

POLITECNICO DI TORINO

Master's Degree in Computer Engineering



Master's Degree Thesis

Study and development of fault tolerant operating systems for aerospace applications

Supervisors

Prof. Luca STERPONE

Ing. Daniele RIZZIERI

Ing. Sarah AZIMI

Candidate

Salvatore Gabriele LA GRECA

July 2022

Summary

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

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

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

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

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

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

Acknowledgements

ACKNOWLEDGMENTS

*“HI”
Goofy, Google by Google*

Table of Contents

| | |
|--|------|
| List of Tables | 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's Microblaze soft-core | 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 |
| 4 Analysis and hardening of a FPGA Design with a MicroBlaze | 30 |
| 4.1 How SEUs affect the MicroBlaze? | 31 |

| | | |
|----------|---|-----------|
| 4.2 | Strategies and adopted solutions | 37 |
| 4.3 | Development of a watchdog | 39 |
| 4.3.1 | What is a watchdog? | 39 |
| 4.3.2 | How to implement a watchdog? | 39 |
| 4.3.3 | How to harden the watchdog? | 44 |
| 4.3.4 | Integration of the watchdog in the design | 49 |
| 4.4 | Design Partial Reconfiguration | 54 |
| 4.4.1 | Vivado Design Flow for Dynamic Function Exchange | 55 |
| 4.4.2 | DFX with MicroBlaze in Vivado 2021.1 | 58 |
| 4.4.3 | Xilinx DFX Controller | 60 |
| 4.5 | Integration of the watchdog and the DFX | 61 |
| 4.6 | A script to automatize the process | 64 |
| 4.6.1 | Script for partial bitstream to C header generation | 64 |
| 5 | Experimental Analysis | 65 |
| 5.1 | Fault Injection | 65 |
| 5.2 | Experimental Results | 65 |
| 6 | Conclusions | 66 |
| 6.1 | Future Work | 66 |
| A | Watchdog FSM - VHDL Code | 68 |
| B | Galileo | 72 |
| | Bibliography | 73 |

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 [13] | 31 |
| 4.2 | Fault injection result for the basic MicroBlaze design | 36 |
| 4.3 | Detailed explanation of the states of the FSM | 43 |
| 4.4 | Voter truth table. The red cells indicate the faulty output. | 45 |
| 4.5 | Output signal matrix for the no-tmr version. | 47 |
| 4.6 | Output signal matrix for the no-tmr version. | 58 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Simplified schematic of a FPGA cell | 5 |
| 2.2 | The intrinsic 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 Wiki- media 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 | [8]Overview of a Microblaze SoftCore | 19 |
| 3.4 | Example of Block Design | 20 |
| 3.5 | Basic scheme of a fault injection tool [9] | 27 |
| 4.1 | Schematic of a basilar MicroBlaze design | 32 |
| 4.2 | Resulting hierarchy of the MicroBlaze design | 33 |
| 4.3 | Resulting floorplan of the MicroBlaze design, with the PBLOCK on the top side. Microblaze cells are highlighted in red | 35 |
| 4.4 | High level scheme of the fault tolerant design. | 38 |
| 4.5 | Timing diagram of a very basic watchdog. | 40 |
| 4.6 | Timing diagram of a more sophisticated watchdog. | 40 |
| 4.7 | Possible digital circuit implementation of a watchdog. | 41 |
| 4.8 | Timing diagram of a more sophisticated watchdog. | 43 |
| 4.9 | Triple Modular Redundancy (TMR) scheme. | 44 |
| 4.10 | 1-bit voter circuit scheme. | 45 |
| 4.11 | Basic TMR scheme vs. full TMR circuit scheme. | 46 |
| 4.12 | Input signal matrix for the no-tmr version. | 47 |
| 4.13 | Interfaces of the TMR Watchdog. | 48 |
| 4.14 | Instantiation of the Watchdog IP. | 51 |
| 4.15 | Watchdog IP configuration wizard. | 52 |
| 4.16 | Example of partial reconfiguration modules. | 54 |

| | | |
|------|---|----|
| 4.17 | Example of modules grouped under a partial reconfiguration area. . | 55 |
| 4.18 | Partition definition with two reconfigurable modules, each one with its own wrapper and .xci file. | 57 |
| 4.19 | Flow used to generate a fully working Reconfigurable MicroBlaze design. | 60 |
| 4.20 | Basic scheme of the DFX Controller flow. | 60 |
| 4.21 | ICAP Interface. | 64 |

Acronyms

SEU

Single Event Upset

COTS

Commercial Off-The-Shelf

FPA

Field Programmable Gate Array

ASIC

Application Specific Integrated Circuit

CLB

Configurable Logic Block

LAB

Logic Array Block

LUT

Look-up Table

HDL

Hardware Description Language

CPU

Central Processing Unit

DSP

Digital Signal Processing

CMOS

Complementary Metal-Oxide Semiconductor

TMR

Triple Module Redundancy

SEE

Single Event Effect

SOC

System On Chip

Chapter 1

Introduction

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

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

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

To understand better the problem, we can start from a real-world example, a piece of history. On September 22, 2021, the ESA's INTEGRAL spacecraft autonomously entered into emergency safe mode [1]. INTEGRAL is a space telescope for observing gamma rays, and it was launched into Earth 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: due to the fact that the spacecraft was spinning, data from the spacecraft were only reaching ground control in a difficult way, and the batteries were quickly discharging because of the missing orientation of the solar panels towards the Sun. ESA was going to lose a 19-years old space telescope.

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

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

- *Reliability*: the probability of a system to work as expected, continuously, in a given period of time (usually it coincides with the period of time of the mission itself).
- *Availability*: the probability of a system to work as expected at a generic moment in time, in the future.
- *Safety*: the ability of a system to work in a given environment, without any risk of serious damage.

With the increasing need for protection against unwanted effects caused by radiations, since the first interplanetary mission in the 60s with the Mariner 2 mission, there have been an increasing number of studies and techniques developed to deal with the problem. At the hardware level, there are *hardware mitigation techniques*, where usage of radiation-tolerant hardware components are used and hardware created with those components is called *radiation-hard* or *rad-hard* for simplicity. In most of the cases, *COTS* (Commercial Off-The-Shelf) hardware [2] is used, which is a hardware meant to be used in a generic environment, and on top of that logical mitigation techniques [3] are used to protect the system from the effects of radiation. The latter solution is easier to implement, and it is more efficient than the former one.

1.1 Thesis Motivation

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

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

For this thesis, the usage of FPGAs has one big advantage, among other things: randomly generated SEU faults can be injected easily without using any sophisticated [4] 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 needs to undergo a lot of tests [5], 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 in the implemented solutions, it is better to introduce a few background concepts. In particular, concepts about how FPGAs works, what kind of radiations exists and how FPGAs are affected by them.

2.1 Hardware Technology

2.1.1 FPGA Architecture

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

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

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

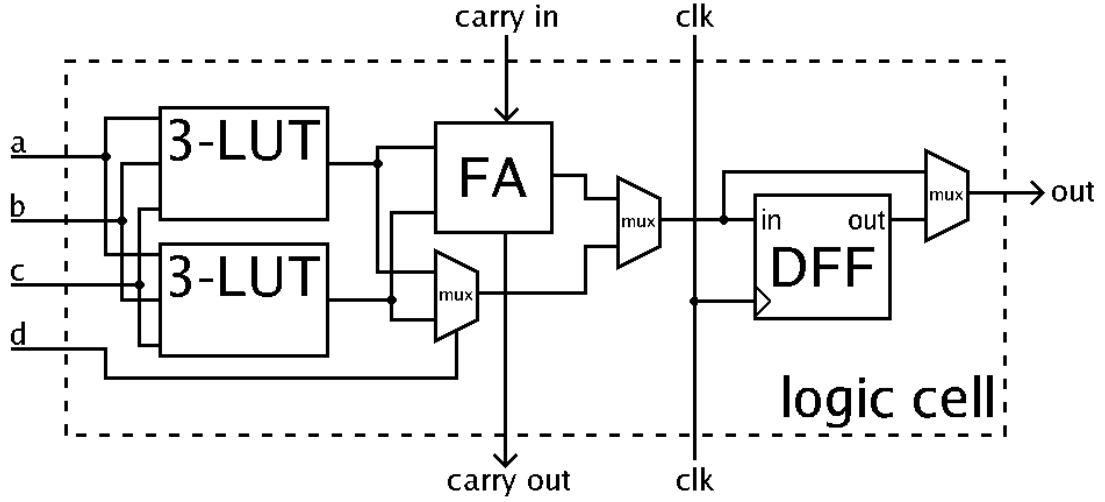


Figure 2.1: Simplified schematic of a FPGA cell

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

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

Once the User's Application is designed (i.e. the description of the FPGA is written), the design needs to be mapped onto the FPGA's hardware resources. This is done using the Vendor's specific software and it is in charge of deciding which FPGA's LE is assigned to which subpart of the description and how each LE is configured. Then, all the LEs need to be connected between 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 one 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 to be in-system programmable (so the FPGA chip can be programmed without unmounting it from the board and from the system itself) and re-programmable (can be programmed as many times we want), but require external boot devices. Because the SRAM is a volatile memory, when the FPGA is powered off, the configuration memory content is lost. An external memory where the bitstream can be retrieved is required in order to re-program it. The SRAM approach is based on CMOS.

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

2.1.2 FPGAs vs. ASICs

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

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

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

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

To recap:

- ASICs circuits are faster, less power-hungry than FPGAs.
- ASICs are more complex to design and implement (hence more expensive) than FPGAs.
- FPGAs are more 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 the hardware technology. FPGAs are typically much less radiation hardened than ASICs, so they are more prone to SEUs as well as lower total ionizing dose tolerance, but there are techniques to reduce these deficiencies. However, FPGAs are used on a lot more missions nowadays than 15 years ago, for all the reasons that make FPGAs a better choice than ASICs.

As an example, Mars Exploration Rovers were something like 90% ASICs. The last JPL's Martian Rover, 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 be a great help in this aspect and Perseverance rover is an example. It utilizes an almost decade-old FPGA technology (Xilinx Virtex-5, introduced in May 2006 on 65 nm technology) as one of the main processing units. This unit is responsible for rover entry, descent and landing on Mars. Once the rover is landed, this unit would be useless and would become a *dead hardware*. However, it is based on a FPGA hardware so it has been reprogrammed by NASA engineers from Earth to handle computer vision tasks.

