Radiation Hardened System Design with Mitigation and Detection in FPGA

Hampus Sandberg
Master of Science Thesis in Electrical Engineering

Radiation Hardened System Design with Mitigation and Detection in FPGA
Hampus Sandberg
LiTH-ISY-EX--16/5008--SE

Supervisor: Dr. James Storey
CERN, BE-BI-BL

Examiner: Kent Palmkvist
ISY, Linköpings universitet

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Copyright © 2016 Hampus Sandberg
Abstract

FPGAs are attractive devices as they enable the designer to make changes to the system during its lifetime. This is important in the early stages of development when all the details of the final system might not be known yet. In a research environment like at CERN there are many FPGAs used for this very reason and also because they enable high speed communication and processing. The biggest problem at CERN is that the systems might have to operate in a radioactive environment which is very harsh on electronics. ASICs can be designed to withstand high levels of radiation and are used in many places but they are expensive in terms of cost and time and they are not very flexible. There is therefore a need to understand if it is possible to use FPGAs in these places or what needs to be done to make it possible.

Mitigation techniques can be used to avoid that a fault caused by radiation is disrupting the system. How this can be done and the importance of understanding the underlying architecture of the FPGA is discussed in this thesis. A simulation tool used for injecting faults into the design is proposed in order to verify that the techniques used are working as expected which might not always be the case. The methods used during simulation which provided the best protection against faults is added to a system design which is implemented on a flash based FPGA mounted on a board. This board was installed in the CERN Proton Synchrotron for 99 days during which the system was continuously monitored. During this time 11 faults were detected and the system was still functional at the end of the test. The result from the simulation and hardware test shows that with reasonable effort it is possible to use commercially available FPGAs in a radioactive environment.
Acknowledgments

I would like to thank James and Swann for giving me the opportunity to work on this project and learn about radiation effects in electronics. It has been a truly eye-opening experience for me. I would also like to thank all other colleagues and people I have met during my time at CERN for their kindness, openness and willingness to share their knowledge about physics and electronics.

Geneva, November 2016
Hampus Sandberg
Contents

Notation ix

1 Introduction 1
1.1 Motivation ........................................ 1
1.2 Purpose ........................................... 1
1.3 Problem formulation ................................. 2
1.4 Delimitations ...................................... 2

2 Theory 3
2.1 FPGA ............................................... 3
   2.1.1 Logic cells .................................... 3
   2.1.2 Hard blocks ................................... 4
   2.1.3 Configuration and process technology ............ 4
   2.1.4 Radiation tolerant FPGAs ....................... 6
2.2 Faults from radiation ............................... 6
   2.2.1 Ionizing radiation ............................. 7
   2.2.2 Single event effects ......................... 7
2.3 Fault mitigation techniques ......................... 8
   2.3.1 Triple modular redundancy ..................... 8
   2.3.2 Finite state machines ......................... 10
   2.3.3 Block RAM / FIFO .............................. 10
   2.3.4 Scrubbing .................................... 11
   2.3.5 Inputs and outputs ............................ 11
   2.3.6 VCC and GND ................................ 12
   2.3.7 Temporal redundancy ......................... 13
2.4 Fault detection .................................... 14
2.5 Fault Injection .................................... 15
   2.5.1 Simulation-based .............................. 16
   2.5.2 Emulation-based ............................... 17

3 Method 19
3.1 Fault injection tool ............................... 19
   3.1.1 Bit-flip faults ............................... 20
## Contents

3.1.2 Transient faults ........................................... 20
3.2 Simulated implementations ........................................... 20
  3.2.1 V1 - Baseline design ........................................... 21
  3.2.2 V2 - Baseline + One-hot FSM ................................. 21
  3.2.3 V3 - TMR .................................................. 21
  3.2.4 V4 - TMR with correct keepers .............................. 22
  3.2.5 V5 - TMR + One-hot FSM .................................... 23
3.3 Synthesis tool and optimizations .................................... 23
3.4 Metrics ................................................................. 24
  3.4.1 Resource usage ............................................... 24
  3.4.2 Simulation time and error count .............................. 24
3.5 Hardware implementation ........................................... 24
  3.5.1 Installation ................................................. 25
  3.5.2 Logging software ........................................... 27
  3.5.3 Dosimeter measurement ...................................... 27

4 Results ................................................................. 29
  4.1 Simulation ....................................................... 29
    4.1.1 Bit-flip fault injection .................................... 29
    4.1.2 Transient fault injection ................................... 32
    4.1.3 Resource usage ............................................ 33
  4.2 Hardware implementation ........................................ 35
    4.2.1 Fault detect events ......................................... 35
    4.2.2 Current draw ............................................... 37
    4.2.3 Dosimeter measurement ..................................... 37

5 Discussion ............................................................ 39
  5.1 Simulation ....................................................... 39
    5.1.1 Bit-flip injections ......................................... 40
    5.1.2 Transient injections ........................................ 41
    5.1.3 Resource usage ............................................ 42
  5.2 Hardware implementation ........................................ 43
    5.2.1 Block RAM and FIFO ........................................ 44
    5.2.2 FPGA orientation .......................................... 44
    5.2.3 Current draw ............................................... 44
    5.2.4 Dosimeter measurement ..................................... 45

6 Conclusions ........................................................... 47
  6.1 Simulation ....................................................... 47
  6.2 Hardware implementation ........................................ 48
  6.3 Future Work ..................................................... 48

Bibliography ............................................................... 49
## Notation

### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Field-programmable gate array</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-specific integrated circuit</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial off-the-shelf</td>
</tr>
<tr>
<td>LUT</td>
<td>Lookup table</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-access memory</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static random-access memory</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary metal–oxide–semiconductor</td>
</tr>
<tr>
<td>TMR</td>
<td>Triple modular redundancy</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite state machine</td>
</tr>
<tr>
<td>FIFO</td>
<td>First in, first out</td>
</tr>
<tr>
<td>DUT</td>
<td>Device under test</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time</td>
</tr>
<tr>
<td>UART</td>
<td>Universal asynchronous receiver/transmitter</td>
</tr>
<tr>
<td>LVDS</td>
<td>Low-voltage differential signaling</td>
</tr>
<tr>
<td>ECC</td>
<td>Error-correcting code</td>
</tr>
<tr>
<td>SEE</td>
<td>Single event effect</td>
</tr>
<tr>
<td>SET</td>
<td>Single event transient</td>
</tr>
<tr>
<td>SEU</td>
<td>Single event upset</td>
</tr>
<tr>
<td>RPL</td>
<td>Radio-photoluminescence</td>
</tr>
</tbody>
</table>
1

Introduction

1.1 Motivation

In a research environment like at CERN there are often needs to evaluate and prototype new ideas for sensors and detectors. How the system should operate can be hard to know from the start of the project and can often change over the lifetime as the users gain knowledge about the strength and limitations of the system. This requires a flexible system design where parameters can be changed quickly and new functionality can be added over time. The reprogrammability of a field-programmable gate array (FPGA) makes it a good candidate for this. In addition, they are nowadays available with a vast amount of resources at a reasonable price.

Operating electronics in a radioactive environment is a difficult task and for FPGAs there are some particular problems that have to be considered. Radiation-hardened application-specific integrated circuits (ASICs) can be designed to meet the tough requirements radiation present. The problem is that they require a long time to design, evaluate and produce at a cost that is not reasonable for small quantities. They also provide limited configurability once the specification for the chip has been locked down. It is therefore of interest to evaluate the possibility of using commercial-off-the-shelf (COTS) FPGAs with some clever firmware design to mitigate errors caused by radiation.

1.2 Purpose

The purpose of this thesis project is to evaluate different techniques that can be used in an FPGA design to make it more radiation hardened. Different comparisons in terms of resource usage and fault tolerance will be done in order to provide some guidelines on how to design a radiation-hardened system.
1.3 Problem formulation

The following questions will be answered in this thesis.

- What radiation-hardening techniques can be used when designing a read-out system with FPGAs?
- Where can/should radiation-hardening be implemented in the system?
- How are the FPGA resources used in the different implementations?
- How can errors be detected in an active system?

1.4 Delimitations

The following delimitations are made to keep a consistent and straightforward approach during the project.

- Minimal power consumption is not a goal of this thesis. It is assumed that the system architecture is designed to provide enough power.
- Performance in terms of data throughput or highest clock frequency is not the main objective. Focus is put on enough performance for the system to be functional.
- External circuitry, e.g. power supply circuits, oscillators, etc. that are needed to run the FPGA are not discussed in regards to radiation-hardening.
- The focus is on single event effects. Total ionizing dose effects are not evaluated because of the limited time for testing.
In this chapter relevant research and publications are discussed that influenced the design decisions.

