained in Section 4.6.2. However, this is not enough because the array would be allocated in the `.text` section of the .elf file, that is placed at low memory addresses (below the `0x10000000` target). To solve this, in the linker script generated by Vitis (regarding the ARM0 project), a new memory portion is defined: it points to the second half of the DDR memory. Then, a new section called `.partialbs` is created. The following is an extract of the linker script:

```
/* Define Memories in the system */
MEMORY
{
   ps7_ram_0 : ORIGIN = 0x0, LENGTH = 0x30000
   ps7_ram_1 : ORIGIN = 0xFFFF0000, LENGTH = 0xFE00
   /*2nd DDR half*/
   2ndhalf   : ORIGIN = 0x10000000, LENGTH = 0x10000000
}
/* Define the sections, and where they are mapped in memory */
SECTIONS
{
  .text : {
    KEEP (*(.vectors))
    *(.boot)
    *(.text)
    /*-------*/
    *(.vfp11_veneer)
    *(.ARM.extab)
    *(.gnu.linkonce.armextab.*)
  } > ps7_ddr_0
/** other sections **/
  .partialbs : {
    *(.partialbs)
  } > 2ndhalf
}
```

GCC (the C compiler used by Vitis) allows to map memory regions as preferred as well as to force the placement of a certain variable in a certain place in memory. This can be achieved as follows:

```
u32 __attribute__((section(".partialbs"))) data[] = {
  0xFFFFFFFF,
  0xFFFFFFFF,
  0xFFFFFFFF,
  0xFFFFFFFF,
  0xFFFFFFFF,
  0xFFFFFFFF,
  /**....**/
  0xAA995566, // SYNC WORD
  /**....**/
};
```

4.5.2 How to enable the ICAP port

For this work, the ARM core is not only used to load the partial bitstream in memory. As explained previously, the ARM cores hold full control of the two configuration ports (ICAP and PCAP) for security purposes. Before executing the partial configuration, the ARM core must release the hold on the ICAP and this is done in two consecutive steps:

1. The ARM core disables the PCAP (because they are mutually exclusive).
2. The ARM core enables the ICAP.

This can be done easily with the XDevCfg driver offered by Xilinx:

```
#include "platform.h"
#include "data.h" // Contains the partial bitstream array
#include "xdevcfg.h"

int main() {
  XDcfg XDcfg_0;

  XDcfg_Config *conf = XDcfg_LookupConfig(XPAR_XDCFG_0_DEVICE_ID);
  XDcfg_CfgInitialize(&XDcfg_0, conf, conf->BaseAddr);
  XDcfg_DisablePCAP(&XDcfg_0);
  XDcfg_SelectIcapInterface(&XDcfg_0);

  while(1);
}
```

4.5.3 ICAP instantiation

The second last step is to connect the DFX Controller to the ICAP port. For security reasons, it is not directly available to the user. It is an HDL primitive that needs to be manually instantiated in the design so it needs to be manually inserted in the HDL code generated by the Block Design Tool and connected to the ICAP interface of the DFX Controller.

The ICAP port can be accessed via different blocks, but the most simple is the ICAPE2. It offers different input and outputs port, as shown below:

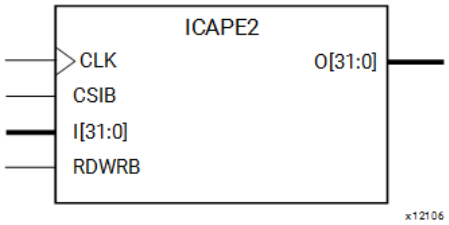


Figure 4.22: ICAP Interface.

The following is a short description of the ports:

Port	Data Width	Description
`CLK`	`1 bit`	The clock signal (max 100 MHz $\rightarrow$ 32 bit data bus = $3.2GB/s$).
`CSIB`	`1 bit`	The chip select signal (active low).
`RDWRB`	`1 bit`	The read/write signal.
`I`	`32 bit`	The input data.
`O`	`32 bit`	The output data.

Table 4.7: ICAPE2 Interface description.

The ICAP interface can be use used to monitor (for example via a Integrated Logic Analyzer IP, as explained in Section 3.5) the configuration process when it is used as port for delivering bitstreams. The `O` port of the ICAPE2 block is a 32-bit bus, but only the lowest byte is used. The mapping of the lower byte is as follows:

Register 4.5: ICAP (`O` port)

CFGERR_B Configuration Error:
0 = A configuration error has occurred.
1 = No configuration error.

DALIGN Sync word (`0xAA995566`) received:
0 = No sync word received.
1 = Sync word received.

RIP Readback in Progress:
0 = No readback in progress.
1 = A readback is in progress.

IN_ABORT_B ABORT in progress.
0 = Abort is in progress.
1 = No abort in progress.

The following is the VHDL code for instantiating the ICAP port in a design:

```
ICAPE2_inst : ICAPE2 generic map (
   DEVICE_ID => X"3651093",     -- Simulation only.
   ICAP_WIDTH => "X32",         -- Input/Output data width.
   SIM_CFG_FILE_NAME => "NONE"  -- Simulation only.
) port map (
   O => O,          -- 32-bit output: Configuration data output bus
   CLK => CLK,      -- 1-bit input: Clock Input
   CSIB => CSIB,    -- 1-bit input: Active-Low ICAP Enable
   I => I,          -- 32-bit input: Configuration data input bus
   RDWRB => RDWRB   -- 1-bit input: Read/Write Select input
);
```

An example of the ICAPE2 during the initial phases of the configuration process is shown in Figure 4.23. When the trigger is asserted (`vsm_VS_0_hw_triggers_1`), the DFX Controller starts sending the configuration data taken from the partial bitstream uploaded in the memory. The ICAP's `O` port initially is at `0x9B` that

means *no cfg error*, *no sync word*, *no readback in progress*, *no abort in progress*. When the SYNC WORD is received, the ICAP's `O` port is updated to `0xDB`: only 1 bit changed, indicating that the sync word was received.

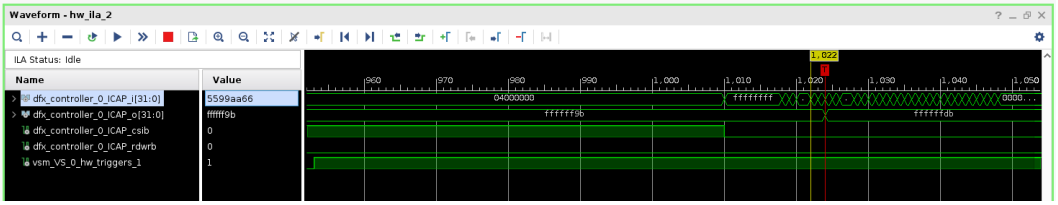


Figure 4.23: ICAP Interface as seen by an ILA core.

4.5.4 Connection of the Watchdog and the DFX Controller

All the actors are now ready to be integrated. In particular, the *timeout* signal of the watchdog should be used as a trigger for the DLX to start the reconfiguration. Moreover, when the DFX finishes the reconfiguration, both the MicroBlaze and the Watchdog must be reset to avoid any unexpected behavior and to restart the watchdog's timer itself (as explained in Section 4.3.2, the watchdog is designed to remains in the *DOOMED* state until a reset arrives).

First, the DFX Controller is configured to allow hardware triggers. This means setting up a signal width of 1 bit for the `vsm_VS_0_hw_triggers` signal and to assign the MicroBlaze RM previously configured as the RM to load when the triggers arrive. The problem is that the watchdog outputs a 3-bit wide signal as a timeout signal, due to the TMR design. This can be used to feed a final voter to convert a 3-bit wide signal to a 1-bit wide signal. At this point the voter represents a single point of failure, but as explained the probability of its failure is low. However, if its probability is not sufficiently low, it is possible to use a different voter implementation like the following one:

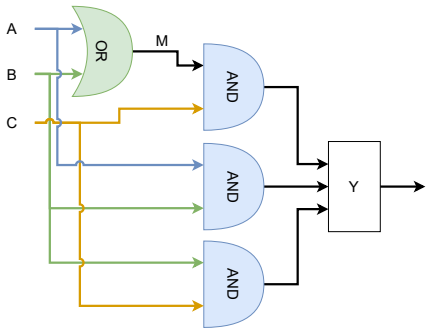


Figure 4.24: Enhanced majority voter design.