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

Another advantage of using FPGAs is the faster time-to-space. There are different points that help in achieving this advantage. Not only the development on FPGA is faster than on ASICs (cost of design, development and fabrication of an ASIC are not present), but the most important thing is that there are many and many changes in the processing unit is architecture during project's development phase. There is usually a very strict 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 change of the bitstream or of the software running on a *soft-core* take a lot less time to certify.
- Software and Hardware development can be done in parallel. This is a great advantage for the software development team, because a first iteration of the hardware can be prepared and ready to use by the software team faster and the software team can start to work on the software itself.

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

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

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

2.2 Radiations

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

2.2.1 Radiation sources

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

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

2.2.2 Radiation problems on Earth: the Super Mario 64 glitch

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

Even if there is a big magnetosphere protecting the planet's surface, some charged particles still escape and travel until they reach the ground and some everyday device. In 2013, a player was challenging another player in Nintendo's Super Mario 64 game. Suddenly, Mario was teleported into the air, saving crucial time and providing an incredible advantage in the game. The glitch caused the attention of a lot of players, and a \$1000 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 own list, which is impossible in the voting system. The official explanation was "the spontaneous creation of a bit at the position 13 in the memory of the computer". it is not a coincidence that the value added was exactly 4096, in hexadecimal $0x1000$, that is 2^{12} .

2.2.3 Types of radiation

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

Total ionizing dose

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

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

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

Displacement ionizing dose

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

2.2.4 Single Event Effects

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

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

Single Event Latchup

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

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

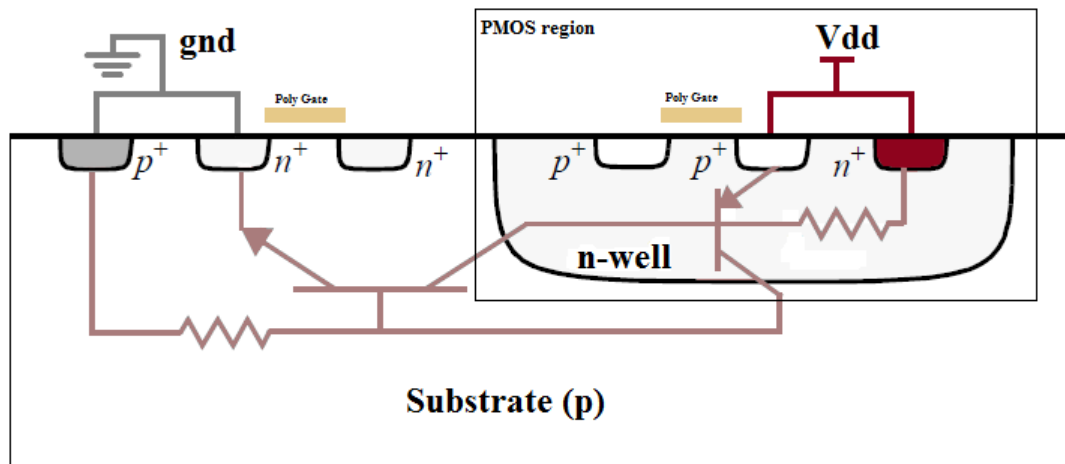


Figure 2.2: The intrinsic BJTs in the CMOS Technology that can cause a Latchup. Deepoon, CC BY-SA 3.0, via Wikimedia Commons

Single Event Burnout

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

Single Event Gate Rupture

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

Single Event Upset

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

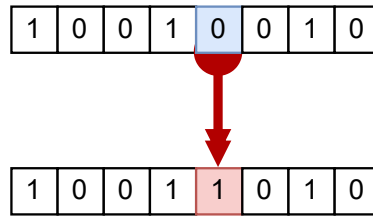


Figure 2.3: Example of a Single Event Upset in a memory element.

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

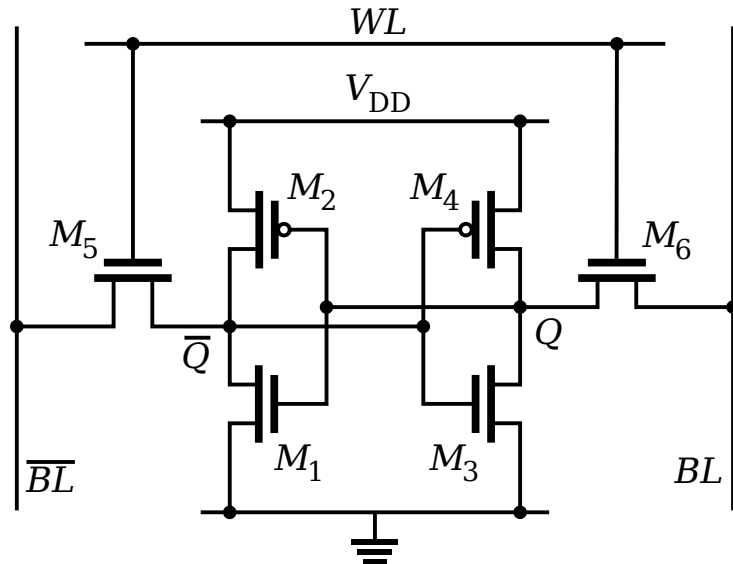


Figure 2.4: Simple SRAM Cell layout. Inductiveload, Public domain, via Wikimedia Commons.

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

As explained in Section 2.1.1, most FPGAs' memory configuration are based on SRAM technology. A fault that occurs in a configuration SRAM of a 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 single error and detect one or double errors, or SEC-DED-DAEC [7] (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.

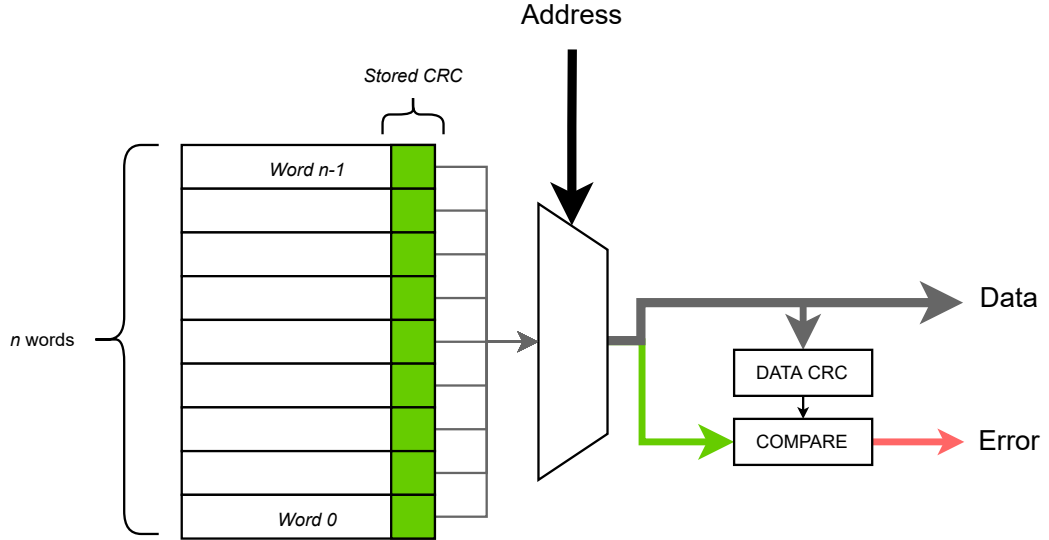


Figure 2.5: Example of error-detection circuitry in SRAM.

In a FPGA, the configuration SRAM is not utilized the same way as traditional SRAM. Indeed, a direct (logic) connections from the configuration to the user logic exists. If a 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 to a routing modification.

