Institute of Computer Architecture and Computer Engineering (ITI)
Computer Architecture Department
Pfaffenwaldring 47
D-70569 Stuttgart

Master Project Nr. 3097

Development of an Error Detection and Recovery Technique for a SPARC V8 Processor in FPGA Technology

Andrew Boktor

Major:	Embedded Systems Engineering, INFOTECH
Examiner:	Prof. Dr. habil. Hans-Joachim Wunderlich
Supervisor:	M.Sc. Rafal Baranowski
Starting Date:	18$^{\text{th}}$ of October, 2010
Submission Date:	19$^{\text{th}}$ of April, 2011
CR Classification:	B.8.1, C.4, C.5.3, C.1.3

University of Stuttgart
Faculty of Computer Science, Electrical Engineering and Information Technology

Institut für Teschnische Informatik (ITI)
Abteilung Rechnerarchitektur (RA)
Pfaffenwaldring 47
D-70569 Stuttgart

Masterarbeit Nr. 3097

Development of an Error Detection and Recovery Technique for a SPARC V8 Processor in FPGA Technology

Andrew Boktor

Studiengang:	Embedded Systems Engineering, INFOTECH
Prüfer:	Prof. Dr. habil. Hans-Joachim Wunderlich
Betreuer:	M.Sc. Rafal Baranowski
begonnen am:	18. Oktober 2010
beendet am:	19. April 2011
CR-Klassifikation:	B.8.1, C.4, C.5.3, C.1.3

Universität Stuttgart
Fakultät Informatik, Elektrotechnik und Informationstechnik

To my mother Elham, my father Mofid, and my brother Peter

Abstract

Field-Programmable Gate Arrays (FPGAs) found widespread use in many areas of applications, including safety and mission-critical systems. More and more manufacturers are choosing to implement designs on FPGAs. However, SRAM-based FPGAs are proven to be much more prone to Single Event Upsets (SEUs) compared to traditional Application-Specific Integrated Circuit (ASIC) designs. Moreover, SEU affects FPGAs in more severe ways compared to ASIC. Techniques to provide fault-tolerance for SRAM-based FPGAs become essential to maintain their advantages over other technologies.

This thesis presents a fault-tolerance technique for pipeline architectures in FPGA technology. It provides fault-tolerance against SEUs in the design and is able to detect faults in the FPGA configuration. It also proposes an additional mechanism that detects all SEUs independent of their location. Pipeline operation can be resumed with known techniques of partial reconfiguration. Both designs occupy a much smaller area compared to known techniques such as TMR in combination with Scrubbing. They introduce no additional time penalty in case of fault-free operation. Fault injection and simulation were used to validate the design and calculate the fault coverage.

Acknowledgments

The work accomplished throughout this Master's thesis and degree would not have been done the way it was, and the output achieved would not be the same without the help and support of many people. I would like to particularly thank everyone of the following for his/her help.

My mother **Elham Fahim**, my father **Mofid Boktor**, and my brother **Peter Boktor** for their prodigious support all over the project, for their patience all over my travel period in Stuttgart, and for always being there for me.

Prof. Dr. habil. Hans-Joachim Wunderlich for encouraging the project idea and welcoming me to work with his department.

M.Sc. Rafal Baranowski my supervisor, for his great interest in the project, his follow-up and encouragement during the thesis period of 6 months.

Prof. Dr. Paul J. Kühn for handling the course acknowledgment issues and his great help in achieving this Master's Degree.

Marleine Daoud for her help and ideas throughout the project, her reviews, her encouragement, her dedication and her patience during the work period and during my full stay in Stuttgart.

List of Figures

List of Tables

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
CLB	Configurable Logic Block
COTS	Commercial-Off-The-Shelf
DUT	Device Under Test
DWC	Duplication With Comparison
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
GPL	GNU General Public License
HDL	Hardware Description Language
IP	Intellectual Property
ISA	Instruction Set Architecture
LUT	Look-Up Table
MIPS	Microprocessor without Interlocked Pipeline Stages
MUX	Multiplexer
NMR	N-Modular Redundancy
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level
SEU	Single Event Upset
SPARC	Scalable Processor Architecture, a RISC ISA developed by Sun Microsystems
SRAM	Static RAM
TMR	Triple Modular Redundancy
VHDL	VHSIC HDL

VHSIC	Very-High-Speed Integrated Circuit
XST	Xilinx Synthesis Technology

Contents

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) have been forcing their way, slowly but steadily, into many safety-critical applications. FPGAs can be found in application ranging from embedded systems to space vessels. A design can be implemented on an FPGA at a very low cost and in a very short time compared to traditional Application Specific Integrated Circuits (ASICs), thus eliminating upfront, non-recurring costs associated with ASIC technologies. Due to their low cost and low turnaround time, they present a very attractive alternative for small/medium size projects. They also allow testing different changes to a design at virtually no additional cost.

Unfortunately, FPGAs turn out to be much more vulnerable to Single Event Upsets (SEUs) caused by radiations that alters the state of memory elements[BBB+04].

Typically, memory cells other than FPGA configuration memory constitute less than 2% of all memory on an FPGA. The rest, 98%, are configuration memory cells[AT05]. This is the reason why FPGAs are particularly vulnerable to SEUs. In addition to that, SEUs hitting the FPGA configuration memory cause errors that have a semi-permanent behavior; they are not corrected unless the FPGA configuration is re-written. Moreover, those errors, do not only change the data handled by a design, they change its behavior as well.

