Politecnico di Torino

Master Degree in Electronic Engineering

Master Thesis

High-Energy Physics Fault Tolerance Metrics and Testing Methodologies for SRAM-based FPGAs

A case of study based on the Xilinx Triple Modular Redundancy (TMR) Subsystem

Advisor
prof. Marco Parvis
Co-Advisor
prof. Michelangelo Agnello

Candidate
Emanuele Canessa

Contents

1 Introduction 1

2 Radiation Effects on Field-Programmable Gate Arrays 3

2.1 FPGA Technologies 3

2.2 Radiation Effects 5

2.2.1 Ionizing Radiation 6

2.2.2 Radiation Measurements 7

2.3 Single Event Effects 8

2.3.1 Single Event Upsets 10

2.3.2 Single Event Transients 10

2.3.3 Destructive Single Event Effects 11

2.4 Single Event Effects on SRAM-based FPGAs 13

2.4.1 Configuration RAM Upsets 13

3 Available Techniques for Fault Tolerance 17

3.1 Spatial Redundancy 17

3.1.1 Duplex Architectures 18

3.1.2 Majority-Voting Architectures 18

3.2 Information Redundancy 22

3.2.1 Parity Code 23

3.2.2 Hamming Code 24

3.3 Temporal Redundancy 25

4 Metrics for Fault Tolerance 27
4.1 Dependability . . . 27
4.1.1 Dependability Attributes . . . 28
4.1.2 Dependability Threats . . . 33
4.1.3 Dependability Means . . . 35
4.2 Fault Classification . . . 35
4.3 Metrics for Single Event Effects on FPGAs . . . 36
4.3.1 Cross Section . . . 37
4.3.2 Measurement of SEE Sensitivity . . . 37
4.3.3 SEU Sensitivity on FPGAs . . . 38

5 Radiation Hardness Design Validation 41
5.1 Fault Injection . . . 41
5.1.1 Fault Injection Procedure . . . 42
5.1.2 Limitations of Fault Injection . . . 42
5.1.3 The Xilinx Soft Error Mitigation IP . . . 43
5.1.4 The JTAG Configuration Manager . . . 46
5.2 Ground Testing . . . 48
5.2.1 Testing Methodology . . . 48
5.2.2 Radiation Decay . . . 48
5.2.3 Tests after Retrieval . . . 49

6 Characterization of the Xilinx Triple Modular Redundancy Subsystem 51
6.1 Xilinx Microblaze and TMR Subsystem . . . 51
6.1.1 Recovery of the Microblaze Subsystem . . . 52
6.2 Benchmarks for Radiation Testing . . . 54
6.2.1 The Algorithm of Choice . . . 55
6.3 Microprocessor Testing Metrics . . . 56
6.3.1 Working status of a processor . . . 56
6.3.2 Severity Analysis . . . 56

6.3.3 Mean Time to Failure Evaluation . . . 57
6.4 Testing Procedure and Architecture . . . 58
6.4.1 Single Module Testing . . . 58
6.4.2 Tabletop Testing . . . 60
6.4.3 Ground Testing . . . 61

7 Results, Conclusions and Future Work 63
7.1 Results . . . 64
7.1.1 Tabletop Testing Results . . . 64
7.1.2 Ground Testing Results . . . 65
7.2 Conclusions and Future Work . . . 65

A A Large Ion Collider Experiment 67
A.1 Upgrade of the Inner Tracking System . . . 67
A.1.1 Readout Electronics . . . 69

B Advanced Encryption Standard 71
B.1 Working Principle . . . 71
B.2 AES Versions . . . 71
B.3 Implemented Algorithm . . . 72
B.3.1 `aes.h` . . . 73
B.3.2 `aes_constants.h` . . . 75
B.3.3 `aes.c` . . . 77

C Xilinx Microblaze and TMR Subsystem 83
C.1 Configuration Scripts . . . 83
C.1.1 `generate_mb.tcl` . . . 83
C.2 Firmware . . . 89
C.2.1 `test_micro.c` . . . 89
C.2.2 `mb_recovery.S` . . . 91

List of Figures 97

List of Tables 101

Acronyms 103

Bibliography 107

Chapter 1

Introduction

Field-Programmable Gate Arrays have become more and more actractive to the developers of mission-critical and safety-critical systems. Thanks to their reconfigurability properties, as well as their I/O capabilities these devices are often employed as core logic in many different applications, like:

- Aerospace and Defense;
- ASIC Prototyping;
- Audio;
- Automotive;
- Broadcast;
- Consumer Electronics;
- Distributed Monetary Systems;
- Data Center;
- **High-Energy Physics**;
- High Performance Computing;
- Industrial;
- Medical;
- Scientific Instruments;
- Security systems;

- Video and Image Processing;
- Wired Communications;
- Wireless Communications.

On top of that, the use of soft microcontrollers can ease the complexity related to the some of the control logic of these devices, allowing to easily develop new features without having to redesign most of the control logic involved.

However, for application safety-critical and mission-critical like Aerospace and High-Energy Physics these devices require a further analisys on radiation effects. The main matter of this thesis, that has been developed in collaboration with the Conseil Européen pour la Recherche Nucléaire (CERN) A Large Ion Collider Experiment (ALICE), for the planned Inner Tracking System (ITS) Upgrade, are discussed the fault tolerance metrics and the testing methodologies that can be applicable to soft microprocessor cores running on FPGAs.

In Chapter 2 are discussed the effects of radiation on FPGAs, as well as the main units of measure involved. Particular attention is then dedicated to the so-called Single Event Effects.

In Chapter 3 are discussed the main techniques employed to protect digital designs load onto FPGAs.

In Chapter 4 are discussed the main metrics that are available to classify the effects of faults in these devices, with particular emphasis to the ones employed for Single Event Effects.

In Chapter 5 are discussed the available techniques for radiation hardness design validation. In particular, are presented the working schemes for tabletop testing and ground testing.

In Chapter 6 are introduced the metrics and the testing methodologies that have been used to characterize the Xilinx TMR Subsystem against radiation effects.

Finally, in Chapter 7 are presented the results of the characterization process and the conclusions, as well as the possible future work associated to this matter.

Chapter 2

Radiation Effects on Field-Programmable Gate Arrays

Field-Programmable Gate Array (FPGA)s are becoming more and more attractive in many fields of applications due to their reconfiguration capabilities. FPGAs, however, are highly sensible to ionizing radiation. This weakness makes them very prone to radiation-induced memory upsets.

2.1 FPGA Technologies

There are three major types of FPGA technologies on the market:

- Antifuse-based;
- Flash-based;
- Static RAM (SRAM)-based.

Antifuse-based FPGAs

Antifuse-based FPGAs were the most used technology in radiation environments. This family of Field-Programmable Gate Arrays are characterized by having a One-Time Programmable (OTP) memory, thus making the configuration permanent after the first programming. The fuse technology is the less susceptible to radiation effects: once a fuse is "blown", the change is permanent. The price to pay for this technology is high. First of all, there is no reconfiguration capability: every time that the firmware changes, the device have to be changed. Secondly, the capabilities are very limited for its economic price: the technology involved in antifuse-based FPGAs is often very old.

Flash-based FPGAs

Flash-based Field-Programmable Gate Arrays are a point in the middle between antifuse-based and SRAM-based. The configuration memory bits are stored in a flash memory that provided an highly -but limited- number of reprogramming cycles. Like the antifuse technology, this technology is also non-volatile. While the radiation susceptibility is higher than antifuse-based FPGAs, the hardware capabilities are less restricted. Their use in radiation environments, though, is limited by the flash memory technology adopted, *floating gate*. The transistors used for this technology are easily degraded by the presence of charges in their gates, this is a strong limitation if the memory is hit by radiation particles that can easily move charges in this location.

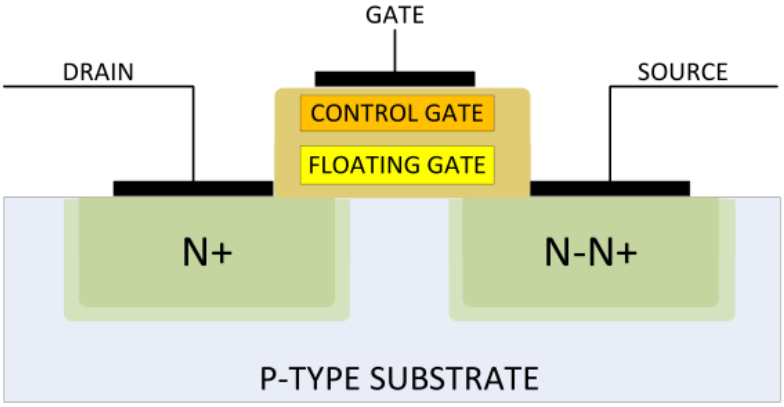


Figure 2.1: Floating Gate NMOS Transistor: The accumulation of charges in the Floating Gate prevents the transistor from working as expected, eventually, the value stored is changed when the charge exceed the threshold.

Static RAM-based FPGAs

Finally, SRAM-based FPGAs are characterized by having all the configuration bits stored in a Static RAM. Although this choice leads to potentially an infinite number of reconfiguration cycles, the memory itself is volatile and it is the most susceptible to radiation effects among the others. The strength of this family of devices resides in the technology adopted, that is the best available on the market. It is also important to note that an external memory has to be present in order to reprogram the Configuration RAM in case of power loss.

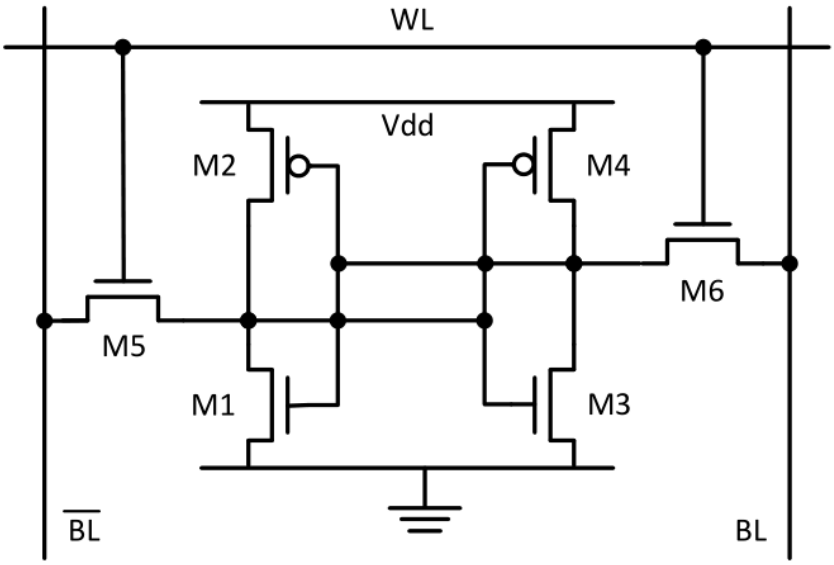


Figure 2.2: SRAM Cell: the effect of a particle striking through one of the M1..M4 transistors could flip the value stored the memory cell by the positive feedback of the structure.

Feature	SRAM	Flash	Antifuse
Reprogrammability	High	Medium-Low	One-Time
Volatile	Yes	No	No
Memory Size	High	Medium	Small
Radiation Sensitivity	High	Medium-Low	Low-None
Total Dose Tolerance	Medium-High	Low	High

Table 2.1: Comparison of FPGA technologies in terms of the main considered parameters in radiation environments.

2.2 Radiation Effects

Radiation is very common in many different environments, it can be emitted by natural sources as well as artificial sources. This section will give a brief description of the theory and the various units of measure used later on.

2.2.1 Ionizing Radiation

Ionizing Radiation identifies any kind of radiation that is able to produce, directly or indirectly, the ionization of atoms or molecules of the material that cross, i.e. the extraction of one or more electrons from them.

It is defined *Directly Ionizing Radiation* the kind of radiations that is composed by charged particles, like:

- electrons;
- protons;
- α-particles;
- β-particles;
- heavy ions.

On the other hand, *Indirectly Ionizing Radiation* could be caused by particles without charge, like neutrons, and high-energy electromagnetic radiation, like photons, γ rays and X-rays.

Ionization radiation is hazardous for an electronic circuit, since the effect of ionizing an atom or a molecule can change the behavior of an electronic circuit. Finally all the moving charged particles are influenced by the effect of an electromagnetic field, where it is present, due to the Lorentz force.

α-particles

An α-particle is an Helium nucleus, made of two neutrons and two protons and is a very highly ionizing particle. For this reason the α-particle lose their energy in a short path inside the material and can be easily shielded with a few centimeters of air or a thin thickness shielding material like a sheet of paper.

β-particles

The β-particles are electrons or positrons emitted from radioactive atoms. Their energy spectrum can vary from a few keV up to 10 MeV and it is dependent by the emitting atoms. As the other ionizing radiations, a simple and low-cost Geiger counter can detect beta particles, although without the information about their energy. Beta particles can be easily stopped in the material: for instance, a 1 MeV beta particle can be stopped by a thin ($\sim$ 1 mm) Aluminum foil. On the other

hand, β-particles crossing materials with high atomic numbers (Z) can produce Bremsstrahlung radiation (*photons*) that can easily penetrate the material. Besides, the positrons can annihilate and produce two photons of 0.511 MeV.

Neutrons

Neutrons, per se, are not able to cause *direct ionization* having zero electrical charge. Their interaction with the material, instead, can cause recoil in the nuclei present; finally, the nuclei's can cause subsequent ionization in other atoms. Having zero electrical charge, these particles have a greater penetration capability with respect to the particles discussed above.

2.2.2 Radiation Measurements

In order to identify its effects, radiation has to be measured. In the following section are discussed briefly the main units of measure used.

Total Ionizing Dose

One of the most common units of measure that are used for radiation is the so called *absorbed dose* or *Total Ionizing Dose (TID)*. This quantity is often measured in Gray (Gy) or, less frequently, in radians (1 Gy = 100 rad). The TID has a direct correlation with the energy that has been absorbed by the material: in fact, an absorbed dose of 1 Gy corresponds to an absorbed energy of 1 $\frac{\text{J}}{\text{kg}}$.

The absorbed dose has also a biological significance, but it is also necessary to take into account the type of radiation considered. This operation requires the definition of a *weighting factor*, w_r, for each radiation type. Using the previously defined weighting factors, each pair of radiation type and energy is multiplied by the correspondent weighting factor, therefore obtaining a "weighted" absorbed dose, called *equivalent dose* and measured in Sievert (Sv).

Linear Energy Transfer

Another important unit of measure to describe is called *Linear Energy Transfer (LET)*. This quantity models the interaction between different radiation types and materials: it represents the quantity of energy that has been released on the material by an incoming radiation.

LET is defined as the amount of energy that an ionizing particle transfers to the material traversed per unit length. It can be defined using the following formula:

$$\text{LET} = \frac{dE}{dx} \tag{2.1}$$

Where dE is represents the quantity of energy that has been transferred and dx represents the distance traveled in the material. Although it can be expressed in Newton (N), most often the unit of measure used to express this quantity is $\frac{\text{MeV}}{\text{cm}}$.

Different particles have different Linear Energy Transfer. For instance, α-particles are often referred as High-LET, while others –like β-particles– are defined as Low-LET.

Finally, *Effective Linear Energy Transfer* (LET_{eff}) is often used when the LET has been already characterized using a perpendicular beam to the material. This quantity is expressed as follows:

$$\text{LET}_{\text{eff}} = \frac{\text{LET}}{cos(\theta)} \tag{2.2}$$

Fluence

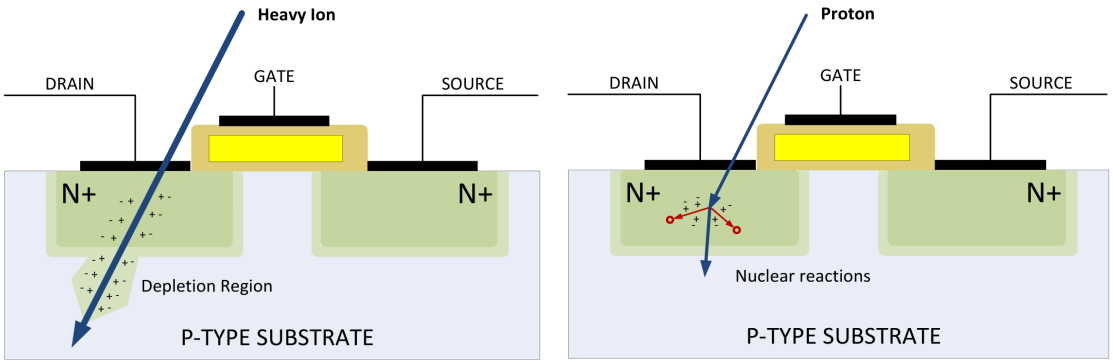
Fluence is the *Flux* integrated over a period of time. The *particle fluence* defines the number of particles passing through a spherical surface during a specified period of time ΔT.

$$\Phi = \int_{\Delta T} \phi \; dt \tag{2.3}$$

Where Φ is the fluence, ϕ is the flux.

2.3 Single Event Effects

Single Event Effect (SEE) is a generic term that describe the type of effects that can be caused by a single particle striking a silicon device. Necessary condition for a Single Event Effect to come true is that the penetrating particle has a sufficient LET to cause a ionization.



(a) Heavy Ion striking through a transistor and creating a ionization path

(b) Proton inducing nuclear reactions in a transistor

Figure 2.3: Single Event Effects on transistors: the effect of striking particles can activate the transistors.

In particular, there are four main forms of Single Event Effect:

- *Single Event Upset (SEU)*;
- *Single Event Transient (SET)*;
- *Single Event Induced Burnout (SEB)*;
- *Single Event Gate Rupture (SEGR)*;
- *Single Event Latchup (SEL)*;
- *Single Event Snap-Back (SESB)*;
- *Single Event Hard Error (SEHE)*.

The first two families of errors are often referred to as *Soft Errors*; the term derives from the fact that this type of errors can be cleared by power cycling the circuit.

The last five families, instead, are examples of *Hard Errors*: these errors lead to a permanent misbehavior of the circuit; to recover from an hard errors it is often necessary to replace the whole device.

2.3.1 Single Event Upsets

Single Event Upsets are a special form of Single Event Effects: they model the effect of a striking particle that hits a memory element in a sequential circuit and flip its value. Among the other types of SEEs, SEUs are the less destructive events that can be caused by striking particles.

These errors manifest themselves with a high probability in devices that contain large memory elements: this is a common denominator in FPGAs.

Single Event Functional Interrupts

Single Event Functional Interrupt (SEFI) is a particular type of SEU that takes place when one of the basic functionality of the circuit is interrupted due to the upsed. Common examples of SEFIs are particles that hits the clock tree configuration bits in FPGAs.

Multiple-Cell Upsets and Multiple-Bit Upsets

With *Multiple-Cell Upset (MCU)* identifies a special type of SEU that change the state of two or more logic cells. These cells are usually physically adjacent, so that a single particle can hit partially all of them.

A particular case of MCU is represented by a *Multiple-Bit Upset (MBU)*: in this case the cells whose value have been flipped by the particle are inscribed by being part of the same word. These effects are very destructive in terms of functional behavior of the circuit: in fact, error correction codes are usually not able to correct more than one bit flip per word. For this reason, many hardware manufacturers produce their own memory where cells of the same word are interleaved by cells of other words, so that the possibility of having an MBU is greatly reduced.

2.3.2 Single Event Transients

Another special kind of SEE is represented by the *SET*. This type of soft errors models a change in the timing of a signal. The circuit behavior induced by a SET can be easily modeled as a glitch in a signal propagating through the circuit. [4]

If the voltage transient caused by the particle striking through a node in the combinational logic is captured by a storage element, it can lead to a state change. In this case the SET resulted in a SEU in a memory element.

2.3.3 Destructive Single Event Effects

In this category are included all the SEEs that can cause permanent damage to the integrated circuits on which they arise.

Single Event Induced Burnout

Single Event Induced Burnouts affects usually the power transistors present in a circuit. It corresponds to a trigger of their parasitic bipolar structure, that is followed by a positive feed-back. The feedback increase rapidly the current flowing therefore producing a *burnout* in the transistor affected.

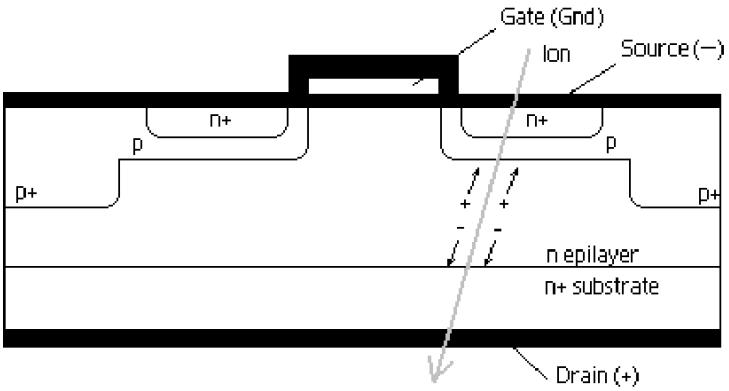


Figure 2.4: Single Event Induced Burnout on a MOS Transistor: the parasitic bipolar structure is excited, the followed by a positive feedback that increase the temperatures.

This type of effect is quite rare in both ASICs and FPGAs.

Single Event Gate Rupture

A Single Event Gate Rupture, also called *Single Event Dielectric Rupture (SEDR)*, represent the destructive rupture of the dielectric present in a transistor (usually is the gate oxyde). The rupture of the dielectric cause the formation of a conducting path, in the case of SEGR a permanent leakage gate current is added.

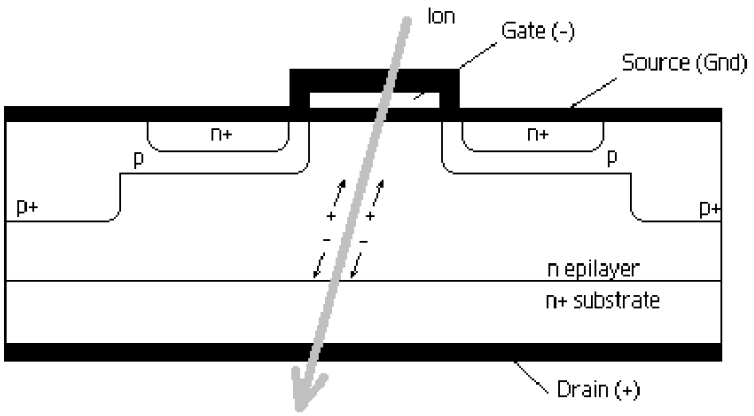


Figure 2.5: Single Event Gate Rupture on a MOS transistor: the dielectric present in the gate of the transistor is pierced.

This hard errors are always destructive, and the only way to protect a component against these effects is to force electrical conditions such that their generation is not possible.

Single Event Latch-Up

Single Event Latchup, unlike the previous, is a *potentially* destructive effect. It consists of the triggering of a parasitic PNPN thyristor structure in the device. The effect alone is not destructive per se, however, the current generated tends to increase over time due to the rising temperature. If not stopped by powering off the device soon enough, a thermal destruction is likely to occur.

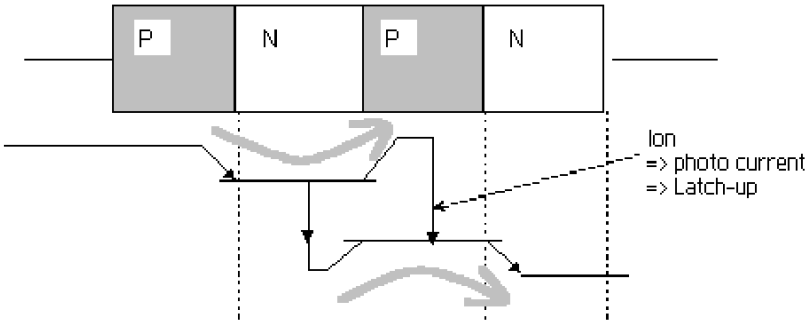


Figure 2.6: Single Event Latch-Up on a PNPN thyristor structure: the particle excites the implicit thyristor structure that starts conducting due to the positive feedback.