The above scheme [37] is made of a 1-bit OR gate, three 2-bit AND gates and a 3-bit OR gate. This design allows reaching a higher Fault Masking Ratio (FMR), specified as the ratio of the total number of correct voter output states in the presence of internal and/or external faults, which are masked, divided by the total number of potential internal and/or external fault occurrences. From the given definition, it may be understood that FMR has to be high (ideally 1) to achieve good (absolute) fault tolerance. A normal majority voter has an FMR of 42.86 % while the enhanced majority voter has an FMR of 75.00 %.

Secondly, when the DFX Controller ends the reconfiguration process, the signal `vsm_VS_0_rm_reset` is asserted for 1 clock cycle, as specified in the RM's configuration. This is conncted to the `Reset` signal of the MicroBlaze that is active high, feeding an OR gate together with the active-high `mb_reset` signal arriving from the *Processor System Reset* IP. In this way, both the DFX Controller and the PSR are able to reset the CPU when required.

To reset the watcdog, the reset signal needs to be negated because the Watchdog's reset signal is active low, as other AXI IPs are. Because the Watchdog can be reset both by the DFX Controller and the PSR, as for the MicroBlaze, a NOR gate is fed with the one coming from the DFX Controller and the one coming from the PSR in the active-high form, that is `peripheral_reset`.

Finally, the DFX Controller's ICAP port is connected to the ICAPE2's instance. The DFX Controller ICAP's `I` port is connected to the ICAPE2's `O` port, and the DFX Controller ICAP's `O` port is connected to the ICAPE2's `I` port.

4.5.5 DFX Decoupler: what is it?

There is one important aspect that was not mentioned in the previous sections. During the Partial Reconfiguration of a partition, unpredictable signal activity can happen between the Reconfigurable Partition and the remaining part of the system. To overcome this problem, Xilinx provides the Dynamic Function eXchange (DFX) Decoupler IP, for a complete logical isolation capability for DFX designs.

During the reconfiguration process, the DFX Controller asserts a decouple signal that remains high for the entire duration of the process. This signal is used to decouple the system from the logic under reconfiguration and avoid strange signal activities. For example, can happen that an AXI write request signal is asserted wrongly, leading to a modification of a random memory location with an unexpected value. By decoupling the AXI interface from the rest of the system, this can be avoided.

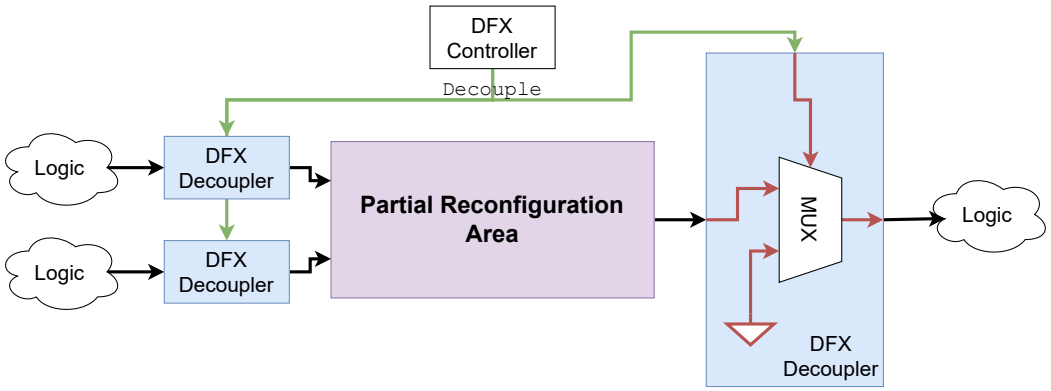


Figure 4.25: DFX Decoupler scheme.

In a complex system, like the case under study, there are a lot of signals that need to be decoupled. Luckily, the DFX Decoupler IP's Configuration Wizard offers the ability to create decoupler interfaces for each standard interconnection like AXI or LMB. For what concerns a simple MicroBlaze architecture, the following is the adopted solution:

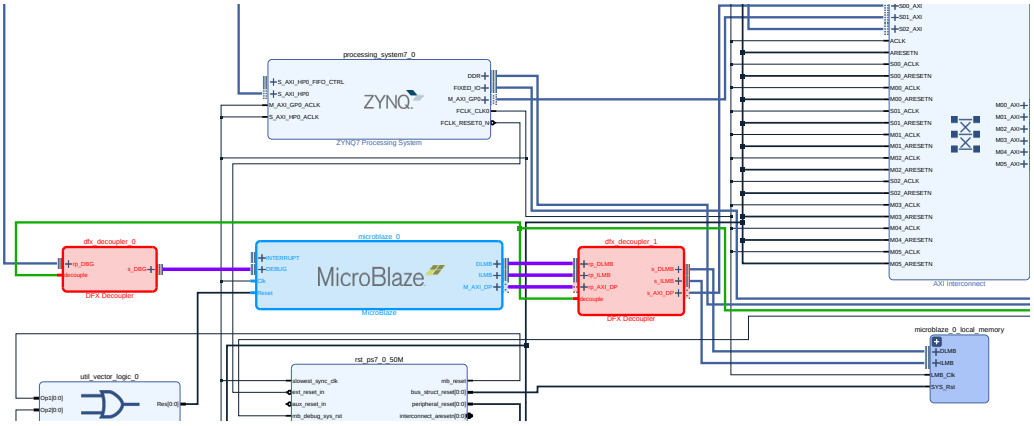


Figure 4.26: Decoupling of a Reconfigurable MicroBlaze region.

4.6 From a manual workflow to a fully automated one

The presented workflow allows enabling partial reconfiguration of a MicroBlaze, even on an old version of Vivado. The workflow is presented in Figure 4.20 and is made of 5 major steps, summarized as follows:

1. Starting from a working project based on Block Design, with a MicroBlaze core, a DFX Controller and a Watchdog, the HDL description from the BD is extracted.
2. A new project is created and the previous HDL description is copied into it. The MicroBlaze instance is replaced with its own wrapper and the wrapper instantiates the MicroBlaze. All the .xci files from the base project are copied into the new project.
3. The new project is converted into a DFX-based project and the MicroBlaze wrapper is added in a new reconfigurable partition as Reconfigurable Module (RM). Finally, a PBLOCK is defined for the MicroBlaze wrapper.
4. The new project is implemented and bitstreams are generated.
5. From the base project, the XSA is generated, files within it are extracted and the bitstream is substituted with the one generated in the previous step. Then, all the files are inserted in a new .xsa archive to generate the new .xsa.

Thus, the workflow is made of five simple steps that require a certain amount of time to be completed. The designer needs to execute all of them every time she/he wants to change something in the original project. Hence, the workflow is not suitable for a continuous integration process and some sort of automation is required.