A way to workaround this problem could be using radiation hardened FPGAs. However, radiation hardened FPGAs are a few fold more expensive than Commercial-Off-The-Shelf (COTS) ones. Hardened FPGAs are also a few technology generations behind COTS FPGAs. For price sensitive applications, COTS still present an attractive option despite their SEU vulnerability[AT05, VSC+04].

A new generation of fault-tolerance techniques have evolved to solve this problem. Their aim is providing fault-tolerance for FPGAs against the new class of faults. This

allows the use of COTS FPGAs for high reliability and high availability systems.

1.1 Motivation

1.1.1 The Need for Fault-Tolerance

SEU-tolerance techniques have been developed as the counter measure engineers take in reaction to the increased vulnerability to soft errors. Fault-tolerance techniques are used to allow the system to continue its correct operation even when faults occur.

For critical applications like airplane control systems or space missions, the correct operation of the electronic components becomes absolutely essential since a failure would result in catastrophic consequences. Techniques to provide fault-tolerant electronics have been developed for exactly that reason. Those fault-tolerance techniques range from simple radiation shielding of the electronics to very complex designs that detect faults and correct them.

For ASIC designs, many of those fault-tolerance techniques were appropriate and yielded very good results. However, with the new capabilities FPGAs offered to developers, FPGAs have forced their way into those sensitive applications. Unfortunately, due to the new technology, traditional fault-tolerance techniques are not suitable anymore for the new FPGA technology. Thus, new fault-tolerance techniques, specifically tailored for FPGAs, had to be developed.

1.1.2 FPGAs are Different

FPGAs adopt a very different approach at implementing the user's design. They employ general purpose, programmable units that could be programmed to perform the required functionality. Those general purpose units are accompanied by a set of programmable switches that can also be programmed to yield the required interconnections between the general purpose units.

The programmability of the FPGA components requires most of the FPGA area to be used to store these configurations. Due to the large area used for the configuration logic and memory, this logic and memory are affected greatly by soft errors. Opposed to soft errors in ASIC designs, where re-writing the value would replace the faulty value with a new one, FPGA configuration memory faults cannot be corrected in the same way. This calls for new techniques to detect those configuration faults and a way to recover from them.

1.2 Goals/Objectives

This thesis aims at constructing a fault-tolerance technique for pipelined microprocessors running on FPGAs. The technique would detect and correct SEUs in user bits. Moreover, it would detect SEUs in FPGA configuration memory. The area overhead for the presented technique is expected to be in the range of 220%-260%. An implementation of the technique will be made using the LEON3 soft core processor. Fault injection will be used to validate the design and calculate the fault coverage.

1.3 Outline

Chapter 2 introduces concepts that are used extensively in the explanation of the proposed design. Moreover, it provides an overview of previous work that is important for this thesis. In *Chapter 3*, the architecture of the proposed fault-tolerant design is discussed in details. First, it is introduced on a theoretical level. Then, the implementation using the LEON3 processor is explained. Afterwards, *Chapter 4* discusses the methodology used to validate the proposed design and calculate the fault coverage. *Chapter 5* presents the results from the validation and a brief discussion about the reasoning behind them. *Chapter 6* summarises the thesis and gives an outlook about open problems that might need to be addressed in the future to improve or complete the presented design.

Chapter 2

Background and State of the Art

This chapter introduces concepts that are needed to understand this thesis. It also gives an overview about related work that has been done in the past in the area of fault-tolerance for Field-Programmable Gate Arrays (FPGAs) and microprocessors. The related work influenced the design proposed in this thesis either through adoption of the same mechanism or adopting similar mechanisms that are known to yield good results.

Section 2.1 introduces the architecture of FPGAs and an overview about how they work. This introduction is essential to understand the fault model presented in *Section 4.1*. *Section 2.2* introduces faults, their classifications and the impact of each type on the system. *Section 2.3*, *Section 2.4* and *Section 2.5* introduce different fault-tolerance approaches that are used for Application-Specific Integrated Circuit (ASIC), FPGA and microprocessors, respectively. Since the approach presented by this thesis is a combination of those three types, the different approaches presented in this section are relevant. A brief introduction to the LEON3 processor, which is used for the implementation of the work done for this thesis, is given in *Section 2.6*. *Section 2.7* gives an overview about software functional test for processors. *Section 2.8* discusses methods considered for evaluating fault impact on the design. Finally, *Section 2.9* explains the contribution of this thesis and its distinction from the State of the Art.

2.1 Architecture of FPGAs

Field-Programmable Gate Arrays (FPGAs) are digital integrated circuits designed to be programmed by the developer after manufacturing. The ability to be programmed led to FPGAs being capable of implementing any logic that an Application-Specific Integrated Circuit (ASIC) is able to [VH98]. FPGAs have no upfront costs that need to be paid as in the case of ASIC. This makes them ideal for small and medium size

projects. Moreover, they allow rapid prototyping, validation and testing of designs with a dramatic decrease in the cost and turnaround time.

FPGAs contain Configurable Logic Blocks (CLBs) that can be programmed to carry out a special function along with programmable interconnects. This section will talk in detail about how the simplest CLBs are structured and how interconnects work. Knowledge of the FPGA structure and functionality will be needed later on in this thesis.

In this section, a simplified FPGA architecture will be described with details about which components are used to make an FPGA and how these components are configured and connected together to provide the needed functionality.

2.1.1 Configurable Logic Blocks

Configurable Logic Blocks (CLBs) are the heart of FPGAs. In order to provide a programmable logic, FPGAs use CLBs that can be programmed to implement a small logic function.

In FPGAs, Look-Up Tables (LUTs) act to replace traditional gates in ASIC designs. For instance, if the LUT in *Figure 2.1* had the truth table of a three input OR gate in its memory, like the one in *Figure 2.2*, the LUT would function as an OR gate. The advantage of LUTs is that they can be reconfigured to perform different functions as required.