Single Event Snap-Back

Single Event Snap-Back is very similar to the effect produced by a SEL, the only difference is that it occurs withing a single MOS structure. Similary, an high current is generated between drain and source region, amplified by the intrinsic bipolar transistor placed in between. The high current, as in the other, generates a localized heating that could lead to permanent damage if the device is not power down.

Single Event Hard Error

Finally, the outcome caused by a Single Event Hard Error is very similar to a SEU: a memory cell's bit flips its value. The difference is that the change is semi-permanent or permanent, for this reason a SEHE is often called *stuck bit error* or *hard fault*.

2.4 Single Event Effects on SRAM-based FPGAs

As discussed in Section 2.1, the different technologies used for FPGAs are characterized by different tolerance to radiation. The lack of functionality of antifuse- and flash-based devices forces the usage of SRAM-based FPGAs to implement complex designs with strict requirements. These devices, though, presents a strong susceptance to Single Event Effects.

The most common type of SEE present in SRAM-based FPGAs, as discussed in Section 2.3.1, are indeed the Single Event Upsets. These special type of soft-errors can result in a number of error modes in different parts of the FPGA.

2.4.1 Configuration RAM Upsets

In a SRAM-based FPGA, the most sensible component to SEUs is surely the *Configuration RAM (CRAM)*. This memory holds information about:

- *Look-Up Table (LUT)* contents;
- *User Data* contents;
- *Input/Output (I/O)* configuration;
- *Routing* configuration.

It is important to note that non all SEUs lead to errors: for instance, there may be some configuration bits that are either not used or even disabled. Xilinx defined a set of special bits, called *essential bits*, as a subset containing only the bits that are essential for the specific design that is loaded onto the FPGA. Flipping a Xilinx's essential bit value leads to misbehavior(s) in the design.

Finally, errors affecting this memory are often called *static errors* because they will not disappear until actively corrected by either *scrubbing* or complete reconfiguration.

LUT Contents Errors

LUTs are used to configure the logic function of combinational logic inside the FPGA. Every bit in these memory elements defines the output of the combinational block given a particular input. In case of an upset, the logic function implemented changes, modifying the behavior of the circuit described.

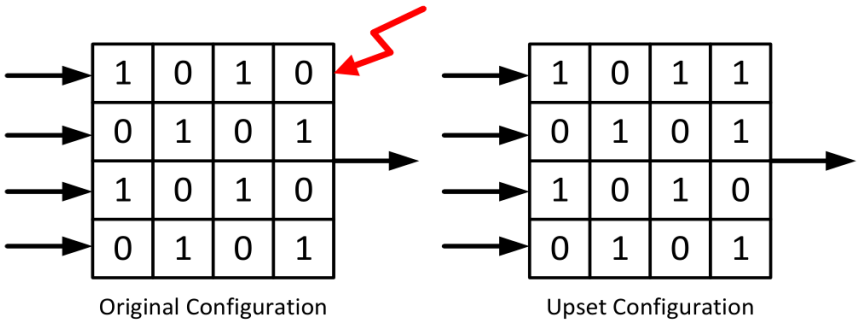


Figure 2.7: Single Event Upset in a LUT: the logic function implemented is changed

User Data Errors

In a FPGA, the user data memory is composed of all the memory elements that are used inside the design. These memory elements usually include:

- *D Flip-Flop (DFF)*;
- *Block RAM (BRAM)*;
- *Distributed RAM (DRAM)*.

The contents of these components can change at any time during the operating time of the design; for this reason, errors affecting these configuration bits are not even considered permanent. To correct errors present in the user data space it is necessary to employ techniques for design mitigation, discussed in Chapter 3.

I/O Configuration Errors

FPGAs have many different I/O configuration capability. Usually, all the pins available are configurable as input, output, and bidirectional buffers. An error affecting the configuration bits responsible for this feature can potentially lead to permanent damage of the device: for instance, a pin that was previously configured as an input could be reconfigured as output, leading to short circuits.

Routing Configuration Errors

Three main categories of routing elements can be affected by SEUs:

- *Programmable Interconnect Point (PIP)*;
- Multiplexers;
- Buffers.

A PIP is the simplest interconnection available in an FPGA, it connects point A to point B using a transistor that is driven by a single configuration bit. A SEU on a PIP interconnection could create an unwanted open or short circuit.

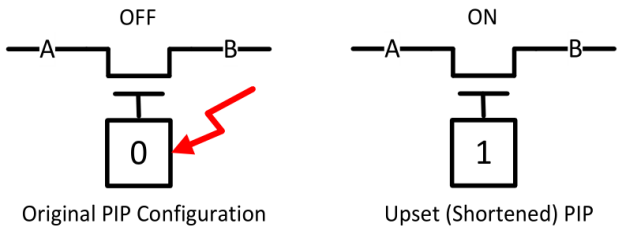


Figure 2.8: SEU on a PIP: an high impedance path is now connected

Multiplexers are one of the most widely used components in FPGAs. Similarly to PIPs, their select signal is driven by one or more configuration bits. The effect of an error on one of these elements could lead to an undefined behavior of the design: for example, an upset could change a MUX configuration in such a way that it selects now the input from an unused, unconnected component.

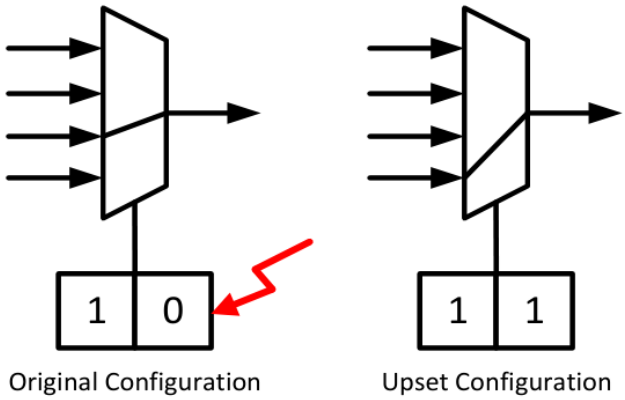


Figure 2.9: SEU on a MUX: another input is selected

Finally, buffers are similar to PIPs. Buffers are often used to propagate clocks or for I/O purposes. An error on one of these elements could cause a variety of different effects, ranging from the interruption of the clock distribution to more severe errors, like a wire driven by two buffers at the same time (short circuit).

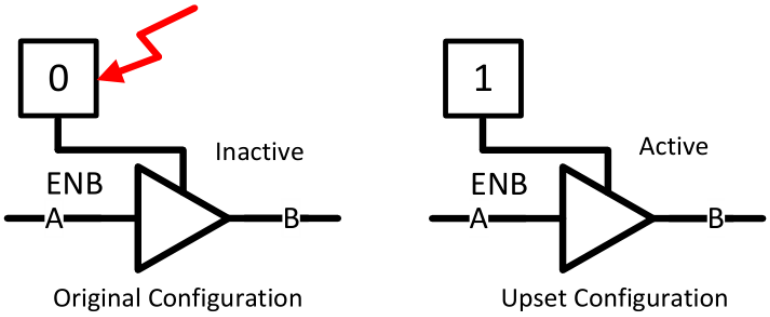


Figure 2.10: SEU on a Buffer: an high impedance output is now driven

Chapter 3

Available Techniques for Fault Tolerance

To overcome the problems caused by radiation effects on integrated circuits, there are multiple techniques that can be exploited.

Currently, there are many solutions applicable at different levels of abstraction from the silicon that can be identified in two major categories:

- *Hardened Technologies*;
- *Mitigation Techniques*.

The following sections briefly describe some of the system-level mitigation techniques that are applicable to a digital design in order to increase its reliability; those techniques are not mutually exclusive, meaning that they can be mixed in order to obtain better results.

3.1 Spatial Redundancy

Spatial Redundancy, also called *Hardware Redundancy*, encloses all the mitigation techniques that trade hardware area occupation to achieve better performances in terms of reliability against non-destructive *Single Event Effects* (SEE).

The concept that pools all the Hardware Redundancy techniques is based on:

- the replication of the same hardware block;
- the comparison/voting of the outputs.

The replication leads not only to increased area, but as well as power occupation and routing difficulty overheads.

It is important to notice that the weak element of this configuration is the comparator/voter placed at the end of the two hardware blocks: the presence of a fault in this component completely defeat the purpose of duplicating the design. With that said, in the vast majority of digital design, the cross section (i.e. the probability that a single event can manifest) of this component is several orders of magnitude less than the probability of the presence of a fault in the original hardware block.

Depending on the level of replication, these techniques may provide:

- *Error Detection*;
- *Error Detection* and *Error Correction*.

3.1.1 Duplex Architectures

Duplex architecture, also called *Duplicate With Comparison*, uses only two instances of the same hardware block to produce two outputs from the same inputs. Only one of the two blocks is actually used to provide to the environment the output value, the other one is just compared with the first one.

If, for instance, there is a mismatch between the two values, the comparator will notify it to the external world. At this point, being unable to recognize which is the correct output and which is not, the operation is usually retried.

The ability to detect the presence of a fault in one of the two duplicated hardware blocks is called *Error Detection*.

This approach is often not used due to the inability to self recover from the presence of an error: additional management logic must be added in order to properly handle the correction process.

3.1.2 Majority-Voting Architectures

Majority-Voting Architectures are identified by the presence of an odd number N of hardware blocks, being $N \geq 3$.

The underlying idea is to feed all the replicated blocks with the same input, then use a majority voter that decides the correct output based on the value to which most of the blocks agree.

This family of techniques provide the ability not only to detect the presence of an error (*Error Detection*) but also to automatically correct it and provide the expected value at the output (*Error Correction*).

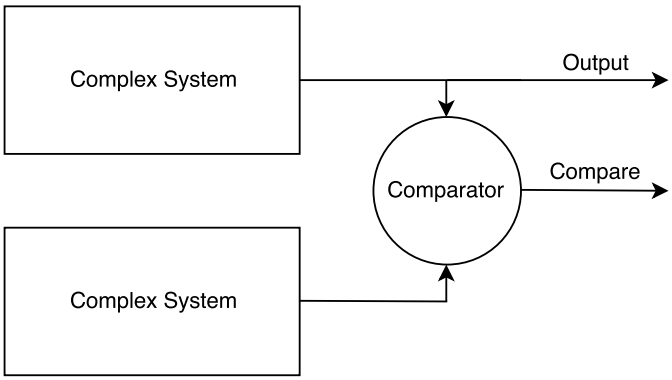


Figure 3.1: Duplicate with Comparison scheme: the systems are duplicated and their outputs compared for errors

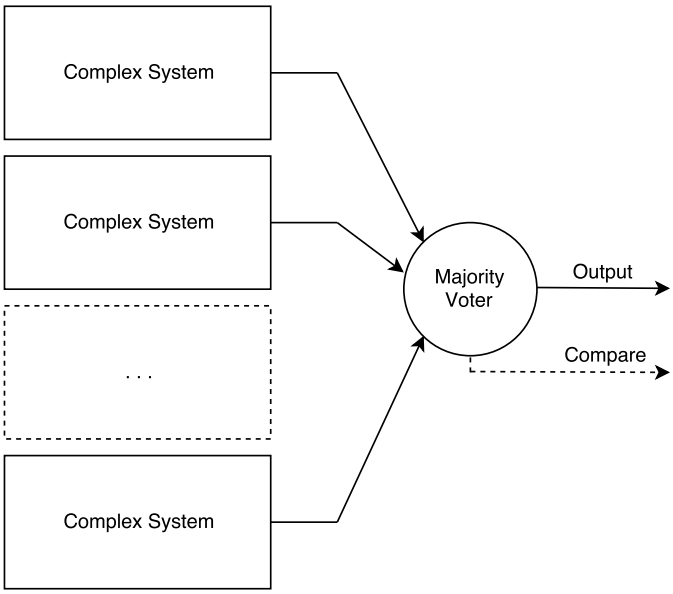


Figure 3.2: N-Modular Redundancy scheme: the systems are replicated N times, the voter decides the correct output using a majority voter scheme

Triple Modular Redundancy

A special set of these architectures is called *Triple Modular Redundancy* (TMR), where the number N of replicated hardware blocks is fixed to 3.

The error detection and correction capabilities, combined to the smallest area overhead among the majority-voting architectures, have made the TMR the most common technique used for design mitigation.

For the sake of simplicity, the following considerations are focused on this particular kind of architecture, but they can easily expanded to any value of N.

Block TMR (BTMR)

To introduce the first problem of this approach, that is in general the main issue of all the error correction techniques, let's first introduce the simpler version.

Its working principle is the simplest: the input data is triplicated and feeds all the blocks, than their outputs are voted by the majority voter.

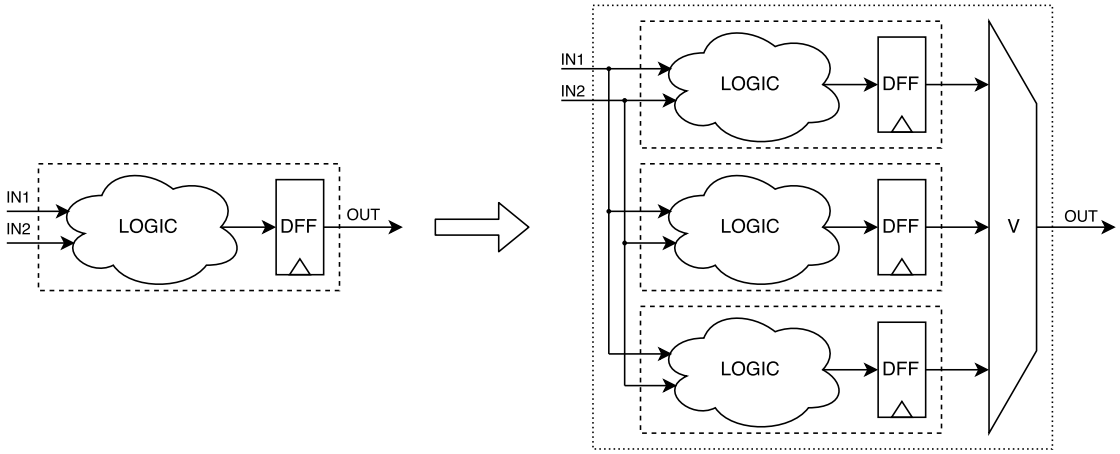


Figure 3.3: Block TMR scheme: a block is triplicated, including its memory elements, and then a voter decides the correct output using a majority voter scheme

The main problem of this scheme shows itself when the various replicas of the hardware block contain registers, a common scenario in all synchronous designs. The presence of a memory element in the replicated block imply an internal state; in case of an error, the internal state may drift from the correct state, leading to permanent errors at the output.

Having one block that always provide wrong results defeats completely the Error Correction capabilities of the system: in case another error occurs in one of the non-faulty blocks, the voter will not be able anymore to mask its presence.

A direct consequence can be highlighted by modeling the non-protected system and the one that encompasses Block TMR. The reliability of the non-protected system can be expressed as follow:

$$R(t) = e^{-\lambda t}$$

Where $R(t)$ is the Reliability function of time t, λ is the failure rate of the system.

Similarly, for the triplicated system:

$$R(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t}$$

The plot shown in Figure 3.4 highlights the problem: as the time passes, the probability to have a fault that alters the state of one memory element increase, thus reducing the reliability of the system. Block TMR is valid if the system is periodically reconfigured and reset, otherwise after a given amount of time $\tau(\lambda)$, the non-protected system will offer a greater reliability compared to the triplicated one.

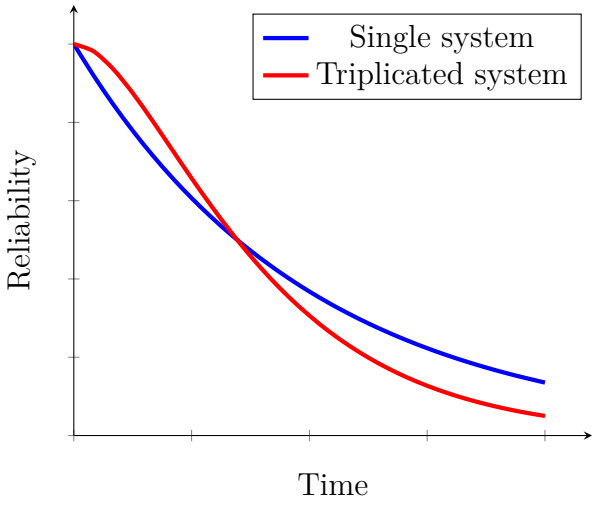


Figure 3.4: Reliablity comparison between a single system and a triplicated system with BTMR, the triplicated system is more likely to fail after a period of time

Distributed TMR (DTMR)

The solution to the problem presented above presents overhead in terms of area and routing difficulty.

Every flip-flop is triplicated, as well as the combinational logic; voters are added after every tuple of memory elements to vote and restore their state. With this method, the fault is masked internally, and the state of the hardware block is always restored the next clock cycle by the feedback network.

Distributed TMR always offers a greater reliability compared with the non-mitigated single block system.

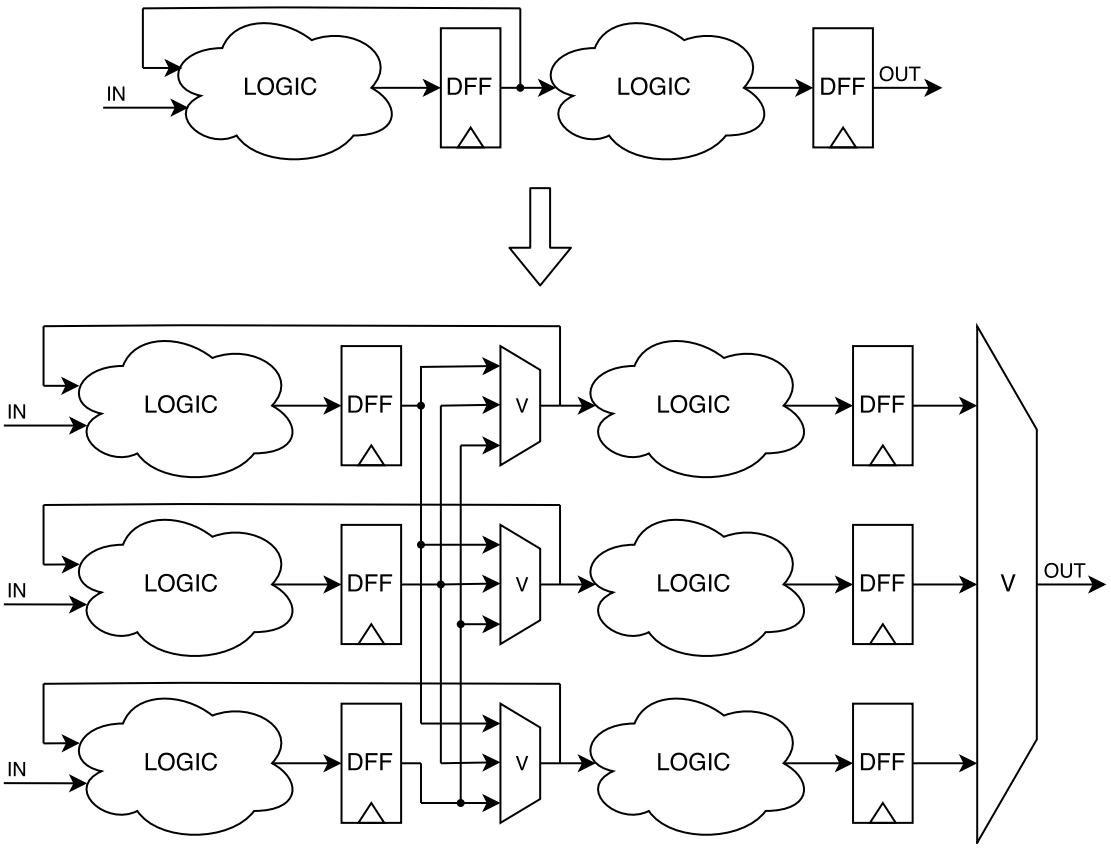


Figure 3.5: Distributed TMR scheme: the block is triplicated at each step, the inputs to the next step are always voted and the correct state is always restored from the voted output

3.2 Information Redundancy

While Information Redundancy techniques are often employed in data transmission and storage to improve the bit error rate, they have some applications in design mitigation. These techniques rely on the presence of additional, redundant, data bits that can be used to:

- identify the presence of an error in the non-redundant bits; (Error Detection)
- correct one or more errors (Error Correction).

The process that stands behind all the information redundancy techniques uses:

- an encoding function $F(D)$ that takes as input the original data D and returns the encoded value K;

Figure 3.6: Information Redundancy technique: a redundant part is added to the original data

- a decoding function $F^{-1}(K)$ that takes as input the encoded value K and returns the original data D.

The data is stored only in its encoded version K. The function $F(D)$ is tuned to maximize the possibility to identify an error in the unique data.

3.2.1 Parity Code

Parity code is the simplest among the possible information redundant techniques, as it adds only one bit of redundant data. This bit, called *parity bit* gets a different interpretation depending on the parity version used:

- Even Parity: the parity bit is asserted when the number of ones present in the data word, excluding the parity bit itself, is odd.
- Odd Parity: the parity bit is asserted when the number of ones present in the data word, excluding the parity bit itself, is even.

This technique is employed in many serial communication problems, and in general in applications where the probability of an error in the data transmitted or stored is negligible.

Moreover, the error detection capability of this technique is limited only to an *odd* number of errors: if, for instance, there are two errors in the same words that flip two bits, this technique will not detect any data corruption, since the parity bit value is consistent with the number of ones present in the word.

Cyclic Redundancy Check

Cyclic Redundancy Check (CRC) is the generalized version of Parity Code technique, where instead of only one redundant bit there are more. Like its simpler version, CRC is only able to provide error detection capabilities.

The technique define a simple hash function that is designed to maximize the error detection capabilities. Unlike Parity Code, using more than one redundant bit allows to detect different families of errors.

Cyclic codes are in general easy to implement, with a relative low hardware overhead, making them a preferable solution in many applications. However, the lack of error correction capability do not make CRC suitable for time-critical applications, where there is not the possibility of data recovery by retry.

3.2.2 Hamming Code

Hamming Code, also called Error Correction Code (ECC), was introduced by Richard Hamming in 1950. This redundant technique allows both error detection and error correction in the non-redundant bits.

As of today, Hamming Code refers to a specific (7,4) code that uses 3 redundant parity bits to encode 4 data bits in a word of 7 bits. In this particular configuration, often called *Hamming(7,4)-code*, the additional bits are capable of Single Error Correction (SEC).

While discussing the Hamming Code, it is important to introduce a new concept: the Hamming Distance. The Hamming Distance between two strings s_1 and s_2, of equal length, is defined as the number of positions at which the corresponding symbols are different.

In other words, the Hamming(7,4)-code is able to detect and correct errors up to an hamming distance of one.

Single Error Correction/Double Error Detection

Although Hamming Code is able to detect and correct single bit errors, the original implementation is not able to detect if more than one error is present in the original word. Note that if an error correction is tried on a word that presents two errors, the result of this operation is still an incorrect word. To overcome this problems, during the last years there were presented various extensions to the original Hamming Code that enable the Double Error Detection (DED) capability. The most common one adds a parity bit to the original (7,4) code to enable this feature.

This family of extensions to the original Hamming Code is called Single Error Correction/Double Error Detection (SECDED) and it is often employed in memory designs.

3.3 Temporal Redundancy

Temporal Redundancy techniques are based on the idea of sampling the result of an operation at different instants of time, and then proceed with a comparison between the sampled data. This category of techniques is more often employed in software application rather that in hardware, due to its area overhead required for the latter.

The software implementation is quite straightforward: the same operation is repeated multiple times, storing the results in different memory locations. When the needed number of operation is reached, a comparison is performed to detect and even correct the presence of an error.