2.1 FPGA

Modern FPGAs commercially available today provide a plethora of functionality and resources and it is difficult to provide a description that covers the many different FPGA implementations. The basic concept is a configurable array of hardware resources with simple functions like registers, logic and signal routing, but also more complex and specific functions like transceivers, processor cores and more. FPGA vendors provide series of FPGAs with different capabilities that designers can choose from depending on their specific need.

2.1.1 Logic cells

Logic cells are the basic blocks inside FPGAs which can be configured by the user to perform various functions. The actual implementation is different between the vendors and also sometimes between the FPGA series of a specific vendor. In general, a logic cell consists of lookup tables (LUTs), D-type flip-flops, multiplexers and other combinational logic, e.g. full-adders and carry propagation logic. A simplified schematic of a logic cell can be seen in figure 2.1.

Logic cells can be considered very simple when compared to hard blocks and the description the vendors provide on the actual implementation of the logic cell can therefore be quite accurate. This is key when it comes to understanding where there can be weak points that are susceptible to faults caused by radiation. It also allows the designer to take advantage of the available resources to make the design more robust.
A common factor for all of these building blocks is that they are built using different combinations of transistors. This is important to note as it is these transistors that form the active devices that provide the functionality of the system and any fault in them may cause a problem.

2.1.2 Hard blocks

The more complex functions are considered hard blocks as they have limited range of configurability and is often designed as a specific circuit inside the FPGA. Because of this, the designer does not know how the actual implementation in the chip is done. This is an advantage in terms of abstraction as there is no need to know all the details to use a complex block. Focus can therefore instead be put on other parts of the design, but in the case of radiation hardening this lack of insight should be considered as a drawback. In order to protect a system fully it is important to have access to as much detail about the circuits as possible to find any weak points. If the designer decides to use hard blocks it is important to do radiation tolerance tests in order to see how the circuitry is affected as done in [13], [17].

A common hard block available in FPGAs is block random access memory (RAM). This consists of a collection of memory cells together with logic and registers for reading and writing to the cells. One common use of the block RAM is as a first in, first out (FIFO) queue where data is stored in an organized order. To keep track on where data should be written and read in the FIFO, registers are needed as well as logic to increment and decrement these. However, the actual implementation of this will be different depending on what vendor and what FPGA is being used. The important thing to note is that using block RAMs or FIFOs will introduce extra circuitry that can make the system more susceptible to errors caused by radiation. If the vendor does not clearly specify that the block RAM is radiation hardened it has to be considered as a weak spot.

2.1.3 Configuration and process technology

The configuration of all the resources inside the FPGA require a memory element where the information is stored. This memory is what defines the system the designer wants and it is therefore important that the content is not altered during operation. Three common types of process technologies used for this configuration memory is SRAM, flash and antifuse.
In antifuse the memory element is a normally high resistance path which can be made into a low resistance path by applying a specific programming voltage across the antifuse. This means that once the element has been programmed it will stay in that state forever. For radiation hardening this is a good feature as it provides an assurance that the configuration will stay constant, but for reconfigurability it is bad as once a connection has been made it cannot be changed.

For SRAM based FPGAs the configuration is stored in a static random-access memory (SRAM) inside the chip. Because of the properties of this type of memory the configuration will be lost when power is removed from the memory and a process of loading the data into the memory has to be done when power is first applied. The memory cell is constructed with transistors that enables writing and reading to random locations at random times, hence the name, but there are multiple variations of SRAM memory cells. A schematic of a common six transistor SRAM memory cell can be seen in figure 2.2. As with most things, the FPGA vendors do not give an exact description on how the cells are designed and it is therefore hard to fully understand all weaknesses. The total number of cells varies depending on how much resources the FPGA has. As an example, the Xilinx 7 series FPGAs have between 17 million and 450 million cells [2]. If a memory cell like the one in figure 2.2 is used in one of these devices the total number of transistors will be at least between 102 million and 2.7 billion.

Flash based FPGAs instead use a flash memory cell where the information is stored as a charge in a floating gate [5]. An illustration of this can be seen in figure 2.3. One bit of information only requires one floating-gate transistor which leads to multiple advantages. The first and maybe the most obvious one is the reduction of transistors that can be in the order of 1/6 if compared to the six transistor SRAM cell. By storing the information as a charge in a floating gate the information will persist even after power has been removed from the system and the memory is therefore non-volatile. The programming is done by applying a certain voltage to the control gate that forces electrons into the floating gate where they are trapped because of the isolation around it. With electrons present in the floating gate the electric field from the control gate is partially canceled.
which changes the threshold voltage of the transistor. By having two threshold voltages, one where the floating gate is charged and one where it is not, it is possible to read out if a logical \textit{HIGH} or \textit{LOW} is stored in the cell.

One drawback of using flash memory cells is that they require extra processing steps to construct the floating gate. For the case of SRAM memory cells, it is possible to use a standard CMOS process for the complete memory cell. There can also be problem with limited lifetime for flash cells in radiation. In [19] they found that operating in an unbiased mode, which means read-only, enabled the cell to withstand a higher radiation dose. When the cell was biased (write-mode) they saw the device fail below 200 Gy, while unbiased it failed at around 500 Gy. This is a good result for flash based FPGAs as most of the time during operation the flash cells are only read. A more thorough radiation test of the different parts of a flash memory was presented in [4]. One of the conclusion from this paper is that after a certain dose the memory is no longer programmable. It is worth noting that the memory cell used in a flash based FPGA will be different from the ones presented in the papers. The only way to fully understand how a certain device will be affected by radiation is to perform a test on that actual device.

\subsection{2.1.4 Radiation tolerant FPGAs}

Some of the FPGA vendors provide what they call rad-hard or rad-tolerant devices that are supposed to be designed and tested to work in harsh environments [7], [22]. Using these devices might be necessary depending on the requirements of the system but it is worth noting that they are not as easy to purchase as regular COTS FPGAs. Radiation testing takes time and manpower which will increase the price of the FPGA. These advantages and disadvantages have to be considered when designing a system in order for it to meet all of the specifications.

\section{2.2 Faults from radiation}

Electronics which are subjected to radiation can experience faults in different ways. Some of these faults are discussed below.
2.2 Faults from radiation

2.2.1 Ionizing radiation

If radiation contains enough energy to free one or more electrons from atoms or molecules it is called ionizing radiation. The SI unit for this is the gray (Gy) which is defined as one joule of absorbed energy in one kilogram of matter [10]:

\[ 1 \text{ Gy} = 1 \frac{J}{kg} \]

With this definition in mind it is clear that ionizing radiation and electronics do not go well together. This is because energy will be deposited in the device (the matter) which logically might cause some type of disturbance, i.e. fault. On earth there is continuous cosmic radiation all around us, but because of the low level and the small dimensions of electronics it does not cause much disturbance. When electronics is put in an environment where the radiation level is order of magnitudes higher compared to cosmic the effect will be more visible.

Another common unit for ionization radiation is the rad which is defined as:

\[ 1 \text{ rad} = 0.01 \text{ Gy} \]

2.2.2 Single event effects

One group of faults caused by ionizing radiation is called single event effects (SEE). These can be divided into permanent faults, where the device is destroyed forever and temporary faults, which the device is able to recover from. The focus in this thesis is on temporary faults or more specifically, single event transients (SET) and single event upsets (SEU). Permanent faults are hard to mitigate once the hardware has been fixed because they are very dependent on the device in use.

The principle behind SEEs can be demonstrated with figure 2.4, which depicts a CMOS transistor with source, drain and gate. When a charged particle crosses through a region of the transistor it will cause electron-hole pairs to be generated. This means that positive charged holes and negative charged electrons are free to move around in that region. Unfortunately, it is not always that simple as the particle can create secondaries which in turn also can generate free electron-hole pairs along their path. Depending on where the electron-hole pairs were generated they may or may not give a significant effect. The most sensitive part is the region around the drain as depicted in figure 2.4 [12], [11]. This is because the drain usually has a bias voltage applied to it which creates a reverse-biased p-n junction with a depletion region between the drain and the substrate. Soon after the electron-hole pairs were generated, the charge will start to move around in order to restore an equilibrium. A simplified way of looking at the process is that the free electrons will start to move towards the drain which has the high potential applied to it. This will cause a transient current in the transistor and that is the main cause of SETs.