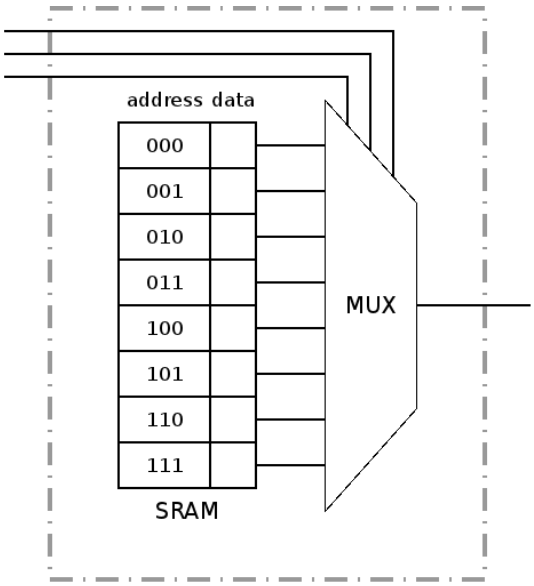
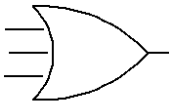


Figure 2.1: *LUT diagram*



000	0
001	1
010	1
011	1
100	1
101	1
110	1
111	1

Figure 2.2: *OR gate and truth table*

As can be seen in *Figure 2.3*, LUTs are often used in FPGA CLBs to provide the ability for the CLBs to perform an arbitrary function. Usually, CLBs will contain a flip-flop (FF) as well to give the ability to function as registers for the output.

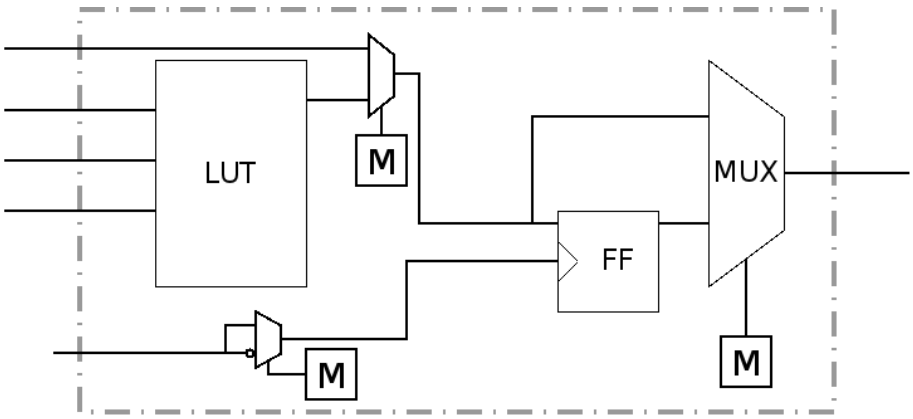


Figure 2.3: *CLB diagram [KNCR04]*

Figure 2.3 depicts a CLB with three inputs and one output. The three inputs to the CLB go directly to the inputs of a LUT. The output of the CLB is either registered or not, as selected by the right-most multiplexer. The FF can register the data on the rising or falling edge of the clock as selected by the left-most multiplexer. Based on the select line of the middle multiplexer, the LUT can be left out and only the register used.

This dynamic configuration allows the CLB to use the FF only, the LUT only or the complete LUT-FF pair. The boxes with 'M' in the figure depict the Static Random-

Access Memory (SRAM) configuration memory. As seen, this memory controls the configuration of the CLB.

2.1.2 Configurable Interconnects

Considering that each CLB can only perform a very small logic function, to construct bigger designs, CLBs need to be connected in an efficient and configurable way. As can be seen in *Figure 2.4*, FPGAs are basically a matrix of CLBs with connective wires in between. To establish the configurable interconnections that are needed to construct bigger designs, FPGAs use programmable switches at the intersections of wires. Those programmable switches can be configured to route the signals in many different ways. Based on the design requirement, the appropriate configuration is given to them. Like CLBs, the programmable switches also have the configuration that controls their behavior stored in SRAM.