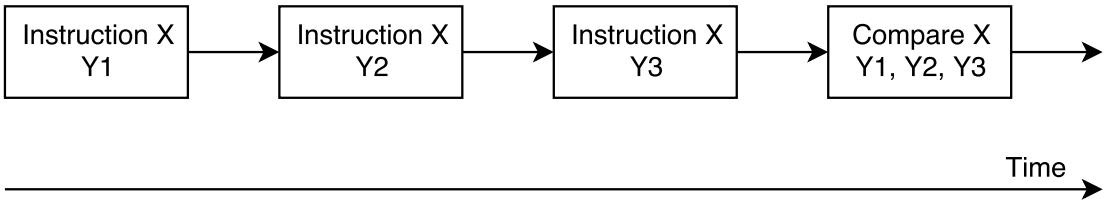


Figure 3.7: Time Redundancy in Software: the same operation is repeated multiple times and then the result is compared and voted

In the case of hardware implementation, the approach is slightly different and requires the data to be stable at the input of the circuit for a period that is N times longer the original one, being N the number of comparisons. Depending on the complexity of the function that has to be replicated, there are two possible implementations.

- An additional circuit is added to replicate the same operation N times, reusing the same hardware for consecutive cycles; this is usually suited for complex, area-expensive, functions that are difficult to replicate.
- The entire hardware is replicated N times and the clock is delayed properly, such that the N independent hardware blocks are queried at different time instants.

Temporal Redundancy techniques are often not employed due to the high computational time and area overhead.

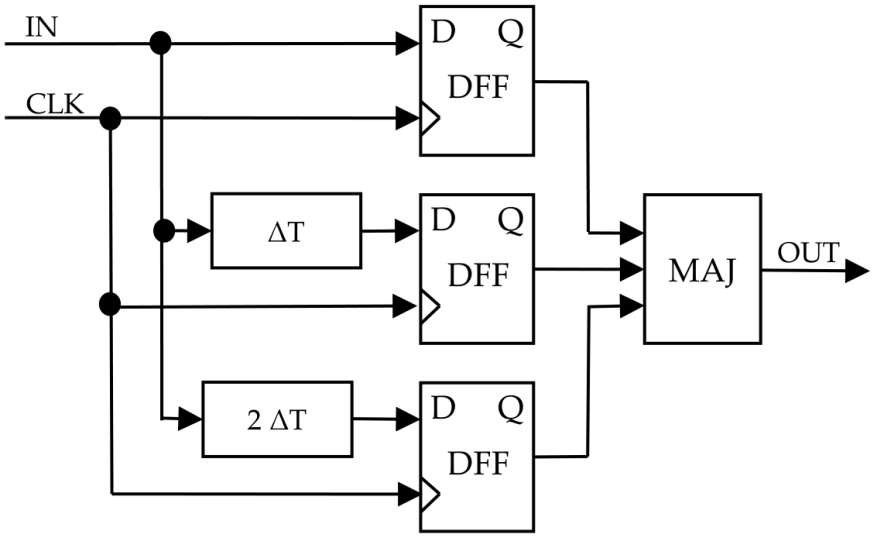


Figure 3.8: Time Redundancy in Hardware: delays are added to repeat the same operation at different time instants

Chapter 4

Metrics for Fault Tolerance

A fundamental task when working on fault tolerance is the definition of the so-called *metrics*: a standard of measurement that can provide informations on how well the system is performing.

Before defining what are the metrics for fault tolerance it is necessary to describe what is the *mission* of a product, that is, in short, its purpose. The mission can be characterized by:

- a *function*, that is what is the system expected to produce;
- a *duration*, that is the amount of time during which the system should perform its task.

In the following sections are presented some of the main metrics that are available to classify systems and provide standardized benchmarks.

4.1 Dependability

Dependability is one of the key parameters used to assess the quality of a product. Dependability is the property that characterize a *dependable* system, and it is defined as:

> *"The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers."* [9]

This property is used in many different fields, and it can be defined using three different class of parameters:

- *Attributes* – to assess the dependability of a system;
- *Threats* – to affect the dependability of a system;
- *Means* – to increase the dependability of a system.

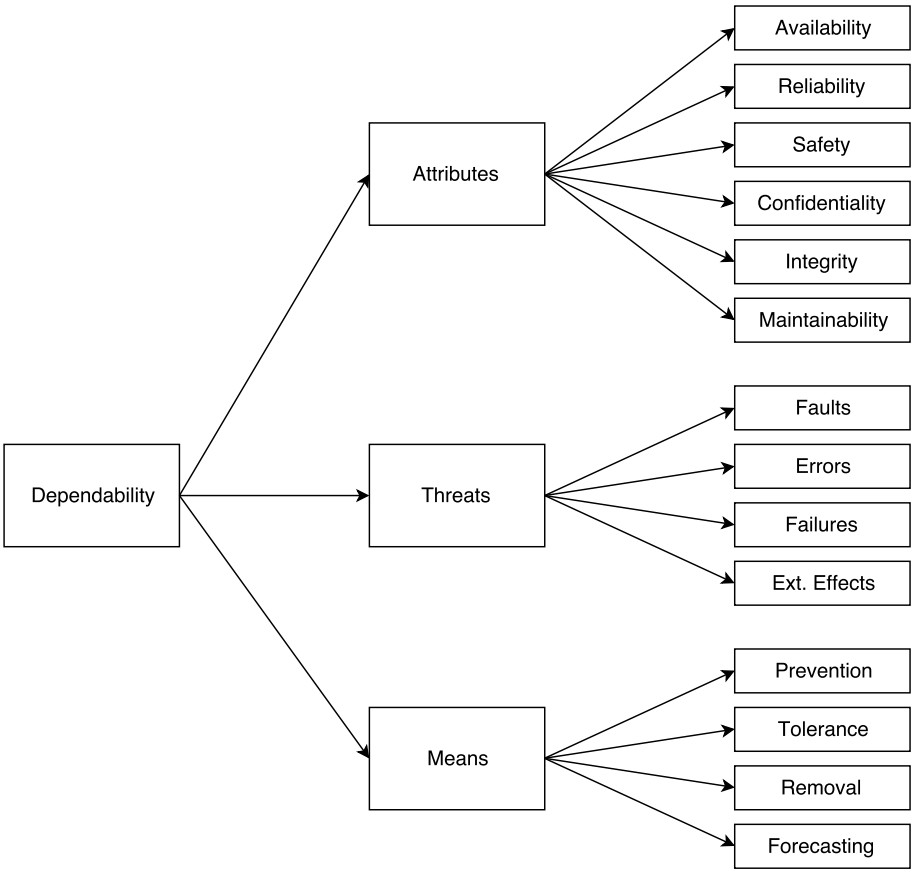


Figure 4.1: Dependability graph: the internal structure of dependability, divided by class

4.1.1 Dependability Attributes

Attributes are used to asses the dependability of a system using a scientific, analytical, repeatable approach. Some of the main attributes include:

- Reliability;
- Maintainability;

- Availability;
- Safety.

Reliability, MTTF, Failure Rate, FIT

Reliability is the attitude of an object to behave as expected, in defined conditions, for a certain amount of time. It is defined as the probability that a system will satisfactorily perform its intended function under given circumstances for a specified period of time.

The *reliability function* $R(t)$ is defined as the conditional probability that a system is in operational conditions after the time instant t.

$$R(t) = P_{\text{working}}(\tau > t) = \int_t^{\infty} f(x)\ dx \tag{4.1}$$

Where $P_{\text{working}}(t)$ represents the probability of being in a working state at time t, τ is a random variable and $f(t)$ represents the *failure probability density function*.

Another unit of measure to quantify the reliability of a system is defined by the *Mean Time to Failure (MTTF)*. This quantity represents the average time before a failure occurs in the system.

$$\text{MTTF} = E[\tau] = \int_0^{\infty} t \cdot f(t)\ dt \tag{4.2}$$

Where $E[\tau]$ is the expected value of the random variable τ, defined in Equation 4.1.

Finally, *failure rate* (also called *hazard rate*) is defined as the number of failures over a period of time.

$$\lambda = \frac{\#\text{FAILURES}}{\Delta t} \tag{4.3}$$

Where λ is the failure rate, Δt is the period of time considered.

Failure rate is usually expressed in terms of *Failure in Time (FIT)*, that represents the number of failures over a period $\Delta t = 10^9$ h.

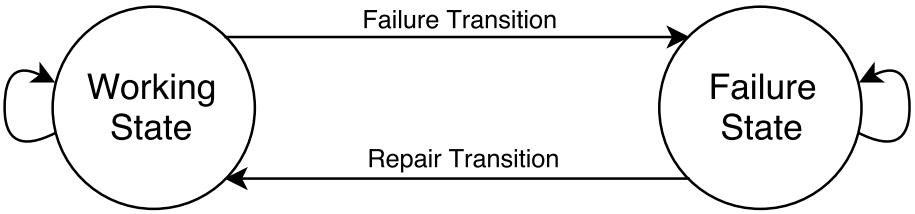


Figure 4.2: The product life cycle of a repairable system: it transitions from working to failure state and vice versa using failure and repair transitions

Repairable Systems, MTBF and MTTFF

The following attributes can be defined when dealing with a subset of all the possible systems, called *repairable systems*. A repairable system is characterized by the ability of being *repairable*, and its life cycle can be modeled as a diagram with two states: *working state* and *failure state*.

The transitions between these two states are regulated by the alternation of two processes: *repair-to-failure* and *failure-to-repair*. The former is regulated by the random variable τ (defined in Equations 4.1 and 4.2), that represents the *time-to-failure* of the system; the latter is instead related to another random variable, θ, that represents the *time-to-repair*.

Similarly to *non-repairable* systems, can be characterized using a quantity similar to MTTF, the Mean Time Between Failures (MTBF): it represents the average amount of time between a failure and the consequent one.

This capability could have consequences depending on the truthfulness of the assumption *"system good as new after repair"*. If the assumption is not considered true, another parameter has to be accounted: *Mean Time to First Failure (MTTFF)*. From now on, the aforementioned assumption will be considered true, therefore the following condition holds.

$$\text{MTTF} = \text{MTBF} = \text{MTTFF} \tag{4.4}$$

Maintainability and MTTR

Another useful attribute to characterize dependability is represented by the *Maintainability*. This quantity represents the probability that a *reparable* system can be repaired in a defined environment within a specified amount of time.

$$M(t) = P_{\text{repaired}}(\theta < t) \tag{4.5}$$

Where $M(t)$ represents the maintainability as a function of time t, θ is a random variable representing the time to repair.

Similarly to reliability, this quantity can also expressed in terms of *Mean Time to Repair (MTTR)*: it represents the average time required to repair a system.

$$\text{MTTR} = E[\theta] = \int_0^\infty t \cdot m(t)\ dt \tag{4.6}$$

Where $E[\theta]$ is the expected value of the random variable θ, $m(t)$ is the *repairability probability density function* ($m(t) = \frac{dM(t)}{dt}$).

Another useful parameter used is represented by the *repair rate*, defined as the number of repairs over a period of time.

$$\mu = \frac{\#\text{REPAIRS}}{\Delta t} \tag{4.7}$$

Availability

Availability represents the ability for a repairable system to be operational at a generic instant of time. It differs from reliability since it does not refer to a period of time, but rather to a single instant of time.

$$A(t) = P_{\text{working}}(t) \tag{4.8}$$

It is important to notice that the availability is independent on the failure-repair cycles already occurred, meaning that this attribute reflects also what is the behavior of the system with respect of its repairability.

To better define it, it is necessary to review the two state process that models a repairable system. This process can in fact be modeled by a Markov chain, where the probability of moving from the Available State to the Unavailable State is related to the failure rate λ, and the opposite is related to the repair rate μ.

The state change is modeled with the following formula:

$$\mathbf{p}'(t) = \mathbf{p}(t) \cdot \mathbf{Q} \tag{4.9}$$

Where $\mathbf{p}'$ and $\mathbf{p}$ represent respectively the current and the next state of the Markov chain and $\mathbf{Q}$ is the *transition matrix*.

$$\mathbf{Q} = \begin{bmatrix} -\lambda & \lambda \\ \mu & -\mu \end{bmatrix} \tag{4.10}$$

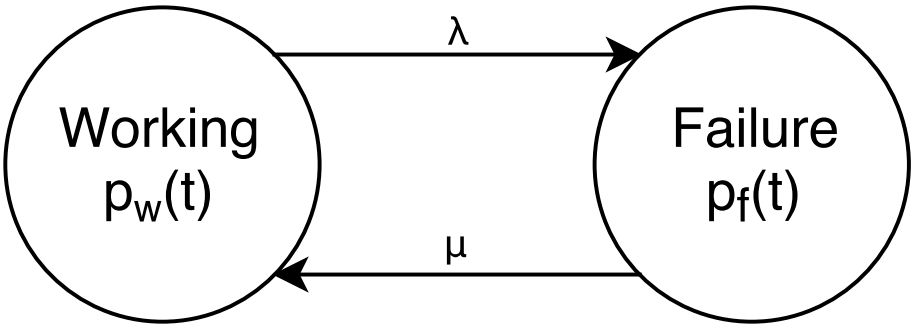


Figure 4.3: Markov chain representation of a repairable system: the transition probabilities are defined by the failure rate (λ) and the repair rate (μ)

Substituting the value of $\mathbf{Q}$ in the Equation 4.9 leads to the following system of equations.

$$\begin{cases} \frac{dp_{\mathrm{w}}(t)}{dt} = -\lambda \cdot p_{\mathrm{w}}(t) + \mu \cdot p_{\mathrm{f}}(t) \\ \frac{dp_{\mathrm{f}}(t)}{dt} = \lambda \cdot p_{\mathrm{w}}(t) - \mu \cdot p_{\mathrm{f}}(t) \end{cases} \tag{4.11}$$

Where $p_{\mathrm{w}}(t) = A(t)$ and $p_{\mathrm{f}}(t) = 1 - A(t) = U(t)$, also called *Unavailability*. The initial conditions for the above system of equations assume that the Markov chain starts from the working state, so $p_{\mathrm{w}}(0) = 1$ and $p_{\mathrm{f}}(0) = 0$.

By solving the system of Equation 4.11 allow to express the Availability, $A(t)$, and the Unavailability, $U(t)$, as function of the failure and repair rate.

$$A(t) = p_{\mathrm{w}}(t) = \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} \cdot e^{-(\lambda+\mu)t} = A_{\infty} + A_{\mathrm{trans}} \tag{4.12}$$

$$U(t) = p_{\mathrm{f}}(t) = \frac{\lambda}{\lambda + \mu} - \frac{\lambda}{\lambda + \mu} \cdot e^{-(\lambda+\mu)t} = U_{\infty} - A_{\mathrm{trans}} \tag{4.13}$$

These two equations are characterized by a constant term, often called *steady-state* term, and a transient one, that is multiplied by an exponential. A common condition for a repairable system is that the time required to repair it is negligible compared to the time required to experience a failure.

$$\mathrm{MTTF} >> \mathrm{MTTR} \implies \lambda << \mu \tag{4.14}$$

For this reason, the Equations 4.12 and 4.13 can be simplified with the following.

$$A(t) = A_{\infty} = \frac{\mu}{\lambda + \mu} = \frac{\mathrm{MTTF}}{\mathrm{MTTF} + \mathrm{MTTR}} \tag{4.15}$$

$$U(t) = U_\infty = \frac{\lambda}{\lambda + \mu} = \frac{\text{MTTR}}{\text{MTTF} + \text{MTTR}} \tag{4.16}$$

Finally, availability is often also expressed in terms of ratio between the *uptime* and the total time elapsed.

$$A(t) = \frac{\text{UPTIME}}{\text{UPTIME} + \text{DOWNTIME}} \tag{4.17}$$

4.1.2 Dependability Threats

Threats are phenomenas that can affect the mission of a system by interfering with its components. First of all, it is important to define the possible outcome of a threat present in a digital circuit. In fact, it can manifest itself differently depending on its type, on the structure of the circuit and on the mission accomplished by the application. The following list provides the four possible outcomes that can be related to the presence of a threat.

- Fault;
- Error;
- Misbehavior;
- External Effect.

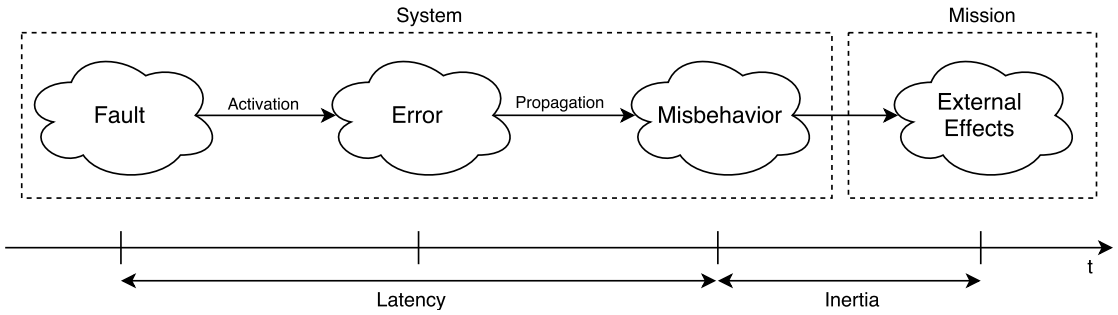


Figure 4.4: Fault life cycle: the fault is activated into an error, the error is propagated into a misbehavior, depending on the type of the misbehavior the external effects can be different

Faults

A fault is simply the presence of a defect in a system. In Chapter 2, for instance, are discussed the possible outcomes of Single Event Effects on FPGAs: in this case, the presence of a bit in the Configuration RAM whose value is flipped with respect to its nominal condition can be considered a fault in the system.

Attention has to be paid to the fact that a fault may or may not be *activated*, so in some cases its presence can be masked (for instance, the CRAM bit flipped could be unused) and an error is never triggered. Finally, faults are in general not observable without using expensive techniques.

Errors

Errors represent an *internal* discrepancy between the expected behavior and the actual one. Its presence is dictated by the activation of the fault present in the circuit. An example of an error could be an internal component whose state has drifted away from the correct one. Errors can be observed using specialized mechanisms, like an hardware debugger.

Similarly to what has been said for faults, errors may or may not be *propagated* into an actual misbehavior.

Misbehaviors

A misbehavior, also called *failure*, represent an *external* discrepancy between the expected behavior and the actual one. To have a misbehavior in a system most of the time imply a failure in its mission.

External Effects

The external effects are cause by the presence of a failure in a system. Depending on its the severity, the impact on the service delivered can be different: for instance, having an error at the output that is not distinguishable from a good one can severely impact on the mission, while a detectable misbehavior can be identified and corrected, therefore reducing its effect.

Latency and Inertia

When discussing about dependability threats, a special analysis has to be performed on two timing parameters: *latency* and *inertia*.

Latency of a fault is defined as the amount of time between its occurrence and its manifestation as a misbehavior on the system. This quantity can be influenced by many different factors:

- The *utilization frequency* of the component affected: an highly utilized one have in general a lower latency.
- The *time of occurrence*: a fault that occurs during an active time of a component has higher chances of being propagated.
- The *observation level*: depending on how the component is observed, the fault propagation may be delayed.

In general, latency should be as small as possible: higher values of latency can lead to the accumulation of many unspotted faults in the system.

On the other hand, inertia represents the quantity of time that elapses from the manifestation of a failure and the beginning of its consequences on the mission. High values for inertia are preferred, since the large time window gives more time for the correction of the system.

4.1.3 Dependability Means

The techniques adopted to increase the dependability of a system are called *means*. The following four techniques are complementary:

- *Fault Prevention*, that defines techniques adopted to prevent faults from occurring;
- *Fault Removal*, that defines techniques used to remove a fault from the system;
- *Fault Tolerance*, that defines techniques utilized to deal and mask the presence of a fault (discussed in Chapter 3);
- *Fault Forecasting*, that defines techniques to forecast the presence of faults, so that their effects can be circumvented.

4.2 Fault Classification

To complete the characterization analysis of a system, faults have to be classified using various parameters:

- *Type*: that identify the class the fault belongs to, for example a fault that is changing the value of a memory location is called a *memory fault*, while a fault that modify the logic function of a block is called *logic fault*.

- *Locality*: that is the location in which the fault is placed. Faults in critical components can impact severely the mission of the system.

- *Latency*: the interval of time from its occurrence and its manifestation as a misbehavior, as discussed in Section 4.1.2.

- *Frequency*: that represents the average time of occurrence of the same fault.

- *Severity*: that is the magnitude of the fault's effect on the system's mission. This parameter is strongly dependent on the fault type and the fault locality.

Depending on the severity level, a fault can be classified as:

- *Critical fault*: that represents a fault that prevents the system's mission to be carried out until the repair is completed. The frequency of these category of faults have to be very low or non-existent.

- *Major fault*: this type of fault is very similar to a critical fault with the difference that a temporary workaround can be applied in order to avoid strong consequences on the mission. Major faults can manifest themselves with a slightly higher frequency compared to the critical ones.

- *Minor fault*: this category includes all the faults that have few secondary effects on the system's behavior, that usually don't affect the mission carried.

4.3 Metrics for Single Event Effects on FPGAs

As discussed in Chapter 2 (Section 2.4), Single Event Effects are a common denominator in FPGAs that have to work in radiation environments. There is therefore the need to be able to classify the sensitivity of the device against SEEs.

These properties are strongly dependent on technology parameters used to produce the integrated circuit on which the Configuration RAM is implemented. In the following sections some of the main quantities to consider are explained and discussed.

4.3.1 Cross Section

To characterize the immunity of a digital circuit against SEUs, it is necessary define the *Cross Section*.

The cross section represents the probability of having a SEE on an integrated circuit, and it is proportional on the area occupied. This quantity is experimentally measured by counting the number of events per unit fluence. The cross section is highly affected by:

- the particle type;
- the Linear Energy Transfer of the particles;
- the angle of incidence of the beam.

More on these parameters in Chapter 2.

Cross section is generally a function of LET for fluxes that are composed mainly of ions, while for for protons and neutrons it is usually expressed as a function of energy.

$$\sigma_{\text{ion}}(\text{LET}) = \frac{\#\text{EVENTS}}{\Phi_{\text{ion}}} \tag{4.18}$$

$$\sigma_{\text{n,p}}(\text{E}) = \frac{\#\text{EVENTS}}{\Phi_{\text{n,p}}} \tag{4.19}$$

Where σ is the cross section, Φ is the fluence.

When this quantity is referring to Single Event Effects in general, it is expressed in terms of $\frac{\text{cm}^2}{\text{device}}$. In the special case of Single Event Upsets, however, is expressed in terms of $\frac{\text{cm}^2}{\text{bit}}$ and usually denoted as σ_{bit}.

4.3.2 Measurement of SEE Sensitivity

To identify the so-called *SEE Sensitivity* of a device, one or more of them have to be placed under beam, while keeping the other operational conditions to normal. The test performed consists in an irradiation of these devices in such a way that the number of events can be counted. Since all the flux parameters are actors during these tests, their values are recorded as well.

A raw indication of the cross section could be the following:

$$\sigma = \frac{N_{\text{avg}}}{\Phi \cdot cos(\theta)} \tag{4.20}$$

Where N_{avg} is the average number of events per device, Φ is the fluence and θ is the incidence angle (0° if the beam is perpendicular to the device).

Influencing Factors

The Equation 4.20 takes into account only few affecting factors, the cross section, actually, is influenced on many more parameters:

- Particle Energy;
- Angle of Incidence (θ);
- Temperature;
- Total Ionizing Dose;
- Operational Mode;
- Stored Data Pattern;
- Clock Frequency;
- Static or Dynamic Test;
- Electrical Bias applied to the device;
- Current-limiting conditions;
- Reset conditions;
- Device Portion tested.