4.6.1 The automatation script

Luckily, all the flow can be almost easily scripted. In the following, the developed script is presented by dividing it into different subparts and each subpart is described by a dedicated pseudo-code algorithm. The overall flow is divided into five main blocks:

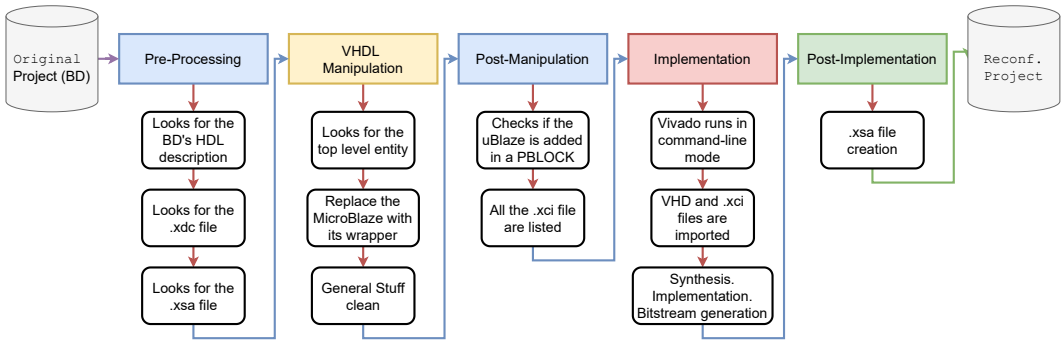


Figure 4.27: Scheme of the script flow.

Pre-processing block

The pre-processing block is the first block to be executed in the automatic workflow script. It takes as input the original project path and checks if the project is a valid candidate for the workflow:

1. Given the project path, the project name `$pn` is extracted.
2. The script checks for the existence of the `$pn.gen` folder. This is a required folder that contains the generated HDL description of the Block Design. The script recursively checks for a path matching `*/bd/*/synth/$dn.vhd`. If the path is found, means that the script knows the block design name `$dn` and found its VHDL description file.
3. The script searches any .xdc file that may contains the PBLOCK definition in the `$pn.srcs` folder. However, the designer can supply its own .xdc file. The PBLOCK is not checked at this time.
4. The script searches for the .xsa file. If not found, the script terminates asking the user to create it from within Vivado before running the script.

VHDL Manipulation block

If the Pre-processing block runs successfully, the HDL Manipulation block can be executed. This is the most compilcated one. It takes as input the previous VHDL description file `$dn.vhd` and a secondary VHDL file named `adding.vhd` that contains partial definitions of some hardware that the designer wants to add to the system outside the Block Design, like for example the ICAPE2.

1. The whole `$dn.vhd` file is parsed. Each entity is extracted and stored separately in a list of entities. Each entity is made of libraries, entity definition and architecture content. The same is done for the `adding.vhd` file.
2. Each entity is analyzed, looking for a MicroBlaze instance in its architecture. If found, the entity is marked as the top-level entity. The user can decide to indicate another entity as top level one, if the found entity is not the one he wants to use or if it is wrong. Each of the next steps is performed on the top-level entity.
3. The script looks for the MicroBlaze's component declaration.
4. The script looks for the MicroBlaze instance, given the previous component name.
5. The script creates a new entity, named `ublaze_wrapper` and:

   (a) Extracts the ports declaration from the MicroBlaze component and adds them to the new entity ports definition. The entity part of the whole wrapper is created.

   (b) The architecture is defined. First, the original MicroBlaze component is appended as it is, then it is instantiated and each port is connected to the ports of the entity wrapper under creation.

   (c) The newly created entity is saved in a new file `ublaze_wrapper.vhd`.

6. The original microblaze component is replaced by the new wrapper component. The instance is left as it is, only the instance name is changed, referencing the new wrapper component.

7. Some synthesizer attributes referenced to the older MicroBlaze instance are now referenced to the new wrapper instance.

8. If in the base project, the DLX Controller's ICAP interface is made external through a port, Vivado automatically creates some attributes and entity ports to let them connect outside. They are removed.

9. The definitions from the `adding.vhd` file are now merged. For each component in the architecture defined in the `adding.vhd` file, the script adds both the component and the relative instance to the top entity. It looks for the ICAPE2 instance too and adds it to the top entity (it is trated as special, becuase in this case there is no component declaration as other components).

10. All the entities are appended in a new file `top.vhd`.

Post-manipulation block

This is the third last block to be executed. At this point in time, if everything succeeded, the script knows the top-level name, the MicroBlaze instance name and any other useful information about the design. It proceeds with the following steps:

1. The script knows the MicroBlaze wrapper instance name and now looks for the PBLOCK where it is assigned in the previous found .xdc file. If this is not found, the script terminates with an error.

2. The script prepares a list of all the .xci files in the base project. It searches for possible .xci file related to the `adding.vhd` file in the `adding_xci` folder and appends them too.

Implementation block

Now the scripts created all the files and information to create a new project and implement it. Vivado is launched in command-line mode and a TCL script is sourced to perform the following steps:

1. The script creates a new project and VHDL is set as the active language.
2. Imports the `top.vhd`, `ublaze_wrapper.vhd` and the .xdc file into the new project.
3. Takes the .xci list previously prepared and adds each file to the new project. They are references to the original files, not a copy.
4. Enables the project as a DFX-based project. Creates a definition partition and a reconfigurable module. The wrapper is added to the partition. A single reconfigurable configuration is created.
5. The project is synthesized and implemented. The bitstreams are generated both in .bit and .bin formats.

Post-implementation block

This is the last block to be executed. It takes as input the generated full bitstream and the previously found .xsa file. The scripts unzip the .xsa file, substitute the old bitstream with the new one and re-zip the .xsa file.

4.6.2 Script for partial bitstream to C header generation

Because of the choice to store the partial bitstream in memory by using the same .elf file executed by the ARM core to configure the PS, a script is needed to generate the C header file that contains the partial bitstream.

An error to avoid is the usage of the bitstream in the .bit format instead of the .bin one to perform a partial reconfiguration via the ICAP interface. As explained in Section 3.3.2, the difference between the two is that the .bit format contains a header while the .bin format does not. Hence, the header can have a different length, depending for example on the design name chosen by the user, thus can create an offset inside the bitstream. As an example, it can shift the SYNC WORD by 8 bits. The ICAP interface is not able to evaluate the header nor is capable of evaluating a shifted bitstream. Hence, the ICAP remains in the *NOSYNC* state or it goes in the *CFGERR* state. The .bit file is only useful to Vivado and XSCT because of the header, otherwise, it is only a waste of memory and must be not used outside Vivado tools.

The script's job is to prepare a `u32` C array and fill it with the content of the partial .bin bitstream. The script allows the creation of an array with a specific size. It accomplishes this by adding some `NOP` instructions to the array at the end, only if the size of the bitstream is less than the required one. This can be useful if the DFX Controller is configured with a specific size for the partial bitstream to be loaded, and the designer discovers only at the end that the generated partial bitstream is smaller than the configured one. In this way, is possible to avoid changing the DFX Controller's settings and to perform the implementation again.

Moreover, the script allows writing the content in little-endian or big-endian format. This is useful because if the bitstream is saved in memory with the wrong format and the ICAP reads it, it does not recognize the SYNC WORD and hence does not performs the partial reconfiguration.

The following is an example of the script usage:

```
format=big
required_size=389928
align=0 # do not touch!
output=data.h

./to_header path/to/partial.bin $format $required_size $align >
   $output
```

Chapter 5

Experimental Analysis

The following is a description of the environment that has been set up for the fault injection tests, using chosen benchmark applications and the hardware design preparation behind the related choices. The second part of this chapter illustrates in detail the obtained experimental results, highlighting some interesting considerations.

5.1 Fault Injection Environment

Before proceeding with the fault injection campaign, which aims to demonstrate the effectiveness of the developed fault tolerance design, the Fault Injection Tool requires some extra hardware for a good understanding of the obtained results.

Watchdog Configuration

The watchdog IP, during the design stage, is configured through its customization wizard with a default timeout value of 2 seconds (two times the clock frequency) and it is started by default. This choice has been taken because under normal operational conditions, the MicroBlaze is capable of starting the watchdog and setting a correct timeout value.

Instead, with the chosen fault injection method, the MicroBlaze directly starts with a bit-flip in his configuration, hence it may be not able to start correctly the watchdog or set a valid timeout value. Therefore, the campaign results would not be valid.