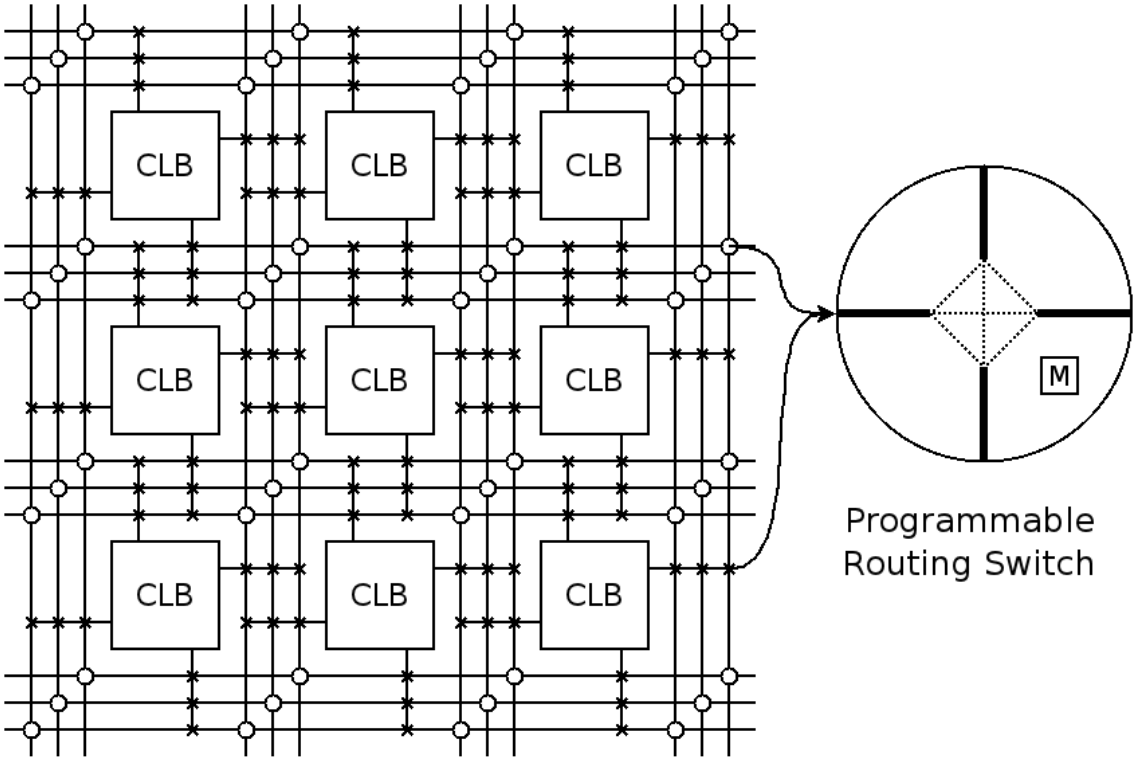


Figure 2.4: *FPGA diagram*

2.1.3 FPGA Configuration

To facilitate the configuration process of FPGAs, most FPGAs have an external flash memory that holds the configuration. The configuration bitstream is then loaded

to the SRAM upon power up. When programming, the configuration, in a form of bit stream that represents the configurations needed for all components within the FPGA (CLBs and switches), is loaded to the flash memory.

After the configuration is transferred to the SRAM upon power up, the SRAM is actively used during the operation of the FPGA. Thus, any faults that strike this memory are likely to cause misbehavior of the board.

2.1.4 Synthesis, Placing and Routing

Designs targeted for FPGAs are usually written in a high level Hardware Description Language (HDL). HDLs specify the functionality of the design. After the HDL for a design has been written and simulated, a synthesis tool is used to convert this design to a design implementation using complex logic gates. After the synthesis of a design, a *Place and Route* is performed. During the *Place and Route* phase, each part of the design is mapped to a specific CLB and the routing configurations are generated. The output of the synthesis, placing and routing is then written to a file in the form of a bit stream that is used later to configure an FPGA board.

2.2 Faults

Since the aim of the work done on this thesis is to achieve a fault-tolerant design, it is important to give an overview of *faults*. In this thesis, we are particularly interested in soft errors since soft errors are the most common type of errors to affect FPGAs. Furthermore, the occurrence of a soft error does not permanently impact the functionality of the circuit and can be fully recovered from. Thus, fault-tolerance techniques targeted at soft errors are feasible and highly beneficial.

Soft errors are caused by Single Event Upsets (SEUs) in the combinational logic or register components. SEUs themselves are caused by an energized particle impacting the surface of the design. The energy of the particle can temporarily transform the state of a transistor. An SEU might or might not cause a soft error in the design depending on many factors, among which are electrical masking, logical masking, and/or temporal masking[AT05, SKK+02]. If at that instant the signal is registered, then the fault is not masked and will be represented by the wrong value in the register. The energized particles can also hit a sensitive part in the implementation of a flip-flop and cause an instant state change.

In the case of FPGAs, SEUs can cause soft errors in two different, distinguishable locations: user bits or configuration bits. The faults will have a different effect on the design based on where they are located. In the following two subsections, the effects of faults in both locations will be discussed in details.

2.2.1 User Bit Faults

Faults in user bits are represented by flips of some signals or flip-flop states in the user-defined design. For instance, in *Figure 2.3* the faults would be in the flip-flop or the output signal.

Effect on design

Basically, user bit faults correspond to soft errors in traditional Application-Specific Integrated Circuit (ASIC) design. They do not modify the design but rather cause wrong output. When a soft error occurs in user bits, it is correctable by writing a new value to the erroneous location.

2.2.2 FPGA Configuration Faults

On the contrary to user bit faults, configuration faults occur in the FPGA configuration memory. Since the user-defined design does not write to the FPGA configuration memory during its normal operation, those faults cannot be corrected using a fault-tolerant design. To correct an FPGA configuration memory fault, a write has to be made to the location of the fault. This is only possible during the FPGA configuration. Thus, a partial or full re-configuration is needed to correct faults in FPGA configuration memory.

Effect on design

The effect of a fault in the configuration bits will depend on the location of the fault. For instance, a fault in the Look-Up Table (LUT) memory will result in that particular LUT performing a different logic function other than the intended one (e.g. OR vs. XOR). A fault in the configuration of whether the output of a Configurable Logic Block (CLB) is registered or not will result in a change of synchronization. Furthermore, a fault in the configuration of the programmable switches will result in faulty connections. Despite that, all faults in configuration bits have one common characteristic: they are all persistent until a new configuration is written to the configuration memory.

In contrast to faults in the user bits, the configuration faults may not instantly affect the output. For example, a fault in the LUT configuration memory will not be detectable from within the design unless the inputs of the LUT select that particular faulty bit as the output. The same is true for the routing configurations, the fault will not be seen from within the design unless the faulty link is used. It is a challenge to detect those faults since foreknowledge of the exact inputs to expose the error is impossible.

2.3 Fault-Tolerance for ASIC

For traditional Application-Specific Integrated Circuit (ASIC) designs, there exists a multitude of techniques that can be used to achieve fault-tolerance. Those techniques can also be implemented on FPGAs to provide fault-tolerance for faults in the user bits. In this section, two of the most mature fault-tolerance techniques for ASIC designs are discussed.