4.3.3 SEU Sensitivity on FPGAs

As discussed in previous sections, the cross section in an indication of the sensitivity of integrated circuit to radiation effects. As stated in Section 4.3.1, a cross section per bit (σ_{bit}) is used for Single Event Upsets. Starting from here, it is possible to evaluate the cross section of a device by simply multiply its value by the number of memory bits present:

$$\sigma_{\text{device}} = \sigma_{\text{bit}} \cdot N_{\text{bits}} \tag{4.21}$$

From this calculation, it is possible to evaluate the number of Single Event Upsets as function of the fluence.

$$U = \Phi \cdot \sigma_{\text{device}} \tag{4.22}$$

Where Φ is the integrated flux: fluence, defined in Equation 2.3.

For other purposes, it is also convenient to calculate the upset rate of the circuit, as a function of the average flux, ϕ_{avg}:

$$F_{\text{U}} = \phi_{\text{avg}} \cdot \sigma_{\text{device}} \tag{4.23}$$

The upset rate is useful to calculate the various requirements in terms of correction rate.

Chapter 5

Radiation Hardness Design Validation

Once the design part is completed and all the mitigation techniques have been implemented in the design, it is important to validate it to ensure a correct behavior in radioactive environments. There are multiple techniques to simulate the effect of a particle beam that hits a FPGA.

5.1 Fault Injection

Fault Injection (also called *Tabletop Testing*) is the simplest, yet incomplete, method to estimate the reliability of the design under any given beam.

Fault Injection is a technique that uses internal and/or external peripherals to inject errors in the CRAM of the FPGA. The procedure of injecting an error do accurately simulate the effect of a Single Event Upset caused by a particle hitting the silicon integrated circuit.

FPGA Configuration RAM Structure

Before discussing what is the mechanism behind the process of fault injection it is necessary to explain the internal structure of the *Configuration RAM* present in FPGAs.

The CRAM is organized as an array of *frames*, similarly to a wide Static RAM. Each frame is subdivided into *words*, that are usually 32bit each. Each bit present into these words represent a specific configuration bit used to configure the various parts present in the FPGA.

Due to the high density of these memories, information redundancy techniques are often employed in order to reduce the probability of errors in the configuration. Almost all the devices have a CRC protection, some others instead, have a much more effective ECC protection.

Some FPGA vendors, on top of this information redundancy techniques, often organize the memory cells so that those of the same word are not physically adjacent. In the example of Xilinx's Ultrascale Field-Programmable Gate Arrays the word bits for CRAM are interleaved by one bit of other words.

5.1.1 Fault Injection Procedure

The Fault Injection procedure works as follows:

1. A *configuration frame* is read from the FPGA's Configuration RAM;
2. Within the configuration frame, one or more bits are flipped using the xor logic function, therefore making it a faulty frame.
3. The faulty frame is wrote back to the Configuration RAM, replacing the original one.

When performing fault injection, it is usual to randomize the frames, words, and bits in order to simulate better the effect of a particles hitting the FPGA without a defined pattern, that is close to what happens if the device is put under a particle beam.

With this technique, however, it is possible to force an ECC error by selecting properly two bits of the same word to simulate a worst case scenario where the bits are not correctable automatically.

5.1.2 Limitations of Fault Injection

Although Fault Injection is a valid method to estimate the reliability of a design, it is not able to simulate all the effects that a striking particle could cause; the following lines provide a brief description of the benefits and limitations of this technique.

First of all, it is important to mention that some configuration frames present in the Configuration RAM are write-protected until a complete reprogram of the memory contents. These locations usually holds the values of the internal memory elements, such as registers, Distributed RAM and Block RAM. This is a big limitation of this method: to simulate the presence of an error in these locations, it is necessary to reprogram completely the FPGA.

With that said, taking into account the fact that these memory elements represent a small percentage of the total configuration RAM size, this technique is able to predict the behavior under beam with a sufficient enough confidence level.

5.1.3 The Xilinx Soft Error Mitigation IP

The Xilinx Soft Error Mitigation (SEM) IP is a solution provided by Xilinx to detect, correct and inject faults on Ultrascale FPGAs. This patent does not prevent the arise of soft errors, but rather it provides a method to better manage them at system level. [13]

This Intellectual Property is a valid example of what is called an *internal* scrubber: it is, in fact, configured as a peripheral on the FPGA, and it utilizes a dedicated interface to address directly the Configuration RAM of the device on which it is configured.

Error Classification on SEM IP

The SEM IP, among all its features, have the possibility of classify the faults that are present on the CRAM of the device. This is a proprietary technology of Xilinx, called Xilinx Essential Bits Technology, that uses an algorithm to identify which are the *essential* bits for a design. Essential bits are, in short, a subset of all the configuration bits available: they are essential in the sense that a changing a value of these bits changes the function implemented by the design. [5]

Xilinx also defines the so-called *prioritized* essential bits, a subset of the essential bits that are weighted by metrics defined by the user. An example of this could be the configuration bits of a device with an high utilization rate. On top of that, there are the *critical* bits: those are bits whose change is likely to kill the entire design, like the configuration bits for the clock distribution. [5]

The Soft Error Mitigation supports error classification using an external Read-Only Memory (ROM), interfaced using the built-in SPI interface, that contains a list of only the essential bits for the design. In this way, in case of an *uncorrectable error*, it is possible to identify if this error involves essential bits or not, and act consequently.

Features and Capabilities

This Intellectual Property can be generated in various different modes, depending on the requirements:

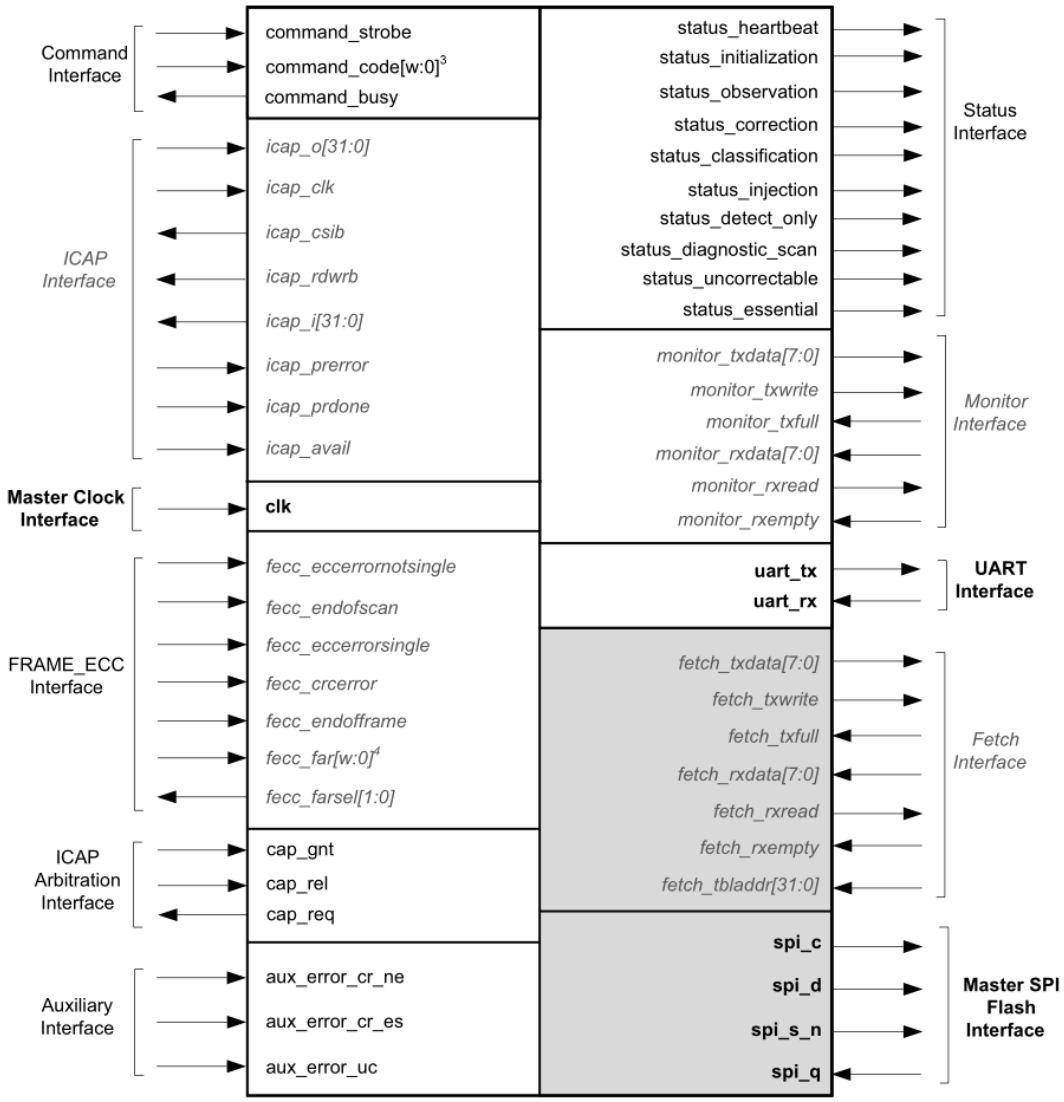


Figure 5.1: SEM IP block description: all the input and outputs ports are listed

- Mitigation and Testing;
- Mitigation only;
- Detect and Testing;
- Detect only;
- Emulation;

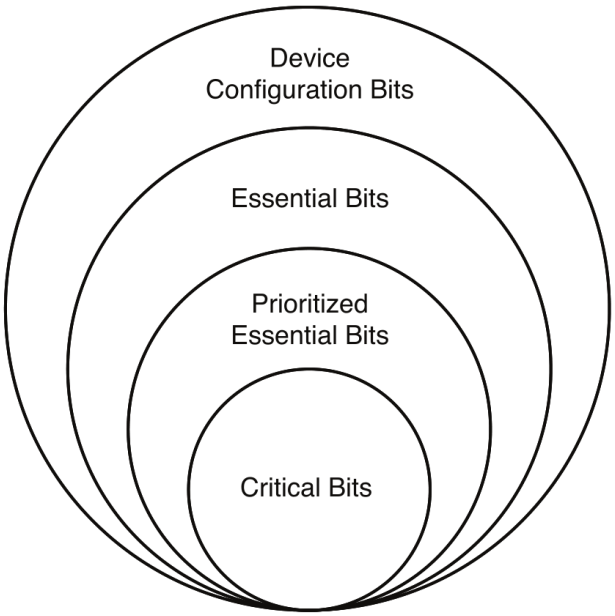


Figure 5.2: Xilinx Essential Bits: configuration bits can be classified based on their priority levels

- Monitoring only.

The mitigation modes enable error detection, correction and classification. In the case of Mitigation and Testing, it is also possible to perform error injection on almost all the CRAM configuration bits. The detect modes are similar to mitigation, with the only difference that error correction is disabled. Finally, emulation mode is useful to evaluate the effects of SEUs on the system, in this case only error injection is possible.

This peripheral have a command interface to receive commands from other components present on the FPGA. A common solution is to provide an UART interface to the external world, so that it is possible to monitor the current status of the device with a dedicated interface.

Working Principle

Assuming to have generated the SEM IP with the most feature-rich mode, Mitigation and Testing, it is possible to:

- detect errors;
- correct errors;

- inject errors.

All of these operation are possible thanks to a dedicated interface to the Configuration RAM, the Internal Configuration Access Port (ICAP). This interface enables a direct, fast communication from FPGA to the configuration memory. For this reason the SEM IP can detect errors present with a latency between 22ms and 58ms, depending on the size of this memory. [13]

To ensure error detection and correction capabilities, the addressable memory location are continuously read as fast as possible. In case one frame presents a CRC error, correction has to be performed: in the case of a single bit error with ECC enabled, the correction is automatic, and the IP only takes care of rewriting the correct value in the corrupted memory location. If, instead, a multiple bit error is present, the Hamming codes implemented are not able to correct its value: in this case it is necessary to classify the bits affected using the error classification capability, if enabled. If one or more bits are classified as essential, or if the classification is disabled, the IP will show an *uncorrectable error* message that states that recovery is impossible: in this case it is necessary to reprogram either the configuration frame or, directly, the entire device, in order to restore the original configuration.

Fault injection, on the other hand, can be performed using two different type of addressing:

- *Linear Frame Address (LFA)*;
- *Physical Frame Address (PFA).*

The difference between the two is that the former have the property of being linear, from 0 to a maximum value that depends on the size of the FPGA CRAM. The latter instead is closer to the actual cell placement: in fact, internally, LFAs are translated into PFAs. Removing this level of abstraction create intrinsic "holes" in the address space, trying to inject an error to these locations is simply discarded, and no action is taken.

5.1.4 The JTAG Configuration Manager

The *JTAG Configuration Manager (JCM)* is a custom platform that features an application processor connected to FPGA fabric, developed at Brigham Young University (BYU). It is composed of two parts, a Linux software library that runs on the embedded ARM core present, and a custom JTAG controller that is implemented as a custom IP on its FPGA. With this platform it is possible, by connecting to the JTAG ports of a Xilinx FPGA, to implement the *blind scrubbing* and the *fault injection.* [14].

Blind Scrubbing Procedure

Blind scrubbing refers to the operation of continuously rewriting the configuration frames present on a FPGA in order to correct the eventually present errors. It is the simplest, yet the most effective method to increase the reliability of those devices, especially when an high upset rate is expected from the field application.

This technique is often employed when external devices are used, like in the case of the JCM, but the same functionality can be accomplished if a custom peripheral is designed and implemented directly on the target FPGA.

First of all, the *golden* configuration bitfile is uploaded to a dedicated memory used by the blind scrubber either directly or indirecly, by reading back the contents of the freshly programmed CRAM (this procedure is called *readback*). Once the golden bitfile is loaded into the memory the blind scrubbing can be started: it consists of an infinite loop that addresses all the words available on the target FPGA. The process is repeated until there is the need of resetting the target device: in this case, the process is stopped and the device have to be reconfigured completely.

The blindness of this process, however, have its own disadvantages. The first one is that an external golden memory has to be provided, and an error on this memory is likely to cause catastrophic effects on the device that it is supposed to protect. A second disadvantage is given by the fact that, independently of the correctness of the data stored in memory location, it has to be overwritten at every scrub cycle: this has strong consequences on the fault correction latency that are strongly dependent on the speed of the interface used and on the size of the configuration memory.

Fault Injection on JCM

Using an external device that manages the fault injection on the configuration bits presents various advantages with respect to an internal peripheral.

First of all, being the internal peripheral able to faults on virtually the whole device, while performing sessions of random fault injection there is the possibility that the error injected can break the peripheral itself, forcing to reconfigure the FPGA to regain the control. This problem does not subsist in the case of external platforms like the JCM, the advantage, then, is that it is possible to inject an arbitrary number of faults without having to continuously check the working status of the fault injector, therefore easing the process.

Secondly, the external platform does not consume resources on the FPGA: implementing an internal scrubber requires the utilization of internal resources like LUTs, D Flip-Flops, Block RAMs and Distributed RAMs. This point is also important since additional utilization of resources could make a difference in terms of routing difficulty, that in the specific case of fault injection can alter the results.

This approach have also drawbacks: similarly to what has been discussed for the blind scrubbing, the interface speed can be a problem, even though usual rates for fault injection are slow enough not to notice any difference using the JTAG interface.

5.2 Ground Testing

After a longer session of fault injection, usually the designs are placed under a particle beam to evaluate their performances in real life scenarios.

For this type of test, called *ground testing*, can give accurate results if the beam to which the FPGA is exposed to is equivalent to the final operational conditions in which the system have to work. As stated in Chapter 2, different particles have a different interaction with the silicon integrated circuit of the device: for example, neutrons are more likely to cause multiple upsets with a strike of a single particle.

5.2.1 Testing Methodology

To have a comprehensive overview of the performances of a design on a FPGA, however, multiple beam tests take place, with different fluxes that are usually orders of magnitude higher than the operating ones. The reason for this choice is that a reasonable higher flux is increasing the statistics in terms of upset rate: a device that is supposed to work for one day under a proton particle flux of 10^3 1/s can be simulated for 1000 days if it is put under a beam of 10^6 1/s of the same particles.

During these tests the status of the *Device Under Tests* is constantly monitored. The first scope of interest is represented by the functional behavior of the board, to allow the retrieval of the data some test points are placed in the design in order to verify its correct behavior over time. The second scope of interest is instead represented by physical parameters concerning the electrical conditions and temperature of the DUT: the reason for this choice is to detect and prevent destructive effects like Single Event Latchup before they can damage permanently the integrated silicon.

To have an accurate prediction and statistic of the behavior of the system under test, it is usual to put more than one prototype of the same final product under particle flux. In this way, it is in fact possible to "multiply" the number of hours of beam time by the number of prototypes that are employed for the test.

5.2.2 Radiation Decay

After the radiation campaign, the prototypes that have been exposed to the particle flux have to be stored in a controlled room where the radioactive elements can

complete their radioactive decay process. During this period only a small group of specialized people can access to that room, for this reason it is usually necessary to wait until the radioactive decay can be considered complete. An usual period for this process is around two weeks (14 days) after which it is possible to retrieve the prototype(s).

5.2.3 Tests after Retrieval

After the retrieval, more intensive tests are performed using dedicated instruments to verify that all the prototypes are still fully functional in all of their parts. For complex boards, it is necessary to verify everything starting form the power distribution to the functional behavior of all the components placed onto them.

Once all the tests are completed, assuming that no component showed any functional failure after the beam test, it is possible to ooze the results of the radiation campaign. After that, in case the results are not filling the requirements in terms of reliability and/or availability, it is necessary to review some steps in the design in order to increase these statistics.

Chapter 6

Characterization of the Xilinx Triple Modular Redundancy Subsystem

The main matter of this thesis is represented by the characterization of the Xilinx Triple Modular Redundancy (TMR) Subsystem, an Intellectual Property developed by Xilinx that is made to increase the dependability of their soft microprocessor core, the Microblaze. In the following sections are presented the main structure of the TMR Subsystem, the testing procedure and the testing architecture employed to characterize and compare the performances against radiation effects.

6.1 Xilinx Microblaze and TMR Subsystem

As stated in the introduction, Xilinx has developed over the years a soft microprocessor core called *Microblaze*. It is a *Reduced Instruction Set Computer (RISC)* optimized for implementation on Xilinx FPGAs, and has the property of being highly customizable using generation scripts. [15]

For the sake of this thesis, the base core has been configured with the most conservative settings in terms of area:

- 40 MHz Clock;
- No Instruction and Data Cache;
- No Branch Target Cache;
- No Memory Management Unit (MMU);

- No Barrel Shifter;
- No Integer Multiplier and Divider;
- No Floating Point Unit (FPU);
- 8 kB for Instruction and Data Memory.

This core serves as the base to build the Triple Modular Redundancy Subsystem, that is, in short, a set of IPs developed by Xilinx that are designed in order to be able to manage automatically and mask the presence of the faults that affects the Microblaze soft core. [16]

Similary to what said for the Microblaze embedded core, there are many configuration options that can be used to generate the Subsystem, that has been configured to triplicate the soft core and its peripherals, without a Watchdog counter and the Soft Error Mitigation Interface. The reason for the lack of this interface is dictated by the fact that the SEM IP shows an comparable susceptibility to soft errors than the core itself, therefore making its presence not useful for testing purposes.

6.1.1 Recovery of the Microblaze Subsystem

An important role in the TMR Subsystem IP is played by the *TMR Manager* component. This is the core component of the subsystem: it handles the presence of faults by continuously analyzing the comparator statuses. In case one of them presents a mismatch, a special interrupt-like signal, called *Break* is asserted and the Microblazes present in the design are forced to start the recovery process.

During this process the cores are forced to perform the following list of operations: [16]

1. the software is interrupted by the break signal, that cause the call of the software break handler function.
2. the break handler stores all the internal registers to the data RAM. during this process, the data is automatically corrected by the voters present at the output of each processor.
3. the break handler resets all the microblaze cores present by executing a special instruction that resets also the status of the TMR manager.
4. after reset, the values of the registers placed in RAM are read and restored.

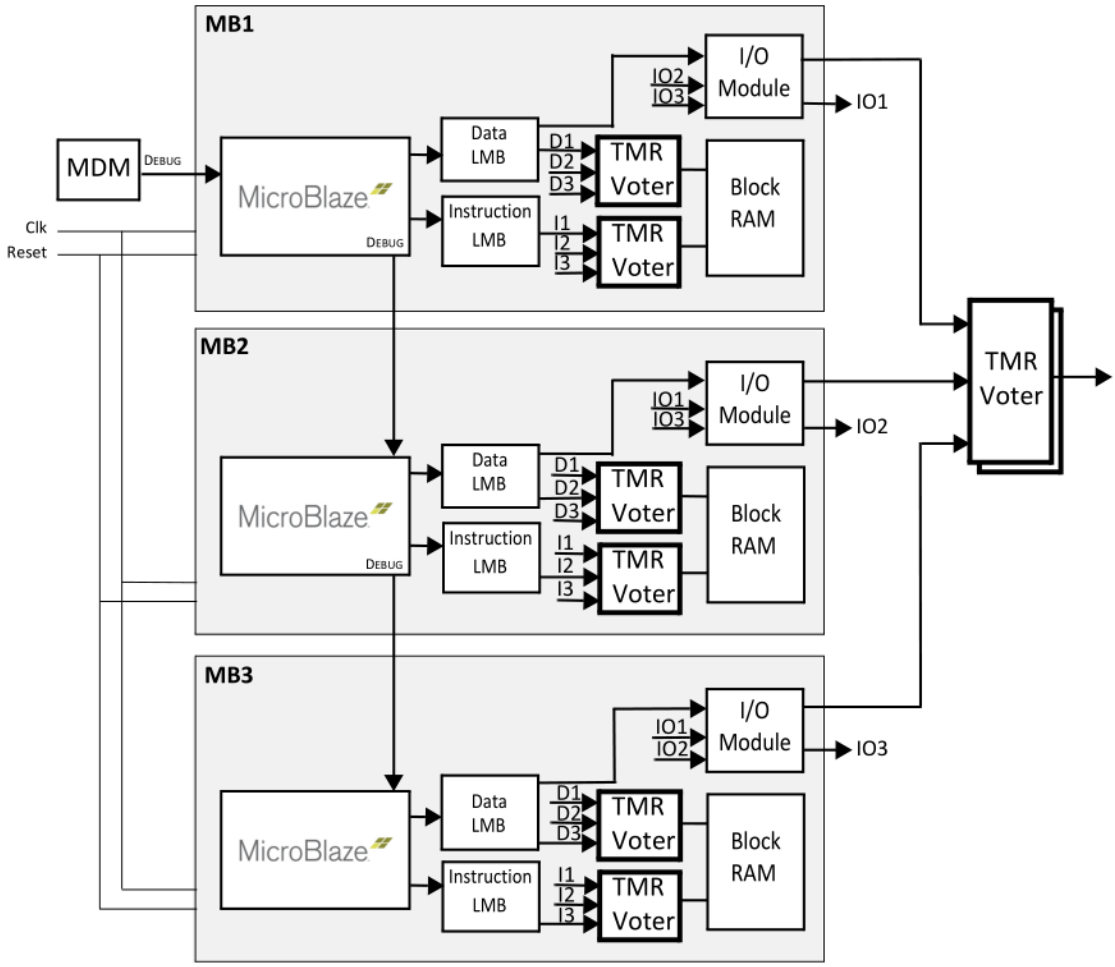


Figure 6.1: TMR Subsystem block diagram: the Microblaze core is triplicated as well as its peripherals and its memory, the outputs are voted

5. a special return resumes the execution exactly at the place where the break occurred.

During this process, the processor subsystem is *unavailable*. The shortness of the recovery process ensures high levels of availability for the core. For real time applications there is also the possibility of masking the break signal during time-critical parts of the code executed, de facto delaying the restore of the subsystem, that works with a working scheme similar to the Duplicate with Comparison (Lockstep).