Hardware to count the number of timeouts

In order to understand if the watchdog is capable of covering a good number of faults by expiring, the design has been equipped with a UP counter. The counter is reset once at each FGPA programming with a new full bitstream. The counter is incremented every time the watchdog expires by connecting the CLK port to the timeout signal, and its value is obtained via an AXI GPIO peripheral that is connected to the AXI Interconnect.

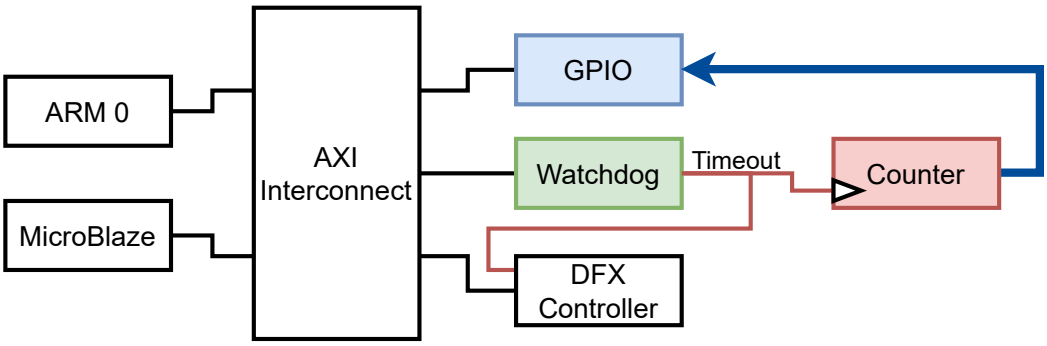


Figure 5.1: Hardware schematic to count the number of times the watchdog time outs.

Both the MicroBlaze and the PS can access it. Because the MicroBlaze is under test and may not be able to access correctly the GPIO peripheral, the value is read from one of the ARM cores in the PS side.

Benchmark Firmware

As explained in previous sections, a good benchmark software is required for a good fault injection campaign for two reasons:

- A good benchmark software can stress various aspects of the MicroBlaze's hardware (ALU, Decode Unit, Controller Unit, etc.) and reveal a hidden fault.
- A good benchmark software can self-test itself (checking the correctness of the produced results) and trigger the reconfiguration when needed.

The following is an extract of the firmware running on the MicroBlaze, that computes the Fibonacci series up to a certain point and checks the results via a simple checksum check:

```
int main() {
  int i;
  uint64_t op1 = 0, op2 = 0, res = 0, checksum = 0;
  GBcnCtrl hBcn;

  i = XPAR_BEACON_WATCHDOG_0_S00_AXI_BASEADDR;
  GBcnCtrl_Initialize(&hBcn, i);

  print("started? ", GBcnCtrl_IsStarted(&hBcn) ? 1 : -1);
  i = hBcn.modules->module0.DATAREG;
  print("timeout: ", i ? i : -1);
  GBcnCtrl_SetTimeoutValue(&hBcn, XPAR_CPU_CORE_CLOCK_FREQ_HZ<<1);
  GBcnCtrl_Start(&hBcn);
  print("started? ", GBcnCtrl_IsStarted(&hBcn) ? 1 : -1);
  i = hBcn.modules->module0.DATAREG;
  print("timeout: ", i ? i : -1);

  op1 = op2 = 1;
  print("Fibonacci current value ", op1);
  print("Fibonacci current value ", op2);

  while(1) {
    checksum ^= (res = op1 + op2);
    if(res > 0xfffff) {
      res = 1;
      op2 = 0;
      print("\n\rDONE_1 DONE_1 DONE_1\r\n");
      break;
    }

    print("Fibonacci current value ", (uint64_t)res);
    op1 = op2;
    op2 = res;
    for (i = 0; i < 1e5; i++); // a bit of delay
    GBcnCtrl_Toggle(&hBcn);
  }

  printt("CHECKSUM: ", checksum);
  if (checksum == 1673873) {
    print("DONE_2 DONE_2 DONE_2\r\n");
    for (i = 0; i < 5e6; i++); // delay
    GBcnCtrl_Toggle(&hBcn); // all fine!
  } else {
    print("WRONG CHECKSUM!\r\n");
    while(1); // stops toggling because wrong checksum
  }
}
```

How the fault injection tool access the number of expired times

The Fault Injection Tool has been enhanced to support the retrieval of the number of times the watchdog expires at each run. It does it via an XSCT script that appends in a file the retrieved number, as follows:

```
connect -url tcp:127.0.0.1:3121

# selects the ARM #0 core
targets -set -nocase -filter {name =~ "*A9*#0"}
set outfile1 [open "faulty_bitstreams/uB_results/dfx_cnt.txt" a+]

# reads the value from the GPIO register
puts $outfile1 [mrd -value 0x41200008 1]
close $outfile1
```

5.1.1 Watchdog Inhibition

In some cases may be useful to inhibit the watchdog. This allows to detach the timeout signal from the DFX Controller, thus the reconfiguration is not triggered. This is useful for example to debug the firmware. In this case, the firmware stops kicking the watchdog during the interruption of the software and thus the MicroBlaze is automatically reconfigured and restarted. To allows the inhibition by toggling a physical switch on the board, the following hardware scheme is adopted:

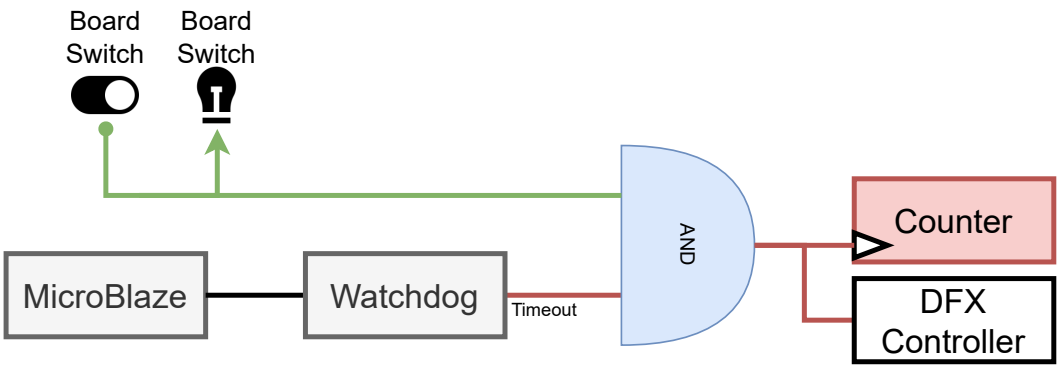


Figure 5.2: Hardware schematic to inhibit the watchdog using a physical switch.

5.2 Experimental Results

In this section, the main obtained results are presented and discussed. During the overall test period, ten campaigns of fault injection have been performed. Each campaign is made of 100 injections. Some of them are aborted, thus the mean number of completed injections per run is around 96, leading to a total of 966 injections.

For each fault, there can be three possible outcomes, and each one can have a different reason behind it:

- Correct result:
  1. The fault has not been excited by the software so it is naturally masked and the result is automatically correct.
  2. The fault caused a CPU halt (no detected output on the UART). The system corrected the fault and the final output is correct. It is identical to the golden one, hence it is marked as correct.
- The output is different from the golden one (SDE):
  1. The fault has not been detected by the watchdog, thus it is not fixed.
  2. The MicroBlaze noticed the fault while executing the program and printing messages and it has been fixed. However, the output is different from the golden one, even if the final result is correct. Marked as SDE anyway. An example is shown in Appendix C.
  3. The MicroBlaze noticed the fault and tried to fix it, unsuccessfully. The output is different and the result remained incorrect.
- The MicroBlaze is halted:
  1. The hang is detected by the watchdog, but it could not be fixed.
  2. The MicroBlaze does not output anything on the UART, even if the watchdog is kicked correctly. No output means that the CPU is in the hang state but a correct kicking act means that the CPU is working so the reconfiguration is not triggered.