2.3.1 Triple Modular Redundancy (TMR)

Triple Modular Redundancy or TMR is a technique that achieves fault-tolerance through hardware redundancy. TMR designs instantiate the functional component three times and employ a majority voter on the output. The voter guarantees that in case of a single fault, the faulty output will be voted out and the correct output will be provided. TMR is associated with an area overhead of more than 200% over the original design. That means that the fault-tolerant version will occupy more than three times as much area as the original design. [BMS07]

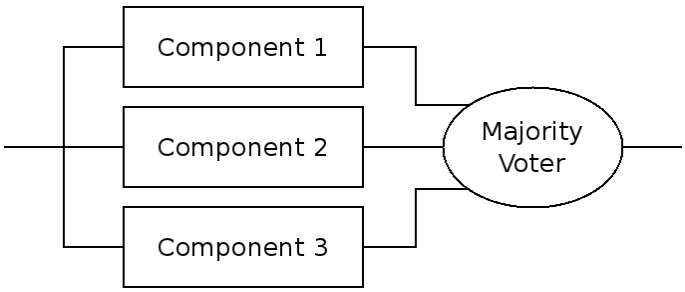


Figure 2.5: *Triple Modular Redundancy (TMR)*

Figure 2.5 shows how TMR creates three instances of a component and compares their output using a majority voter to eliminate the effect of faults. TMR can be generalized in the form of N-Modular Redundancy (NMR) where more than three instances of a component are used. [BW89]

2.3.2 Duplication With Comparison (DWC)

Duplication With Comparison or DWC, as the name implies, duplicates the design and employs a comparator at the outputs. DWC is a simple technique that can only detect faults, but not correct them [BQS07]. *Figure 2.6* illustrates how DWC would be used. In the design presented in this thesis, DWC will be used to detect faults. This is discussed in more details in *Section 3.3*.

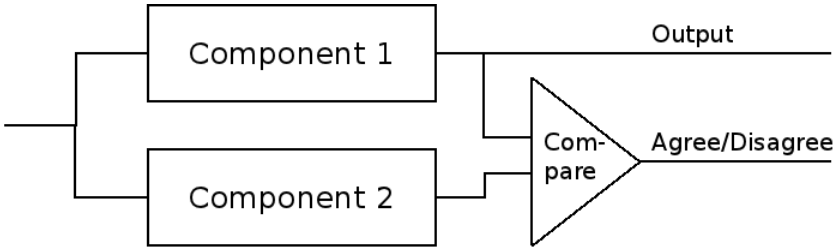


Figure 2.6: *Duplication With Comparison (DWC)*

2.4 Fault-Tolerance for FPGAs

As mentioned in *Section 2.2*, Field-Programmable Gate Arrays (FPGAs) require the consideration for additional fault locations when implementing a fault-tolerant design. More specifically, faults in FPGA configuration memory have to be considered separately since they have a different effect on the design compared to faults in the user bits. [CCS00]

Many approaches have been proposed that would provide fault-tolerance specifically targeted at FPGA configuration memory faults. Independent of the technique applied, traditional fault-tolerance techniques targeted at Application-Specific Integrated Circuit (ASIC) are used to provide fault-tolerance for the user bits [AT05]. This section will briefly discuss fault-tolerance techniques specifically targeted at FPGAs.

2.4.1 Readback

Readback is a technique developed for FPGAs to detect and correct Single Event Upsets (SEUs) in the configuration memory. The technique periodically reads each frame of the configuration memory and compares it with the bit stream originally uploaded to the FPGA board. When a discrepancy is found between the two, the mechanism writes the correct value back to that single defective frame.

Readback is superior to other techniques because it keeps the configuration memory in read mode for most of the time and in write mode only for the period necessary to correct the fault. This decreases the probability that an upset in the configuration logic will be propagated to the configuration memory. On the other hand, readback requires a hardware implementation of the algorithm that reads the memory periodically and corrects it when an error is found. [CCS00]

2.4.2 Scrubbing

Scrubbing is a simpler technique to achieve fault-tolerance for FPGA configuration memory. Scrubbing simply writes the configuration memory periodically, which ensures

that any faults that were present are overwritten. It does not require any comparison or detection of faults, it simply overwrites the data whether it contains faults or not. There is an interval between each scrub, this interval should be adapted for the frequency of SEUs in the particular environment.

Despite the simplicity of this mechanism, it leaves the configuration memory in write mode for a longer period of time compared to readback. In case of a low error rate, the scrubbing interval can be lengthened thus decreasing the amount of time the memory is in write mode. [CCS00, KNCR04]

Both scrubbing and readback should be used along with another fault-tolerance technique that can tolerate single faults. This is necessary because scrubbing and readback do not correct faults immediately. They are rather used to prevent fault accumulation.

2.5 Fault-Tolerance for Microprocessors

Technologies that allow configuring the Field-Programmable Gate Array (FPGA) without any disruption to the running design exist [CCS00]. Thus, the techniques discussed in *Section 2.3* and *Section 2.4* can be used with any design. However, other techniques exist that are specific to microprocessors. These techniques depend on the architecture and functionality of microprocessors. The most well known of those techniques are briefly explained in this section.

This thesis presents a design based on pipeline rollback to recover from user bit faults and to detect faults in FPGA configuration memory. More details about the employed techniques are given in *Chapter 4*.