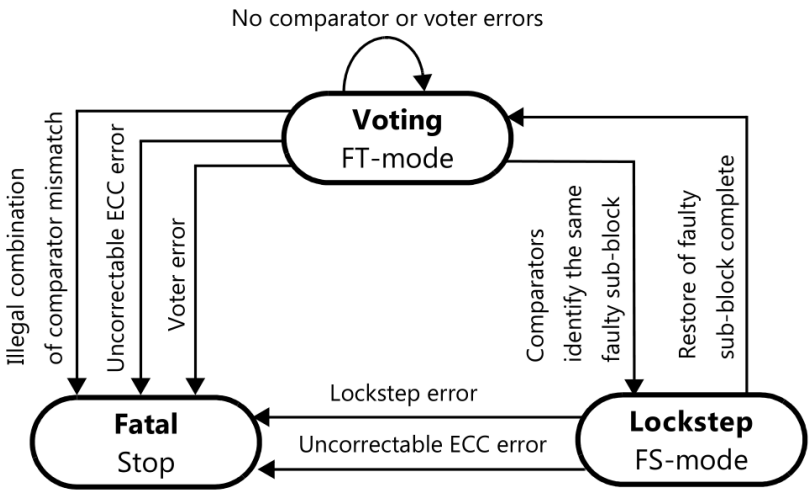


Figure 6.2: TMR Manager state transition in case of an error: starting from Voting mode, an error move the state to Lockstep mode, where the only two out of three processors are working with a Duplicate with Comparison scheme

6.2 Benchmarks for Radiation Testing

The first step towards the characterization of the behavior of a microprocessor against radiation effects is represented by the choice of the algorithm running on it. A microprocessor, indeed, always require a software running on it to produce meaningful results.

In the particular case of radiation benchmarking, the results produced by the core processor play an important role to to identify if it is behaving correctly or not. Although there are present many performance benchmarks for processors, like the Dhrystone or Whetstone synthetic benchmarks, there are no standardized equivalents for radiation testing. For this reason, in many cases, already available performance benchmarks are used in place of dedicated radiation benchmarks. [17]

Benchmarks for radiation testing have different requirements to meet, since the performance that have to be measured is not related to the number of operations executed over a time interval. The main interest during radiation benchmarks is instead represented by the assessment of the working status of the microprocessor tested. An ideal radiation benchmark algorithm should meet the following requirements:

- Fast: the benchmark should be able to highlight the presence of errors in the microprocessors on which it runs onto as soon as possible, therefore reducing

the fault latency. This requires the continuous production of results to compare against.

- Exhaustive: the benchmark should be able to stress completely every component composing the microprocessor. In this way it is possible to propagate the error into a misbehavior of the system.

- Small: due to the memory limitations of embedded microprocessors, the benchmark should occupy as few memory locations as possible for both code and data memory.

- Generic: the benchmark should be "universal", meaning that it should be easily portable to different microprocessor architectures.

Excluding the last requirement, the first three are easily covered if an algorithm to detect *stuck-at* faults is employed for this type of test. Although this type of algorithm is able to maximize is proven to give the best results in terms of speed, exhaustiveness and size, it is strongly affected by the architecture of the microprocessor tested and definitively not portable to other's.

6.2.1 The Algorithm of Choice

After careful consideration the solution for the radiation benchmark coincided with the *Advanced Encryption Standard (AES)* algorithm. The reasons for this choice are presented in the following paragraphs.

First of all, it is a fast algorithm that works on small chunks (128 bit) of data at a time and produces continuously results to be compared against. It works by reading a block of data to be encoded, and it outputs it right after the encoding; the reduced size of the block allows it to produce results at high rate.

Secondly, for the purpose of testing an integer-only microprocessor it is exhaustive: this is the case for the configured Microblaze core. The operations performed are able to test most of the core on which it runs on, allowing to easily highlight the presence of errors in these components. In a cryptographic algorithm every bit in a stored in a register is used and it matters for the successfulness of the encoding.

Finally, there are many open source implementations of this algorithm available online, some of them already optimized to run on microprocessors. AES can be found implemented in many different programming languages, including the omnipresent C language used for microprocessor programming, that are easily portable and architecturally independent.

6.3 Microprocessor Testing Metrics

After the definition of the algorithm of choice for radiation benchmarks, it is necessary to define the metrics and the procedure used to characterize the microprocessors. In the following sections are discussed the main actors involved in these subjects.

The first step toward characterization process requires to be able to identify if a core is behaving correctly or not. If it is not, it is also important to be able to distinguish between the different possible outcomes and assign severity based on the potential consequences that they can cause on the mission. In the next sections are discussed these two topics.

6.3.1 Working status of a processor

For the type of the test that have to be performed, that is representative of the worst case scenario for the processor, there are only four possible working statuses for it:

- *Active and Working* – that represents the nominal conditions: the processor is both in running state and it is producing the correct results at the outputs.
- *Active and Not Working* – that represents the conditions where the processor is in running state but it is not producing the correct results.
- *Not Active and Working* – that represents the conditions where the processor is not in running state (i.e. it is not producing any output) but it was working correctly up to that point.
- *Not Active and Not Working* – that represents the conditions where the processor was already not producing the correct results, and it stopped producing any. This state is not really meaningful for the analysis performed but it is indeed one of the possible outcomes.

6.3.2 Severity Analysis

Now that the possible working statuses of a processor are defined, it is important to identify what are the threats associated with each working state.

As said previously, *Active and Not Working* represents the conditions correspondent to a processor that seems to run correctly, but the results that it produces are

not correct. This status is definitively the most hazardous among all of them: the fact that the core is produce what seem to be correct results, but in reality they are not, can have serious consequences on the mission carried. In fact, the only method to verify the correctness of the data that is produced by a core is to have already the expected values stored, or to produce them at runtime with an error-free circuit. These two solutions completely defeat the purpose of use of a microprocessor from the beginning. As said, the consequences of a bad output can be catastrophic in systems that require an high reliability of their components; for instance, a core could be utilized to control the opening of an airplane door: a wrong output value can potentially open it during a flight, causing a catastrophe.

On the other hand, *Not Active and Working* status can be detected more easily and usually there is also the possibility of performing corrective actions before the system's mission is affected; for these reasons, the severity associated to this status is lower than the previous one. The detection of this kind of status can be implemented both at hardware level and at software level. The first one consists in the monitoring of a so-called *heartbeat* signal coming from the processor: this is a non-constant periodic signal that indicates the normal operation of the core. When this signal stops, corrective actions have to be performed in order to restore the working state. A second one, instead, make use of a *watchdog counter*: this counter is usually designated to generate a reset when it reaches zero, task of the software running on the core is to periodically reset it to the original value.

6.3.3 Mean Time to Failure Evaluation

To complete the characterization of an embedded core, it is necessary to evaluate its Mean Time to Failure. This quantity, as stated in Chapter 4, represents the average time required to experience a failure on one system; it is now necessary to be able to express it by counting the failures on the cores tested in relation to the number of faults present on the board. Starting from the number of faults, then, it is possible to evaluate the time required to experience them, allowing to express them as MTTF.

For this purpose there is no distinction between a core that stopped running (*Not Active and Working*) and a core that produces wrong results (*Active and Not Working*): although there are difference in terms of severity, the failure is still perceived in both cases. By keeping track then of the number of non-operational cores over an extensive test in which the number of faults present can be counted, it is possible to evaluate the average number of upsets required to experience a misbehavior.

$$U_{\text{avg}} = \frac{\sum_{i=0}^{c-1} E_i}{c} \tag{6.1}$$

Where c represents the number of cores tested, E_i represents the quantity of errors required to produce a misbehavior for i-th core tested.

From this point it is possible, by revisiting Equation 4.22, to express the time required to experience an upset, as function of flux.

$$t_{\text{avg}}^{\text{upset}} = \frac{U = 1}{\phi_{\text{avg}} \cdot \sigma_{\text{device}}} \tag{6.2}$$

Considering Equation 6.1 and substituting the value of U, it possible to express the average time to experience a misbehavior as a function of flux, that is correspondent to the MTTF of the core.

$$t_{\text{avg}}^{\text{failure}} = t_{\text{avg}}^{\text{upset}} \cdot U_{\text{avg}} = \frac{\sum_{i=0}^{c-1}}{c \cdot \phi_{\text{avg}} \cdot \sigma_{\text{device}}} = \text{MTTF} \tag{6.3}$$

6.4 Testing Procedure and Architecture

After the definition of the metrics used to evaluate the performances against radiation effects for the embedded cores, it is necessary to define a testing procedure to follow.

The testing procedure have slight differences depending on its type, tabletop testing or ground testing. The former, as described in Section 5.1 is in general faster than the latter due to the possibility to inject faults at an almost arbitrary rate. The latter, instead, as described in Section 5.2, is able to produce more accurate results. The differences on the procedures reflect the ones on the architectures, for this reason two different architectures have been developed.

6.4.1 Single Module Testing

To be able to highlight the operational status of a microprocessor core, there are requirements that have be to be met. In the following paragraphs are discussed the main components required to accomplish this task.

First of all, there is the need of both the source data and the golden result for each processor: they store respectively the input and output (expected) patterns for each core tested. They could not even be present on board, therefore relying on a communication to the outside world: although this approach can save area, it slow all the operation down to the speed of the link employed. For this reason, these two

components are implemented using a triplicated BRAM with ECC enabled: this ensure to be able to work at the operational speed of each core.

As said before, the source data is a set of the input patterns required by the core; in the specific case of the AES algorithm, the source data is composed by:

- the *key* used to encode the data;
- a *plaintext* that represents the input patterns for the processor.

The *golden result* is instead required to have a reference to compare against. In the specific case of the AES algorithm, it is composed by the *ciphertext*, that is the result of the encoding of the plaintext with the key provided.

Based on the comparison between the value produced by the *Device Under Test (DUT)*, i.e. the core tested, and the expected value coming from the golden result memory, it is possible to identify its operational status. For this reason, two saturating up-counters are added:

- a *Loop Counter*;
- an *Error Counter*.

The former is incremented every time that a comparison between the result from the processor and its respective "golden" is performed. By continuous reading this counter it is possible to identify if a core is active or not based on the history of the values. When a core is active this counter counts up until it saturates, while if the core is stalled it remains constant over time. Based on the speed of the interface and the frequency of the output of the algorithm it is impossible to read consecutively two identical values.

The latter is instead incremented every time that a comparison is performed, but there was a mismatch between the values at the output. This counter holds the value 0 until an error occurs: in this case two are the possible outcomes:

- the error counter starts counting up, that is representative of the *Active and Not Working* status;
- the error counter increase and stop counting afterwards, that is representative a non-permanent error.

With these two counters it is then possible to evaluate the four possible operational statuses of a core in real time, by reading continuously the values that they holds.

Finally, there is also the need for an interface used to communicate these values to the external world, where they are analyzed and stored to complete the characterization. For this purpose, a SPI interface was used.

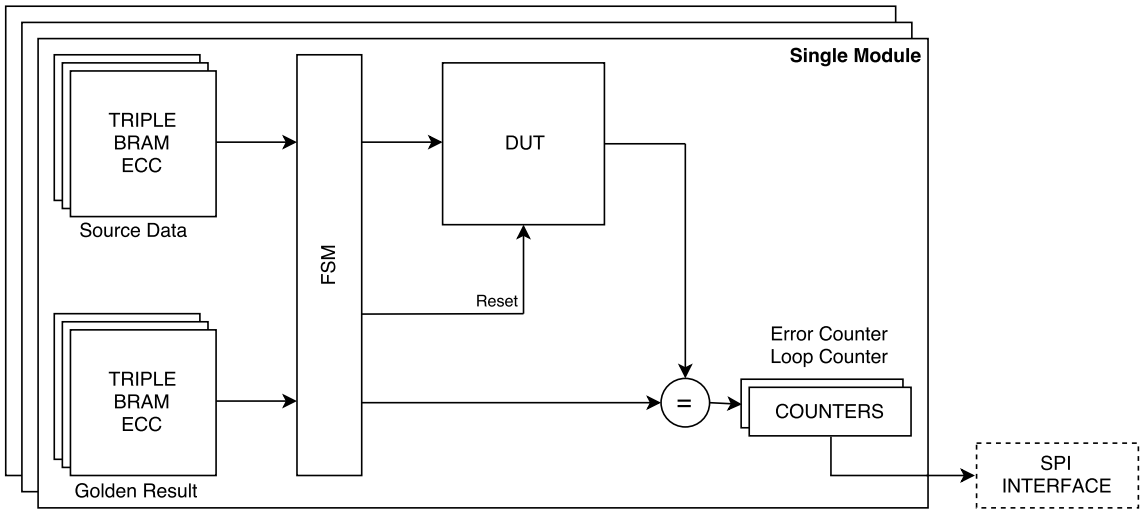


Figure 6.3: Single Module architecture: the source data is read by the DUT, and compared with the golden results. Counters are placed to verify the operational and working state of the DUT.

6.4.2 Tabletop Testing

The first round of irradiation is performed during tabletop testing. During this phase a design containing multiple copies (20) of each core tested have been loaded onto the FPGA, then a fault injection campaign have been started. Having multiple copies of each core can improve two aspects of this test:

- The speed of the simulation – the increased number of cores running in parallel reduces the number of tests that have to be performed;
- The systematic effects given by the placement of the processor core on the FPGA.

For fault injection purposes, and to ease the complexity and improve the speed of the tests, an external JCM device was connected to the JTAG port of the FPGA. On the remote PC, acting as an SPI master, instead, a script was in charge of the following actions:

1. Configuring the FPGA with a clean, correct bitstream;
2. Start reading valued from the core counters present;
3. Start fault injection, injecting one fault every 100 ms;

4. Keeping track of the current status of every core present in the design: when a core changes its operating status the following operations are performed:
   (a) Stop the fault injection;
   (b) Save the state of all counters present in the design;
   (c) Save the current number of faults injected;
   (d) Resume the fault injection;
5. Stopping the procedure and restarting the process when one of the following conditions was met:
   - All the microprocessors present were either in Not Available state or in Not Working state;
   - The testing circuitry failed, meaning that a fault have been registered on one of the components responsible for the data transferred to the PC.

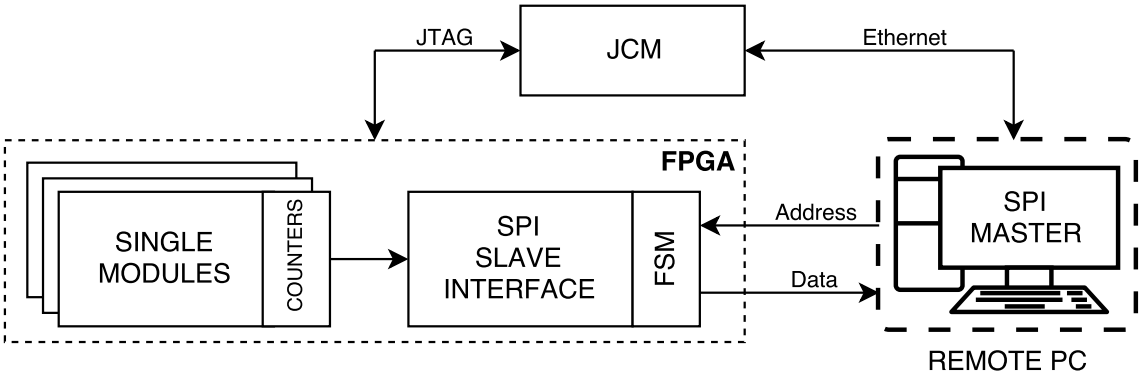


Figure 6.4: Tabletop testing block diagram: the single modules are interfaced using SPI from a remote PC, that controls the fault injection procedure using JCM

6.4.3 Ground Testing

For ground testing the approach is different. First of all, being a part of a more complex system containing more elements not correlated to the microprocessors that had to be tested, the number of cores placed on the FPGA had been reduced to 1 per type, instead on 20.

The FPGA used was mounted on a custom board developed for the ALICE Experiment, called *Readout Unit*. This board features a custom chip, called *Giga-Bit Transceiver - Slow Clock Adapter (GBT-SCA)* that was used to enable the SPI communication over optical fiber. [18]

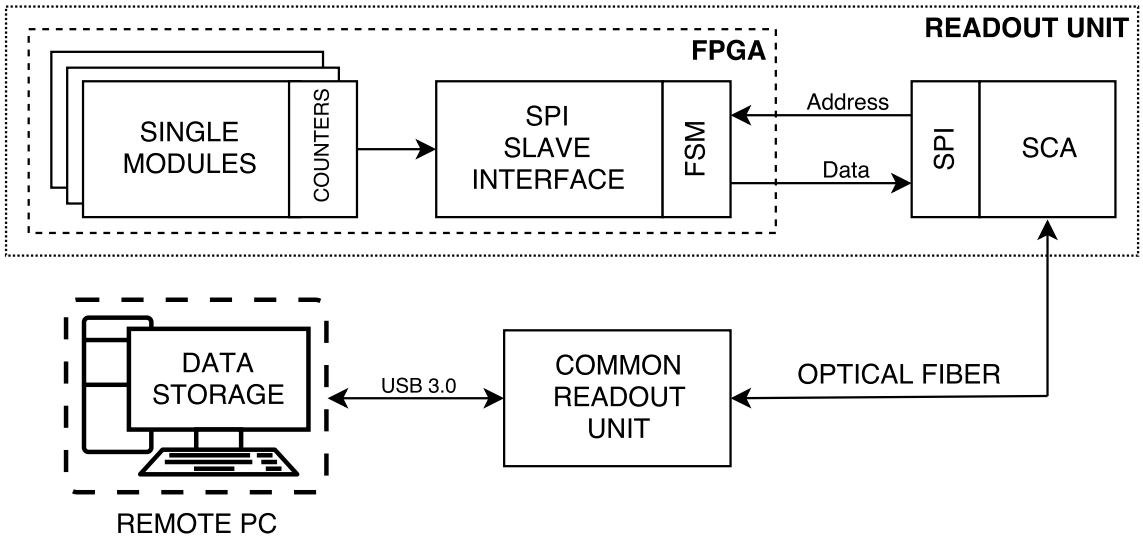


Figure 6.5: Ground testing block diagram: the single modules are interfaced using a custom chip called SCA, that allows SPI communication over optical fiber. A remote PC serves for data storage, and it uses a Common Readout Unit to communicate over optical fiber.

The remote PC had a different task, while before it was actively used in the process of controlling the fault injection process and the reconfiguration of the FPGA, now it only serves the purpose of data storage. The values of the counters are, in fact, read with many more other values coming from different components and peripherals involved in the project.

After the data taking process, the available data is analyzed by cross-checking with the values of the fluence irradiating the FPGA and the timestamps of counter values. Starting from this point it is possible, using the equations discussed in Chapter 4, to evaluate the equivalent number of upsets registered on the device.

Chapter 7

Results, Conclusions and Future Work

In this thesis have been presented techniques and architectures that can be employed for radiation testing of microprocessors, in collaboration with the CERN ALICE ITS group. In the following sections are presented the results relative to the characterization of the Xilinx Microblaze and TMR Subsystem.

Furthermore, the same testing procedure and architecture have been employed for the characterization of an open source microprocessor soft core: the Murax VexRiscv SoC. On top of this implementation have been applied custom mitigation techniques using Synopsys Synplify. The results were finally compared in terms of reliability, being the two alternatives similar in features, resource occupation and performances.

In Table 7.1 are presented:

- a list of the core tested, comprehensive of the mitigation techniques employed;
- the FPGA main resources utilized for each resource type.

Resource	**Xilinx Microblaze**		**Murax VexRiscv**			
Type	**Single**	**TMR**	**Single**	**ECC**	**TMR**	**TMR+ECC**
LUTLUT	2665	9933	915	942	7965	8129
FF	3086	11652	1008	1022	3374	3440
BRAM	2	6	3	4	9	12

Table 7.1: Comparison between resource utilization of each core flavor

7.1 Results

In this section are presented the results of both tabletop and ground testing, first in terms of average number of upsets required to change the operational status, and then in terms of Mean Time to Failure, considering the nominal flux for the ALICE Experiment:

$$\phi_{\text{avg}} = 10^3 \text{ Hz} \tag{7.1}$$

Considering the Xilinx Ultrascale XCKU040 FPGA as reference for the cross section and making use of Equation 6.2, the following average upset period can be calculated:

$$t_{\text{avg}}^{\text{upset}} = \frac{U = 1}{\phi_{\text{avg}} \cdot \sigma_{\text{device}}} = 3690 \text{ s} \tag{7.2}$$

Where $\sigma_{\text{device}} = \sigma_{\text{bit}} \cdot N_{\text{bits}} = 2.55^{-15} \cdot 106269009 = 2.71^{-7}$.

7.1.1 Tabletop Testing Results

For tabletop testing purposes, six different designs were produced, one for each core tested as in Table 7.1. Each one of them contained 20 replicas of the same core, as explained in Chapter 6, on which fault injection have been performed using JCM.

During these tests, a total of 75881726 faults have been injected on the six different designs produced, that yielded a total of 102550 cores in either Operational and Not Working (F) or Not Operational and Working (S) state. The following table presents the results in terms of average number of upsets required, and the correspondent MTTF with the particle flux of Equation 7.1.

	Xilinx Microblaze		**Murax VexRiscv**			
	Single	**TMR**	**Single**	**ECC**	**TMR**	**TMR+ECC**
U_{avg}	583	286	599	885	1042	1329
%F	45.82	3.76	35.31	39.07	37.61	47.04
%S	54.18	96.24	64.69	60.93	62.39	52.96
MTTF [h]	598	293	614	907	1068	1362

Table 7.2: Tabletop testing results

7.1.2 Ground Testing Results

For the purpose of ground testing, differently from tabletop testing, due to limited resource requirements, only one single peripheral containing one copy of each core subsystem was integrated into the ReadOut Unit firmware. For this reason, due to both possible systematic effects and not sufficient beam time, the results are not presented in terms of equivalent Mean Time to Failure.

During these tests, only 125 cores per core type have been tested, and in many occasions it was not possible to establish the number of faults required to break them, as they were still working at the end of the beam test. In the following table are presented cores in Operational and Working (W), Operational and Not Working (F) and Not Operational and Working (S) state.

	Xilinx Microblaze		**Murax VexRiscv**			
	Single	**TMR**	**Single**	**ECC**	**TMR**	**TMR+ECC**
U_{avg}	1403	1025	1984	3117	21524	114343
%F	19.80	2.97	17.82	13.86	0.00	0.99
%S	25.74	50.50	21.78	12.87	5.00	0.00
%W	54.46	46.53	60.40	73.27	95.00	99.01

Table 7.3: Ground testing results

7.2 Conclusions and Future Work

As shown in the previous section, the results of the characterization of the Xilinx Microblaze and TMR Subsystem IPs do not match the open source counterpart, especially in the case of the Triple Modular Redundancy Subsystem.

Speaking in terms of pure reliability, the TMR Subsystem is completely outperformed by even the single Microblaze core, that takes advantage of its reduced resource usage. On the other hand, analyzing the results from a security point of view, as discussed in Section 6.3.2, the Subsystem is without doubt the most secure solution among the possible options.

There are many possible solutions that may be implemented to improve the reliability of this IP, that have been tested in its first official release to the public. First of all, the internal Soft Error Mitigation IP could be employed to improve the error correction latency. Secondly, there are many software-based solutions, as discussed in Chapter 3, that may be implemented to further improve the reliability of the system. Finally, another path could be represented by the employment of Synopsys Symplify to provide a custom solution to be compared against.

For what concerns instead the testing procedure and, consequently, the testing architecture, there are possible future improvements that include:

- The use of other algorithms for radiation benchmarking; one of the possible solution may be the development of a dedicated benchmark for this purposes.
- The use of a more sophisticated testing procedure, that has not been implemented due to limitations in terms of data bandwidth.
- The employment of different FPGA families from different vendors, to possibly compare the same core on different platforms.