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

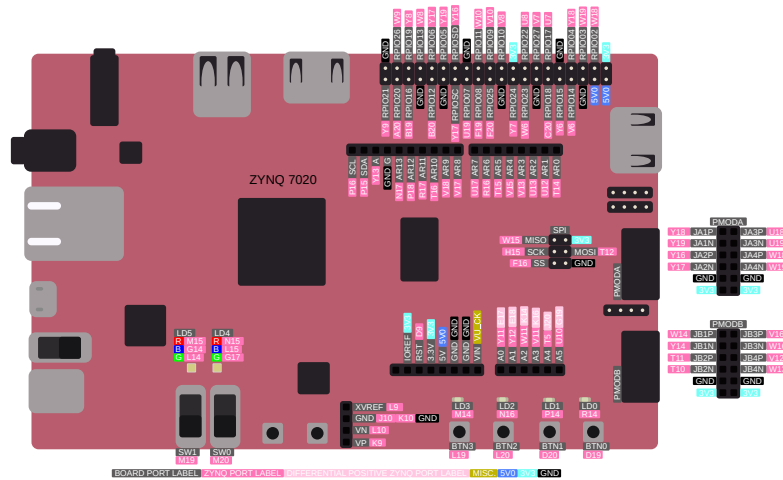
Chapter 3

Thesis Background

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

3.1 PYNQ-Z2 Development Board

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



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

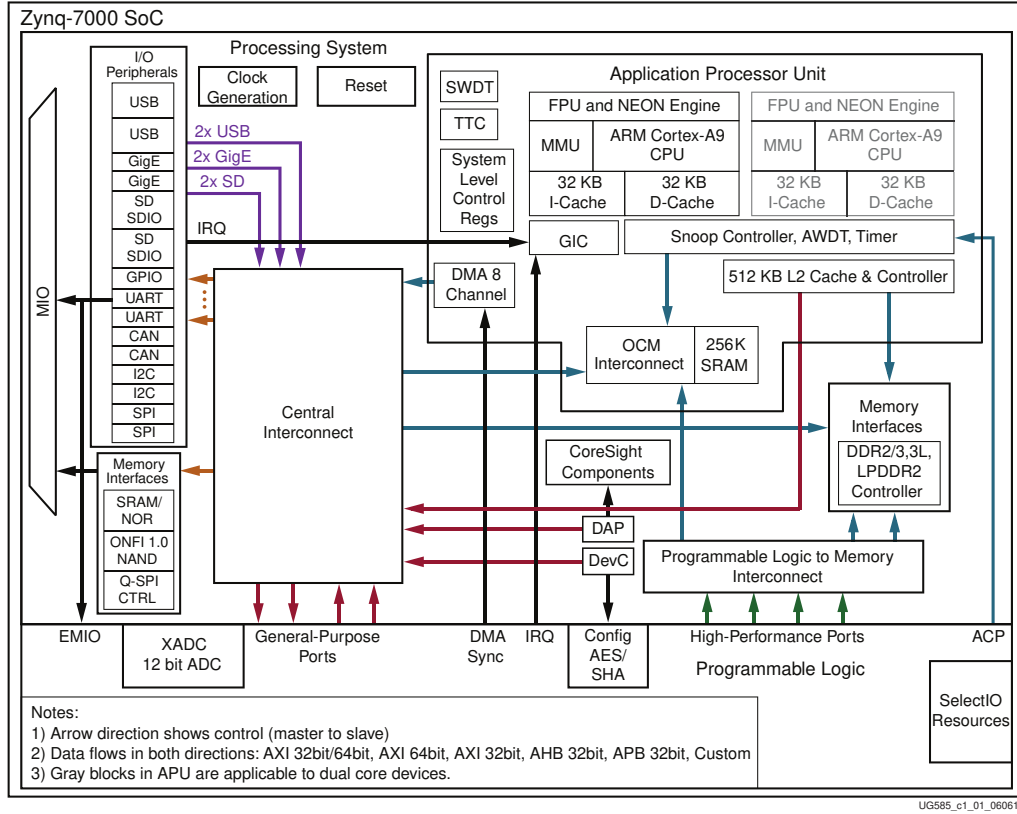


Figure 3.2: Schematic of ZYNQ 7020 SoC

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

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

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

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

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

Table 3.1: ZYNQ 7020 SoC Memory Map

3.2 Xilinx's Microblaze soft-core

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

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

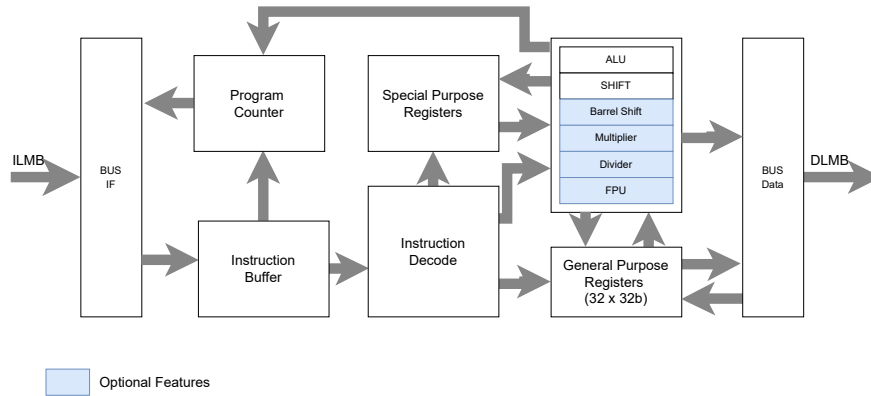


Figure 3.3: [8]Overview of a Microblaze SoftCore

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

3.3 Xilinx FPGA Standard Design Flow

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

Together with the HDL description, Vivado offers an IP catalog. IP stands for Intellectual Property, and each IP is an already developed and tested design ready to be integrated into the user's own design. An example of IP offered by default is the Microblaze IP, that 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 into 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 by means of 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 means of scripts. The interesting thing is that each action performed by the user in the GUI corresponds to an exact TCL command that can be seen from the TCL Console available in the IDE. This 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 (XDC) and a set of IP instantiations.

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

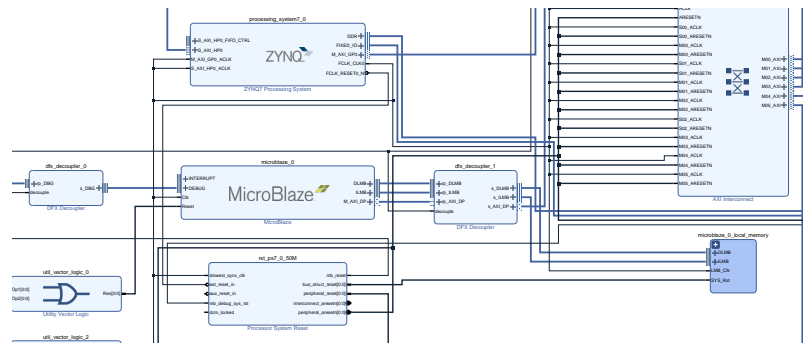


Figure 3.4: Example of Block Design

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

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

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

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

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

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

minimize the circuit delay (i.e. maximize the speed). The placement tries to follow the constraints defined in the XDC file. If it is not possible to 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 to each other 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 a FPGA, the .bit file is the default format generated by Vivado.

Bitstream Header

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

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

| | | | | | | |
|----|-----------|----------|----------|----------|----------|------------------|
| 1 | 00000000: | 00090ff0 | 0ff00ff0 | 0ff00000 | 0161002a |a.* |
| 2 | 00000010: | 64657369 | 676e5f31 | 3b557365 | 7249443d | design_1;UserID= |
| 3 | 00000020: | 30584646 | 46464646 | 46463b56 | 65727369 | 0xFFFFFFFF;Versi |
| 4 | 00000030: | 6f6e3d32 | 3032312e | 31006200 | 0c377a30 | on=2021.1.b..7z0 |
| 5 | 00000040: | 3230636c | 67343030 | 0063000b | 32303232 | 20clg400.c..2022 |
| 6 | 00000050: | 2f30362f | 31360064 | 00093133 | 3a34363a | /06/16.d..13:46: |
| 7 | 00000060: | 30340065 | 003dbafc | ffffffff | ffffffff | 04.e.=..... |
| 8 | 00000070: | ffffffff | ffffffff | ffffffff | ffffffff | |
| 9 | 00000080: | ffffffff | ffffffff | 000000bb | 11220044 |".D |
| 10 | 00000090: | ffffffff | ffffffff | aa995566 | 20000000 |Uf ... |

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

Raw Bitstream

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

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

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

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

- Bus Width Auto Detection
- Sync Word
- FPGA Configuration

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

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

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

| | | | | | | | |
|------|----|---------|----|----|----|----------------|---|
| 31 | 29 | 28 | 27 | 26 | 13 | 10 | 0 |
| Type | OP | Address | | | | Payload Length | |

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

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

| OP | Description |
|----|-------------|
| 00 | NOP |
| 01 | Read |
| 10 | Write |

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

For NOP operations, that 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 does not contain any address field, to extend the payload length maximum value. Thus, the configuration logic will use the address field of the previous type 1 packet to determine the address of the type 2 packet. The flow would be a NOP packet with a valid address field followed by a type 2 packet.

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

| Register | Address | Length | Description |
|----------|---------|----------|--|
| CRC | 00000 | Fixed | Automatical updated register: when a packet is received, the configuration logic computer the CRC incrementally and updates the register. |
| FAR | 00001 | Fixed | Start address for the next read or write operation for the configuration memory |
| FDRI | 00010 | Variable | Register where configuration data are wrote. This is the real content of the configuration memory of the FPGA, the one indicating how the physical cells are used and the interconnections |
| CMD | 00100 | Fixed | Used to perform one-shot actions. For example the <i>RCRC</i> resets the CRC register or the <i>START</i> 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 for 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 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 informations 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 is able to 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), or FreeRTOS based or petalinux based (if available).

Once a project is created, software developers can start writing their own code (usually in C) and to customize the Board-Support Package (BSP) that allows to change 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:

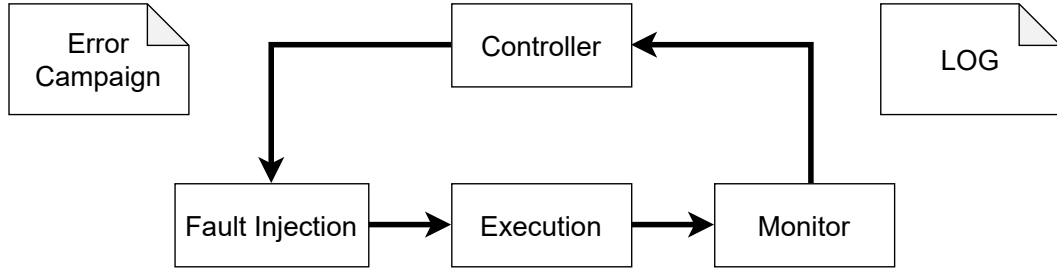


Figure 3.5: Basic scheme of a fault injection tool [9]

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 where faults are injected in the system, according to a specific error campaign.
- *Execution*: the output of the fault injection is executed.
- *Monitor*: the behaviour 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 in 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 perturbing 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 of the without contact there is an external source that produces physical phenomenon such as a heavy ion radiation that produces a faults in the integrated circuits.
- *Software-Based fault injection*: consists in 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 [10] are inserted into the source code of the software or at the assembly level after the compilation of the original

source code. 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 behaviour of the software.

- *Simulation-Based fault injection*: simulation-based fault injection is a technique that allows to simulate the system and to inject faults in it. The faults are injected in 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. An example of simulation-based fault injection is SST [11].
- *Emulation-Based fault injection*: emulation-based fault injection is a technique consisting in real implementation in a 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 behaviour of the system under test.