Depending on where a transistor is located in the circuit the SET may or may not give rise to a SEU. A SEU will occur if the transient is propagated and latched into a memory element. As an example one can look at transistor M3 in the SRAM...
cell in figure 2.2 and assuming node Q has a logic HIGH value. If a SET occurs at the drain of this transistor it will pull node Q low which, if it lasts for a long enough time, causes the internal latch of the memory cell to switch state.

2.3 Fault mitigation techniques

Fault mitigation techniques aims to take care of faults in such a way that they cause minimal or no damage at all. Some techniques for this will be presented here.

2.3.1 Triple modular redundancy

The most common method discussed in papers is redundancy or more specific, triple modular redundancy (TMR). The idea is to mask faults by having multiple copies of a module and perform a voting on the outputs using a majority voter. In the case of TMR there are three copies that are compared and if one of them is wrong the output of the voter will still give a correct value. If two are wrong the output of the voter will also be wrong. The majority voter can be designed as a simple logic function as seen in figure 2.5 and table 2.1. Because of this added logic there is a speed penalty that has to be considered as it may cause timing issues. Luckily for FPGAs the majority voter maps very well to one LUT which helps keeping the combinatorial path short [8], [18].
Table 2.1: Truth table for majority voter

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Input C</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The TMR described so far is on a bit logic level but it could also be applied on a higher module level. Figure 2.6 illustrates this where Module 1A, Module 1B and Module 1C are three copies of an arbitrary module with a specific function. An important aspect here is the dashed feedback path going from the output of the majority voter to the inputs of the modules. It might be needed in order to correct the errors introduced in any of the modules, otherwise the error will be kept forever. In some cases, where the modules do not have to store the current value for more than a clock cycle the feedback is not needed. This means that the input always contains a new value which should be processed by the modules and the results voted on. The input and output signal can here be more than one bit wide. One example of this could be triplicating a complete adder. The majority voter could then check the output value from the three adders and make a decision on what the actual value should be. This might cause issues as faults could occur inside this higher abstraction module that are not fixed but it limits the amount of resources required. The trade-off between abstraction level and resource usage is something the designer has to have in mind during the design process as implementing full TMR on bit level might not fit in the FPGA that is used.
2.3.2 Finite state machines

Finite state machines (FSMs) are of particular interest when it comes to fault mitigation as they often perform some type of control in the system. If a fault would occur in a FSM it might cause the system to reset or enter an invalid state that causes damage. A FSM consists of a set of memory elements where the current state is stored and combinational logic that sets the outputs and the next state. The state memory can be encoded in many different ways which will affect resource usage, timing and the susceptibility for radiation faults. Some common encodings are:

- Binary encoding: one unique binary number per state
- Gray encoding: similar to binary but two successive state encodings can only differ by one bit
- One-hot encoding: one bit per state is only active

An example on how the different encodings could look like for a FSM with four states can be seen in table 2.2. In [6] they have compared these three encodings in terms of resource usage, performance and reliability with the conclusion that one-hot have the lowest percentage of errors even though it requires more register compared to binary. The reasoning behind this is that for one-hot encoding there are many invalid states that can easily be detected as only one bit should be active at any given time. By adding a detection circuit that can sense these errors it could be possible to recover from them gracefully. If the invalid states are not taken care of, one-hot encoding could be worse which is why the actual implementation has to be studied and understood. Another disadvantage of one-hot encoding is that it requires one bit for every state. For small state-machines this will not be a big problem but as the number of states increases so does the number of bits needed. More bits mean that the sensitive cross section will increase and the system will be more susceptible to radiation effects.

<table>
<thead>
<tr>
<th>State</th>
<th>Binary</th>
<th>Gray</th>
<th>One-hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>State A</td>
<td>00</td>
<td>00</td>
<td>0001</td>
</tr>
<tr>
<td>State B</td>
<td>01</td>
<td>01</td>
<td>0010</td>
</tr>
<tr>
<td>State C</td>
<td>10</td>
<td>11</td>
<td>0100</td>
</tr>
<tr>
<td>State D</td>
<td>11</td>
<td>10</td>
<td>1000</td>
</tr>
</tbody>
</table>

*Table 2.2: Finite state machine encoding example*

2.3.3 Block RAM / FIFO

As discussed earlier the biggest problem with block RAM and FIFOs is that it is hard to know how they are actually implemented. Finding a good mitigation technique for these can therefore be difficult. Some FPGAs provide error detection
and/or correction circuitry as part of the block RAM which can be a good start. This way it could be possible to discard and maybe correct data that has been corrupted. It would also be possible to design a separate error detection and correction circuit where mitigation techniques discussed earlier are deployed. This could be combined with the most basic block RAM entity available to form a memory element which is potentially more robust than the one provided by the vendor.

As FIFOs are essentially block RAMs with some added registers and logic they are good candidates for mitigation techniques. A custom FIFO with TMR for the registers and error correction and detection added would provide better observability and protection than the default FIFO provided by the vendor tools.

### 2.3.4 Scrubbing

SRAM-based FPGAs contain a lot of memory cells to hold the configuration as discussed in section 2.1. A fault in any of these could potentially change the interconnections inside the FPGA and therefore also modify the system behavior. Because of the large number of these cells and their importance, care has to be taken to ensure any faults are mitigated. A first step would be to reload the content of the configuration memory with the correct values at given time intervals or whenever a fault has been detected. This is what is called scrubbing and it is a technique that is required in a SRAM-based FPGA in order to avoid errors [3]. Luckily this is a common feature in modern FPGAs nowadays and is sometimes called reconfiguration. Some even make use of error-correction codes (ECC) which enables the system to locate single-bit errors and detect multiple bit errors [21].

The important thing to note about scrubbing is that it will cause downtime of the system. Care must therefore be taken to make sure the scrubbing does not disturb the operation of the system in an intolerable way. Partial reconfiguration where only a subset of the configuration memory is rewritten at a time might decrease the downtime to a manageable level.

### 2.3.5 Inputs and outputs

The main purpose of fault mitigation is to remove singular points of failure in the system. This also includes the interface which the FPGA uses to communicate with the outside, i.e. the inputs and outputs of the actual chip. For the case of inputs one can understand that if a fault is present in the input circuitry for a pin, that fault will be propagated to the rest of system. Having a single point of failure in the beginning of a chain will of course make any mitigation done later useless. One technique that can be used here is to triplicate the number of input pins that are sensing the same signal coming to the chip. If one of the three inputs has an error it can be detected and mitigated before the signal is sent to the next stage.

Similarly, outputs from the FPGA can experience faults. This can be critical if the application is relying on the output signals to be fault-free and reliable. If all signals in the FPGA have been kept triplicated up until the outputs they
have to be brought together if only one signal should be routed to the external interface. Having three output buffers connected together and driving one signal on the circuit board could potentially be dangerous if one buffer wants to drive HIGH while another wants to drive LOW. This is basically causing a short circuit through the two output buffers. What needs to be done is to disable and put a buffer in a high-impedance state whenever it is disagreeing with the other two. A minority voter, which has the truth table seen in table 2.3, can be used to enable and disable the high-impedance state of the buffers. As can be seen in the table, whenever Main Input A is disagreeing with Input B and Input C the output goes HIGH.

Another option for both inputs and outputs would be to keep the triplicated signals outside the chip also. The problem with this is that it requires whatever is connected at the other end of the signal path to support three signals as input and output. A hopefully obvious drawback of triplicating inputs and outputs is that it requires three times the number of pins on the FPGA. This will have a huge impact on overall design as it means that a bigger FPGA with more pins has to be used which increases the cost and size. It also requires a board where pins are connected together three per group which basically means a custom board has to be made. In some applications this technique might be necessary in order to get the required reliability.

### 2.3.6 VCC and GND

The high flexibility of the FPGA fabric means that the components building up the logic cells and hard blocks will have inputs that might not be needed for a particular design. This means that they have to be driven to VCC or GND to enable or disable whatever the inputs might be controlling. A simple example is the chip-enable signal on registers. If this functionality is not needed, i.e. the register should store the data each clock cycle, the chip-enable signal has to be tied to VCC. How this is done will be different depending on what FPGA is used and the designer has to be aware of these extra signals. One potential implementation of this is having constant keeper signals at each logic cell or group of cell to which these extra signals can be routed. If the keeper signals are susceptible to faults

<table>
<thead>
<tr>
<th>Main Input A</th>
<th>Input B</th>
<th>Input C</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 2.3: Truth table for minority voter*
caused by radiation, there will be a single point of failure which is not good.