The following one is a summary of the obtained results from the conducted campaigns:

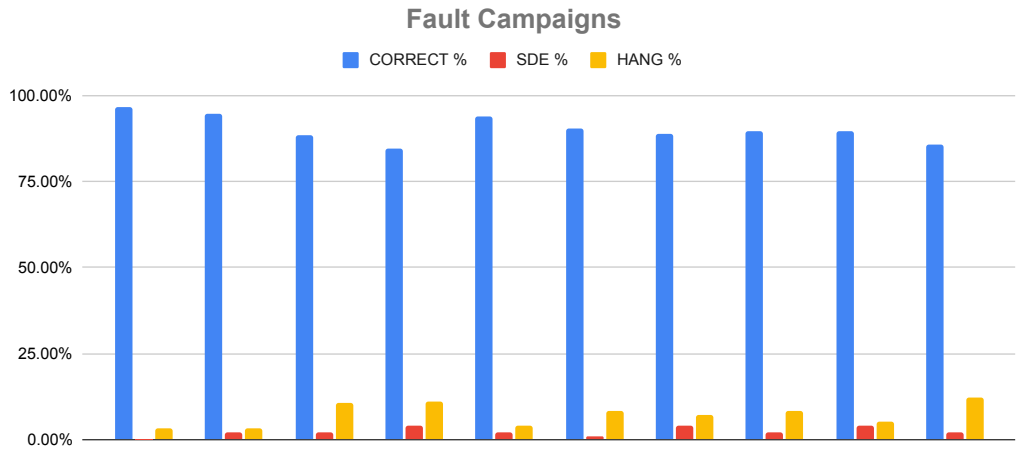


Figure 5.3: Chart representing the executed Fault Injection campaigns.

Each value is represented as the percentage of the total number of injections for each campaign. The blue ones represent the correct results. The overall correct results among all the campaigns are 872, 27 of which have been corrected by the fault tolerance system.

Among the red ones representing SDEs, the total is 23. Among these 23 SDEs, 11 have been successfully corrected, while the other 12 have not been corrected. Among those 12 not corrected SDEs, 1 has been detected but the system could not fix them, while the remaining 11 have not been detected at all. This can be easily fixed by increasing the quality of self-test routines in the firmware, looking for example for differences in the produced UART output and the expected one or by comparing written memory values with the expected ones. Consequently, this may help in increasing the coverage of those faults.

The remaining yellow cases represent situations where the MicroBlaze is completely halted, even 1fter one or multiple reconfigurations have been performed. There is a total of 71 hangs, 1 of which has not been detected at all. As the not detected SDEs, this can be fixed by increasing the quality of self-test routines. The remaining cases are the uncoverable ones.

Those values are presented in the following charts:

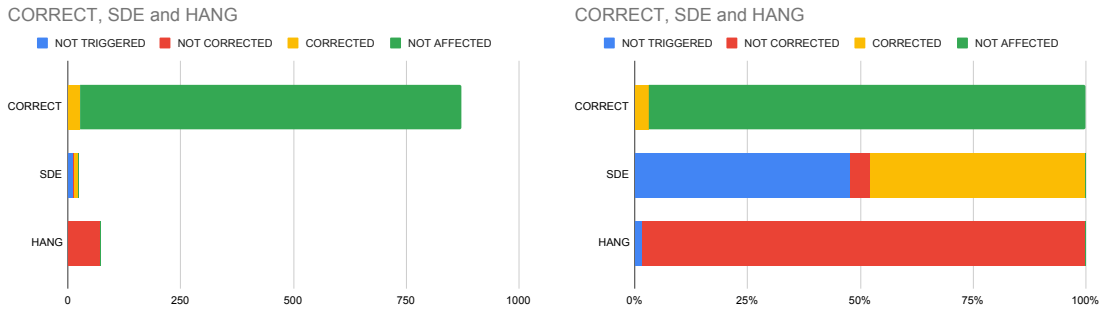


Figure 5.4: Charts showing the times the reconfiguration is triggered (or not) and how many times it solved the issue (or not).

As a side note, the unrecoverable cases are particular cases where it is not possible to say if the overall fault tolerance system is working or not. Unfortunately, the bitstreams cannot be fully controllable and the used fault injection tool randomly injects bit flips in an area that is thought to be 100% dedicated to the MicroBlaze. Unfortunately, in the reality it is not true: a bitflip could affect a configuration instruction instead of a configuration bit, leading to a misconfiguration anywhere else in the overall FPGA, thus it cannot be fixed by the partial reconfiguration.

By improving the quality of the self-test routines, the fault tolerance system may be able to detect the faults and fix them. As an example, the software has been enhanced to detect errors in memory access and division or modulus operations. Those two mathematical operators are used to convert an integer into its string representation. The two operators are mutually tested at each operation by first applying the division operator then it is again multiplied by the dividend and finally the computed modulus is applied. This is the definition of the remainder. The following is the implementation:

```
1  while (a != 0) {
2    /* module operation test */
3    pmod = a % 10;
4    if ((a/10)*10 + pmod != a || a - (a/10)*10 != pmod)
5      while(1); // HALT. Triggers the watchdog.
6
7    str[i--] = pmod + '0';
8    if (str[i + 1] - '0' != pmod) // Checks stored value
9      while(1); // HALT. Triggers the watchdog.
10
11   a /= 10;
12 }
```

In addition, the firmware checks that each character sent over the UART is the same as the one expected. This is done by comparing the expected character with the one received. If they are different, the MicroBlaze is halted, triggering the watchdog. To achieve this, the UART has been instantiated with a loopback connection from the *TX* channel to the *RX* channel, as follows:

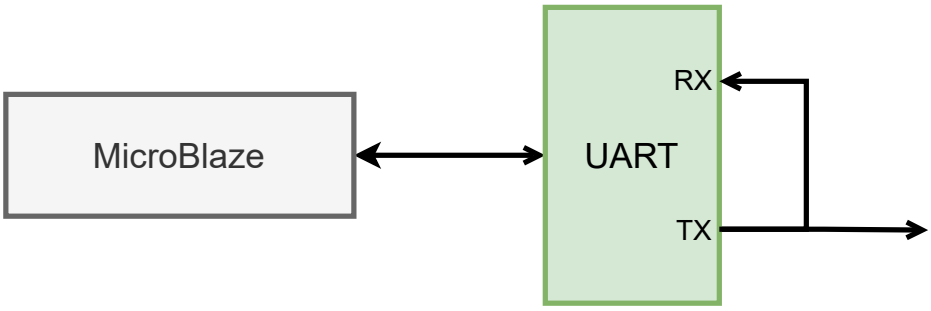


Figure 5.5: UART loopback schematic.