2.5.1 Toleration

One way to tolerate faults is to completely ignore those known to have no effect on the design. For instance, a fault in the data input to the register file in a cycle where the write enable is not asserted is known to have no effect on the system. Such errors can be completely ignored and no recalculation of the data is needed. [GPV02]

2.5.2 Rollback

Rollback generally refers to reverting to a pre-error state and redoing the affected computations. The pre-error state reverted to can vary from one clock cycle to many instruction cycles in the past. Rollback is considered as a form of temporal redundancy since some calculations are repeated. [GPV02]

Micro Rollback

Micro Rollback takes the smallest step possible to the past. In a processor pipeline, a micro rollback would be done by freezing all stages except the erroneous one and just repeating the computations in that stage. [GPV02]

Stage Rollback

A Stage Rollback is necessary if the operands for a Micro Rollback are no longer available, thus the complete pipeline is rolled back one stage. [GPV02]

Pipeline Rollback

If the operand of some operation in the register file were overwritten, then a Pipeline Rollback would be needed. A pipeline rollback would restart the pipeline to a state that was present several clock cycles in the past. [GPV02]

Macro Rollback

A Macro Rollback is a rollback to a specific point called rollback point. It is used in the case that many operands in the register file have been destroyed. [GPV02]

2.6 LEON3

LEON3 is a soft core pipeline processor that is part of the GRLIB IP library. The VHDL source code of LEON3 is released under the GPL license. LEON3 is an implementation of the SPARC version 8 architecture. The fault-tolerance mechanism proposed in this thesis is introduced to the LEON3 processor integer pipeline. The processor is fully synthesizable for Field-Programmable Gate Arrays (FPGAs) which makes it very suitable as a base for the implementation of the practical tasks of this thesis.

The LEON3 integer pipeline is a RISC pipeline composed of seven stages: Fetch, Decode, Register-Access, Execute, Exception and Write-Back. The register file, instruction cache, data cache, multiplier and divider are located outside the integer pipeline.

2.7 Software Functional Test

To validate the processor, software functional tests were chosen. Functional tests perform the testing on the device under test (DUT) in a black box fashion. They simply perform a set of operation that would verify that the DUT is functioning properly.

2.8 Methods for Evaluating Fault Impact

To evaluate fault impact on the design, there exists a multitude of methods ranging across actually bombarding the functional design with radiation [QGK+05, FCS+00], simulating it [VSC+04], and theoretically estimating the effects of a fault [BBB+04]. In this thesis, it was necessary to evaluate the fault coverage of the proposed design. An existing fault injection and simulation environment was reused for this task, together with software for functional tests.

In the following subsections, basic definitions and techniques for fault injection and simulation are presented.

2.8.1 Fault Injection and Simulation

In a fault injection experiment, a fault is deliberately introduced to the design and the effect is observed. The fault injection can be done by different means, among which is using mutants. [Lev00]

In the experiments done in this thesis, simulations with fault injection are performed to calculate the fault coverage of the proposed design. A deeper explanation of the exact technique used to conduct the experiment is presented in *Chapter 4*.

Mutant Based Simulation

Mutant based simulation is a type of fault injection. A mutant is an entity that is modified or mutated to model the fault effect. Usually, entities such as gates or Look-Up Tables (LUTs) are mutated to model the fault. The mutated entity behaves as if it had a fault, thus allowing the analysis of the design under the effect of faults. [Lev00, JAR+94]

2.9 Contribution Beyond the State of the Art

This thesis presents a new approach that combines three different concepts from distinct areas of research to provide fault-tolerance for microprocessor implementations running on Field-Programmable Gate Arrays (FPGAs). A study of fault-tolerance techniques targeted at Application-Specific Integrated Circuit (ASIC) in general, at FPGAs and at microprocessors was done and the most suitable of each was selected. The proposed approach combines Duplication with Comparison (DWC) with pipeline rollback to achieve a low area fault-tolerant design for microprocessors in FPGA technology.

Originally, DWC was used because of its simplicity to detect faults [BQS07]. Implemented in FPGA, DWC does not distinguish between faults in user bits and in configuration memory. The proposed technique employs pipeline rollback to a state before the error was detected to correct the faults in user bits. Faults in the user bits are effectively corrected after the rollback while faults in the configuration memory manifest themselves in the exact same way after the rollback. The proposed approach thus detects and distinguishes them from user bit faults at no additional cost. The proposed approach only detects and does not correct faults in the FPGA configuration memory. However, it can be coupled with dynamic reconfiguration to correct them.

This thesis proposes also an alternative mechanism. This mechanism occupies a smaller area than the original one. It only detects faults in the pipeline independent of their location (user bits *vs.* configuration memory). Known partial reconfiguration mechanisms can be used to recover from the faults.

Chapter 3

Proposed Architecture

In this thesis, a pipelined processor architecture that achieves fault-tolerance in FPGA technologies is proposed. The design aims at detecting Single Event Upsets (SEUs) in the user bits as well as recovering from them. It also aims at detecting SEUs in the FPGA configuration bits.

This chapter details the specifics of the proposed design in *Section 3.1*. Details about how the approach works and how it handles cases specific to processor pipelines are described. *Section 3.3* explains how the proposed design can be applied to the LEON3 soft-core processor. An alternative mechanism that detects faults independent of their location can be found in *Section 3.2*.

In the following chapter, the techniques employed for validating the design by the means of fault-injection are discussed.

3.1 Proposed Fault-Tolerance Mechanism

This section explains the proposed fault-tolerance approach on a theoretical level. A simple Microprocessor without Interlocked Pipeline Stages (MIPS)-like architecture is used to explain the most important concepts. Details about the implementation of the proposed technique using the LEON3 processor is found in *Section 3.3*.

3.1.1 Introduction to MIPS

A block diagram of a simple MIPS-like pipeline is seen in *Figure 3.1*.