A mitigation technique for this would be to make sure all unused signals for the primitive components used are connected manually to known signals. This involves using one or three input pins that are tied to a known voltage externally, for example tied to ground. Using only one input pin will give rise to the same problem as discussed earlier but fortunately the constant can be shared inside the FPGA. It is therefore safe to say that three more pins should not affect the overall system design that much. The three inputs can then be used in the triplication circuitry inside the FPGA. If a commercially available FPGA board is used it might not have any pins connected to a constant voltage like this, but most boards have some exposed pins which means it should be possible to implement this without much hassle. The problem lies in the fact that the designer has to manually extract all the signals which should be tied to VCC or GND. This would be a very time consuming job and might also lead to issues with the synthesis and routing as it introduces a lot of extra signals.

### 2.3.7 Temporal redundancy

Temporal redundancy is taking advantage of the time span for which a fault is present. This is different from TMR which is a spatial redundancy, i.e. it is spread over an area whereas temporal is spread over time. The basic principle is to now triplicate the circuit but drive each copy with clocks that have slightly different phases. Figure 2.7 gives a simplified view of this where Clock A, Clock B and Clock C have different phases. A big drawback of this technique is that it requires three clocks to be generated and routed throughout the FPGA. The generation part would require some sort of clock management circuit which in itself could be sensitive to radiation and it might contain analog parts. Routing of the three clocks would require global nets which is a scarce resource inside FPGAs. This shows that it might not be an appropriate mitigation technique for some systems. The conclusion from [18] also suggest temporal redundancy does not provide greater reliability compared to regular TMR.
Fault detection is taking advantage of the triplicated signals that TMR introduce. By comparing the three signals a truth table as table 2.4 can be constructed. If all signals are the same it indicates no error and if one or two signals differ an error is present. The truth table can be mapped to the logic seen in figure 2.8. As the fault detector is basically a three input lookup table it can be easily implemented as a LUT inside the FPGA. The fault detect signal should reset at the next clock edge if the TMR was done correctly. This is because a fault should be corrected by the feedback path introduced by the TMR at the next clock edge.

Every register which has TMR applied to it will have a fault detect signal coming from a fault detector. This could easily add up to a lot of signals which all have to be routed to a common fault detect module where the counting takes place. In order to ease the implementation and reduce the number of signal paths, the signals are combined locally in the different modules. This is illustrated in figure 2.9 where each module itself can contain a similar combination circuit.

The biggest downside with this combination circuit is that if two or more fault detectors indicate a fault during the same clock cycle it will only count as one. This cannot be avoided without introducing too many signals in the design and a compromise has to be made. The important thing though is that it will still be able to detect when one or more faults occur. Another improvement that can be seen in figure 2.9 is the addition of a register in some fault detect signal paths. As we want to detect faults from modules that are physically located all
over the FPGA the path can be quite long. Adding a register in selected places will cut the long paths and help improve the overall performance. One thing to note here is that these extra registers do not have TMR and are therefore not protected. This can be seen as both a feature and a bug. If a fault occurs in one of these registers and it changes the value from LOW to HIGH it will be seen as a valid fault detect signal, which is good. On the other hand, if it changes the value from HIGH to LOW it will mask out a fault detect signal from somewhere else. For the latter it has to happen at the correct clock cycle, which is possible but it limits the probability.

All the fault detect signals and combined fault detect signals are routed to a fault detect controller. This is where the actual fault counting happens. The first step is to combine the remaining signals to one final fault detect signal. Whenever this signals goes HIGH a counter is incremented. One way to get some positional data for the fault detect would be to have multiple counters for different fault detect inputs. Again, a compromise has to be made where the number of signals and the wanted fault detect information has to be considered.

2.5 Fault Injection

Fault injection is a way of introducing one or many faults in a system in a controlled manner and observe the response from it. This can be done in many ways and care has be taken in order to understand the strengths and the limitations of the fault injection technique that is used. In [24] a thorough comparison is done between various fault injection techniques highlighting their benefits and drawbacks and categorizing them into the following categories:

- Hardware-based
- Software-based
- Hybrid
- Simulation-based
- Emulation-based
The first three categories require a system with actual hardware and/or software available and can therefore only be used at a late stage in the design process. This is not very useful for a prototype design where time and hardware iterations are limited. It is therefore of interest to use simulation tools and/or emulating the design with fault injection to get some valuable data on the performance of the system.

2.5.1 Simulation-based

In the case of simulation-based fault injection, a high-level model written in e.g. VHDL or Verilog can be used together with a test bench. This enables the designer to early on in the process start injecting faults and see that the desired behavior is observed. The actual injection can be done in multiple ways and they can be divided into two categories.

First there are methods that take advantage of capabilities in the HDL-language that is used and most often require some addition and/or modification to the source code. In [9] they use resolution functions, that is part of the VHDL language, to have multiple drivers controlling the value of a signal. This method requires very small modifications of the source code and is therefore an interesting approach but there is a limitation that makes it unsuited for injecting bit-flips faults. Resolution functions works by connecting one of many drivers to a given signal depending on a condition. This means that the actual signal is not really modified when a fault is injected, but rather it is replaced for a certain amount of time with a fault signal. This works great for stuck-at faults but not for bit-flips.

Similar to this is the use of saboteurs and mutants. The basic idea behind the first is to add components either in series or in parallel with the signal of interest, that are capable of modifying the signal value at any given time. Using resolution functions as described earlier can be seen as a saboteur placed in parallel with the signal.

In the case of mutants, the idea is to replace a functional component with a similar version that has some added capability and can therefore be considered a mutated version of the original. An example could be an AND-gate that will always give a LOW output when the mutation is activated using extra signals controlled by the test bench. The strength with especially mutants is that they can be designed to model many different fault modes, but they can be complicated to design and use in the system [14].

The other approach is to use commands that are built-in to the simulator to modify the value of the signals. This does not require modification of the source code but the functionality that is available depends on which simulator is used. The big advantage with this method is that fault injection can be done early in the design phase on a high-level model which will help the designer to find weak points in the system. It is also possible to use the same fault injector from the behavioral model all the way to the synthesized netlist with timing information included [20].

In the simulator ModelSim SE from Mentor Graphics there is a command called signal_force that can either deposit or freeze a value on a signal [1]. Doing
a freeze on a signal gives a similar result as resolution functions in VHDL in the sense that it disconnects the normal driver from the signal and replaces it with a fault driver. To connect the normal driver again a `signal_release` command has to be done. Using the `freeze` option, it is possible to inject transient faults but in order for it to cause a fault it has to occur across a clock edge. This is because the faulty signal value can only be latched into a storage element at this point in time. With the `deposit` option the value of the signal is instead changed to the value that was deposited on it and can therefore be used to inject bit-flip faults.

### 2.5.2 Emulation-based

The main drawback with simulation-based fault injection is the long time it takes to run the simulation on the computer. With emulation it is possible to remove this issue by running the design in an FPGA on a commercially available development board for example. This requires the design to be synthesizable for that particular FPGA, which might not be what is used for the final product, and can therefore be rather time consuming. To be able to inject faults, mutants or saboteurs have to be introduced and a also module that controls these to emulate faults happening in the design.
This chapter presents the different methods used for implementation and how the system was tested and verified.

3.1 Fault injection tool

The fault injection is done in a test bench where the device under test (DUT) is stimulated with a functional test vector and a fault test vector. In the functional monitor the output of the DUT is compared to a fault-free golden run. If a difference is seen in the monitor it indicates a weakness with the fault mitigation. The fault monitor is reading back the number of faults that are detected in the DUT which can later be compared against the number of injected faults. This gives an indication on how efficient the detection is. An overview of the setup can be seen in figure 3.1.

The functional test vector is the same for all implementations to give com-

![Figure 3.1: Fault injection test bench](image-url)
parable results. To simulate the randomness of faults caused by radiation, the injection node is taken randomly from a text file with all possible nodes. The delay time between two faults is also random but limited to a range to get a specific average injection rate. 1 ps is the lowest possible delay as that is the resolution used in the simulator. A summary of the delay times used during simulation can be seen in Table 3.1.

Longer delay time will lead to longer simulation time which has to be taken into consideration. This means that a compromise has to be made to be able to run a simulation with full coverage in a reasonable time. In [21] they have calculated a mean time between upsets in block RAMs in the order of hundreds of seconds. If similar values would be used in simulation it would take an extreme amount of time and is therefore not appropriate. The important thing during the simulation is to inject faults in all possible nodes to make sure the design can handle it. Much higher injection rates are therefore used.