Appendix A

A Large Ion Collider Experiment

CERN's A Large Ion Collider Experiment (ALICE) is one of the largest experiments in the world devoted to research in the physics of matter at an infinitely small scale. Located at the Large Hadron Collider (LHC), this experiment is committed to the study of heavy ion collisions, with a center of mass energy of approximately 5.5 TeV per nucleon. The main objective of the experiment is represented by the study of dark matters at high densities and temperatures.

To achieve this objective, the ALICE detector is composed of two main components:

- a central part, called Inner Tracking System (ITS) – mainly composed of detectors used to study hadronic signals and dielectrons;
- a forward muon spectrometer – used to study quarkonia behavior in dense matter.

A.1 Upgrade of the Inner Tracking System

As said in the introduction, the ITS represents the central part of the ALICE detector. It is embedded in a large magnet and it covers $\pm 45°$ over the full azimuth. Its basic functions are the following:

- determination of the primary vertex and of the secondary vertices necessary for the reconstruction of charm and hyperon decays;
- particle identification and tracking;

- improvement of the momentum and angle measurements for the Time Projection Chamber (TPC).

The ALICE Experiment is planning to upgrade the ITS during the second LHC shutdown, in the years 2019-2020. The new ITS will be composed of 7 concentric layers of pixel detectors, each layer will be 1.5 m long, with the outer radius of 40 cm. This arrangement of sensors will create a 12.6 Gpx camera. [19]

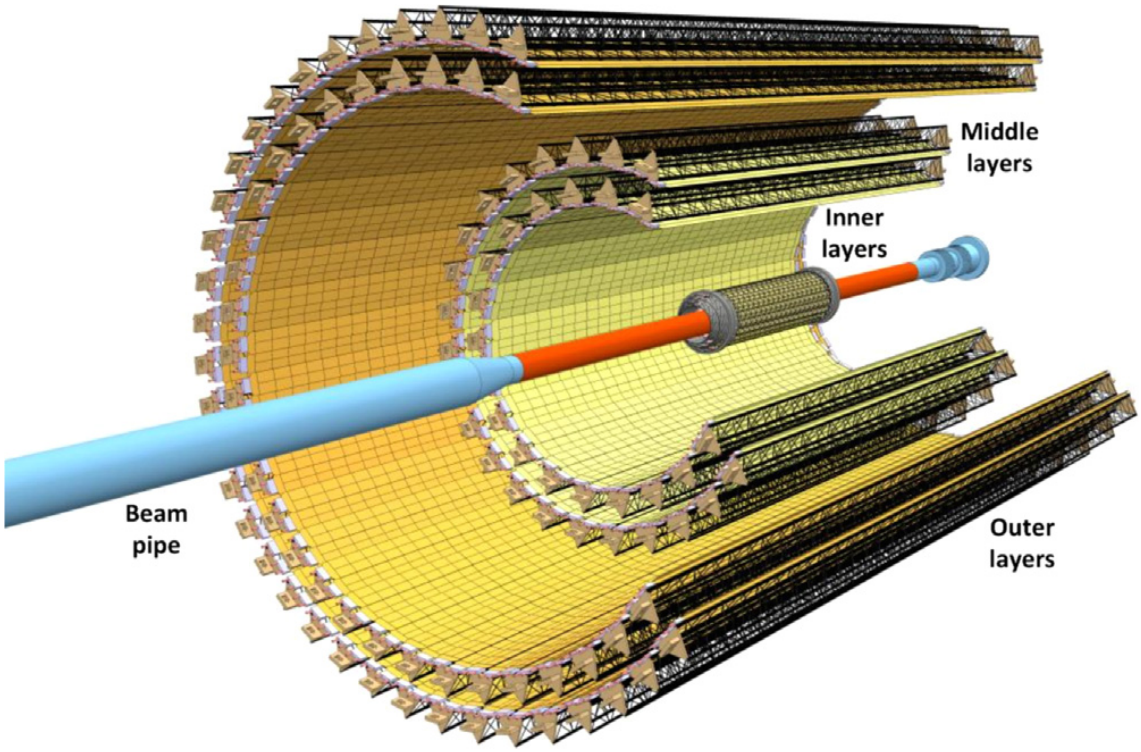


Figure A.1: Sensors layout of the upgraded ALICE-ITS: 2 outer layers, 2 middle layers and 3 inner layers

All the seven layers will be equipped with the ALPIDE chip, that embeds the sensitive part and the read-out electronics within the same piece of silicon. This chips are organized in *staves*, each stave is composed by a different number of aligned ALPIDE chips:

- Innner Barrel stave: 9 ALPIDE chips;
- Middle and Outer Barrel stave: 14 ALPIDE chips, organized in two *half-staves*.

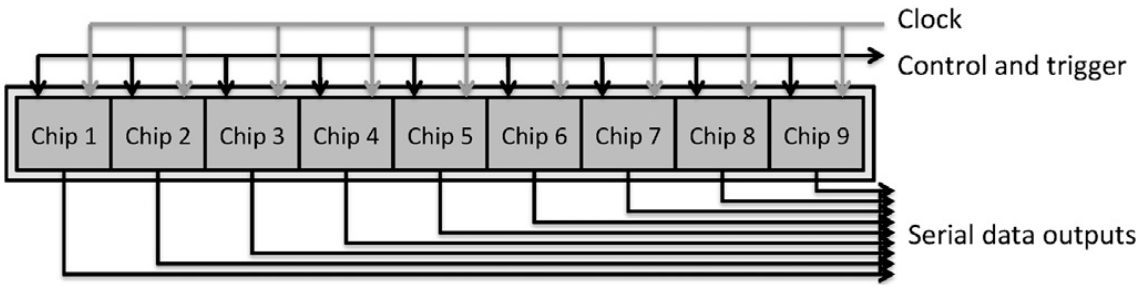


Figure A.2: The Inner Barrel stave readout architecture: 9 ALPIDE chips are organized in a straight line, operating at 1.2 Gbit/s

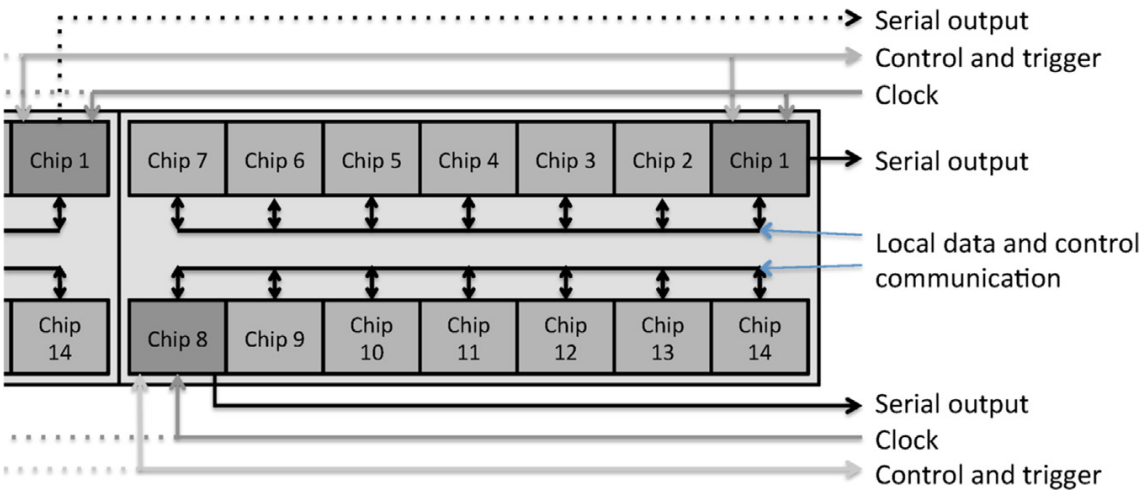


Figure A.3: The Outer Barrel stave readout architecture: 14 ALPIDE chips are organized in two half-staves, each one composed by 7 sensors operating at 0.4 Gbit/s

A.1.1 Readout Electronics

To carry out the incoming data from the sensors present in the ITS, a set of dedicated hardware (Readout Electronics) is placed nearby them. [20]

The readout electronics plays a fundamental role in this scheme. Being close the beam collision, it is strongly affected by radiation effects. The core logic of the readout electronics is composed by the Readout Unit (RU), that is custom board that features a Xilinx XCKU060 FPGA. This board takes care of interfacing multiple ALPIDE chips to the Common Readout Unit (CRU) while syncing with the triggers coming from the ALICE Trigger System. In particular, one single RU is connected to:

- 9 Inner Barrel ALPIDE chips, operating at 1.2 Gbit/s;

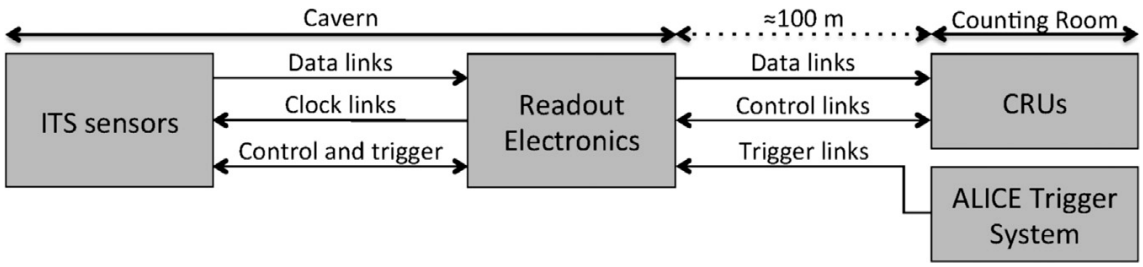


Figure A.4: High level architecture of the upgraded ITS: the sensors are directly interfaced by the readout electronics, that is connected to Common Readout Units and synchronized with the ALICE Trigger System

- 8 or 14 Middle Barrel ALPIDE chips, operating at 0.4 Gbit/s;
- 8 or 14 Outer Barrel ALPIDE chips, operating at 0.4 Gbit/s;
- the Common Readout Unit, that receives the data coming from the sensors;
- the ALICE Tigger System, that sends triggers synchronized with the LHC clock.

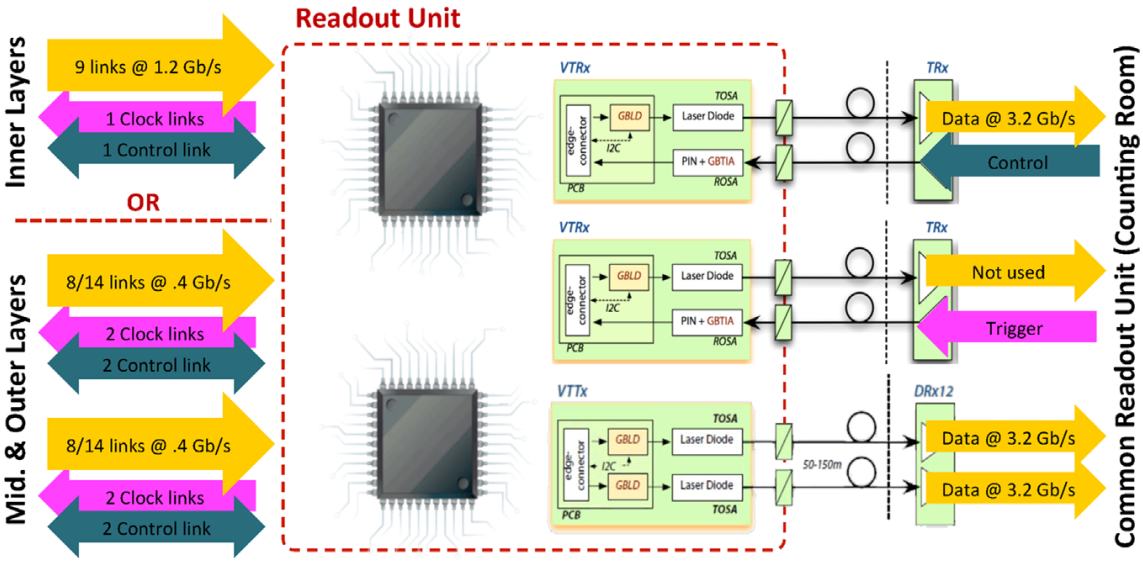


Figure A.5: Readout Unit architecture, it interface various sensors operating a different speeds with the Common Readout Unit, while being synchronized with the triggers coming from the ALICE Trigger System

Appendix B

Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data. AES is a subset of the so-called *Rinjdael cipher*, a family of ciphers with different key and block sizes. AES is a fast algorithm to implement both in software and in hardware, that defines a block size of 128 bit and a key size of 128, 192 or 256 bit, depending on the version used.

B.1 Working Principle

The algorithm works using a *substitution-permutation* network, that is a series of linked mathematical operations. This network takes two inputs, a *plaintext* and a *key*, and applies several *rounds* of *substitution boxes (S-boxes)* and *permutation boxes (P-boxes)*; the result obtained is called *ciphertext*. A S-box takes care of substituting a small block of bits with another one; the operation must be invertible to ensure the possibility of decryption. A P-box, instead, takes care of performing a permutation of the bits present in a small block. After the application of the substitution and permutation boxes, a *round key* is obtained by employing some group operations, typically *xor*.

B.2 AES Versions

AES operates on a 4x4 matrix of bytes, called *state*, on which are applied the same operations for a number of cycles that depends on the key size.

- 10 cycles for 128 bit keys (AES-128);

- 12 cycles for 192 bit keys (AES-192);
- 14 cycles for 256 bit keys (AES-256).

Finally, there are many modes of operation for the cipher. The simplest one is called *Electronic Codebook (ECB)*: in this case each block is encrypted separately, by applying the same set of operations to each block encoded. An alternative to this solution is called *Cipher Block Chaining (CBC)*: in this case the product of the previous encoding is combined to the plaintext of the next one. [23]

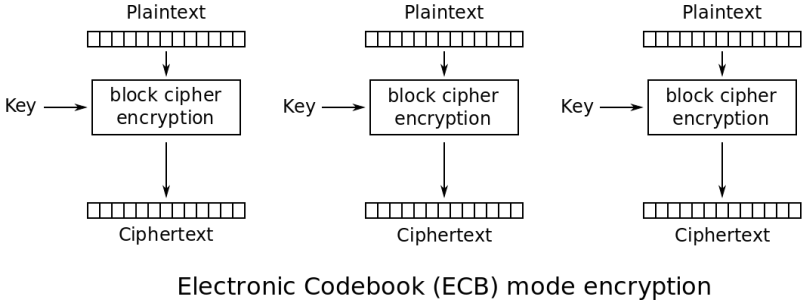


Figure B.1: AES ECB Encryption: the same set of operations are applied to different blocks of the input data, treating them separately.

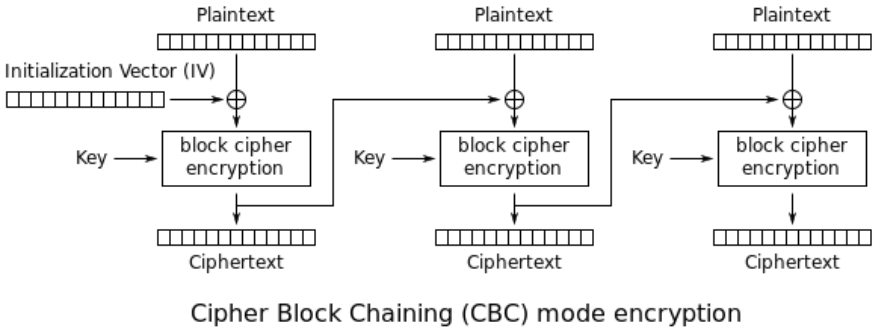


Figure B.2: AES CBC Encryption: the output of the previous encoding is combined with the block of data encoded at the next step.

B.3 Implemented Algorithm

In this section is presented the implemented lightweight version of the AES algorithm, written in C language.

B.3.1 aes.h

```
#ifndef _AES_H_
#define _AES_H_

#include <stdint.h>

/* Name definitions:
 *  - Nb: The number of columns comprising a state in AES.
 *  - Bs: Block size in bytes AES is 128b block only.
 *  - Nk: Number of 32-bit words in a key.
 *  - Nr: Number of rounds of for encryption.
 *  - Ks: Ik size in bytes.
 *  - Ke: Expanded Ik size in bytes.
 *  - Rk: Round Ik.
 *  - Ik: Input Ik.
 *  - Iv: Input Vector.
 *  - St: State.
 */

#define AES128 1
#define ECB_ENC 1

#define Nb 4
#define Bs (Nb * Nb)

#if   AES128

#define Nk 4
#define Nr 10
#define Ks 16
#define Ke 176

#elif AES192

#define Nk 6
#define Nr 12
#define Ks 24
#define Ke 208

#elif AES256

#define Nk 8
#define Nr 14
#define Ks 32
#define Ke 240

#else

#error "No AES version defined!"

#endif /* AES */


#if ECB | ECB_ENC
void ecb_encrypt(uint8_t* dst, const uint8_t* src, const uint8_t* key);
#endif /* ECB | ECB_ENC */

#if ECB | ECB_DEC
void ecb_decrypt(uint8_t* dst, const uint8_t* src, const uint8_t* key);
#endif /* ECB | ECB_DEC */
```

```
#if CBC | CBC_ENC
void cbc_encrypt(uint8_t* dst, const uint8_t* src, const uint8_t* key,
                 const uint8_t* iv);
#endif /* CBC | CBC_ENC */

#if CBC | CBC_DEC
void cbc_decrypt(uint8_t* dst, const uint8_t* src, const uint8_t* key,
                 const uint8_t* iv);
#endif /* CBC | CBC_DEC */

typedef uint8_t state_t[Nb][Nb];

#endif //_AES_H_
```

B.3.2 aes_constants.h

```
#ifndef AES_CONST_H
#define AES_CONST_H 1

static const uint8_t sbox[256] = {
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
  0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
  0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
  0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
  0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
  0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
  0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
  0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
  0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
  0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
  0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
  0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
  0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
  0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
  0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
  0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
  0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};

static const uint8_t rsbox[256] = {
  0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,
  0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
  0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
  0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
  0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d,
  0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
  0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2,
  0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
  0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,
  0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
  0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,
  0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
  0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,
  0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
  0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02,
  0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
  0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea,
  0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
  0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85,
  0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
  0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
```

```
  0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
  0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20,
  0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
  0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31,
  0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
  0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d,
  0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
  0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0,
  0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
  0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,
  0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
};

static const uint8_t Rcon[11] = {
  0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
  0x80, 0x1b, 0x36
};

#endif /* ifndef AES_CONST_H */
```