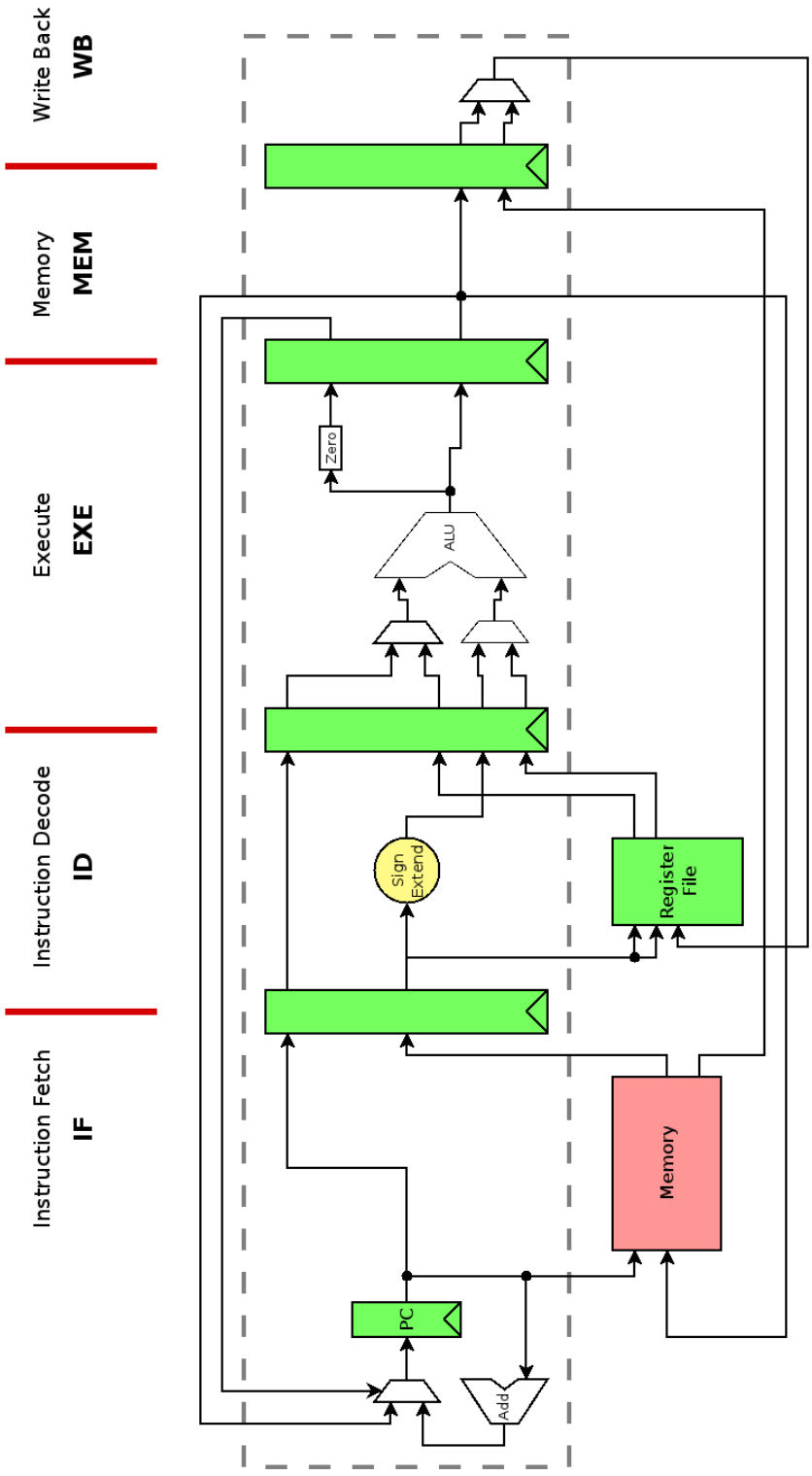


Figure 3.1: *A simple MIPS-like pipeline*

Stages

The MIPS pipeline has five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory (MEM), and Writeback (WB).

IF: The IF stage is responsible for calculating the next Program Counter (PC) and fetching the instruction from the instruction memory.

ID: The ID stage is responsible for decoding the instruction into control signals, requesting the operands from the register file, and extending the sign of the immediate if an immediate is present.

EXE: The EXE stage is responsible for ALU operations and comparison with "zero" for the calculation of the branch decision. It is also responsible for calculating the memory address if a memory access instruction is there.

MEM: The MEM stage is responsible for either reading data or writing data to the memory.

WB: The WB stage is responsible for writing the result (either from EXE or from MEM) to the register file.

3.1.2 Fault Detection via DWC

The design proposed in this thesis aims at using Duplication with Comparison for fault detection of faults in user bits as well as FPGA configuration faults. For the pipeline in *Figure 3.1*, this could be achieved using the following steps:

Duplicating the pipeline. Two instances of the pipeline are created.

Comparing pipeline states. The signals that represent the state of the pipeline will be compared and a fault is detected if there is a mismatch between these signals. The signals that represent the state of the pipeline in this case are the values registered inside the inter-stage registers. If a fault is present somewhere within the design, whether in the configuration or in the user bits, a mismatch should be found between the two pipeline instances.

Figure 3.2 shows a diagram of the duplicated pipeline and the comparators.