<table>
<thead>
<tr>
<th>Simulation run</th>
<th>Delay range</th>
<th>Average injection rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1ps to 0.01us</td>
<td>200 000 000 faults/s</td>
</tr>
<tr>
<td>2</td>
<td>1ps to 0.1us</td>
<td>20 000 000 faults/s</td>
</tr>
<tr>
<td>3</td>
<td>1ps to 1us</td>
<td>2 000 000 faults/s</td>
</tr>
</tbody>
</table>

*Table 3.1: Fault delay and injection rate*

### 3.1.1 Bit-flip faults

To simulate bit-flip faults the command `signal_force` in the ModelSim simulator is used together with the `deposit` option. The current value of the node has to be read before injecting the fault as the faulty value should be the inverse of the current value. This is done with the command `init_signal_spy`.

### 3.1.2 Transient faults

Transient faults are injected with `signal_force` and using the `freeze` option. As discussed in section 2.5, freezing a signal disconnects the normal driver from the signal. To return to normal operation it has to be connected back again after some time has passed. This is done with a `signal_release` command and using a random delay time between 1ps and 100ps.

### 3.2 Simulated implementations

Five different implementations were used during the simulation test and they are presented below. The implementations are simulated before synthesis and therefore without any extra timing information. This is done to avoid any difference in the implementations caused by optimizations during synthesis.
3.2 Simulated implementations

**Figure 3.2: Schematic of baseline design**

**Figure 3.3: Schematic of triplicated registers**

### 3.2.1 V1 - Baseline design

This implementation is the functional baseline without any fault mitigation. The reasoning behind this was to have a reference for the later implementations and also to get an idea on how a design without protection would perform. A simple schematic can be seen in figure 3.2. For this version the enable signal (en) of registers can be used to hold values in the register for longer than one clock cycle.

### 3.2.2 V2 - Baseline + One-hot FSM

From the results in [6] one-hot encoded FSMs seemed to be more resistant to faults and therefore for this implementation all FSMs were modified to use this encoding. Previously the encodings were all set to be binary.

### 3.2.3 V3 - TMR

As a first and simple step all registers in the baseline design were triplicated and the output of these were fed through a majority voter. In figure 3.3 the basic idea can be seen. The enable signal of the registers can no longer be used. This is because the registers should always be updated with the voted value in order to clear out any faulty values. The multiplexer at the input is now instead acting as the enable signal in the sense that it can select whether to use the voted value or a new input value each clock cycle.

For all the assignment inside processes in the VHDL code, which indicates a register, a change was made to assign the value to the now triplicated register. An example code of this can be seen in code 3.1.
-- Without TMR
process (clk)
begin
  ...
someRegister <= some_value;
  ...
end process;

-- With TMR
process (clk)
begin
  ...
someRegister_a <= some_value;
someRegister_b <= some_value;
someRegister_c <= some_value;
  ...
end process;

Code 3.1: Code without and with triplicated registers

Looking at the elaborated design after these changes were done there were some registers that were not implemented as expected. Instead of having the keeper signal taken from the output of the majority voter it was taken from the output of the register as can be seen in figure 3.4. The problem with this is that any fault happening in one of the three registers will be kept there forever as the faulty value is fed back to the input instead of the correct voted value.

3.2.4 V4 - TMR with correct keepers

To solve the problem of the keeper signal in the previous implementation a default assignment was added at the beginning of the VHDL process as seen in

Figure 3.4: Schematic of triplicated register with wrong keeper signal
code 3.2. This makes sure that if no other assignment in a clock cycle is done in the process to the register it will assign the output value of the majority voter instead. With this modification the resulting circuit is as in figure 3.3.

```vhdl
process (clk)
begin
    -- Default values
    someRegister_a <= someRegister;
    someRegister_b <= someRegister;
    someRegister_c <= someRegister;
    ...
end process;
```

*Code 3.2: Code for triplicated register with correct keeper signal*

### 3.2.5 V5 - TMR + One-hot FSM

This is a combination of the TMR implementation with correct keepers and using one-hot encoded FSMs.

### 3.3 Synthesis tool and optimizations

Introducing TMR in the design is basically adding more resources than actually needed which the synthesis tool might try to optimize away. To avoid this, it is necessary to tell the tool not to remove the extra registers that have been added. Depending on what synthesis tool is used this might be different and the manual should be consulted to find the appropriate way of doing it. For Xilinx Vivado the attributes seen in code 3.3 can be used and for Microsemi Libero SoC Design code 3.4 can be used. Other settings might also affect how much the tool tries to optimize the design. For Xilinx Vivado the option `-fsm_extraction` should to be set to off and `-keep_equivalent_registers` should be enabled. This will ensure that the synthesis tool do not modify the design that is wanted.

```vhdl
attribute dont_touch : string;
attribute dont_touch of someRegister : signal is "true";
```

*Code 3.3: Attributes in Xilinx Vivado*

```vhdl
attribute syn_preserve : boolean;
attribute syn_preserve of someRegister : signal is true;
```

*Code 3.4: Attributes in Microsemi Libero SoC Design*
3.4 Metrics

To be able to compare the different implementations some common metrics are used.

3.4.1 Resource usage

The different implementations were designed to be synthesizable on a Xilinx Virtex 7 (part number XC7VX485T-2FFG1761C) that is mounted on a Xilinx VC707 Evaluation Kit. Synthesis was done using the built-in tool in Xilinx Vivado version 2015.4 and the attributes and settings described in section 3.3 were used to avoid unwanted optimization. Even though this is not the FPGA that is going to be used in the final design it can be used to compare the different implementations against each other. The two resource metrics that are used are:

- Slice LUT usage: gives an indication on how much combinational logic is needed
- Slice register usage: gives an indication on the number of registers needed

3.4.2 Simulation time and error count

For the fault simulation metric, it is interesting to know how long a simulation runs on average as it gives an indication on how long it takes before an injected fault causes an error. This time is normalized against the baseline design which means that a value > 1 indicates it is better and a value < 1 indicates it is worse than the baseline. The measurement used to compare the different implementations is the average simulation time. When the functional monitor senses a difference between the output of the DUT and the golden run it will break the simulation and restart it. The elapsed simulation time for that particular simulation is saved and an average of all these simulation times are used for comparison. This means that if an implementation is weak the simulation will break sooner and the average simulation time will be low.

The total number of detected faults is counted in the fault monitor. Ideally it should be exactly the same as the number of injected faults. A lower fault count than number of injected faults indicate that some faults occurred too close to each other in time and were combined into a single fault or that the detection is not working properly. If the fault count is higher there might be a TMR somewhere that is not implemented correctly with a feedback to remove the fault the next clock cycle.

3.5 Hardware implementation

Simulation is a good tool to understand the system and fix potential bugs, but at some point actual hardware is needed. For this thesis a board developed at CERN called GEFE (GBT-based Expandable Front-End) was used. This is designed to be
3.5 Hardware implementation

used as an acquisition board in radiation environment and therefore contains components that are made for this purpose. The GEFE is intended to operate in an environment with a total ionizing dose of up to 750 Gy [16], [15]. A photo of the board can be seen in figure 3.5 with annotations for the following important components:

- FPGA, Microsemi ProASIC3E - Flash based
- GBTx - Radiation tolerant transceiver chip for gigabit optical link
- VTRx - Radiation tolerant optical transceiver
- FEASTMP - Radiation tolerant DC/DC converter
- LHC4913 - Radiation tolerant low dropout regulator

3.5.1 Installation

The FPGA was programmed with a version of the firmware that resembles what will be used in the final readout system. The firmware was similar to the last simulated design in the sense that all registers were triplicated correctly and FSMs were one-hot encoded. With everything implemented the total FPGA logic core usage was 74.2% and 66.1% of the block RAM was used. To enable some control and to read back registers inside the FPGA a universal asynchronous receiver/transmitter (UART) link was added and connected to a cable going to the back-end control room. The cable to this room was about 150 meters long. To minimize any potential signal issues, the single-ended UART signals were converted to two low-voltage differential signaling (LVDS)-signals. Another issue was the limited installation time which meant that any modifications to the hardware during installation could only be minor. Luckily, everything worked
as intended and the logging software could be started in the control room on a
computer that was connected to the system.

The board was installed into a crate which was positioned in a rack below the
beam as can be seen in figure 3.6. The green ring highlights the crate in which the
board is installed. Just above the top of the picture is the pipe where the beam
is located inside. It is worth mentioning that the FPGA on the board is placed in
the same horizontal plane as the beam is traveling. If the board was flipped 90
degrees and placed vertically the results could be different. The decision to place
it horizontally was made as it would minimize the area of the FPGA which is
exposed to the beam. A simplified graphic to display the used horizontal position
is displayed in figure 3.7.