The Fault Injection tool used is a kind of Emulation-Based fault injection. 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, like 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 [12] it is possible to obtain a visual low-level representation of the bitstream. This allows to understand how and where 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:

```

1 open_run impl_1
2 set_property BITSTREAM.GENERAL.CRC DISABLE [get_designs impl_1]
3 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 implements, 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 is the chance to detect an injected fault, because of the higher probability to stimulate that fault.

All the remaining Execution's 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:

```

1 Total injected bitflips = 17
2
3     --- FUNCTIONAL ANALYSIS ---
4 Correct results -> 14 [82.35%]
5 Faulty results(SDE) -> 0 [0.0%]
6 MicroBlaze halted -> 3 [17.65%]
7 Total exceptions -> 0 [0.0%]

```

Chapter 4

Analysis and hardening of a FPGA Design with a MicroBlaze

A 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 a 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 one 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 second one are persistent because they directly affect how the bottom hardware 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 complete disconnect internal wires.
- They can change the behaviour 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 [13]

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 allows 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 behaviour when affected by SEUs and how those effects can be mitigated by constructing a series of ad-hoc hardening technique.

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

- *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) stands for 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 to 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).

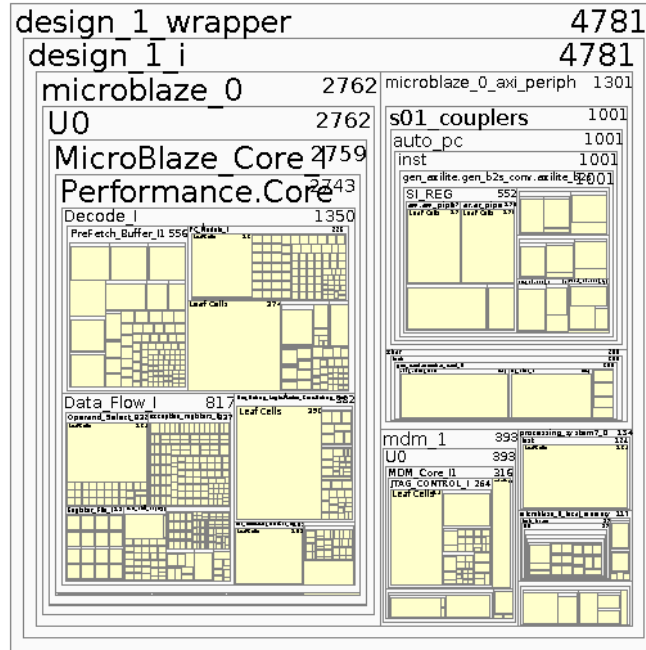
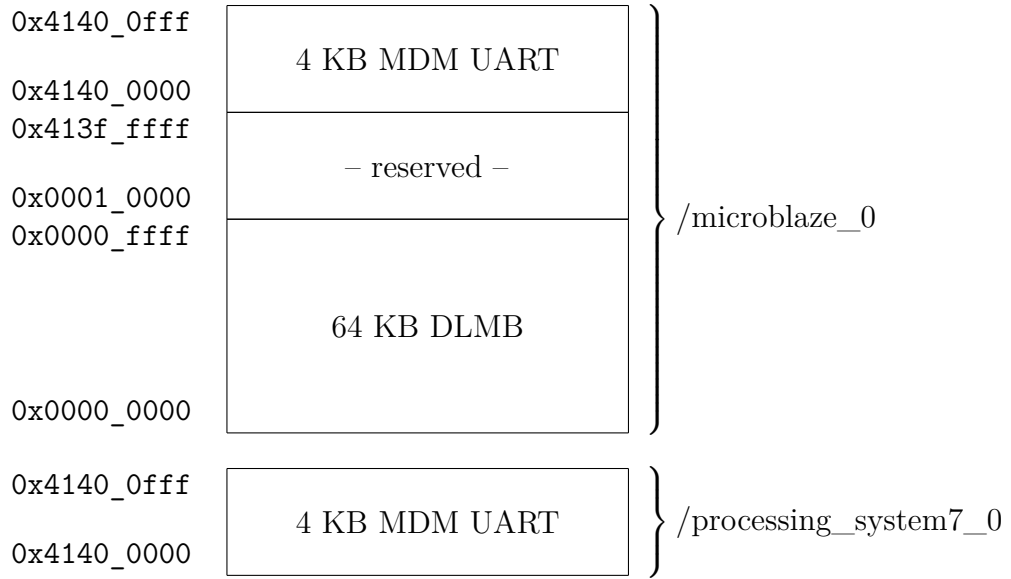


Figure 4.2: Resulting hierarchy of the MicroBlaze design

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 specify 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:

```

1 create_pblock pblock_1
2 add_cells_to_pblock [get_pblocks pblock_1] [
3     get_cells -quiet [
4         list design_1_i/microblaze_0
5     ]
6 ]
7
8 resize_pblock [get_pblocks pblock_1] -add {
9     SLICE_X54Y102:SLICE_X67Y148
10 }
11
12 set_property IS_SOFT 0 [get_pblocks pblock_1]
```


In the above constraints, a PBLOCK called *pblock_1* is first defined. Than 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 final operation, the PBLOCK is marked as a *not soft* PBLOCK, which means that each MicroBlaze cell must be placed 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:

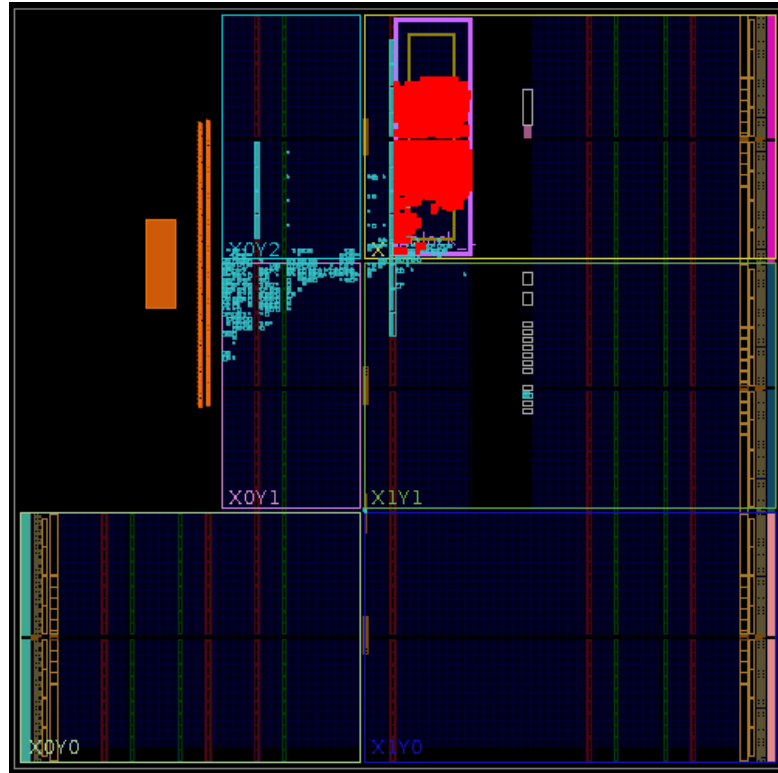


Figure 4.3: Resulting floorplan of the MicroBlaze design, with the PBLOCK on the top side. Microblaze cells are highlighted in red

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 | | |
|-------------------------|--------------|-------------------|
| | <i>Total</i> | <i>Percentage</i> |
| 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 | | |
|---|--------------|-------------------|
| | <i>Total</i> | <i>Percentage</i> |
| 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 all the algorithm 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 measure 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 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 needs 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 is able to 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 [14]. 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 is able itself to detect and mask the error, 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* [15] 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.

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:

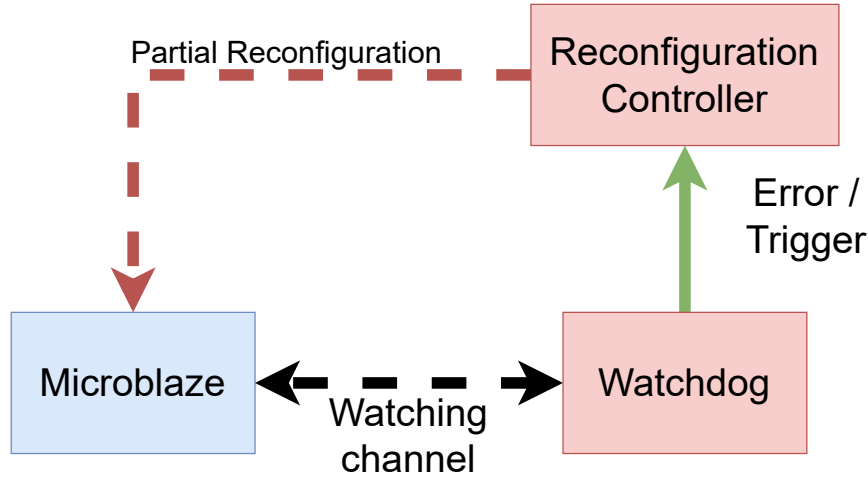


Figure 4.4: High level scheme of the fault tolerant design.

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 be able to detect the error and trigger the reconfiguration.

4.3.1 What is a watchdog?

In computer systems, a watchdog is basically 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 automatic correction of temporary hardware faults. Can be thought 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 which do not interfere with the kicking operation, but still requires 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 are able to 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 basically a timer. This means that

it must have some form of timing, so it needs 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:

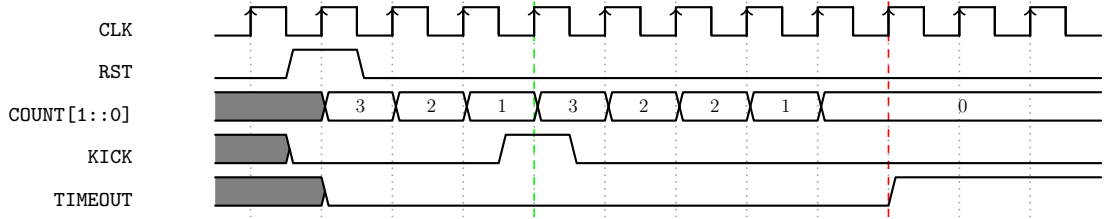


Figure 4.5: Timing diagram of a very basic watchdog.

In figure 4.5, the system initially is in an unknown state. Once the reset signal *RST* arrives, synchronously the watchdog is reseted 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 more sophisticated implementation:

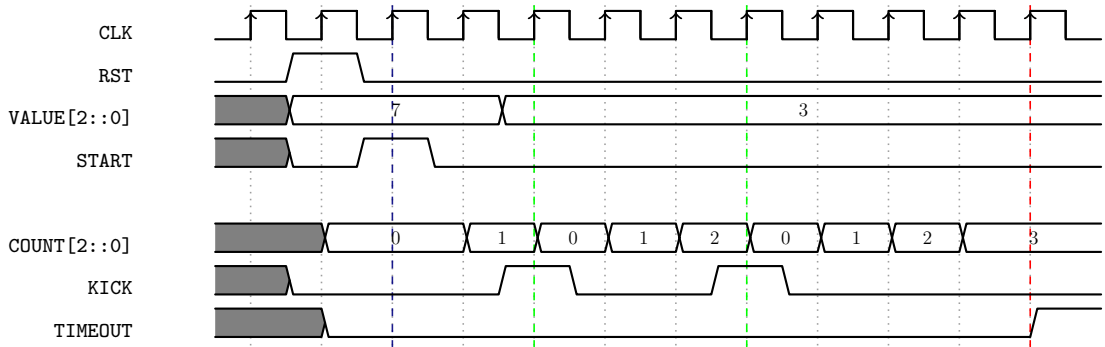


Figure 4.6: Timing diagram of a more sophisticated watchdog.

In the figure above, we have a different working mechanism compared to the previous one. After the reset signal *RST*, the timer is reseted to its initial value 0 and stays in this state until the *START* signal is received. At this point in time,

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 cycles sees a changing 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 cycles the *TIMEOUT* signal is generated (red line).

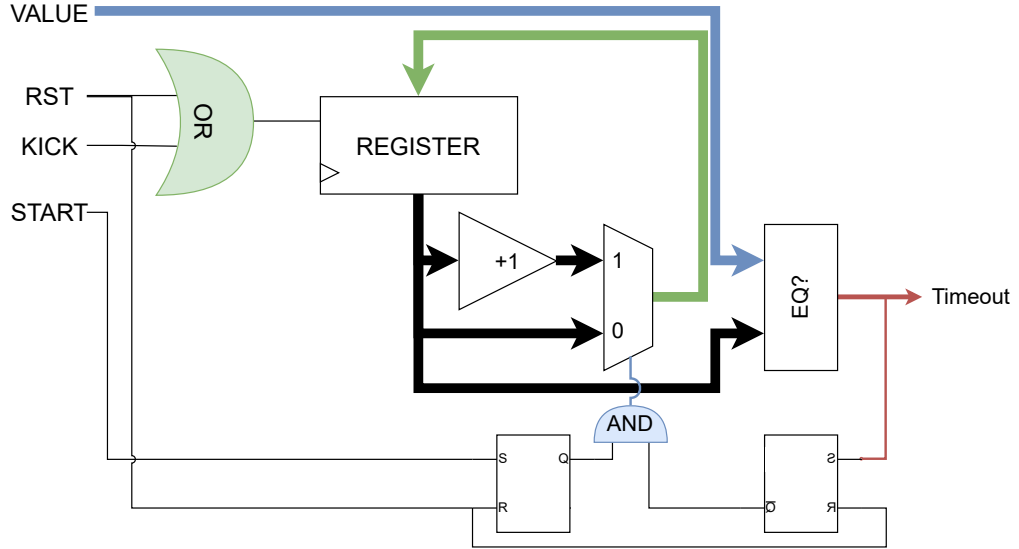
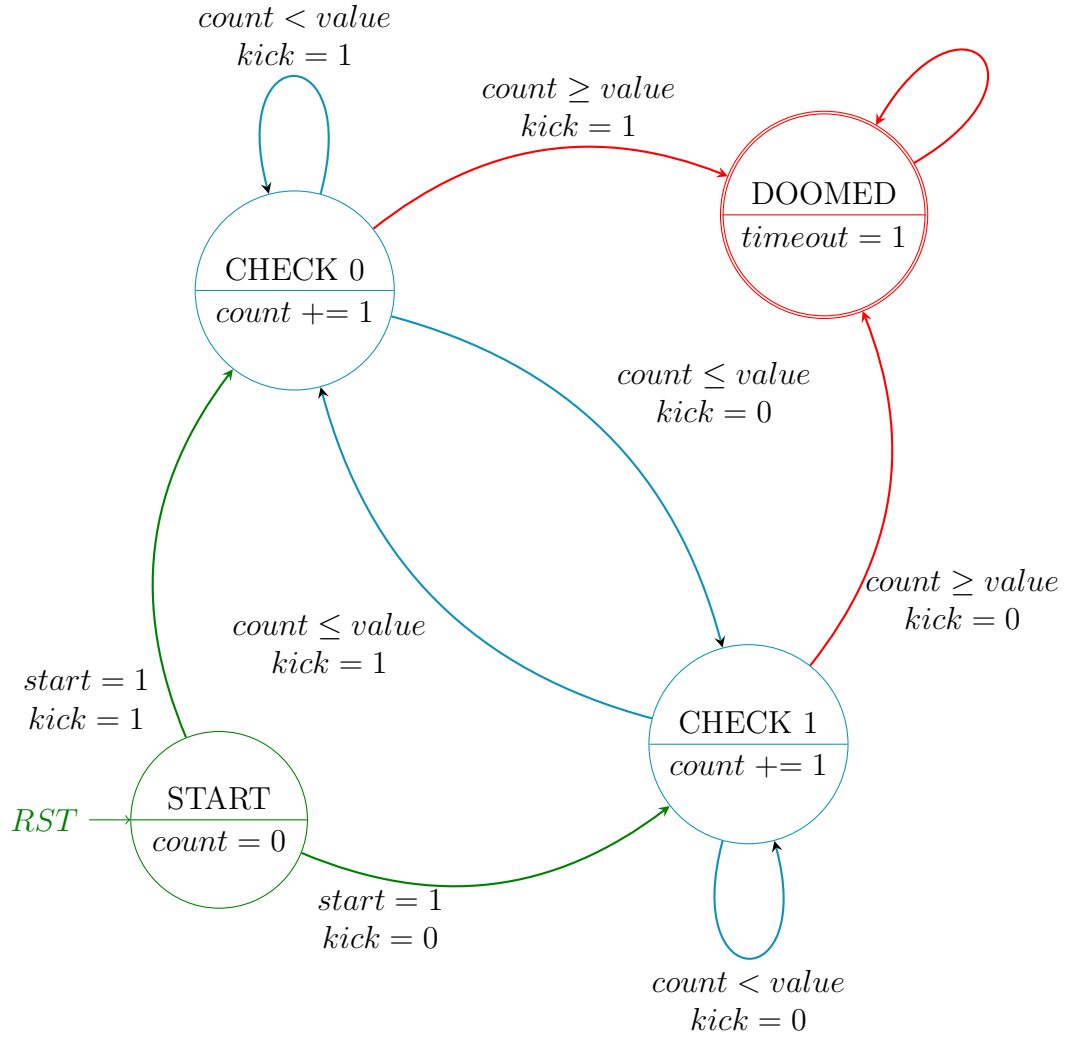


Figure 4.7: Possible digital circuit implementation of a watchdog.

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 a SEU, it can leave the signal stuck in a high state, and the watchdog will continuously reset the timer, masking the real CPU's status.

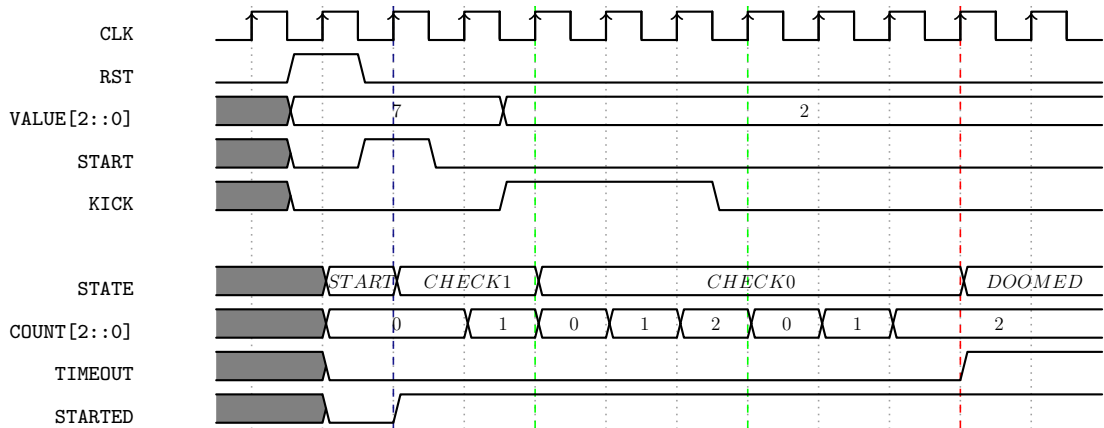
To overcome this problem, a more sophisticated signaling mechanism is used. To keep things simple, the act of kicking is done at every $H \rightarrow L$ or $L \rightarrow H$ transition of the *KICK* signal. This way, if the signal is stuck in a certain state, the watchdog will expire correctly. The final 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 its current state [16]. Inputs only affect the next state, and state transitions happens 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 in a loop between two states 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 in a final state asserting the timeout signal. The FSM will stay in this state until it is resetted 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 | COUNT += 1 TIMEOUT = 0 STARTED = 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). A HLSM is a natural extension of a FSM, used to capture more complex behaviors, including multibit 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 behaviour is presented.

4.3.3 How to harden the watchdog?

Once the watchdog is implemented, it needs to be hardened itself. This is needed to prevent the watchdog from being inhibited by a 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.

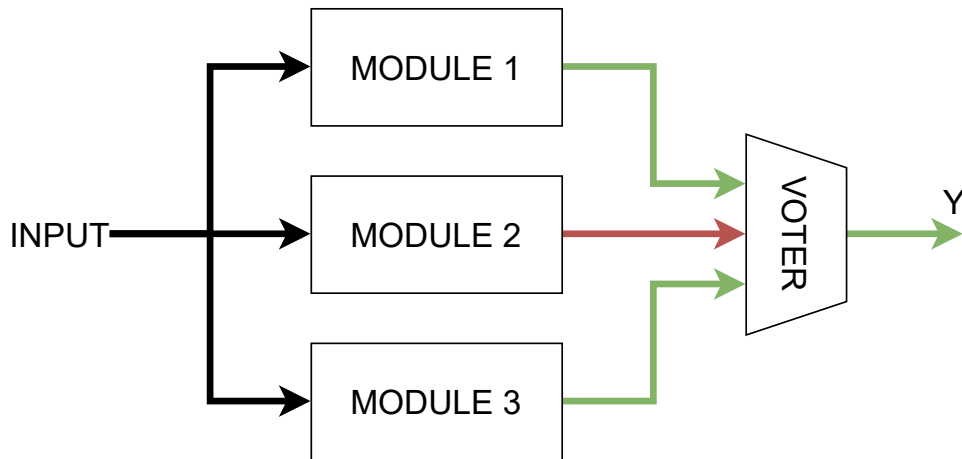


Figure 4.9: Triple Modular Redundancy (TMR) scheme.

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:

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

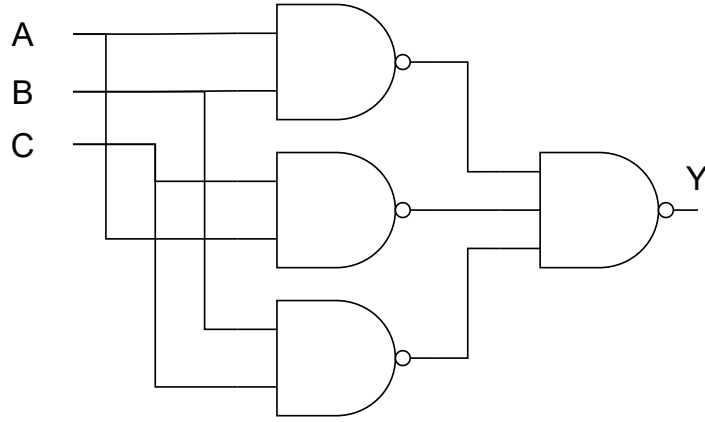


Figure 4.10: 1-bit voter circuit scheme.

The majority gate itself could fail, representing a single point of failure. This can be protected against 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:

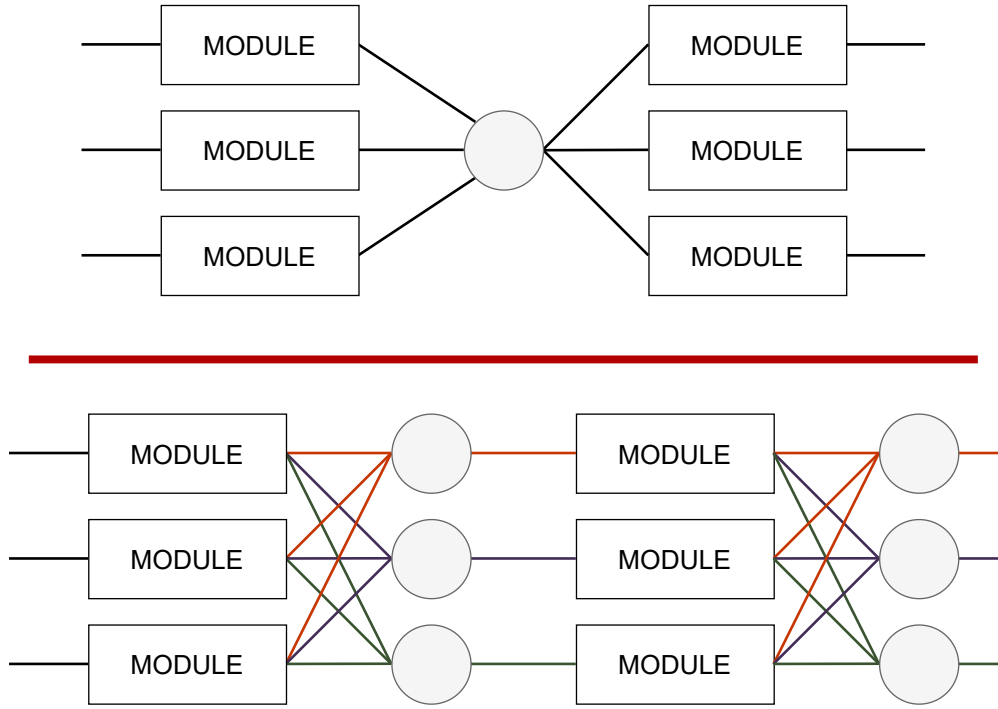


Figure 4.11: Basic TMR scheme vs. full TMR circuit scheme.

Even tho the second scheme doesn't present any single point of failure, most systems sticks with the simplest scheme. This is because the majority 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 reliability 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 set of matrix 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 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.

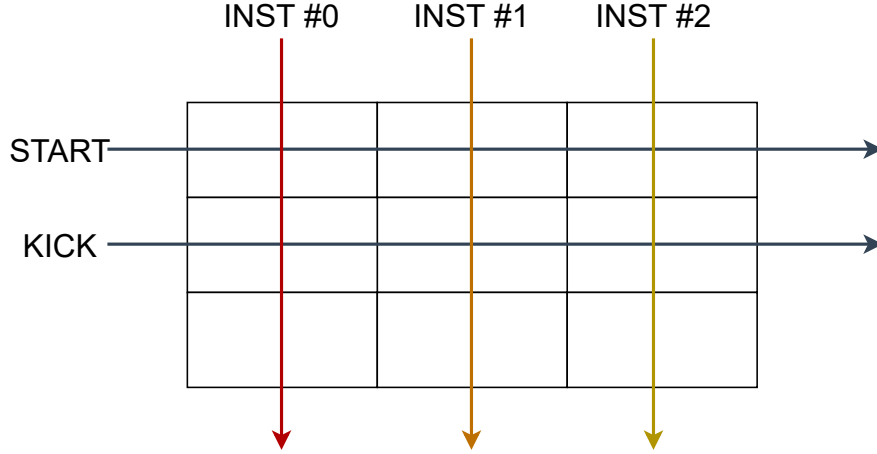


Figure 4.12: Input signal matrix for the no-tmr version.

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 for the output signals of the voters. Three voters creates 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 notmr matrix is configured as the one showed 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 doing 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 in 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 absolutely be 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:

```

1 (* dont_touch = "true" *) wire notmr[3:0][2:0];
2 (* dont_touch = "true" *) wire tmr[3:0][2:0];
3
4 generate
5     genvar jdx;
6     for (jdx = 0; jdx < 3; jdx = jdx + 1) begin
7         for (idx = 0; idx < 4; idx = idx + 1) begin
8             (* dont_touch = "true" *) voter_bus #(
9                 .NBITS(1)
10            ) voter_ith (
11                .DATA_IN0(notmr[idx][0]),
12                .DATA_IN1(notmr[idx][1]),
13                .DATA_IN2(notmr[idx][2]),
14                .DATA_OUT(tmr[idx][jdx])
15            );
16        end
17    end
18 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. 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 well usable. The goal is to have a watchdog that can be easily inserted in 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 a 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 to check its status. The problem is that the watchdog uses discrete signals as 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 directly 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.

For what concerns this thesis work, the second option is the most convenient because there are less 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 a AXI wrapper on top of which the watchdog can be inserted. Hence, the watchdog will be controlled and monitored by the MicroBlaze via registers.

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 registers are:

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).

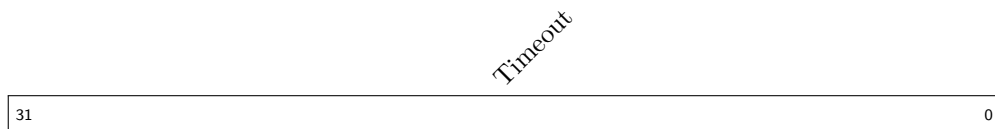
Register 4.2: STATUS REGISTER (0x04 - 0x14 - 0x24)



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 | |
| 31 | 0 |

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.

Once the Watchdog IP repository has been added to the list of IP repositories in Vivado (*Vivado* → *IP* → *Repository* → +), it can be instantiated in the design and connected to a AXI Interconnect to make it accessible to masters like the MicroBlaze:

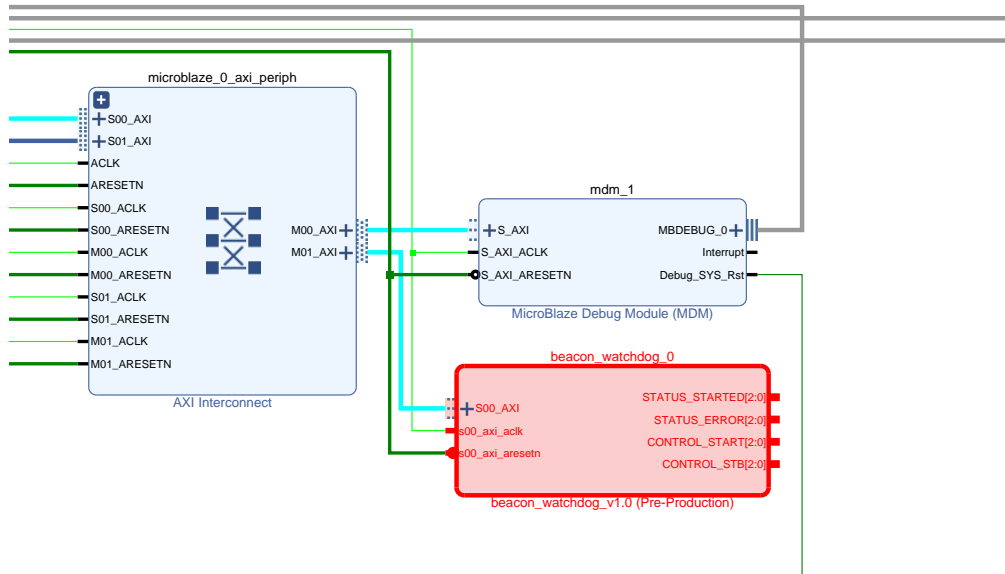


Figure 4.14: Instantiation of the Watchdog IP.

The IP presents two interfaces:

- *S00_AXI*: The AXI interface is used to communicate with the watchdog from a 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 are vectors of 3 positions, 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:

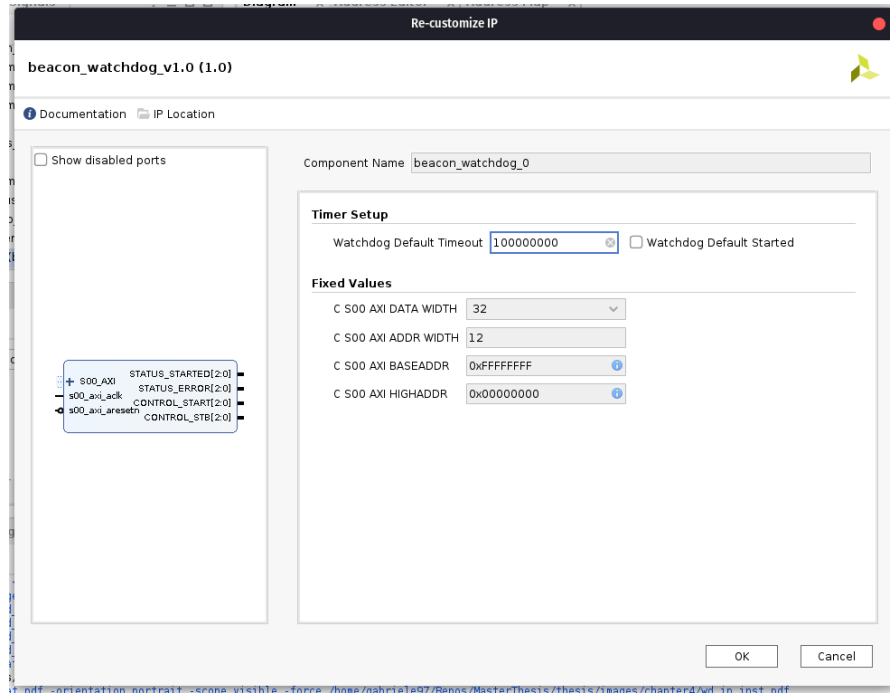


Figure 4.15: Watchdog IP configuration wizard.