B.3.3 aes.c

```
#include <stdint.h>

#include "aes.h"
#include "aes_constants.h"

static void SubWord(uint8_t* word) {
  word[0] = sbox[word[0]];
  word[1] = sbox[word[1]];
  word[2] = sbox[word[2]];
  word[3] = sbox[word[3]];
}


static void RotWord(uint8_t* word) {
  uint8_t tmp;
  tmp     = word[0];
  word[0] = word[1];
  word[1] = word[2];
  word[2] = word[3];
  word[3] = tmp;
}


static void IkExpand(uint8_t* Rk, const uint8_t* Ik) {
  int i;
  uint8_t word[4];

  // The first round key is the key itself.
  for (i = 0; i < Nk; ++i) {
    Rk[(i * 4) + 0] = Ik[(i * 4) + 0];
    Rk[(i * 4) + 1] = Ik[(i * 4) + 1];
    Rk[(i * 4) + 2] = Ik[(i * 4) + 2];
    Rk[(i * 4) + 3] = Ik[(i * 4) + 3];
  }

  // All other round keys are found from the previous round keys.
  //i == Nk
  for ( ; i < Nb * (Nr + 1); ++i) {
    word[0] = Rk[(i-1) * 4 + 0];
    word[1] = Rk[(i-1) * 4 + 1];
    word[2] = Rk[(i-1) * 4 + 2];
    word[3] = Rk[(i-1) * 4 + 3];

    if (i % Nk == 0) {
      RotWord(word);
      SubWord(word);
      word[0] =  word[0] ^ Rcon[i/Nk];
    }
#if defined(AES256) && (AES256 == 1)
    if (i % Nk == 4) {
      SubWord(word);
    }
#endif
    Rk[i * 4 + 0] = Rk[(i - Nk) * 4 + 0] ^ word[0];
    Rk[i * 4 + 1] = Rk[(i - Nk) * 4 + 1] ^ word[1];
    Rk[i * 4 + 2] = Rk[(i - Nk) * 4 + 2] ^ word[2];
    Rk[i * 4 + 3] = Rk[(i - Nk) * 4 + 3] ^ word[3];
  }
}
```

```
static void AddRk(state_t* St, uint8_t* Rk, int round) {
  int i, j;

  for (i = 0; i < 4; ++i) {
    for (j = 0; j < 4; ++j) {
      (*St)[i][j] ^= Rk[round * Nb * 4 + i * Nb + j];
    }
  }
}


static void SubBytes(state_t* St) {
  int i, j;

  for (i = 0; i < 4; ++i) {
    for (j = 0; j < 4; ++j) {
      (*St)[j][i] = sbox[(*St)[j][i]];
    }
  }
}


static void ShiftRows(state_t* St) {
  uint8_t temp;

  // Rotate first row 1 columns to left
  temp        = (*St)[0][1];
  (*St)[0][1] = (*St)[1][1];
  (*St)[1][1] = (*St)[2][1];
  (*St)[2][1] = (*St)[3][1];
  (*St)[3][1] = temp;

  // Rotate second row 2 columns to left
  temp        = (*St)[0][2];
  (*St)[0][2] = (*St)[2][2];
  (*St)[2][2] = temp;
  temp        = (*St)[1][2];
  (*St)[1][2] = (*St)[3][2];
  (*St)[3][2] = temp;

  // Rotate third row 3 columns to left
  temp        = (*St)[0][3];
  (*St)[0][3] = (*St)[3][3];
  (*St)[3][3] = (*St)[2][3];
  (*St)[2][3] = (*St)[1][3];
  (*St)[1][3] = temp;
}


#if XTIME_AS_FUNC
static uint8_t xtime(uint8_t x) {
  return (x << 1) ^ (((x >> 7) & 0x01) * 0x1B);
}
#else
#define xtime(x) (((x) << 1) ^ ((((x) >> 7) & 0x01) * 0x1B))
#endif /* XTIME_AS_FUNC */


static void MixColumns(state_t* St) {
  int i;
  uint8_t Tmp, Tm, St0;
```

```
  for (i = 0; i < 4; ++i) {
    St0 = (*St)[i][0];
    Tmp = (*St)[i][0] ^ (*St)[i][1] ^ (*St)[i][2] ^ (*St)[i][3];

    Tm  = (*St)[i][0] ^ (*St)[i][1]; Tm = xtime(Tm); (*St)[i][0] ^= Tm ^ Tmp;
    Tm  = (*St)[i][1] ^ (*St)[i][2]; Tm = xtime(Tm); (*St)[i][1] ^= Tm ^ Tmp;
    Tm  = (*St)[i][2] ^ (*St)[i][3]; Tm = xtime(Tm); (*St)[i][2] ^= Tm ^ Tmp;
    Tm  = (*St)[i][3] ^ St0;         Tm = xtime(Tm); (*St)[i][3] ^= Tm ^ Tmp;
  }
}

#if MPY_AS_FUNC

static uint8_t Mpy(uint8_t x, uint8_t y) {
  return (
    (((y) >> 0 & 0x01) * (x)) ^
    (((y) >> 1 & 0x01) * xtime(x)) ^
    (((y) >> 2 & 0x01) * xtime(xtime(x))) ^
    (((y) >> 3 & 0x01) * xtime(xtime(xtime(x)))) ^
    (((y) >> 4 & 0x01) * xtime(xtime(xtime(xtime(x)))))
  );
}

#else

#define Mpy(x, y) (                                  \
  (((y) >> 0 & 0x01) * (x)) ^                        \
  (((y) >> 1 & 0x01) * xtime(x)) ^                   \
  (((y) >> 2 & 0x01) * xtime(xtime(x))) ^            \
  (((y) >> 3 & 0x01) * xtime(xtime(xtime(x)))) ^     \
  (((y) >> 4 & 0x01) * xtime(xtime(xtime(xtime(x))))) \
)

#endif /* MPY_AS_FUNC */


#if ECB | ECB_DEC | CBC | CBC_DEC
static void InvMixColumns(state_t* St) {
  int i;
  uint8_t s0, s1, s2, s3;
  for (i = 0; i < 4; ++i) {
    s0 = (*St)[i][0];
    s1 = (*St)[i][1];
    s2 = (*St)[i][2];
    s3 = (*St)[i][3];

    (*St)[i][0] = Mpy(s0, 0x0e) ^ Mpy(s1, 0x0b) ^ Mpy(s2, 0x0d) ^ Mpy(s3, 0x09);
    (*St)[i][1] = Mpy(s0, 0x09) ^ Mpy(s1, 0x0e) ^ Mpy(s2, 0x0b) ^ Mpy(s3, 0x0d);
    (*St)[i][2] = Mpy(s0, 0x0d) ^ Mpy(s1, 0x09) ^ Mpy(s2, 0x0e) ^ Mpy(s3, 0x0b);
    (*St)[i][3] = Mpy(s0, 0x0b) ^ Mpy(s1, 0x0d) ^ Mpy(s2, 0x09) ^ Mpy(s3, 0x0e);
  }
}
#endif


#if ECB | ECB_DEC | CBC | CBC_DEC
static void InvSubBytes(state_t* St) {
  uint8_t i,j;
  for (i = 0; i < 4; ++i) {
    for (j = 0; j < 4; ++j) {
      (*St)[j][i] = rsbox[(*St)[j][i]];
    }
  }
```

```
}
#endif


#if ECB | ECB_DEC | CBC | CBC_DEC
static void InvShiftRows(state_t* St) {
  uint8_t temp;

  // Rotate first row 1 columns to right
  temp        = (*St)[3][1];
  (*St)[3][1] = (*St)[2][1];
  (*St)[2][1] = (*St)[1][1];
  (*St)[1][1] = (*St)[0][1];
  (*St)[0][1] = temp;

  // Rotate second row 2 columns to right
  temp        = (*St)[0][2];
  (*St)[0][2] = (*St)[2][2];
  (*St)[2][2] = temp;

  temp        = (*St)[1][2];
  (*St)[1][2] = (*St)[3][2];
  (*St)[3][2] = temp;

  // Rotate third row 3 columns to right
  temp        = (*St)[0][3];
  (*St)[0][3] = (*St)[1][3];
  (*St)[1][3] = (*St)[2][3];
  (*St)[2][3] = (*St)[3][3];
  (*St)[3][3] = temp;
}
#endif


#if ECB | ECB_ENC | CBC | CBC_ENC
static void Encrypt(state_t* St, uint8_t* Rk) {
  int round = 0;

  AddRk(St, Rk, round);
  round++;

  for (; round < Nr; ++round) {
    SubBytes(St);
    ShiftRows(St);
    MixColumns(St);
    AddRk(St, Rk, round);
  }

  SubBytes(St);
  ShiftRows(St);
  AddRk(St, Rk, round);
}
#endif


#if ECB | ECB_DEC | CBC | CBC_DEC
static void Decrypt(state_t* St, uint8_t* Rk) {
  int round = Nr;

  AddRk(St, Rk, round);
  round--;

  for (; round > 0; --round) {
```

```
    InvShiftRows(St);
    InvSubBytes(St);
    AddRk(St, Rk, round);
    InvMixColumns(St);
  }

  InvShiftRows(St);
  InvSubBytes(St);
  AddRk(St, Rk, round);
}
#endif


static void cpy(uint8_t* dst, const uint8_t* src) {
  int i;

  for (i = 0; i < Bs; i++) {
    dst[i] = src[i];
  }
}


#if ECB | ECB_ENC
void ecb_encrypt(uint8_t* dst, const uint8_t* src, const uint8_t* key) {
  state_t* St;
  uint8_t  Rk[Ke];

  if (dst != src) {
    cpy(dst, src);
  }

  St = (state_t*) dst;

  IkExpand(Rk, key);
  Encrypt(St, Rk);
}
#endif /* ECB | ECB_ENC */

#if ECB | ECB_DEC
void ecb_decrypt(uint8_t* dst, const uint8_t* src, const uint8_t* key) {
  state_t* St;
  uint8_t  Rk[Ke];

  if (dst != src) {
    cpy(dst, src);
  }

  St = (state_t*) dst;

  IkExpand(Rk, key);
  Decrypt(St, Rk);
}
#endif /* ECB | ECB_DEC */


#if CBC | CBC_ENC | CBC_DEC
static void XorWithIv(uint8_t* block, const uint8_t* Iv) {
  int i;

  for (i = 0; i < Bs; ++i) {
    block[i] ^= Iv[i];
  }
}
```

```
#endif /* CBC | CBC_ENC | CBC_DEC */


#if CBC | CBC_ENC
void cbc_encrypt(uint8_t* dst, const uint8_t* src, const uint8_t* key,
                 const uint8_t* iv) {
  state_t* St;
  uint8_t  Rk[Ke];

  if (dst != src) {
    cpy(dst, src);
  }

  St = (state_t*) dst;
  IkExpand(Rk, key);

  XorWithIv(dst, iv);
  Encrypt(St, Rk);
}
#endif /* CBC | CBC_ENC */


#if CBC | CBC_DEC
void cbc_decrypt(uint8_t* dst, const uint8_t* src, const uint8_t* key,
                 const uint8_t* iv) {
  state_t* St;
  uint8_t  Rk[Ke];

  if (dst != src) {
    cpy(dst, src);
  }

  St = (state_t*) dst;
  IkExpand(Rk, key);

  Decrypt(St, Rk);
  XorWithIv(dst, iv);
}
#endif /* CBC | CBC_DEC */
```

Appendix C

Xilinx Microblaze and TMR Subsystem

C.1 Configuration Scripts

In this section is presented the TCL configuration script used to generate automatically the single Xilinx Microblaze and its triplicated version in the TMR Subsystem. This script defines a procedure called `generate_mb` that is designed to take as input name (`bd_name`) a string in the following form:

`<base_name>_tmr(0|1)_ecc(0|1)`

C.1.1 `generate_mb.tcl`

```
proc generate_mb { prj_name bd_name } {

  create_bd_design $bd_name

  set use_tmr [regexp "tmr1" $bd_name]
  set use_ecc [regexp "ecc1" $bd_name]

  set bd_path "${prj_name}/${prj_name}.srcs/sources_1/bd/${bd_name}"
  set bd_file [get_files "$bd_path/${bd_name}.bd"]

  ## CONSTANTS ##################################################################
  set XIP "xilinx.com:ip"
  set MB_ENABLE_UART 1
  set MB_ENABLE_GPIO 1
  set MB_ENABLE_TIMR 1

  ## COMPONENT: MicroBlaze ######################################################
  set mb [create_bd_cell -type ip -vlnv $XIP:microblaze microblaze_0]
  set_property -dict [list \
```

```
CONFIG.C_ADDR_SIZE 32 \
CONFIG.C_AREA_OPTIMIZED 1 \
CONFIG.C_INTERCONNECT 2 \
CONFIG.C_BASE_VECTORS 0x00000000 \
CONFIG.C_FAULT_TOLERANT 1 \
CONFIG.C_LOCKSTEP_SLAVE 0 \
CONFIG.C_AVOID_PRIMITIVES 3 \
CONFIG.C_PVR 0 \
CONFIG.C_PVR_USER1 0x00 \
CONFIG.C_PVR_USER2 0x00000000 \
CONFIG.C_D_AXI 0 \
CONFIG.C_D_LMB 1 \
CONFIG.C_I_AXI 0 \
CONFIG.C_I_LMB 1 \
CONFIG.C_USE_BARREL 0 \
CONFIG.C_USE_DIV 0 \
CONFIG.C_USE_HW_MUL 0 \
CONFIG.C_USE_FPU 0 \
CONFIG.C_USE_MSR_INSTR 1 \
CONFIG.C_USE_PCMP_INSTR 1 \
CONFIG.C_USE_REORDER_INSTR 1 \
CONFIG.C_UNALIGNED_EXCEPTIONS 0 \
CONFIG.C_ILL_OPCODE_EXCEPTION 0 \
CONFIG.C_M_AXI_I_BUS_EXCEPTION 0 \
CONFIG.C_M_AXI_D_BUS_EXCEPTION 0 \
CONFIG.C_DIV_ZERO_EXCEPTION 0 \
CONFIG.C_FPU_EXCEPTION 0 \
CONFIG.C_OPCODE_0x0_ILLEGAL 0 \
CONFIG.C_FSL_EXCEPTION 0 \
CONFIG.C_ECC_USE_CE_EXCEPTION 0 \
CONFIG.C_USE_STACK_PROTECTION 0 \
CONFIG.C_IMPRECISE_EXCEPTIONS 0 \
CONFIG.C_DEBUG_ENABLED 1 \
CONFIG.C_NUMBER_OF_PC_BRK 1 \
CONFIG.C_NUMBER_OF_RD_ADDR_BRK 0 \
CONFIG.C_NUMBER_OF_WR_ADDR_BRK 0 \
CONFIG.C_DEBUG_EVENT_COUNTERS 5 \
CONFIG.C_DEBUG_LATENCY_COUNTERS 1 \
CONFIG.C_DEBUG_COUNTER_WIDTH 32 \
CONFIG.C_DEBUG_TRACE_SIZE 8192 \
CONFIG.C_DEBUG_PROFILE_SIZE 0 \
CONFIG.C_DEBUG_EXTERNAL_TRACE 0 \
CONFIG.C_DEBUG_INTERFACE 0 \
CONFIG.C_ASYNC_INTERRUPT 0 \
CONFIG.C_FSL_LINKS 0 \
CONFIG.C_USE_EXTENDED_FSL_INSTR 0 \
CONFIG.C_ICACHE_BASEADDR 0x0000000000000000 \
CONFIG.C_ICACHE_HIGHADDR 0x000000003FFFFFFF \
CONFIG.C_USE_ICACHE 0 \
CONFIG.C_ALLOW_ICACHE_WR 1 \
CONFIG.C_ICACHE_LINE_LEN 4 \
CONFIG.C_ICACHE_FORCE_TAG_LUTRAM 0 \
CONFIG.C_ICACHE_STREAMS 0 \
CONFIG.C_ICACHE_VICTIMS 0 \
CONFIG.C_ICACHE_DATA_WIDTH 0 \
CONFIG.C_ADDR_TAG_BITS 17 \
CONFIG.C_CACHE_BYTE_SIZE 8192 \
CONFIG.C_DCACHE_BASEADDR 0x0000000000000000 \
CONFIG.C_DCACHE_HIGHADDR 0x000000003FFFFFFF \
CONFIG.C_USE_DCACHE 0 \
CONFIG.C_ALLOW_DCACHE_WR 1 \
CONFIG.C_DCACHE_LINE_LEN 4 \
CONFIG.C_DCACHE_FORCE_TAG_LUTRAM 0 \
```

```
  CONFIG.C_DCACHE_USE_WRITEBACK 0 \
  CONFIG.C_DCACHE_VICTIMS 0 \
  CONFIG.C_DCACHE_DATA_WIDTH 0 \
  CONFIG.C_DCACHE_ADDR_TAG 17 \
  CONFIG.C_DCACHE_BYTE_SIZE 8192 \
  CONFIG.C_USE_MMU 0 \
  CONFIG.C_MMU_DTLB_SIZE 4 \
  CONFIG.C_MMU_ITLB_SIZE 2 \
  CONFIG.C_MMU_TLB_ACCESS 3 \
  CONFIG.C_MMU_ZONES 16 \
  CONFIG.C_MMU_PRIVILEGED_INSTR 0 \
  CONFIG.C_USE_INTERRUPT 1 \
  CONFIG.C_USE_EXT_BRK 0 \
  CONFIG.C_USE_EXT_NM_BRK 0 \
  CONFIG.C_USE_NON_SECURE 0 \
  CONFIG.C_USE_BRANCH_TARGET_CACHE 0 \
  CONFIG.C_BRANCH_TARGET_CACHE_SIZE 0 \
] $mb

set IRQ_PERIPH [list]
set AXI_PERIPH [list]

## COMPONENT: AXI Timer ########################################################
if {$MB_ENABLE_TIMR} {
  set tim  [create_bd_cell -type ip -vlnv $XIP:axi_timer:2.0 axi_timer_0]
  lappend IRQ_PERIPH $tim
  lappend AXI_PERIPH $tim
}

## COMPONENT: AXI GPIO #########################################################
if {$MB_ENABLE_GPIO} {
  set gpio [create_bd_cell -type ip -vlnv $XIP:axi_gpio:2.0 axi_gpio_0]
  lappend AXI_PERIPH $gpio

  set_property -dict [list \
    CONFIG.C_GPIO_WIDTH 32 \
  ] $gpio
}

## COMPONENT: AXI UART #########################################################
if {$MB_ENABLE_UART} {
  set uart [create_bd_cell -type ip -vlnv $XIP:axi_uartlite:2.0 axi_uartlite_0]
  lappend IRQ_PERIPH $uart
  lappend AXI_PERIPH $uart

  set_property -dict [list \
    CONFIG.C_BAUDRATE 115200 \
    CONFIG.C_DATA_BITS 8 \
    CONFIG.C_ODD_PARITY 0 \
    CONFIG.C_USE_PARITY 0 \
  ] $uart
}

## AUTOMATION: MicroBlaze ######################################################
apply_bd_automation \
  -rule xilinx.com:bd_rule:microblaze \
  -config { \
    preset "Microcontroller" \
    local_mem "8KB" \
    ecc "None" \
    cache "None" \
    debug_module "None" \
    axi_periph "Enabled" \
```

```
    axi_intc "1" \
    clk "New External Port (100 MHz)" \
  } $mb

## AUTOMATION: AXI ############################################################
foreach axiPeriph $AXI_PERIPH {
  apply_bd_automation \
    -rule xilinx.com:bd_rule:axi4 \
    -config { \
      Master "/microblaze_0 (Periph)" \
      intc_ip "/microblaze_0_axi_periph" \
      Clk_xbar "Auto" \
      Clk_master "Auto" \
      Clk_slave "Auto" \
    } [get_bd_intf_pins $axiPeriph/S_AXI]
}

## INTERRUPTS #################################################################
if {[llength $IRQ_PERIPH] > 0} {
  set irqconcat [get_bd_cells microblaze_0_xlconcat]
  set_property -dict [list \
    CONFIG.NUM_PORTS [llength $IRQ_PERIPH] \
  ] $irqconcat

  set i 0
  foreach irqPeriph $IRQ_PERIPH {
    connect_bd_net \
      [get_bd_pins $irqPeriph/interrupt] \
      [get_bd_pins $irqconcat/In$i]

    incr i
  }
}

## EXTERNAL CONNECTIONS #######################################################
set_property NAME ext_clk [get_bd_ports Clk]
make_bd_pins_external [get_bd_pins rst_*/ext_reset_in]
set_property NAME ext_rst_n [get_bd_ports ext_reset_in*]

if {$MB_ENABLE_UART} {
  make_bd_pins_external [get_bd_pins $uart/rx]
  set_property NAME mb_uart_rxd [get_bd_ports rx*]
  make_bd_pins_external [get_bd_pins $uart/tx]
  set_property NAME mb_uart_txd [get_bd_ports tx*]
}

if {$MB_ENABLE_GPIO} {
  # make_bd_intf_pins_external [get_bd_intf_pins $gpio/GPIO]
  # set_property NAME mb_gpio [get_bd_intf_ports GPIO]
  make_bd_pins_external  [get_bd_pins $gpio/GPIO_io_i]
  set_property NAME mb_gpio_i [get_bd_ports GPIO_io_i*]
  make_bd_pins_external  [get_bd_pins $gpio/GPIO_io_t]
  set_property NAME mb_gpio_t [get_bd_ports GPIO_io_t*]
  make_bd_pins_external  [get_bd_pins $gpio/GPIO_io_o]
  set_property NAME mb_gpio_o [get_bd_ports GPIO_io_o*]
}

## GROUP CELLS ################################################################
group_bd_cells $bd_name [get_bd_cells]

## SAVE DESIGN ################################################################
regenerate_bd_layout
validate_bd_design
```

```
save_bd_design
write_bd_tcl generated_${bd_name}.tcl -force

## COMPONENT: TMR Manager ##########################################################
if {$use_tmr} {
  set tmr [ \
    create_bd_cell \
      -type ip \
      -vlnv $XIP:tmr_manager:1.0 \
      $bd_name/tmr_manager_0 \
  ]

  set_property -dict [list \
    CONFIG.C_LMB_AWIDTH 32 \
    CONFIG.C_LMB_DWIDTH 32 \
    CONFIG.C_MAGIC1 0x00 \
    CONFIG.C_MAGIC2 0x00 \
    CONFIG.C_NO_OF_COMPARATORS 1 \
    CONFIG.C_UE_IS_FATAL 0 \
    CONFIG.C_STRICT_MISCOMPARE 0 \
    CONFIG.C_USE_DEBUG_DISABLE 0 \
    CONFIG.C_USE_TMR_DISABLE 0 \
    CONFIG.C_WATCHDOG 0 \
    CONFIG.C_WATCHDOG_WIDTH 30 \
    CONFIG.C_SEM_INTERFACE 0 \
    CONFIG.C_SEM_ASYNC 0 \
    CONFIG.C_SEM_HEARTBEAT_WATCHDOG 0 \
    CONFIG.C_SEM_HEARTBEAT_WATCHDOG_WIDTH 10 \
    CONFIG.C_SEM_INTERFACE_TYPE 2 \
    CONFIG.C_BRK_DELAY_WIDTH 0 \
    CONFIG.C_BRK_DELAY_RST_VALUE 0x00000000 \
    CONFIG.C_COMPARATORS_MASK 0 \
    CONFIG.C_MASK_RST_VALUE 0xFFFFFFFFFFFFFFFF \
    CONFIG.C_TEST_COMPARATOR 0 \
  ] $tmr

  ## AUTOMATION: TMR Manager ######################################################

  # Options:
    # bram: "Local"|"Common With ECC" (LMB Memory Configuration)
    # wd: "None"|"Internal" (Software Watchdog)
    # sem_if: "None"|"Included"|"External" (SEM Interface)
    # sem_wd: "0"|"1" (SEM Heartbeat Watchdog)
    # brk: "0"|"1" (Reconfiguration Delay)
    # mask: "0"|"1" (Comparator Test)
    # inject: "0"|"1" (Fault Injection)

  apply_bd_automation \
    -rule xilinx.com:bd_rule:tmr \
    -config { \
      bram "Local" \
      wd "None" \
      sem_if "None" \
      sem_wd "0" \
      brk "1" \
      mask "0" \
      inject "0" \
    } $tmr

  ## EXTERNAL RESET ###############################################################
  move_bd_cells [get_bd_cells /] [get_bd_cells ${bd_name}/rst_Clk_100M]
  set_property NAME reset_generator [get_bd_cells rst_Clk_100M]
  set rst [get_bd_cells /reset_generator]
```

```
    ## VALIDATE AND SAVE ############################################################
    regenerate_bd_layout
    validate_bd_design
    save_bd_design
    write_bd_tcl generated_${bd_name}_tmr.tcl -force
  }

  ## GENERATE IP FILES ###########################################################
  generate_target all $bd_file -quiet

  export_ip_user_files -of_objects $bd_file -no_script -sync -force -quiet
  create_ip_run $bd_file

  export_simulation \
    -of_objects $bd_file \
    -directory ${prj_name}/${prj_name}.ip_user_files/sim_scripts \
    -ip_user_files_dir ${prj_name}/${prj_name}.ip_user_files \
    -ipstatic_source_dir ${prj_name}/${prj_name}.ip_user_files/ipstatic \
    -lib_map_path [list \
      {modelsim=${prj_name}/${prj_name}.cache/compile_simlib/modelsim} \
      {questa=${prj_name}/${prj_name}.cache/compile_simlib/questa} \
      {ies=${prj_name}/${prj_name}.cache/compile_simlib/ies} \
      {vcs=${prj_name}/${prj_name}.cache/compile_simlib/vcs} \
      {riviera=${prj_name}/${prj_name}.cache/compile_simlib/riviera}\
    ] \
    -use_ip_compiled_libs -force -quiet

  ## ASSOCIATE ELF FILE ##########################################################
  set elf_path ../swt/${bd_name}.elf

  set elf_syn [add_files -quiet -fileset [get_filesets sources_1] $elf_path]
  set_property SCOPED_TO_REF ${bd_name} ${elf_syn}
  set_property SCOPED_TO_CELLS [get_bd_cells -hier microblaze_0] ${elf_syn}

  set elf_sim [add_files -quiet -fileset [get_filesets sim_1] $elf_path]
  set_property SCOPED_TO_REF ${bd_name} ${elf_sim}
  set_property SCOPED_TO_CELLS [get_bd_cells -hier microblaze_0] ${elf_sim}
}
```

C.2 Firmware

In this section are presented the main C and assembly files that, used with the AES C files presented in Appendix B, were compiled in an Executable and Linkable Format (ELF) file and therefore loaded on the microprocessors.

Depending on the configuration used, single or triplicated Microblaze, the define `TMR_ENABLE` was set properly.