Figure 3.6: Location of the GEFE board at installation
3.5 Hardware implementation

3.5.2 Logging software

The logging over the UART had two parts. First the software would read back a register in the firmware with the fault detect count stored in it. Whenever this register was not equal to zero one or more fault events had happened in the registers inside the FPGA. The second part involved writing an incremental number to four different registers and reading them back. This made sure the system was still operational in the sense that the content of registers could be modified. All the communication between the computer and the board was written to a text file for later analysis.

The computer was also connected to a power supply that was providing power to the GEFE board through the long cable. Another logging software took care of continuously reading the voltage and current the power supply provided and writing it to a text file. This was done in order to see if the current consumption was affected during the test period. As faults caused by radiation might damage transistors in a way where they conduct an abnormal amount of current, it could potentially be sensed by looking at the total current used by the board. It also gives an indication on how all the components degrade over time, but given the short test period the effects from this might not be visible. An overview of the full system used for the test can be seen in figure 3.8.

3.5.3 Dosimeter measurement

In order to measure the dose the board is exposed to during the test, a passive dosimeter is installed in the same crate. The dosimeter is taken advantage of a phenomenon called radio-photoluminescence (RPL), where an irradiated sample which is exposed to UV-light emits visible light proportional to the dosage [23]. The sample is a small piece of glass that has been doped in a particular way. A great advantage of this type of passive dosimeter is that it is can be stored in a small plastic container close to the equipment of interest.

![Figure 3.7: Orientation of GEFE board inside the crate](image-url)
Figure 3.8: Overview of the hardware used for testing
In this chapter results from the simulation and the hardware implementation are presented.

4.1 Simulation

The result presented below is from the fault injection simulations.

4.1.1 Bit-flip fault injection

The results from the bit-flip fault injection simulations are presented here. Data for detected faults is not available for V1 and V2 as they do not include fault detection circuitry. For all implementations a fault was injected in each node ten times at random moments.

- Table 4.1 is for simulation with 1ps to 0.01us delay time.
- Table 4.2 is for simulation with 1ps to 0.1us delay time.
- Table 4.3 is for simulation with 1ps to 1us delay time.
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Average simulation time</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>2524417 ps</td>
<td>1</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>2482602 ps</td>
<td>0.983</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>1779831 ps</td>
<td>7.051</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>325534357 ps</td>
<td>128.954</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>326136923 ps</td>
<td>129.193</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Injected faults</th>
<th>Detected faults</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>192900</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>194490</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>194330</td>
<td>147653</td>
<td>75.981 %</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>194330</td>
<td>119213</td>
<td>61.346 %</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>195920</td>
<td>119551</td>
<td>61.020 %</td>
</tr>
</tbody>
</table>

Table 4.1: Results for bit-flip fault injection, 1ps to 0.01us delay time

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Average simulation time</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>6709337 ps</td>
<td>1</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>6740292 ps</td>
<td>1.005</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>52520693 ps</td>
<td>7.828</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>9669382850 ps</td>
<td>1441.183</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>9828875000 ps</td>
<td>1464.955</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Injected faults</th>
<th>Detected faults</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>192900</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>194490</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>194330</td>
<td>1523387</td>
<td>783.918 %</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>194330</td>
<td>168149</td>
<td>86.528 %</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>195920</td>
<td>168866</td>
<td>86.702 %</td>
</tr>
</tbody>
</table>

Table 4.2: Results for bit-flip fault injection, 1ps to 0.1us delay time
### 4.1 Simulation

#### Table 4.3: Results for bit-flip fault injection, 1ps to 1us delay time

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Average simulation time</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>20 275 689 ps</td>
<td>1</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>19 932 626 ps</td>
<td>0.983</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>386 704 489 ps</td>
<td>19.072</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>96 880 174 130 ps</td>
<td>4778.145</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>97 461 679 800 ps</td>
<td>4806.824</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Injected faults</th>
<th>Detected faults</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>192 900</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>194 490</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>194 330</td>
<td>15 575 043</td>
<td>8015.739 %</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>194 330</td>
<td>172 875</td>
<td>88.960 %</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>194 330</td>
<td>172 791</td>
<td>88.916 %</td>
</tr>
</tbody>
</table>
4.1.2 Transient fault injection

The results from the transient fault injection simulations are presented here. Data for detected faults is not available for $V1$ and $V2$ as they do not include fault detection circuitry. For all implementations a fault was injected in each node ten times at random moments.

- Table 4.4 is for simulation with 1ps to 0.01us delay time.
- Table 4.5 is for simulation with 1ps to 0.1us delay time.
- Table 4.6 is for simulation with 1ps to 1us delay time.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Average simulation time</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>20745 301 ps</td>
<td>1</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>21827 725 ps</td>
<td>1.052</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>984 291 150 ps</td>
<td>47.446</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>984 021 440 ps</td>
<td>47.433</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>982 697 340 ps</td>
<td>47.370</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Injected faults</th>
<th>Detected faults</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>192 900</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>194 490</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>194 330</td>
<td>1 378</td>
<td>0.709 %</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>194 330</td>
<td>1 409</td>
<td>0.725 %</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>195 920</td>
<td>1 452</td>
<td>0.741 %</td>
</tr>
</tbody>
</table>

Table 4.4: Results for transient fault injection, 1ps to 0.01us delay time

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Average simulation time</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>81 297 761 ps</td>
<td>1</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>83 869 585 ps</td>
<td>1.032</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>9 811 472 060 ps</td>
<td>120.686</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>9 716 287 430 ps</td>
<td>119.515</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>9 761 626 020 ps</td>
<td>120.073</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Injected faults</th>
<th>Detected faults</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>192 900</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>194 490</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>194 330</td>
<td>1 414</td>
<td>0.728 %</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>194 330</td>
<td>1 314</td>
<td>0.676 %</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>195 920</td>
<td>1 401</td>
<td>0.721 %</td>
</tr>
</tbody>
</table>

Table 4.5: Results for transient fault injection, 1ps to 0.1us delay time
4.1 Simulation

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Average simulation time</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>702 263 639 ps</td>
<td>1</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>740 092 194 ps</td>
<td>1.054</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>97 030 585 450 ps</td>
<td>138.168</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>96 775 913 880 ps</td>
<td>137.806</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>97 919 400 520 ps</td>
<td>139.434</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Injected faults</th>
<th>Detected faults</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 - Baseline</td>
<td>192 900</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V2 - Baseline+One-hot</td>
<td>194 490</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>V3 - TMR</td>
<td>194 330</td>
<td>1 368</td>
<td>0.704 %</td>
</tr>
<tr>
<td>V4 - TMR+keeper</td>
<td>194 330</td>
<td>1 435</td>
<td>0.738 %</td>
</tr>
<tr>
<td>V5 - TMR+One-hot</td>
<td>195 920</td>
<td>1 452</td>
<td>0.741 %</td>
</tr>
</tbody>
</table>

Table 4.6: Results for transient fault injection, 1ps to 1us delay time

4.1.3 Resource usage

Table 4.7 displays the resource usage for the different implementations that were used during simulation. This is the resource usage without counting the LUTs and registers inside the FIFOs that are provided by the vendor. In table 4.8 the extra resources introduced by the FIFOs are included. The LUT factor and Register factor are normalized to the baseline design.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>LUTs</th>
<th>LUT Factor</th>
<th>Registers</th>
<th>Register factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1-Baseline</td>
<td>4 463</td>
<td>1</td>
<td>6 660</td>
<td>1</td>
</tr>
<tr>
<td>V2-Baseline+One-hot</td>
<td>4 787</td>
<td>1.073</td>
<td>6 717</td>
<td>1.009</td>
</tr>
<tr>
<td>V3-TMR</td>
<td>33 288</td>
<td>7.459</td>
<td>20 591</td>
<td>3.092</td>
</tr>
<tr>
<td>V4-TMR+Keeper</td>
<td>22 882</td>
<td>5.127</td>
<td>20 591</td>
<td>3.092</td>
</tr>
<tr>
<td>V5-TMR+Keeper+One-hot</td>
<td>23 134</td>
<td>5.184</td>
<td>20 762</td>
<td>3.117</td>
</tr>
</tbody>
</table>