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 from Vitis.

The following is an example of 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 accross the code to cover not only CPU hangs but also other faults (computational errors, self-test code, etc.).

```
1 #include "beacon_watchdog.h"
2
3 /* header file for the hardware parameters,
4  like peripheral addresses */
5 #include "xparameters.h"
6 int main() {
7     GBcnCtrl hBcn; //Handle to the watchdog
8     uint32_t value;
9
10    value = XPAR_BEACON_WATCHDOG_0_S00_AXI_BASEADDR;
11    GBcnCtrl_Initialize(&hBcn, value);
12
13    // Timeout set to 2 seconds (double of the watchdog's clock freq)
14    GBcnCtrl_SetTimeoutValue(&hBcn, XPAR_CPU_CORE_CLOCK_FREQ_HZ << 1);
15    GBcnCtrl_Start(&hBcn);
16
17    value = hBcn.modules->module0.DATAREG;
18    printf("Module 0 Timeout Value: %d\r\n", value);
19
20    value = hBcn.modules->module1.DATAREG;
21    printf("Module 1 Timeout Value: %d\r\n", value);
22
23    value = hBcn.modules->module2.DATAREG;
24    printf("Module 2 Timeout Value: %d\r\n", value);
25
26    while(1) {
27        checksum ^= (res += 2);
28
29        // Example of self test check on checksum
30        if(checksum & 0x3 == 0x0) {
31            // FAULT!! Stop kicking, hardware needs to be reset!
32            while(1); // Waiting for timeout expiration
33        }
34
35        // Kicking (automatic transition detection)
36        GBcnCtrl_Toggle(&hBcn);
37    }
38
39 }
```

4.4 Design Partial Reconfiguration

Partial reconfiguration is a way to change or update a 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 needs to be pushed to the FPGA, and a way to achieve this is to partial 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.

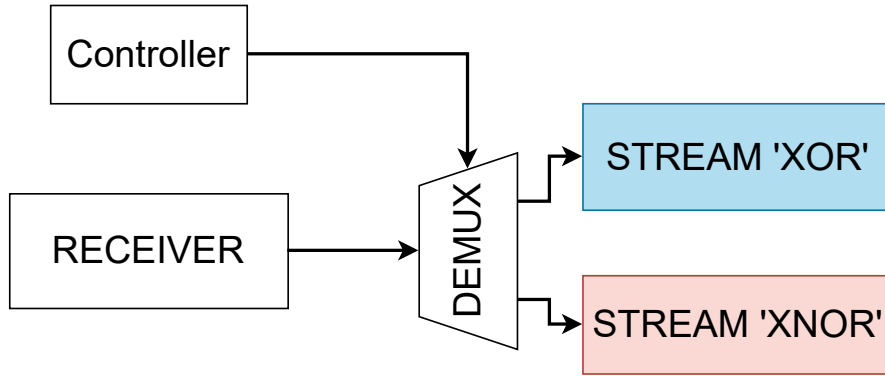


Figure 4.16: Example of partial reconfiguration modules.

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, there are two modules that 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 together. The group is then marked as reconfigurable and the controller, before requesting to compute a checksum to the needed module, it reconfigures the area with the module that the controller designed to compute the checksum. In this way, the system continuous to work in other sub-parts of the design, and meanwhile the partial reconfiguration and the checksum computation is performed. The final design is the following:

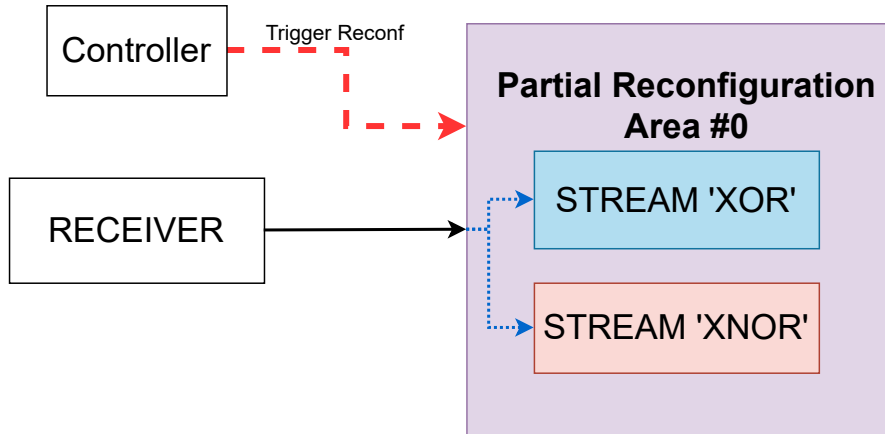


Figure 4.17: Example of modules grouped under a partial reconfiguration area.

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 offers 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 mutual 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 it in the top level module.

As 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 withing a partition only one RM at a time can be available.

A reconfigurable partition is seen from other modules as the same component: other modules do not need to care about the reconfigurable partition. 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 IP 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 the 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 a IP is instantiated, a .xci file is generated. A .xci file is a 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 in 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 me added (via manual selection of the files).

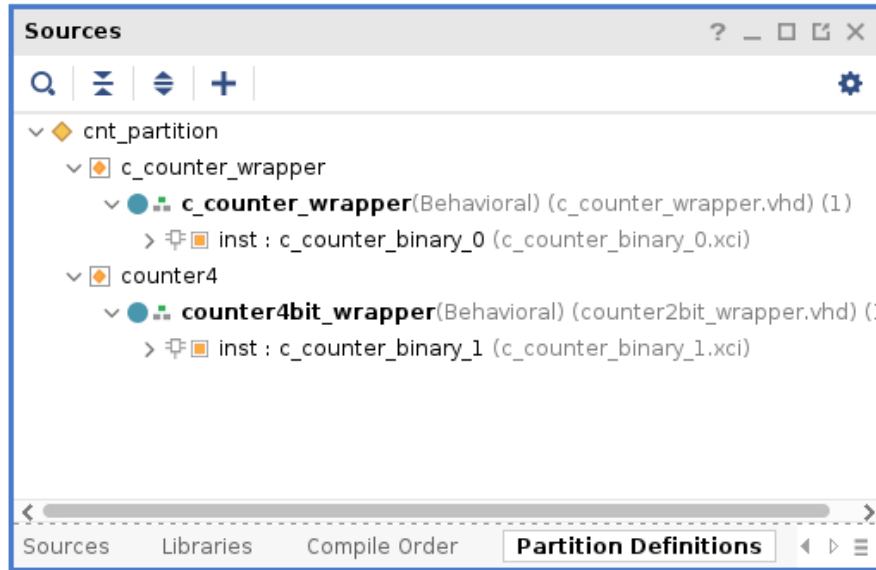


Figure 4.18: Partition definition with two reconfigurable modules, each one with its own wrapper and .xci file.

After the partition definition is completed, it is time to tell to 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 constraints 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 configuration 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: Output signal matrix for the no-tmr version.

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:

```

1 connect
2 target 4 # targets the FPGA
3
4 # loads the full bitstream
5 fpga design_1.bit
6
7 # loads the partial bitstream after the full one has been loaded
  previously
8 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 a 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. software 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 to 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 top level module. This is the chosen way to overcome the first problem: avoid to manually instantiate the MicroBlaze IPs. This creates a new problem: all the IPs definitions (.xci files) used in the Block Design are not available in the 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 still remains. 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 basically 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.

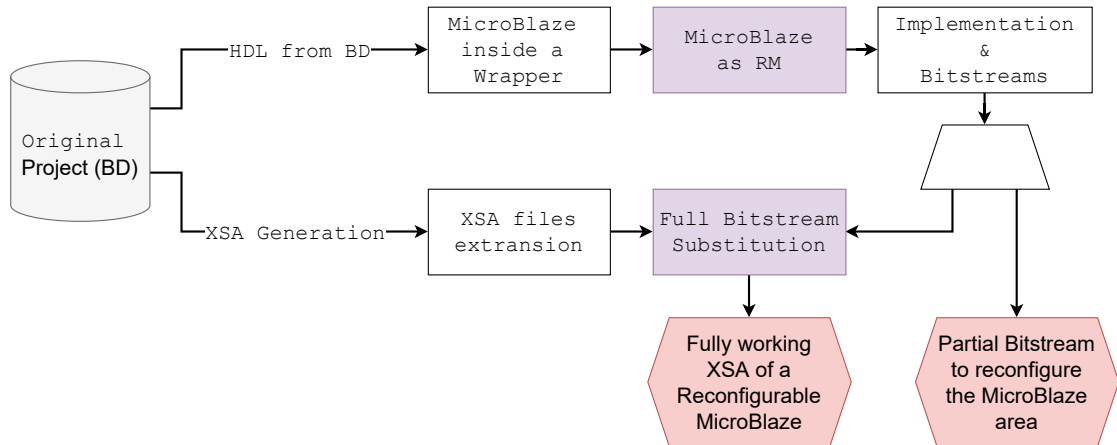


Figure 4.19: Flow used to generate a fully working Reconfigurable MicroBlaze design.

4.4.3 Xilinx DFX Controller

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

Luckily, Xilinx provides an IP that is able to manage the partial reconfigurations from within the FPGA and to ease the process to 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.

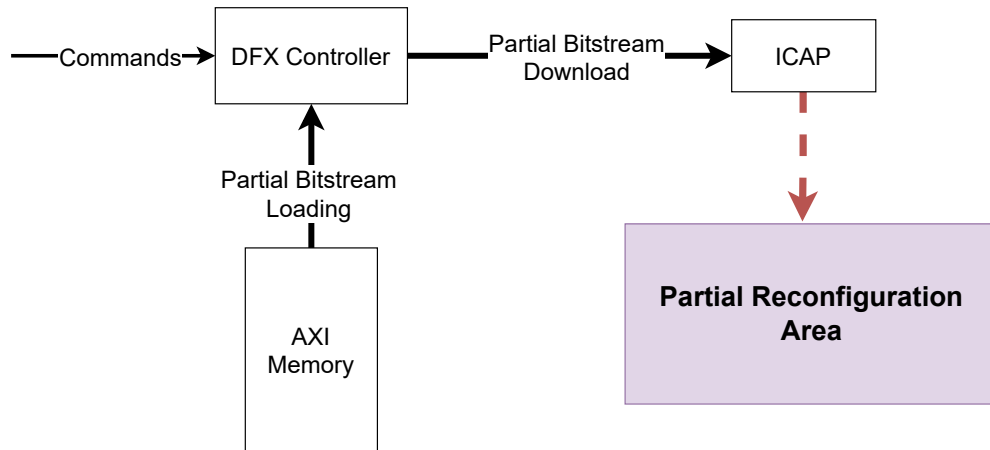


Figure 4.20: Basic scheme of the DFX Controller flow.

As shown in the figure above, the DFX Controller's main job is to fetch the partial bitstream data from a AXI memory peripheral and to 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 Virtual Sockets and each Virtual Socket can manage up to 128 Reconfigurable Modules. A Virtual Socket allow to manage all the RMs inside it as a 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 to define 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 resetted 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 in 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.

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 connect 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 4.6.1. This is not enough because by doing this, 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 definition is created pointing 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:

```

1  /* Define Memories in the system */
2  MEMORY
3  {
4      ps7_ram_0 : ORIGIN = 0x0, LENGTH = 0x30000
5      ps7_ram_1 : ORIGIN = 0xFFFF0000, LENGTH = 0xFE00
6      /*2nd DDR half*/
7      2ndhalf   : ORIGIN = 0x10000000, LENGTH = 0x10000000
8  }
9
10 /* Define the sections, and where they are mapped in memory */
11 SECTIONS
12 {
13 .text : {
14     KEEP (*( .vectors))
15     *( .boot)
16     *( .text)
17     /*-----*/
18     *( .vfp11_veneer)
19     *( .ARM.extab)
20     *( .gnu.linkonce.armextab.*)
21 } > ps7_ddr_0
22
23 /** other sections **/
24 .partialbs : {
25     *( .partialbs)
26 } > 2ndhalf
27 }

```

GCC (the C compiler used by Vitis) allows to define the linker script like the showed one. It allows to force the placement of a certain variable in a certain section. This can be done as follows:

```

1 u32 __attribute__((section(".partialbs"))) data[] = {
2     0xFFFFFFFF,
3     0xFFFFFFFF,
4     0xFFFFFFFF,
5     0xFFFFFFFF,
6     0xFFFFFFFF,
7     0xFFFFFFFF,
8     /* ..... */
9     0xAA995566, // SYNC WORD
10    /* ..... */
11 };

```

For this work, the ARM core is not only used to load the partial bitstream in memory. As explained previously, the ARM cores holds the 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 steps:

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

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

```

1 #include "platform.h"
2 #include "data.h" // Contains the partial bitstream array
3 #include "xdevcfg.h"
4 #include <stdlib.h> // For malloc()
5
6 int main() {
7     XDcfg *XDcfg_0;
8
9     XDcfg_0 = malloc(sizeof *XDcfg_0);
10    XDcfg_Config *conf = XDcfg_LookupConfig(XPAR_XDCFG_0_DEVICE_ID);
11    XDcfg_CfgInitialize(XDcfg_0, conf, conf->BaseAddr);
12
13    XDcfg_DisablePCAP(XDcfg_0);
14    XDcfg_SelectIcapInterface(XDcfg_0);
15
16    /*other code*/
17    return 0;
18 }

```

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 a 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 to be connect to the ICAP interface of the DFX Controller.

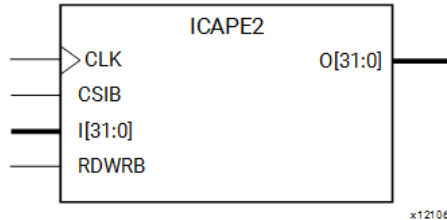


Figure 4.21: ICAP Interface.

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

DFX connected to the watchdog via the hw trigger. ICAPE2. DFX Decoupler.

4.6 A script to automatize the process

4.6.1 Script for partial bitstream to C header generation

Chapter 5

Experimental Analysis

5.1 Fault Injection

5.2 Experimental Results

Chapter 6

Conclusions

Ok there are few caveats like:

- single point of failure in the error/trigger signal (what if the routing of the error/trigger signal is affected by a seu?)

- single point of failure in the TMR output (what if the OR gate is affected by a seu?)

The watching channel (beacon version) is not a single point of failure, because if it is affected by a seu it may be stuck (unconnected net) so watchdog expires and reconf is triggered. However, the stuck remains because the watchdog is not reconfigured.

- the DFX is a single point of failure too.

- the watchdog protects the microblaze from hangs. It can protect from application layer faults only if the firmware running on the microblaze properly signals errors to the watchdog. For example, does checksum controls on the running algorithm, does some self-tests. For example, maybe the algorithm is computing correctly but the UART output is wrong: the firmware, during self-test mode can redirect the output of the UART to the input of the UART, read back what is outputting and see if the output is correct.

- Another thing is to put one more beacon signals to a memory read/write signal. If we expect that every tot ms the system must read from the memory, the watchdog can look at it. if the processor stops working, means that no more transactions are performed and the watchdog expires.

6.1 Future Work

Extension of the watchdog to other types of checks from other sources like direct trigger (signal at 1 => trigger) Extension of the watchdog to other beacon-based activities.

DFX+Watchdog+Microblaze all in a reconfigurable area so they are reconfigured in case of error.

Extension of the script to reconfigure other modules too.

Appendix A

Watchdog FSM - VHDL Code

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use ieee.numeric_std.all;
4
5 entity top_beacon_watchdog is
6     generic (
7         DATA_WIDTH: integer := 32
8     );
9     port (
10         CLK:      in  std_logic;
11         RST:      in  std_logic;
12
13         DATAIN:  in  std_logic_vector(DATA_WIDTH-1 downto 0);
14         START:    in  std_logic;
15         STB:      in  std_logic;
16         TOGRATE:  out std_logic_vector(DATA_WIDTH-1 downto 0);
17         WORKING:  out std_logic;
18         ERR:      out std_logic
19     );
20 end top_beacon_watchdog;
21
22 architecture arch of top_beacon_watchdog is
23
24     type fsm_state is (S_START, S_CHECK_0, S_CHECK_1, S_DOOMED);
25     signal curr_state, next_state: fsm_state;
26     signal curr_timeout, next_timeout:
27         std_logic_vector(DATA_WIDTH-1 downto 0);
28     signal curr_cnt, next_cnt: std_logic_vector(DATA_WIDTH-1 downto 0);
29     signal curr_toggle_rate: std_logic_vector(DATA_WIDTH-1 downto 0);
30
```

```

31 begin
32
33 TOGRATE <= curr_toggle_rate;
34
35 process(curr_state, curr_timeout, curr_cnt, DATAIN, STB, START)
36 begin
37
38     next_state <= curr_state;
39     next_timeout <= curr_timeout;
40     next_cnt <= std_logic_vector(unsigned(curr_cnt) + 1);
41
42     ERR <= '0';
43     WORKING <= '1';
44
45     case(curr_state) is
46
47         when S_START =>
48             next_timeout <= DATAIN;
49             next_cnt <= (others => '0');
50             WORKING <= '0';
51
52             if START = '1' then
53                 if STB = '0' then
54                     next_state <= S_CHECK_1;
55                 elsif STB = '1' then
56                     next_state <= S_CHECK_0;
57                 else
58                     next_state <= S_DOOMED;
59                 end if;
60             end if;
61
62             when S_CHECK_0 =>
63
64                 if unsigned(curr_cnt) < unsigned(curr_timeout) then
65                     if STB = '0' then
66                         next_cnt <= (others => '0');
67                         next_timeout <= DATAIN;
68                         next_state <= S_CHECK_1;
69                     end if;
70                 else
71                     next_cnt <= (others => '0') ;
72                     next_state <= S_CHECK_1;
73                     next_timeout <= DATAIN;
74                     if STB /= '0' then
75                         next_state <= S_DOOMED;
76                     end if;
77                 end if;
78
79

```

```

80     when S_CHECK_1 =>
81
82         if unsigned(curr_cnt) < unsigned(curr_timeout) then
83             if STB = '1' then
84                 next_cnt <= (others => '0');
85                 next_state <= S_CHECK_0;
86                 next_timeout <= DATAIN;
87             end if;
88         else
89             next_cnt <= (others => '0');
90             next_state <= S_CHECK_0;
91             next_timeout <= DATAIN;
92             if STB /= '1' then
93                 next_state <= S_DOOMED;
94             end if;
95         end if;
96
97     when S_DOOMED =>
98         next_cnt <= (others => '0');
99         ERR <= '1';
100
101     when others =>
102         WORKING <= '0';
103         next_state <= S_START;
104
105 end case;
106
107 end process;
108
109 process(clk)
110 begin
111
112     if rising_edge(clk) then
113         if (RST = '1') then
114             curr_state <= S_START;
115             curr_cnt <= (others => '0');
116             curr_timeout <= (others => '0');
117             curr_toggle_rate <= (others => '0');
118         else
119             curr_state <= next_state;
120             curr_timeout <= next_timeout;
121             curr_cnt <= next_cnt;
122
123             if unsigned(next_cnt) = 0 then
124                 if unsigned(curr_cnt) > unsigned(curr_toggle_rate) then
125                     curr_toggle_rate <= curr_cnt;
126                 end if;
127             end if;
128

```

```
129         end if;  
130     end if;  
131  
132     end process;  
133  
134 end arch;
```

Appendix B

Galileo

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

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

numpy

Bibliography

- [1] European Space Agency. *Three hours to save Integral*. 2021. URL: https://www.esa.int/Enabling_Support/Operations/Three_hours_to_save_Integral (cit. on p. 1).
- [2] 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).
- [3] 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).
- [4] 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).
- [5] Boyang Du, Luca Sterpone, Sarah Azimi, David Merodio Codinachs, Véronique Ferlet-Cavrois, Cesar Boatella Polo, Rubén García Alía, Maria Kastriotou, and Pablo 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).
- [6] 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>.
- [7] 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).

- [8] 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).
- [9] 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).
- [10] 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).
- [11] 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).
- [12] 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).
- [13] 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. 31).
- [14] 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. 37).
- [15] 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. 38).
- [16] 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, Princeton 1956, pp. 129–153.» In: *Journal of Symbolic Logic* 23 (1958), pp. 60–60 (cit. on p. 42).