C.2.1 `test_micro.c`

```
#include <stdint.h>
#include "aes.h"

#define TMR_ENABLED    1

#define ECB            0
#define ECB_ENC        1
#define ECB_DEC        0

#define READ_REQUEST   0
#define BLOCKS_ENCODED 255

#define GPIO_BASEADDR  0x40000000
#define GPIO_OFF_DATA  0x0000
#define GPIO_OFF_TRI   0x0004

#define GPIO_DATA_REG  (volatile uint32_t*)(GPIO_BASEADDR + GPIO_OFF_DATA)
#define GPIO_TRI_REG   (volatile uint32_t*)(GPIO_BASEADDR + GPIO_OFF_TRI)

static uint32_t gpio_read(void);
static void gpio_write(uint32_t data);

static void get_data(uint8_t* buf, int size);
static void set_data(uint8_t* buf, int size);

#if TMR_ENABLED
extern void _xtmr_manager_initialize();
#endif

int main() {
  uint8_t key[Ks]; // Container for Key
  uint8_t buf[Bs]; // Container for Data

#if TMR_ENABLED
  _xtmr_manager_initialize();
#endif

  while (1) {
    // Get key from GPIOs.
    get_data(key, Ks);
    set_data(key, Ks);

    int i;
    for (i = 0; i < BLOCKS_ENCODED; ++i) {
#if ECB | ECB_ENC
```

```
      // Test ECB Encoding
      get_data(buf, Bs);
      ecb_encrypt(buf, buf, key);
      set_data(buf, Bs);
#endif
#if ECB | ECB_DEC
      // Test ECB Decoding
      get_data(buf, Bs);
      ecb_decrypt(buf, buf, key);
      set_data(buf, Bs);
#endif
    }
  }

  return 0;
}

static uint32_t gpio_read(void) {
  return *GPIO_DATA_REG;
}

static void gpio_write(uint32_t data) {
  *GPIO_DATA_REG = data;
  *GPIO_TRI_REG  = 0xBADC0FFE;
  *GPIO_TRI_REG  = 0xFFFFFFFF;
}

static void get_data(uint8_t* buf, int size) {
  int i;
  for (i = 0; i < size; i += 4) {
    gpio_write(READ_REQUEST);
    uint32_t data = gpio_read();
    buf[i + 3] = (data >>  0) & 0xFF;
    buf[i + 2] = (data >>  8) & 0xFF;
    buf[i + 1] = (data >> 16) & 0xFF;
    buf[i + 0] = (data >> 24) & 0xFF;
  }
}

static void set_data(uint8_t* buf, int size) {
  int i;
  for (i = 0; i < size; i += 4) {
    uint32_t data =
      ((buf[i + 3] <<  0) & 0x000000FF) |
      ((buf[i + 2] <<  8) & 0x0000FF00) |
      ((buf[i + 1] << 16) & 0x00FF0000) |
      ((buf[i + 0] << 24) & 0xFF000000);
    gpio_write(data);
  }
}
```

C.2.2 mb_recovery.S

```
/*****************************************************************************
 * TMR Manager recovery routines:
 *    - Break Handler
 *    - Reset Handler
 *    - Initialize
 *****************************************************************************/
#define BASE_VECTORS  0x00000000

#define XTMR_BASEADDR 0x44a00000
#define XTMR_CR       (XTMR_BASEADDR + 0x00)
#define XTMR_FFR      (XTMR_BASEADDR + 0x04)
#define XTMR_CMR0     (XTMR_BASEADDR + 0x08)
#define XTMR_CMR1     (XTMR_BASEADDR + 0x0c)
#define XTMR_BDIR     (XTMR_BASEADDR + 0x10)
#define XTMR_SEMSR    (XTMR_BASEADDR + 0x14)
#define XTMR_SEMSSR   (XTMR_BASEADDR + 0x18)
#define XTMR_SEMIMR   (XTMR_BASEADDR + 0x1c)
#define XTMR_WR       (XTMR_BASEADDR + 0x20)
#define XTMR_RFSR     (XTMR_BASEADDR + 0x24)
#define XTMR_CSCR     (XTMR_BASEADDR + 0x28)
#define XTMR_CFIR     (XTMR_BASEADDR + 0x2c)

#define XTMR_MAGIC1   0x46
#define XTMR_MAGIC2   0x73
#define XTMR_CR_MAGIC 0x00017346
#define XTMR_CR_VAL1  0x00010046
#define XTMR_CR_VAL2  0x00007300

#define REG_VAR(reg)  XTMR_Manager_ ## reg
#define SAVE_REG(reg) swi reg, r0, REG_VAR(reg)
#define LOAD_REG(reg) lwi reg, r0, REG_VAR(reg)

/*
 * _xtmr_manager_initialize - Initialize break and reset vector.
 *
 * Save original cold reset vector to global variables.
 * Set up reset vector to branch to _xtmr_manager_reset.
 * Set up break vector to branch to _xtmr_manager_break.
 *
 */

.global _xtmr_manager_initialize
.section .text
.align 2
.ent _xtmr_manager_initialize
.type _xtmr_manager_initialize, @function
_xtmr_manager_initialize:
  /* Push to Stack */
  addik r1, r1, -16
  swi   r6, r1, 0
  swi   r7, r1, 4
  swi   r8, r1, 8
  swi   r9, r1, 12

  /* Clear Registers */
  ori r9, r0, XTMR_CR_MAGIC
  swi r9, r0, XTMR_CR
  swi r0, r0, XTMR_FFR
  swi r0, r0, XTMR_BDIR
  swi r0, r0, XTMR_SEMIMR
```

```
  swi r0, r0, XTMR_RFSR
  swi r0, r0, XTMR_CSCR

  /* Save cold reset vector */
  addik r8, r0, BASE_VECTORS
  lwi r6, r8, 0
  lwi r7, r8, 4
  swi r6, r0, XTMR_Manager_ColdResetVector+0
  swi r7, r0, XTMR_Manager_ColdResetVector+4

  /* Change reset vector */
  ori r6, r0, _xtmr_manager_reset
  bsrli r6, r6, 16
  ori r6, r6, 0xb0000000
  ori r7, r0, _xtmr_manager_reset
  andi r7, r7, 0xffff
  ori r7, r7, 0xb8080000
  swi r6, r8, 0
  swi r7, r8, 4

  /* Initialize break vector */
  ori r6, r0, _xtmr_manager_break
  bsrli r6, r6, 16
  ori r6, r6, 0xb0000000
  ori r7, r0, _xtmr_manager_break
  andi r7, r7, 0xffff
  ori r7, r7, 0xb8080000
  swi r6, r8, 0x14
  swi r7, r8, 0x18

  /* Pop from Stack */
  lwi   r6, r1, 0
  lwi   r7, r1, 4
  lwi   r8, r1, 8
  lwi   r9, r1, 12
  addik r1, r1, 16

  /* Clear MSR BIP by performing an RTBD instead of RTSD */
  rtbd r15, 8
  nop
.end _xtmr_manager_initialize

/*
 * _xtmr_manager_break - Handler for recovery break from the TMR Manager.
 *
 * Save stack pointer in global register.
 * Save all registers that represent the processor internal state.
 * Flush or invalidate all internal cached data: D-cache, I-cache, BTC and UTLB.
 * Call break handler in C code.
 * Suspend processor to signal TMR Manager that it should perform a reset.
 *
 * Handler notes:
 * - There is no need to save exception registers (EAR, ESR, BIP, EDR), since
 *   when the MSR EIP bit is set, break is blocked.
 */

.global _xtmr_manager_break
.section .text
.align 2
.ent _xtmr_manager_break
.type _xtmr_manager_break, @function
_xtmr_manager_break:
  /* Save context to stack */
```

```
  SAVE_REG(r1)
  SAVE_REG(r2)
  SAVE_REG(r3)
  SAVE_REG(r4)
  SAVE_REG(r5)
  SAVE_REG(r6)
  SAVE_REG(r7)
  SAVE_REG(r8)
  SAVE_REG(r9)
  SAVE_REG(r10)
  SAVE_REG(r11)
  SAVE_REG(r12)
  SAVE_REG(r13)
  SAVE_REG(r14)
  SAVE_REG(r15)
  SAVE_REG(r16)
  SAVE_REG(r17)
  SAVE_REG(r18)
  SAVE_REG(r19)
  SAVE_REG(r20)
  SAVE_REG(r21)
  SAVE_REG(r22)
  SAVE_REG(r23)
  SAVE_REG(r24)
  SAVE_REG(r25)
  SAVE_REG(r26)
  SAVE_REG(r27)
  SAVE_REG(r28)
  SAVE_REG(r29)
  SAVE_REG(r30)
  SAVE_REG(r31)
  mfs r1, rmsr
  swi r1, r0, XTMR_Manager_rmsr

  /* Suspend MicroBlaze to signal that a recovery reset should be done */
  suspend
  nop
  nop
  nop
  nop
.end _xtmr_manager_break


/*
 * _xtmr_manager_reset - Handler for recovery reset issued by TMR Manager.
 *
 * Restore stack pointer from global register.
 * Restore MSR to turn on caches.
 * Call reset handler in C code.
 * If C code returns 0, represnting cold reset, jump to saved cold reset vector.
 * Restore all registers that represent the processor internal state.
 * Return from break to resume execution.
 *
 */

.global _xtmr_manager_reset
.section .text
.align 2
.ent _xtmr_manager_reset
.type _xtmr_manager_reset, @function
_xtmr_manager_reset:
  /* Turn on caches if they are used */
  lwi r1, r0, XTMR_Manager_rmsr
```

```
  mts rmsr, r1
  bri 4

  /* Clear Registers */
  ori r9, r0, XTMR_CR_MAGIC
  swi r9, r0, XTMR_CR
  swi r0, r0, XTMR_FFR
  swi r0, r0, XTMR_BDIR
  swi r0, r0, XTMR_SEMIMR
  swi r0, r0, XTMR_RFSR
  swi r0, r0, XTMR_CSCR

  /* Restore context from stack and return from break */
  LOAD_REG(r1)
  LOAD_REG(r2)
  LOAD_REG(r3)
  LOAD_REG(r4)
  LOAD_REG(r5)
  LOAD_REG(r6)
  LOAD_REG(r7)
  LOAD_REG(r8)
  LOAD_REG(r9)
  LOAD_REG(r10)
  LOAD_REG(r11)
  LOAD_REG(r12)
  LOAD_REG(r13)
  LOAD_REG(r14)
  LOAD_REG(r15)
  LOAD_REG(r16)
  LOAD_REG(r17)
  LOAD_REG(r18)
  LOAD_REG(r19)
  LOAD_REG(r20)
  LOAD_REG(r21)
  LOAD_REG(r22)
  LOAD_REG(r23)
  LOAD_REG(r24)
  LOAD_REG(r25)
  LOAD_REG(r26)
  LOAD_REG(r27)
  LOAD_REG(r28)
  LOAD_REG(r29)
  LOAD_REG(r30)
  LOAD_REG(r31)

  /* Return from break to resume execution */
  rtbd r16, 8
  nop
.end _xtmr_manager_reset


/* Declarations of global variables used by the recovery functionality */
.section .data
.align 2
.global XTMR_Manager_ColdResetVector
.global XTMR_Manager_InstancePtr
.global XTMR_Manager_rmsr
.global XTMR_Manager_r1
.global XTMR_Manager_r2
.global XTMR_Manager_r3
.global XTMR_Manager_r4
.global XTMR_Manager_r5
.global XTMR_Manager_r6
```

```
.global XTMR_Manager_r7
.global XTMR_Manager_r8
.global XTMR_Manager_r9
.global XTMR_Manager_r10
.global XTMR_Manager_r11
.global XTMR_Manager_r12
.global XTMR_Manager_r13
.global XTMR_Manager_r14
.global XTMR_Manager_r15
.global XTMR_Manager_r16
.global XTMR_Manager_r17
.global XTMR_Manager_r18
.global XTMR_Manager_r19
.global XTMR_Manager_r20
.global XTMR_Manager_r21
.global XTMR_Manager_r22
.global XTMR_Manager_r23
.global XTMR_Manager_r24
.global XTMR_Manager_r25
.global XTMR_Manager_r26
.global XTMR_Manager_r27
.global XTMR_Manager_r28
.global XTMR_Manager_r29
.global XTMR_Manager_r30
.global XTMR_Manager_r31

XTMR_Manager_ColdResetVector:
  .long 0
  .long 0
XTMR_Manager_InstancePtr:
  .long 0
XTMR_Manager_rmsr:
  .long 0
XTMR_Manager_r1:
  .long 0
XTMR_Manager_r2:
  .long 0
XTMR_Manager_r3:
  .long 0
XTMR_Manager_r4:
  .long 0
XTMR_Manager_r5:
  .long 0
XTMR_Manager_r6:
  .long 0
XTMR_Manager_r7:
  .long 0
XTMR_Manager_r8:
  .long 0
XTMR_Manager_r9:
  .long 0
XTMR_Manager_r10:
  .long 0
XTMR_Manager_r11:
  .long 0
XTMR_Manager_r12:
  .long 0
XTMR_Manager_r13:
  .long 0
XTMR_Manager_r14:
  .long 0
XTMR_Manager_r15:
  .long 0
```

```
XTMR_Manager_r16:
  .long 0
XTMR_Manager_r17:
  .long 0
XTMR_Manager_r18:
  .long 0
XTMR_Manager_r19:
  .long 0
XTMR_Manager_r20:
  .long 0
XTMR_Manager_r21:
  .long 0
XTMR_Manager_r22:
  .long 0
XTMR_Manager_r23:
  .long 0
XTMR_Manager_r24:
  .long 0
XTMR_Manager_r25:
  .long 0
XTMR_Manager_r26:
  .long 0
XTMR_Manager_r27:
  .long 0
XTMR_Manager_r28:
  .long 0
XTMR_Manager_r29:
  .long 0
XTMR_Manager_r30:
  .long 0
XTMR_Manager_r31:
  .long 0
```

List of Figures

2.1 Floating Gate NMOS Transistor: The accumulation of charges in the Floating Gate prevents the transistor from working as expected, eventually, the value stored is changed when the charge exceed the threshold. 4

2.2 SRAM Cell: the effect of a particle striking through one of the M1..M4 transistors could flip the value stored the memory cell by the positive feedback of the structure. 5

2.3 Single Event Effects on transistors: the effect of striking particles can activate the transistors. 9

2.4 Single Event Induced Burnout on a MOS Transistor: the parasitic bipolar structure is excited, the followed by a positive feedback that increase the temperatures. 11

2.5 Single Event Gate Rupture on a MOS transistor: the dielectric present in the gate of the transistor is pierced. 12

2.6 Single Event Latch-Up on a PNPN thyristor structure: the particle excites the implicit thyristor structure that starts conducting due to the positive feedback. 12

2.7 Single Event Upset in a LUT: the logic function implemented is changed 14

2.8 SEU on a PIP: an high impedance path is now connected 15

2.9 SEU on a MUX: another input is selected 16

2.10 SEU on a Buffer: an high impedance output is now driven 16

3.1 Duplicate with Comparison scheme: the systems are duplicated and their outputs compared for errors 19

3.2 N-Modular Redundancy scheme: the systems are replicated N times, the voter decides the correct output using a majority voter scheme . . 19

3.3 Block TMR scheme: a block is triplicated, including its memory elements, and then a voter decides the correct output using a majority voter scheme 20

3.4 Reliablity comparison between a single system and a triplicated system with BTMR, the triplicated system is more likely to fail after a period of time 21

3.5 Distributed TMR scheme: the block is triplicated at each step, the inputs to the next step are always voted and the correct state is always restored from the voted output 22

3.6 Information Redundancy technique: a redundant part is added to the original data 23

3.7 Time Redundancy in Software: the same operation is repeated multiple times and then the result is compared and voted 25

3.8 Time Redundancy in Hardware: delays are added to repeat the same operation at different time instants 26

4.1 Dependability graph: the internal structure of dependability, divided by class 28

4.2 The product life cycle of a repairable system: it transitions from working to failure state and vice versa using failure and repair transitions 30

4.3 Markov chain representation of a repairable system: the transition probabilities are defined by the failure rate (λ) and the repair rate (μ) 32

4.4 Fault life cycle: the fault is activated into an error, the error is propagated into a misbehavior, depending on the type of the misbehavior the external effects can be different 33

5.1 SEM IP block description: all the input and outputs ports are listed . 44

5.2 Xilinx Essential Bits: configuration bits can be classified based on their priority levels 45

6.1 TMR Subsystem block diagram: the Microblaze core is triplicated as well as its peripherals and its memory, the outputs are voted 53

6.2 TMR Manager state transition in case of an error: starting from Voting mode, an error move the state to Lockstep mode, where the only two out of three processors are working with a Duplicate with Comparison scheme 54

6.3 Single Module architecture: the source data is read by the DUT, and compared with the golden results. Counters are placed to verify the operational and working state of the DUT. 60

6.4 Tabletop testing block diagram: the single modules are interfaced using SPI from a remote PC, that controls the fault injection procedure using JCM 61

6.5 Ground testing block diagram: the single modules are interfaced using a custom chip called SCA, that allows SPI communication over optical fiber. A remote PC serves for data storage, and it uses a Common Readout Unit to communicate over optical fiber. 62

A.1 Sensors layout of the upgraded ALICE-ITS: 2 outer layers, 2 middle layers and 3 inner layers 68

A.2 The Inner Barrel stave readout architecture: 9 ALPIDE chips are organized in a straight line, operating at 1.2 Gbit/s 69

A.3 The Outer Barrel stave readout architecture: 14 ALPIDE chips are organized in two half-staves, each one composed by 7 sensors operating at 0.4 Gbit/s 69

A.4 High level architecture of the upgraded ITS: the sensors are directly interfaced by the readout electronics, that is connected to Common Readout Units and synchronized with the ALICE Trigger System . . 70

A.5 Readout Unit architecture, it interface various sensors operating a different speeds with the Common Readout Unit, while being synchronized with the triggers coming from the ALICE Trigger System . 70

B.1 AES ECB Encryption: the same set of operations are applied to different blocks of the input data, treating them separately. 72

B.2 AES CBC Encryption: the output of the previous encoding is combined with the block of data encoded at the next step. 72

List of Tables

2.1 Comparison of FPGA technologies in terms of the main considered parameters in radiation environments. 5

7.1 Comparison between resource utilization of each core flavor 63

7.2 Tabletop testing results 64

7.3 Ground testing results 65

Acronyms

AES Advanced Encryption Standard.

ALICE A Large Ion Collider Experiment.

ASIC Application-Specific Integrated Circuit.

BRAM Block RAM.

BYU Brigham Young University.

CBC Cipher Block Chaining.

CERN Conseil Européen pour la Recherche Nucléaire.

CRAM Configuration RAM.

CRC Cyclic Redundancy Check.

CRU Common Readout Unit.

DED Double Error Detection.

DFF D Flip-Flop.

DRAM Distributed RAM.

DUT Device Under Test.

ECB Electronic Codebook.

ECC Error Correction Code.

ELF Executable and Linkable Format.

FIT Failure in Time.

FPGA Field-Programmable Gate Array.

FPU Floating Point Unit.

GBT-SCA Giga-Bit Transceiver - Slow Clock Adapter.

HEP High-Energy Physics.

I/O Input/Output.

ICAP Internal Configuration Access Port.

IP Intellectual Property.

ITS Inner Tracking System.

JCM JTAG Configuration Manager.

JTAG Joint Test Action Group.

LET Linear Energy Transfer.

LFA Linear Frame Address.

LHC Large Hadron Collider.

LUT Look-Up Table.

MBU Multiple-Bit Upset.

MCU Multiple-Cell Upset.

MMU Memory Management Unit.

MTBF Mean Time Between Failures.

MTTF Mean Time to Failure.

MTTFF Mean Time to First Failure.

MTTR Mean Time to Repair.

OTP One-Time Programmable.

PC Personal Computer.

PFA Physical Frame Address.

PIP Programmable Interconnect Point.

RAM Random Access Memory.

RISC Reduced Instruction Set Computer.

ROM Read-Only Memory.

RU Readout Unit.

SEB Single Event Induced Burnout.

SEC Single Error Correction.

SECDED Single Error Correction/Double Error Detection.

SEDR Single Event Dielectric Rupture.

SEE Single Event Effect.

SEFI Single Event Functional Interrupt.

SEGR Single Event Gate Rupture.

SEHE Single Event Hard Error.

SEL Single Event Latchup.

SEM Soft Error Mitigation.

SESB Single Event Snap-Back.

SET Single Event Transient.

SEU Single Event Upset.

SoC System on a Chip.

SPI Serial Peripheral Interface.

SRAM Static RAM.

TID Total Ionizing Dose.

TMR Triple Modular Redundancy.

TPC Time Projection Chamber.

UART Universal Asynchronous Receiver-Transmitter.

Bibliography

[1] F. Brosser and E. Milh, «Seu mitigation techniques for advanced reprogrammable fpga in space», 128, Master's thesis, 2014.

[2] European Cooperation for Space Standardization, *Ecss-q-hb-60-02a – techniques for radiation effects mitigation in asics and fpgas handbook*, 2016. [Online]. Available: `http://ecss.nl/get_attachment.php?file=2016/09/ECSS-Q-HB-60-02A1September2016.pdf`.

[3] ——, *Ecss-e-hb-10-12a – calculation of radiation and its effects and margin policy handbook*, 2013. [Online]. Available: `http://ecss.nl/get_attachment.php?file=handbooks/ecss-e-hb/ECSS-E-HB-10-12A17December2010.pdf`.

[4] V. Ferlet-Cavrois, L. W. Massengill, and P. Gouker, «Single event transients in digital cmos -a review», *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 1767–1790, Jun. 2013, ISSN: 0018-9499. DOI: `10.1109/TNS.2013.2255624`.

[5] Xilinx Inc., *Soft error mitigation using prioritized essential bits*, XAPP538 (v1.0) April 4, 2012. [Online]. Available: `https://www.xilinx.com/support/documentation/application_notes/xapp538-soft-error-mitigation-essential-bits.pdf`.

[6] D. G. Mavis and P. H. Eaton, «Soft error rate mitigation techniques for modern microcircuits», in *2002 IEEE International Reliability Physics Symposium. Proceedings. 40th Annual (Cat. No.02CH37320)*, 2002, pp. 216–225. DOI: `10.1109/RELPHY.2002.996639`.

[7] M. Pignol, «Dmt and dt2: Two fault-tolerant architectures developed by cnes for cots-based spacecraft supercomputers», in *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, 2006, pp. 1–10. DOI: `10.1109/IOLTS.2006.24`.

[8] B. Kannan and L. E. Parker, «Metrics for quantifying system performance in intelligent, fault-tolerant multi-robot teams», in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2007, pp. 951–958. DOI: `10.1109/IROS.2007.4399530`.

[9] IFIP Working Group 10.4, *Ifip working group 10.4 on dependable computing and fault tolerance*, 1980. [Online]. Available: http://www.dependability.org/wg10.4/.

[10] SEBoK Wiki, *Reliability, availability, and maintainability*, 2018. [Online]. Available: http://www.sebokwiki.org/wiki/Reliability,_Availability,_and_Maintainability.

[11] S. Y. Wang, S. E. Meyer, D. R. Saxena, and M. Y. Lai, «Applications of software fault tolerance testing methodology», in *Global Telecommunications Conference, 1996. GLOBECOM '96. 'Communications: The Key to Global Prosperity*, vol. 1, Nov. 1996, 670–674 vol.1. DOI: 10.1109/GLOCOM.1996.594446.

[12] Xilinx Inc., *Soft error mitigation controller v4.1*, PG036 September 30, 2015. [Online]. Available: www.xilinx.com/support/documentation/ip_documentation/sem/v4_1/pg036_sem.pdf.

[13] ——, *Ultrascale architecture soft error mitigation controller*, PG187 December 20, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/sem_ultra/v3_1/pg187-ultrascale-sem.pdf.

[14] A. Gruwell, P. Zabriskie, and M. Wirthlin, «High-speed fpga configuration and testing through jtag», in *2016 IEEE AUTOTESTCON*, Sep. 2016, pp. 1–8. DOI: 10.1109/AUTEST.2016.7589601.

[15] Xilinx Inc., *Microblaze processor reference guide*, UG984 (v2017.4) December 20, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug984-vivado-microblaze-ref.pdf.

[16] ——, *Microblaze triple modular redundancy (tmr) subsystem v1.0*, PG268 October 4, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/tmr/v1_0/pg268-tmr.pdf.

[17] H. Quinn, W. H. Robinson, P. Rech, M. Aguirre, A. Barnard, M. Desogus, L. Entrena, M. Garcia-Valderas, S. M. Guertin, D. Kaeli, F. L. Kastensmidt, B. T. Kiddie, A. Sanchez-Clemente, M. S. Reorda, L. Sterpone, and M. Wirthlin, «Using benchmarks for radiation testing of microprocessors and fpgas», *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2547–2554, Dec. 2015, ISSN: 0018-9499. DOI: 10.1109/TNS.2015.2498313.

[18] A. Caratelli, S. Bonacini, K. Kloukinas, A. Marchioro, P. Moreira, R. De Oliveira, and C. Paillard, «The gbt-sca, a radiation tolerant asic for detector control and monitoring applications in hep experiments», vol. 10, pp. C03034–C03034, Mar. 2015.

[19] S. Kushpil and A. Collaboration, «Upgrade of the alice inner tracking system», *Journal of Physics: Conference Series*, vol. 675, no. 1, p. 012 038, 2016. [Online]. Available: http://stacks.iop.org/1742-6596/675/i=1/a=012038.

[20] A. Szczepankiewicz, «Readout of the upgraded alice-its», *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 824, pp. 465–469, 2016, Frontier Detectors for Frontier Physics: Proceedings of the 13th Pisa Meeting on Advanced Detectors, ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2015.10.056. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900215012681.

[21] Advanced Encryption Standard, *Advanced encryption standard — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[22] Substitution–permutation network, *Substitution–permutation network — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Substitution%E2%80%93permutation_network.

[23] Block cipher mode of operation, *Block cipher mode of operation — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

[24] D. P. Siewiorek and P. Narasimhan, «Fault-tolerant architectures for space and avionics applications», Mar. 2018.

[25] Xilinx Inc., *Single-event upset mitigation selection guide*, XAPP987 (v1.0) March 18, 2008. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp987.pdf.

[26] M. Mager, «Alpide, the monolithic active pixel sensor for the alice its upgrade», *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 824, pp. 434–438, 2016, Frontier Detectors for Frontier Physics: Proceedings of the 13th Pisa Meeting on Advanced Detectors, ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2015.09.057. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900215011122.