Table 4.7: Resource usage for simulated implementations without FIFOs included
<table>
<thead>
<tr>
<th>Implementation</th>
<th>LUTs</th>
<th>LUT Factor</th>
<th>Registers</th>
<th>Register factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1-Baseline</td>
<td>6057</td>
<td>1</td>
<td>9721</td>
<td>1</td>
</tr>
<tr>
<td>V2-Baseline+One-hot</td>
<td>6381</td>
<td>1.053</td>
<td>9778</td>
<td>1.006</td>
</tr>
<tr>
<td>V3-TMR</td>
<td>34,882</td>
<td>5.759</td>
<td>23,652</td>
<td>2.433</td>
</tr>
<tr>
<td>V4-TMR+Keeper</td>
<td>24,476</td>
<td>4.041</td>
<td>23,652</td>
<td>2.433</td>
</tr>
<tr>
<td>V5-TMR+Keeper+One-hot</td>
<td>24,728</td>
<td>4.083</td>
<td>23,823</td>
<td>2.451</td>
</tr>
</tbody>
</table>

*Table 4.8: Resource usage for simulated implementations with FIFOs included*
4.2 Hardware implementation

Results from the hardware implementation are presented below. The hardware was installed in the Proton Synchrotron particle accelerator at CERN on June 7, 2016 and was taken out on September 14, 2016. The total test time was 99 days during which 11 faults were detected. This gives an average injection rate of \(1.286 \times 10^{-6}\) faults/s. During this period the accelerator was using protons. Data presented here was captured using the setup outlined in section 3.5.

4.2.1 Fault detect events

Different events were observed during the acquisition period and they have been categorized into the following event types:

- Block RAM - Event that indicated a block RAM inside the FPGA has experienced a fault. This attested as a burst of random data being received over the UART by the back-end computer.
- Register - Event that caused the fault detect counter to increment.
- Loss of communication - Event that broke all communication to the GEFE board.

If an event required power to the board to be turned off and on again it was counted as a power cycle event. In table 4.9 all events are listed in chronological order with the event type and power cycling indicated. A graphical representation of the same data can be seen in figure 4.1 which shows a timeline from the start of the acquisition to the end.

<table>
<thead>
<tr>
<th>Date and time (UTC)</th>
<th>Event Type</th>
<th>Power Cycle Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-06-15 07:31:26</td>
<td>Block RAM</td>
<td>No</td>
</tr>
<tr>
<td>2016-06-19 17:29:52</td>
<td>Register</td>
<td>No</td>
</tr>
<tr>
<td>2016-06-27 03:11:32</td>
<td>Block RAM</td>
<td>No</td>
</tr>
<tr>
<td>2016-07-01 17:45:00</td>
<td>Register</td>
<td>No</td>
</tr>
<tr>
<td>2016-07-07 15:10:18</td>
<td>Block RAM</td>
<td>No</td>
</tr>
<tr>
<td>2016-07-07 15:15:59</td>
<td>Register</td>
<td>No</td>
</tr>
<tr>
<td>2016-07-18 06:01:37</td>
<td>Register</td>
<td>No</td>
</tr>
<tr>
<td>2016-08-04 20:21:22</td>
<td>Register</td>
<td>No</td>
</tr>
<tr>
<td>2016-08-10 19:04:39</td>
<td>Register</td>
<td>No</td>
</tr>
<tr>
<td>2016-08-20 16:06:54</td>
<td>Register</td>
<td>No</td>
</tr>
<tr>
<td>2016-08-27 08:10:05</td>
<td>Loss of communication</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*Table 4.9: Fault detect events captured during test*
Figure 4.1: Timeline with fault detect events
4.2 Hardware implementation

4.2.2 Current draw

The current draw of the GEFE board was logged in 30 second intervals as seen in the top plot in figure 4.2. A mean of this data was taken over one hour intervals and presented in the bottom plot of the same figure.

4.2.3 Dosimeter measurement

The readout of the passive dosimeter gave an average reading of 1.6 Gy.
In this chapter the result is discussed and some conclusions are made. It will cover strengths and weaknesses of the methods used and provide some indications on improvements moving forward.

### 5.1 Simulation

The injection rates used during simulation are several orders of magnitude higher than what is expected in a real world scenario. This is done as it enables an easy way of testing at a reasonable cost in terms of simulation time. With the results from the simulation it is possible to understand if an implementation has any particular weak points which was what happened with V3. After running the simulation for V3 for the first time the result was not as good as expected and the design was analyzed in order to detect where the problem was. Once the missing keepers were discovered it became apparent that more care had to be taken to make sure that the HDL code written was actually describing the wanted circuitry.

In the fault injection tool the simulation keeps running until the functional monitor detects something is wrong. This monitoring is done by probing the system at certain locations one after the other. Because of this, there is a slight delay between when an injected fault has caused a fault and when the monitor sees it. A more thorough solution would be to every clock cycle monitor all nodes of interest and compare them to a golden run. Unfortunately, with a large design this would take a lot of processing to perform and the information stored in the golden run would be enormous. The fault injection tool presented in this thesis should be good enough to give a first estimate on how the mitigation and detection techniques deployed in the system performs.

One improvement to the tool which would save the designer some time is an
automated way to extract all the nodes where faults should be injected. This was done manually and took some time to be certain that all nodes were included. Having an automated script for this would help to make sure nothing is missed. During the time designing the fault injection tool no simple way of making such a script could be find which was independent of the tools used and the language. It was therefore decided to do it manually as the design was reasonably small and well structured.

5.1.1 Bit-flip injections

In figure 5.1 the average simulation time is shown for the different implementations and simulation runs when using bit-flips. From this it is apparent that the simulation time increases for each new version of the implementation which is good as it means fewer faults cause the simulation to restart. Another thing to note is the improvement from V1 and V2 is not that big and the same thing goes for comparing V4 and V5. The reason for this has to do with the increased cross-section when going from binary encoded to one-hot encoded state machines. Even though the one-hot encoding makes it easier to detect an invalid state the price to pay is in the extra registers that are needed. Looking at the numbers in table 4.2 it looks like both one-hot implementations improve the result but in table 4.1 this is not the case. This could also be a limitation in the test-bench as a state machine might recover from the faulty state after some time has passed but the functional monitor would still count that as an error. The time it takes for the state-machine to recover to normal operation could potentially be disastrous if it is part of a critical system. In that case it would be better to use mitigation techniques which do not modify the state and which do not disturb the functionality of the state-machine.

From figure 5.1 one can see that there is a slight difference in performance for the three different injection rates. This is most likely due to the fact that the rates used here for the simulation are orders of magnitude higher than expected in reality. With more faults injected every second it will of course decrease the time before a fault causes the simulation to restart. Another important effect of this is the occurrence of faults that are injected during the same clock cycle, i.e. multi-bit faults. The TMR implemented here will not be able to handle these types of faults.

Comparing the number of injected faults and detected faults in table 4.1, table 4.2 and table 4.3 it is possible to see a couple of things. First is that for V3 the number of detected faults is always higher than the other implementations and in the last two simulations it is even higher than the number of injected faults. The problem with V3, as explained in section 3.2, is that once an injection has caused a fault it will be kept forever. As the fault detect circuit checks every clock cycle for an error it will therefore keep incrementing the counter until the simulation stops. It is therefore important to make sure that the counter saturates to its highest value instead of overflowing. Having this comparison between injected faults and detected faults is a good way to make sure the mitigation and detection works as wanted. Secondly, for the other implementations, the number of
detected faults are always lower than the injected. This also has to do with the high injection rate used during simulation. If two faults are injected close to each other in time, but not in the same clock cycle, it can be counted as one fault in the detection circuit. To minimize this effect one would have to have multiple detect counters for different parts of the system and minimize the pipelines for the fault detect signals.

One thing worth mentioning is that there are some fault injections that do not cause an error. This is because some registers are not as critical as others. For example, there is register for a counter that can be turned on or off. If the counter is not used at the moment, i.e. it is turned off, it does not matter if a fault is injected in it. At a later point in time when the counter is enabled it will load a value stored in a limit register and therefore overwrite the faulty value. The register that stores the limit should be considered critical though. This shows that it might be possible to reduce the resource usage by selectively choosing what registers to triplicate.

## 5.1.2 Transient injections

Figure 5.2 shows the average simulation time for the different implementations using transient injections and it has a similar appearance to figure 5.1. Looking at the data in table 4.4, table 4.5 and table 4.6 and comparing it to the equivalent bit-flip tables it shows that the transient faults were not as damaging. The average simulation time for the transient injections is almost 10 times longer than the
bit-flip injections. This can be explained by the fact that a transient will only cause an upset when it happens over a clock edge. It therefore both has to be long enough and occur at the right moment. From the tables one can see that the detection percentage is minuscule which also has to do with the length of the transient.