The firmware implements a custom *print* routine that checks the received character and if it is different from the expected one, it triggers the watchdog. Moreover, it accesses the string memory (to send it) both with byte-oriented access and word-oriented access. The latter is used to check that the string is correctly stored in the memory and that the MicroBlaze does not have any fault in the memory access unit.

```
void print(char *buf) {
  int i; char *pbuf = buf; char ch, tst; u32 *pt;

  for (i = 0; buf[i]; i++); // length computation
  while(*pbuf) {
    XUartLite_Send(phRef, (u8 *)pbuf, 1); // Send the character

    // Receive the character
    while(!XUartLite_Recv(phRef, (u8 *)&ch, sizeof ch));

    /* Tests single byte memory access vs 32 bit memory access */
    pt = (u32 *)((u32)pbuf & 0xfffffffc); /* 32b aligned addr */
    tst = ((*pt >> (((pbuf - (char *)pt)) << 3)) & 0xff);
    if (ch != *pbuf || ch != tst) while(1); // HALT.
    pbuf++; i--;
  }

  if (i || *pbuf) while(1); // just to be sure
}
```

Everything is now set and ready to be tested again. The previous 10 campaigns are executed again, with the same seeds to generate the faults. This means that the tool is able to generate again the same faults as before, thus the new firmware results can be compared with the previous ones:

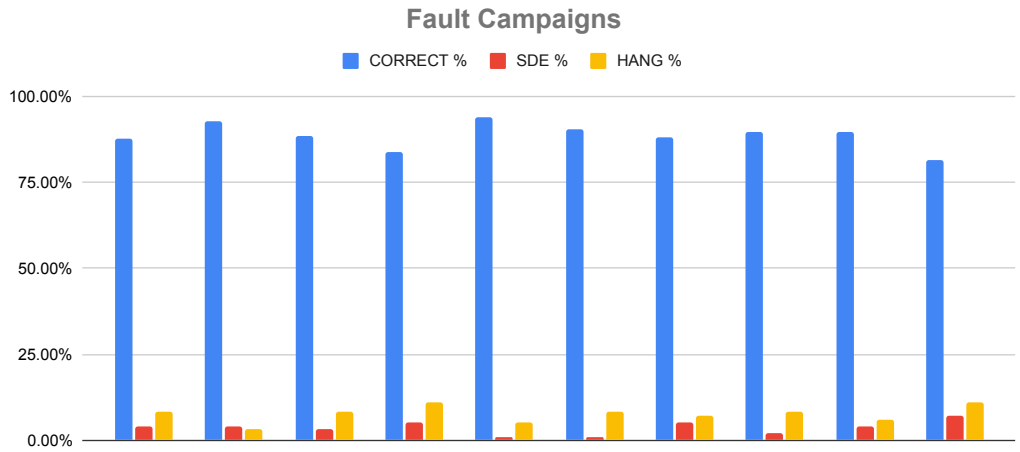


Figure 5.6: Chart representing the repeated Fault Injection campaigns with the new firmware.

The new firmware is able to detect almost all the faults, lowering in the number of not detected among the SDEs to 1 (2.9% among the SDEs) and to 0 among the HANGs, as shown in the following chart:

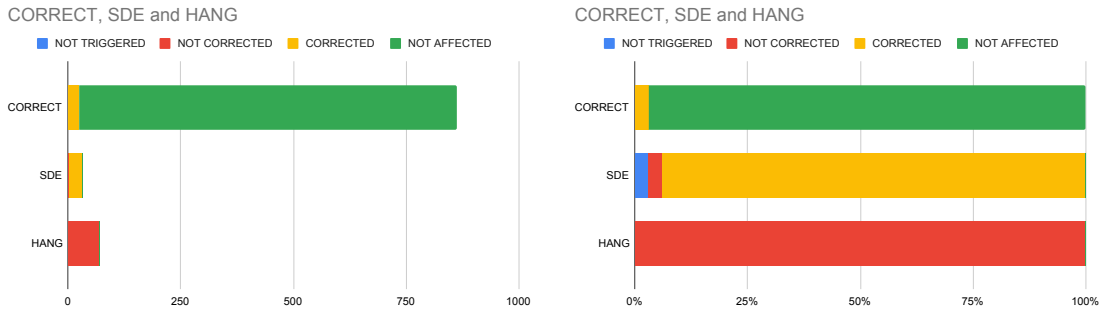


Figure 5.7: Charts showing the times the reconfiguration is triggered (or not) and how many times it solved the issue (or not). Second run with the new firmware.

Chapter 6

Conclusions

This thesis finally reaches its end, It sees the development of a complete fault tolerance system able to increase the dependability of the system itself and to protect in most of cases the MicroBlaze from possible SEUs that can cause faults and errors. It has been achieved by a combination of a fault-tolerant watchdog, completely designed from scratch to be tailored to the needs of the project, and the enabling of partial reconfiguration of a complex IP like the MicroBlaze, by exploiting a little hack that leads to a complete automation script able to convert a normal project designed with normal tools, like the Block Design Tool, in a project with a fully working reconfigurable MicroBlaze.

After the development, fault campaigns took place. Unfortunately, the bitstream manipulation is a lot tricky and fault injection campaigns of the chosen types are not well supported yet, and the tools are limited. Several corrections have been applied to make the output data more readable and useful for further analysis. Moreover, as the main objective, an in-deep evaluation of the fault injection campaign data has been performed, in order to better understand the effects of the injected faults on the system and the way they can be used to improve the dependability of the system itself.

The developed system has been engineered with the idea to be implemented in a more complex one, where almost everything is triplicated, at least for what concerns the most critical parts like a single point of failure in the reconfiguration system. This can be the DFX Controller, the ICAPE2, or even the timeout signal itself can be a SPoF because of a possible fault in the routing configuration.

The developed mitigation technique is unique in its genre, permitting to comprehend which types of faults can be mitigated and which not, without the need for a radiation test conducted in a dedicated facility, using real highly energetic particle beams. Regarding this type of test, another main application of the developed

system is to understand how the system can react to different faults prior to the radiation test itself, and thus 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.

6.1 Future Work

The thesis has been developed with the idea of being implemented in a more complex system, so it can be easily extended for future works. In particular, the watchdog can be developed further to monitor multiple kicking acts from different sources like multiple MicroBlaze instances or add other types of checks like bus activity detection. An example of this could be the monitoring of a read/write signal in a memory bus (like the AXI one). If the system is designed to periodically execute read or write operations from the memory, this can be monitored and if the processor or any other master that reads from the memory stops working, the watchdog can be let expire and trigger a reconfiguration.

Moreover, other modules can be marked as reconfigurable, by extending the presented workflow and the related scripts. And with an extended watchdog, it is possible to partially reconfigure only specific partitions of the system or the whole system.

Speaking of which, the watchdog is protected against single faults but not against fault accumulation. If a difference among the voters is detected, the watchdog itself can be reconfigured.

Appendix A

Watchdog FSM - VHDL Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity top_beacon_watchdog is
  generic ( DW: integer := 32 );
  port (
    CLK:     in std_logic;
    RST:     in std_logic;
    DATAIN:  in std_logic_vector(DW-1 downto 0);
    START:   in std_logic;
    STB:     in std_logic;
    TOGRATE:out std_logic_vector(DW-1 downto 0);
    WORKING:out std_logic;
    ERR:     out std_logic
   );
end top_beacon_watchdog;
```

Listing A.1: Watchdog FSM - VHDL Code (Entity)

```vhdl
architecture arch of top_beacon_watchdog is
  type fsm_state is (S_START, S_CHECK_0, S_CHECK_1, S_DOOMED);
  signal curr_state, next_state: fsm_state;
  signal curr_timeout: std_logic_vector(DW-1 downto 0);
  signal next_timeout:std_logic_vector(DW-1 downto 0);
  signal curr_cnt, next_cnt: std_logic_vector(DW-1 downto 0);
  signal curr_toggle_rate: std_logic_vector(DW-1 downto 0);
begin
```

Listing A.2: Watchdog FSM - VHDL Code (Architecture Header)