As explained in section 2.2 a single event transient is the cause of a single event upset if the timing is right. The problem with simulating transient in a digital simulator is that all the effects cannot be modeled. The basic principle behind it is fundamentally analog as nodes inside the circuit experience voltage swings. In the simulation used here this analog behavior is replaced with a digital where both the rising and falling edge of the signal is instantaneous. How a real transient would affect the circuit in detail requires knowledge of how the device is designed which is not available for a COTS FPGA. It is therefore safe to say that the transient injection does not give any more information than what is already known from the bit-flip injection.

5.1.3 Resource usage

Looking at the register usage one can see the triplication as the three implementations with mitigation has more than three times as many registers compared to the baseline. The reason it is more than three times is because of the added fault detect controller and the pipeline registers for some of the fault detect signals. Comparing $V1$ and $V2$, the added registers from the one-hot encoding is increas-
5.2 Hardware implementation

The test of the hardware implementation was the most interesting as it provided data which is representative of the final installation. As can be seen from the result in section 4.2, the number of events was multiple orders of magnitude lower than what was used during simulations. This was expected as the simulation had to be sped up in order to test all parts of the system. For the majority of the events the system was able to recover by itself which is key for a reliable operation. The last recorded event which disabled the system in some way was not good and indicates a part of the design can get stuck in a state it cannot recover from. As the instrument for which this firmware and hardware is meant for is not part of some beam interlock or safety system this last fault is not critical. The fact that a power cycling returned it to an operational state is good enough for the instrument to be considered functional and useful. As long as it is possible to detect when communication is broken it is possible to automatically cycle the power in the software that controls the system.

It is worth noting that even though 11 events were detected there could have been more faults happening inside the FPGA. The current detection circuitry counts multiple faults occurring close to each other in time as one event. This means that the fault detection system might not be useful for absolute quantita-
tive measurements but more as a qualitative indication. Having an exact number is not the main interest as the system is not meant for radiation measurements. Therefore one can draw the conclusion that the fault detection circuitry is good enough for the requirements.

5.2.1 Block RAM and FIFO

The block RAM events were interesting as they indicate that the FIFOs provided by the vendor are not radiation hardened by default. This was expected as they are not designed to be, but having results to actually show this is valuable. The decision to not implement a custom FIFO with TMR applied was taken based on the limited time available before the installation date. A custom FIFO would have required thorough functional testing as the system would not work without it and the limited time did not allow this. With the result clearly showing that the regular FIFOs are susceptible it would be wise to take the time for this testing before the full system is put into operation.

5.2.2 FPGA orientation

One aspect not evaluated is the orientation of the FPGA in relationship to the beam. The orientation that was used and which is depicted in figure 3.7 might not be optimal and the results presented in this thesis should only be considered valid for this orientation. A thorough test of the best orientation would have required the board to be mounted on a movable platform which can be controlled remotely. This was not possible to do for this test.

5.2.3 Current draw

The current draw presented in figure 4.2 does not show any particular problems. As can be seen in the graph there seems to be a pattern with a period of around 35 hours. This is most likely due to the test software which was writing an incremental value to a couple of registers inside the FPGA. The registers were 8-bit wide which means 256 values from 0 to 255 were written. After a value was written to the registers the software waited 8 minutes before incrementing the value. This gives the following total time for writing all 256 values:

\[
256 \times 8 \text{ min} = 2048 \text{ min} \approx 34.1 \text{ h}
\]

which is basically the period that can be seen in the graph. The most important information from the current logging result is that there was no huge increase in current at any time during the test. If there was, it would have indicated that something had gone wrong in one of the components on the board. 99 days of testing might not give a complete picture of how the system will behave over its full lifetime and it is important that the current is logged in the operational device also to catch any problems.
5.2.4 Dosimeter measurement

The low value of 1.6 Gy was a bit surprising as the estimate for the installation location was in the order of 100 Gy per year which for the 99 days would have given around 27 Gy. This is of course a good surprise as it means the radiation environment might not be as harsh as originally thought. Even though the dosimeter was located in the same crate as the GEF board it could be that the exact location of the dosimeter is a low spot. The question to ask is if it is reasonable to extrapolate the data from the dosimeter to a larger space around it, e.g. the whole crate. A lot of parameters could be varied which would give different results but that having the board installed at location seen in figure 3.6 with the orientation seen in figure 3.7 should give results in the same order of magnitude as this test.
This thesis present some common mitigation and detection techniques that can be used to make a readout system with FPGAs more radiation hardened. It highlights the importance of understanding the details in the FPGA that is used and how the tools interpret the wanted design. Answering where radiation-hardening can and should be implemented is done by first understanding the available resources at hand inside the FPGA but also in terms of the software tools. After this, the system can be designed with enough mitigation and detection to meet the system requirements but without wasting too much resources and time.

The main motivation for this thesis was to investigate the possibility of using COTS FPGAs in a radiation environment at CERN. Having a reprogrammable system is of great importance to projects in an early stage of development and enables new functionality to be added over time. With reasonable effort it is possible to make these FPGAs more radiation-hardened and with detection a qualitative measure of the system can be extracted.

### 6.1 Simulation

A simulation-based fault injection tool is very useful to make sure a system behaves as wanted. The fact that it requires minimal effort and can be done early in the design process makes it an attractive tool for engineers designing systems for FPGAs used in a radiation environment. As a functional simulation is needed anyway it can easily be integrated into the testing procedure. Some drawbacks of this type of tool is that it is impossible to simulate a system using injection rates similar to what can be expected in reality. This means that some corner cases might be missed which can only be found once the system is running on hardware. The amount of time it takes to run the simulations with a lot of random injection is also limiting. In the simulations with an injection rate of 200
000 000 faults/s it took a couple of days to inject all faults. One way to reduce this would be to decrease the number of injections done. In this thesis every node was injected ten times during the simulations which might be a bit excessive. The problem lies in the fact that both location and time can be varied which creates a big parameter space.

### 6.2 Hardware implementation

From the results of the hardware implementation it was decided to continue using the GEFE board for the final implementation of the system scheduled for the end of 2016. With TMR added to the design it is reliable enough for an operational system and any faults requiring a power cycling should be detectable. The fault detection circuitry will be included in the system in order to have an indication of how often faults occur and to monitor the behavior and health for a longer period of time. Software control will be used to monitor the status of the system and perform any necessary actions to keep it running.

### 6.3 Future Work

For the simulation injection tool it would be interesting to automate the node extraction from the design. This was done manually in this thesis which can result in missing nodes. As this automation would look for nodes in the design where fault can be injected it could also be used to automate the process of protecting the nodes. The biggest issue with a tool like that is that it falls into the category of dual-use items which has strict export restrictions. Dual-use means that it can be used for both civilian and military applications and as the tool would create circuits that are radiation tolerant it could be used for harmful activities.

For the hardware implementation the future work would be to investigate further how to make the internal hard-blocks more radiation hardened. This includes custom FIFOs but also any other blocks that the designer might want to make use of.

To speed up the fault injection testing one possibility would be to use emulation based fault injection. How this can be implemented and integrated with a system during development would have to be investigated.
Bibliography


FPGAs are attractive devices as they enable the designer to make changes to the system during its lifetime. This is important in the early stages of development when all the details of the final system might not be known yet. In a research environment like at CERN there are many FPGAs used for this very reason and also because they enable high speed communication and processing. The biggest problem at CERN is that the systems might have to operate in a radioactive environment which is very harsh on electronics. ASICs can be designed to withstand high levels of radiation and are used in many places but they are expensive in terms of cost and time and they are not very flexible. There is therefore a need to understand if it is possible to use FPGAs in these places or what needs to be done to make it possible.

Mitigation techniques can be used to avoid that a fault caused by radiation is disrupting the system. How this can be done and the importance of understanding the underlying architecture of the FPGA is discussed in this thesis. A simulation tool used for injecting faults into the design is proposed in order to verify that the techniques used are working as expected which might not always be the case. The methods used during simulation which provided the best protection against faults is added to a system design which is implemented on a flash based FPGA mounted on a board. This board was installed in the CERN Proton Synchrotron for 99 days during which the system was continuously monitored. During this time 11 faults were detected and the system was still functional at the end of the test. The result from the simulation and hardware test shows that with reasonable effort it is possible to use commercially available FPGAs in a radioactive environment.

Nyckelord:
FPGA, radiation, TMR, fault mitigation, fault injection, fault detection
Uphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydda mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Hampus Sandberg