```
1   TOGRATE <= curr_toggle_rate;
2
3   process(curr_state, curr_timeout, curr_cnt, DATAIN, STB, START)
4   begin
5     next_state <= curr_state; next_timeout <= curr_timeout;
6     next_cnt <= std_logic_vector(unsigned(curr_cnt) + 1);
7     ERR <= '0'; WORKING <= '1';
8
9     case(curr_state) is
10      when S_START =>
11        next_timeout <= DATAIN; next_cnt <= (others => '0');
12        WORKING <= '0';
13        if START = '1' then
14          if STB = '0' then next_state <= S_CHECK_1;
15          elsif STB = '1' then next_state <= S_CHECK_0;
16          else next_state <= S_DOOMED;
17          end if;
18        end if;
19        when S_CHECK_0 =>
20          if unsigned(curr_cnt) < unsigned(curr_timeout) then
21            if STB = '0' then
22              next_cnt <= (others => '0');
23              next_timeout <= DATAIN; next_state <= S_CHECK_1;
24            end if;
25          else
26            next_cnt <= (others => '0'); next_timeout <= DATAIN;
27            next_state <= S_CHECK_1;
28            if STB /= '0' then next_state <= S_DOOMED; end if;
29          end if;
30        when S_CHECK_1 =>
31          if unsigned(curr_cnt) < unsigned(curr_timeout) then
32            if STB = '1' then
33              next_cnt <= (others => '0');
34              next_state <= S_CHECK_0; next_timeout <= DATAIN;
35            end if;
36          else
37            next_cnt <= (others => '0');
38            next_state <= S_CHECK_0; next_timeout <= DATAIN;
39            if STB /= '1' then next_state <= S_DOOMED; end if;
40          end if;
41        when S_DOOMED =>
42          next_cnt <= (others => '0'); ERR <= '1';
43        when others =>
44          WORKING <= '0'; next_state <= S_START;
45    end case;
46  end process;
```

Listing A.3: Watchdog FSM - VHDL Code (Combinational process)

```
1   process(clk)
2   begin
3
4     if rising_edge(clk) then
5       if (RST = '1') then
6         curr_state <= S_START;
7         curr_cnt <= (others => '0');
8         curr_timeout <= (others =>  '0');
9         curr_toggle_rate <= (others => '0');
10      else
11        curr_state <= next_state;
12        curr_timeout <= next_timeout;
13        curr_cnt <= next_cnt;
14
15        if unsigned(next_cnt) = 0 then
16          if unsigned(curr_cnt) > unsigned(curr_toggle_rate) then
17            curr_toggle_rate <= curr_cnt;
18          end if;
19        end if;
20
21      end if;
22    end if;
23
24  end process;
25
26end arch;
```

Listing A.4: Watchdog FSM - VHDL Code (Sequential process)

Appendix B

Watchdog - C drivers

Listing B.1: Watchdog - C drivers - Secondary data types definition

```
typedef union {
    u32 U32VALUE;
    struct f {
        u32     START : 01;
        u32       STB : 01;
        u32 _reserved : 30;
    } FIELDS;
} union_ctrlreg_t;

typedef union {
    u32 U32VALUE;
    struct {
        u32   STARTED : 01;
        u32     ERROR : 01;
        u32 _reserved : 30;
    } FIELDS;
} union_statreg_t;

typedef struct {
    union_ctrlreg_t CONTROLREG;
    union_statreg_t STATUSREG;
    u32             DATAREG;
    u32             TOGGLERATEREG;
} watchdog_module_t;
```

Listing B.2: Watchdog - C drivers - Main data type definition

```
typedef struct {
    union {
        u32 *baseAddress;
        watchdog_module_t *module;
        struct {
            watchdog_module_t module0;
            watchdog_module_t module1;
            watchdog_module_t module2;
        } *modules;
    };
} GBcnCtrl;
```

Listing B.3: Watchdog - C drivers - Driver function prototypes

```
int GBcnCtrl_Initialize(GBcnCtrl *InstancePtr, u32 BaseAddr);
void GBcnCtrl_SetTimeoutValue(GBcnCtrl *InstancePtr, u32 Timeout);
void GBcnCtrl_Start(GBcnCtrl *InstancePtr);
void GBcnCtrl_Toggle(GBcnCtrl *InstancePtr);
u32 GBcnCtrl_GetToggleRate(GBcnCtrl *InstancePtr);
int GBcnCtrl_IsExpired(GBcnCtrl *InstancePtr);
int GBcnCtrl_IsStarted(GBcnCtrl *InstancePtr);
```

Appendix C

Fault Injection - SDE output with correction

```
Hi
Is bcn started? 1
timeout: 200000000
Is bcn started? 1
timeout: 200000000
Successfully ran Hello World application
            Fibonacci current value 1
            Fibonacci current value 1
            Fibonacci current value 2
            Fibonacci current value 3
            Fibonacci current value 5
            Fibonacci current value 8
            Finbonacci current value 13
            Finbonacci current value 21
            Finbonacci current value 34
            Finbonacci current value 55
            ........................................
            Fibonacci current value 46368
            Fibonacci current value 75025
            Fibonacci current value 121393
            Fibonacci current value 196418
            Fibonacci current value 317811

DONE_1 DONE_1 DONE_1
CHECKSUM: 852044
WRONG CHECKSUM!
```

Listing C.1: SDE output - before correction

```
Hi
Is bcn started? 1
timeout: 200000000
Is bcn started? 1
timeout: 200000000
Successfully ran Hello World application
            Fibonacci current value 1
            Fibonacci current value 1
            Fibonacci current value 2
            Fibonacci current value 3
            Fibonacci current value 5
            Fibonacci current value 8
            Finbonacci current value 13
            Finbonacci current value 21
            Finbonacci current value 34
            Finbonacci current value 55
            ........................................
            Fibonacci current value 46368
            Fibonacci current value 75025
            Fibonacci current value 121393
            Fibonacci current value 196418
            Fibonacci current value 317811
DONE_1 DONE_1 DONE_1
CHECKSUM: 1673873
DONE_2 DONE_2 DONE_2
```

Listing C.2: SDE output - after correction

Bibliography

[1] Xilinx. *Touchdown! NASA's Perseverance Rover Lands on Mars with Xilinx FPGAs On Board*. 2021. URL: https://www.xilinx.com/about/blogs/xilinx-xclusive-blog/2021/rover-lands-on-mars-with-xilinx-fpgas-on-board.html.

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

[3] L. Sterpone and M. Violante. «A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs». In: *IEEE Transactions on Nuclear Science* 52.6 (2005), pp. 2217–2223. DOI: 10.1109/TNS.2005.860745 (cit. on p. 2).

[4] L. Sterpone and M. Violante. «Analysis of the robustness of the TMR architecture in SRAM-based FPGAs». In: *IEEE Transactions on Nuclear Science* 52.5 (2005), pp. 1545–1549. DOI: 10.1109/TNS.2005.856543 (cit. on p. 2).

[5] Luca Sterpone and Massimo Violante. «An Analysis of SEU Effects in Embedded Operating Systems for Real-Time Applications». In: *2007 IEEE International Symposium on Industrial Electronics*. 2007, pp. 3345–3349. DOI: 10.1109/ISIE.2007.4375152 (cit. on p. 3).

[6] L. Bozzoli, C. De Sio, B. Du, and L. Sterpone. «A Neutron Generator Testing Platform for the Radiation Analysis of SRAM-based FPGAs». In: *2021 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. 2021, pp. 1–5. DOI: 10.1109/I2MTC50364.2021.9459804 (cit. on p. 3).

[7] Boyang Du, Luca Sterpone, Sarah Azimi, David Merodio Codinachs, Véronique Ferlet-Cavrois, Cesar Boatella Polo, Rubén García Alía, Maria Kastriotou, and Páblo Fernandez-Martínez. «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 (cit. on p. 3).

[8] Engineering National Academies of Sciences and Medicine. *Testing at the Speed of Light: The State of U.S. Electronic Parts Space Radiation Testing Infrastructure*. Washington, DC: The National Academies Press, 2018. ISBN: 978-0-309-47079-7. DOI: 10.17226/24993. URL: https://nap.nationalacademies.org/catalog/24993/testing-at-the-speed-of-light-the-state-of-us.

[9] Jatan Mehta. *Space grade electronics: How NASA's Juno survives near Jupiter*. The Planetary Society. Apr. 17, 2018. URL: https://www.planetary.org/articles/0417-space-grade-electronics.

[10] Heidi Garcia Canizares. «(GeoRadar survey of a test site: verification of underground cartography. Investigation of the floor of the Salone dei Cinquecento in Palazzo Vecchio) Indagine GeoRadar di un sito di test: verifica della cartografia sotterranea. Indagine del pavimento del Salone dei Cinquecento a Palazzo Vecchio.» Corso di laurea triennale in Ingegneria Informatica. Firenze, Italy: Università di Firenze, 2019.

[11] 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.

[12] L. Sterpone, B. Du, and S. Azimi. «Radiation-induced single event transients modeling and testing on nanometric flash-based technologies». In: *Microelectronics Reliability* 55.9 (2015). Proceedings of the 26th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, pp. 2087–2091. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2015.07.035. URL: https://www.sciencedirect.com/science/article/pii/S0026271415301220.

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

[14] Luca Sterpone and Sarah Azimi. «Radiation-induced SET on Flash-based FPGAs: Analysis and Filtering Methods». In: *ARCS 2017; 30th International Conference on Architecture of Computing Systems*. 2017, pp. 1–6.

[15] S. Azimi, B. Du, and L. Sterpone. «Accurate analysis of SET effects on Flash-based FPGA System-on-a-Chip for satellite applications». In: *2016 16th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. 2016, pp. 1–4. DOI: 10.1109/RADECS.2016.8093203.

[16] ECSS Secretariat. *Techniques for radiation effects mitigation in ASICs and FPGAs handbook*. 2016. URL: https://ecss.nl/hbstms/ecss-q-hb-60-02a-techniques-for-radiation-effects-mitigation-in-asics-and-fpgas-handbook-1-september-2016-published/.

[17] Kenneth A. LaBel. *Radiation Effects on Electronics*. NASA. URL: https://nepp.nasa.gov/docuploads/392333B0-7A48-4A04-A3A72B0B1DD73343/Rad_Effects_101_WebEx.pdf.

[18] Nandivada Sridevi, K. Jamal, and Kiran Mannem. «Implementation of Error Correction Techniques in Memory Applications». In: *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*. 2021, pp. 586–595. DOI: 10.1109/ICCMC51019.2021.9418432 (cit. on p. 14).

[19] Matthew J. Gadlage, Paul H. Eaton, Joseph M. Benedetto, Marty Carts, Vivian Zhu, and Thomas L. Turflinger. «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 (cit. on p. 15).

[20] G.C. Cardarilli, F. Kaddour, A. Leandri, M. Ottavi, S. Pontarelli, and R. Velazco. «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.

[21] M. Violante, L. Sterpone, M. Ceschia, D. Bortolato, P. Bernardi, M.S. Reorda, and A. Paccagnella. «Simulation-based analysis of SEU effects in SRAM-based FPGAs». In: *IEEE Transactions on Nuclear Science* 51.6 (2004), pp. 3354–3359. DOI: 10.1109/TNS.2004.839516.

[22] Boyang Du and Luca Sterpone. «Online monitoring soft errors in reconfigurable FPGA during radiation test». In: *2017 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. 2017, pp. 1–5. DOI: 10.1109/I2MTC.2017.7969976.

[23] 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. 19).

[24] Oscar Ruano, Francisco Garcia-Herrero, Luis Alberto Aranda, Alfonso Sanchez -Macian, Laura Rodriguez, and Juan Antonio Maestro. «Fault Injection Emulation for Systems in FPGAs: Tools, Techniques and Methodology, a Tutorial». en. In: *Sensors (Basel)* 21.4 (Feb. 2021) (cit. on p. 27).

[25] Daniele Rizzieri. «Software-Based Radiation Effects Analysis on AP-SoC Embedded Processor». Corso di laurea magistrale in Mechatronic Engineering (Ingegneria Meccatronica). Torino, Italy: Politecnico di Torino, 2021 (cit. on p. 27).

[26] O. Ruano, J.A. Maestro, P. Reyes, and P. Reviriego. «A Simulation Platform for the Study of Soft Errors on Signal Processing Circuits through Software Fault Injection». In: *2007 IEEE International Symposium on Industrial Electronics*. 2007, pp. 3316–3321. DOI: 10.1109/ISIE.2007.4375147 (cit. on p. 28).

[27] V. Sieh, O. Tschache, and F. Balbach. «VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions». In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 1997, pp. 32–36. DOI: 10.1109/FTCS.1997.614074 (cit. on p. 28).

[28] Daniele Rizzieri. *FPGA Bitstream Fault Injector*. https://github.com/danirizziero/FPGA_bitstream_injector. 2021 (cit. on p. 28).

[29] Ludovica Bozzoli, Corrado De Sio, Luca Sterpone, and Cinzia Bernardeschi. «PyXEL: An Integrated Environment for the Analysis of Fault Effects in SRAM-Based FPGA Routing». In: *2018 International Symposium on Rapid System Prototyping (RSP)*. 2018, pp. 70–75. DOI: 10.1109/RSP.2018.8632000 (cit. on p. 28).

[30] Niccolo Battezzati, Luca Sterpone, and Massimo Violante. *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Jan. 2011, pp. 1–220. ISBN: 978-1-4419-7594-2. DOI: 10.1007/978-1-4419-7595-9.

[31] Melanie Berg, AS&D in support of NASA/GSFC. Michael Campola NASA/GSFC| *FPGA Mitigation Strategies for Critical Applications*. NASA. Sept. 21, 2018. URL: https://ntrs.nasa.gov/api/citations/20180006778/downloads/20180006778.pdf.

[32] Ghazanfar Asadi and Mehdi B. Tahoori. «Soft Error Rate Estimation and Mitigation for SRAM-Based FPGAs». In: *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*. FPGA '05. Monterey, California, USA: Association for Computing Machinery, 2005, pp. 149–160. ISBN: 1595930299. DOI: 10.1145/1046192.1046212. URL: https://doi.org/10.1145/1046192.1046212 (cit. on p. 32).

[33] Ken LaBel, Jonathan Pellish, Ray Ladbury: NASA Goddard Space Flight Center. Hak Kim Christina Siedlick: MEI Technologies in support of NASA Goddard Space Flight Cente. *Differentiating Scrub Rates between Space-Flight Applications and Accelerated Single Event Radiation Testing for SRAM based Field Programmable Gate Arrays.* NASA. Apr. 9, 2013. URL: https://nepp.nasa.gov/files/24438/Berg_SEE-MAPLD2013_Scrubbing.pdf (cit. on p. 38).

[34] Luca Bozzoli Ludovica; Sterpone. «Soft-Error Analysis of Self-reconfiguration Controllers for Safety Critical Dynamically Reconfigurable FPGAs». In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Fernando Rincón, Jesús Barba, Hayden K. H. So, Pedro Diniz, and Julián Caba. Cham: Springer International Publishing, 2020, pp. 84–96. ISBN: 978-3-030-44534-8 (cit. on p. 39).

[35] Luca Sterpone, Niccolo Battezzati, and Massimo Violante. «A New Placement Algorithm for the Optimization of Fault Tolerant Circuits on Reconfigurable Devices». In: WREFT '08. Ischia, Italy: Association for Computing Machinery, 2008, pp. 347–352. ISBN: 9781605580920. DOI: 10.1145/1366224.1366228. URL: https://doi.org/10.1145/1366224.1366228 (cit. on p. 39).

[36] Alonzo Church. «Edward F. Moore. Gedanken-experiments on sequential machines. Automata studies , edited by C. E. Shannon and J. McCarthy, Annals of Mathematics studies no. 34, litho-printed, Princeton University Press, Princeton1956, pp. 129–153.» In: *Journal of Symbolic Logic* 23 (1958), pp. 60–60 (cit. on p. 43).

[37] P Balasubramanian and K Prasad. «A Fault Tolerance Improved Majority Voter for TMR System Architectures». In: (2016). DOI: 10.48550/ARXIV.1605.03771. URL: https://arxiv.org/abs/1605.03771 (cit. on p. 69).