3.1.3 User Bits Fault Recovery via Rollback

Upon the discovery of a fault, the following actions are taken to correct the error:

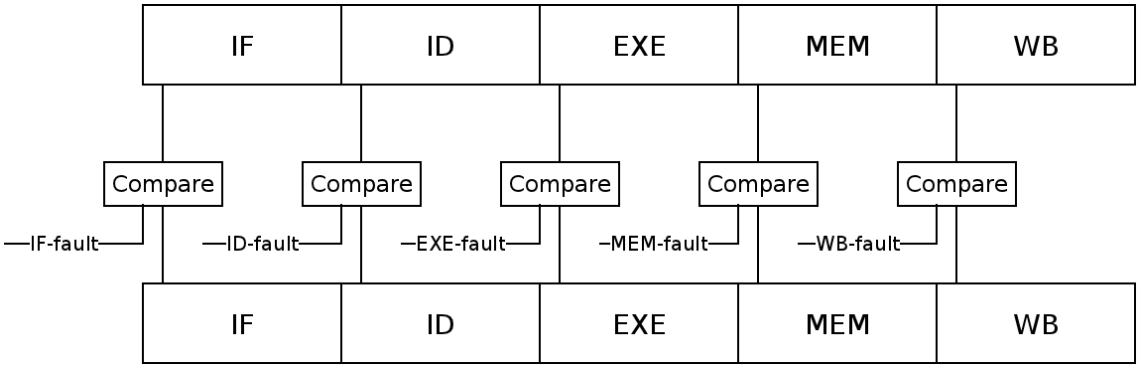


Figure 3.2: *Pipeline duplication and comparators*

Annulling all instructions in the pipeline. Most processor implementations already have an annul signal present that is propagated through the pipeline along with the instruction. Usually this signal is used to annul the instruction in a branch delay slot. This annul signal can be repurposed for this task.

Cancelling the effect of the instruction present in the WB stage. The instruction in the last stage cannot be annulled in a way similar to other instructions since its outputs are not registered. This can be solved through combinational logic at the end of the last stage. In the pipeline presented in *Figure 3.1*, gating the register file write enable signals solves this problem.

Branching the pipeline to the address of the instruction present in the last stage at the instance the fault occurred. This can be achieved in more two ways. A fake branch instruction can be inserted in the pipeline followed by NOP instructions to fill the branch delay slot. Another, more straight-forward way of doing it, is to add combinational logic to the branch enable and branch address signals to override them and cause a branch to the required location.

After those steps are conducted, the pipeline is left to its normal operation, the annulled instructions are repeated and correct operation is assured in the case that the fault was present in the user bits.

3.1.4 Configuration Fault Detection via Persistence of Faults

The proposed design hardly uses any additional area to detect configuration faults. Rather, it takes advantage of the properties of faults in the configuration memory. Faults in configuration memory have a persistent nature, they are not corrected until a configuration of the FPGA is performed. Thus, if a fault persists after the rollback, it is likely that it is located in the configuration memory.

Configuration faults detection can be achieved by using a counter that counts the instruction cycles since the rollback. After five cycles, the pipeline should be back to the original state at the moment of the fault occurrence. The mechanism checks if the fault is detected again. If this is true, it declares detection of a configuration fault. Partial or full reconfiguration of the FPGA could be used to recover.

3.2 Alternative Mechanism

The probability of faults occurring in user bits is very low compared to the probability of faults in configuration memory. This holds true because FPGA configuration memory constitutes 98% of all memory cells on an FPGA [AT05]. Moreover, the initially proposed design often detects user bit faults as configuration faults for the reasons explained in *Chapter 5*. Therefore, an alternative mechanism is proposed that improves on the original design.

The alternative mechanism aims at detecting all faults independent of their location (configuration memory or user bits). After their detection, faults can be corrected using reconfiguration. Since faults in the user bits have a low probability of occurring compared to configuration faults, this mechanism does not introduce a high overhead for correcting faults. The design has a lower area overhead compared to the initial design, and has no impact on the critical path.

The alternative mechanism is identical to the originally proposed mechanism except for the following points:

Signals Monitored: the alternative mechanism monitors all the outputs of the integer pipeline and does not monitor any intermediate signals.

No user bit faults recovery: the alternative mechanism does not recover from faults in user bits. Rather, it just detects faults in the design independent of their location.

The alternative mechanism was implemented on the LEON3 integer pipeline as well.

3.3 Implementation

This section explains the implementation of the proposed architecture on the SPARC V8 compliant LEON3 soft core processor. The work was done on the open source, GPL, version of the LEON3. Following are the different parts of the implementation in detailed description along with solutions to encountered problems.

3.3.1 Pipeline Stage Extraction

The integer pipeline of the LEON3 processor is coded in a single VHDL process. To start working with the processor, different stages of the pipeline needed to be extracted and each declared in its own entity. Those entities are then instantiated in one top-level pipeline entity that includes the complete functionality of the original pipeline design.

A complex programming script was developed to parse the integer pipeline's VHDL code and generate the separate stage entities as well as the new integer pipeline entity that contains the instances of the stages. The stage signals/variables that are read or written by a particular stage were identified manually and stored. Based on this information, the script generates the necessary ports for the stage entities.

The separate stage entities are very useful later on when constructing the comparator. They provide the needed intermediate signals that are monitored. The information gathered by the script during the parsing of the original pipeline code is used as well to generate the comparator.

3.3.2 Pipeline Duplication

As discussed in *Section 3.1*, two instances of the integer pipeline are necessary. A new entity that has an identical port to the integer pipeline was created. This entity serves as a place for the comparator logic as well as the place to instantiate the two needed pipeline instances. The inputs of this parent entity are connected to the inputs of both pipelines. However, the pipeline external outputs (e.g. register file input, data cache input, etc.) take their values from the output of the first pipeline only. This could be done since both pipelines will be kept in synchronization.

3.3.3 Intermediate Signals Exporting

In order to make the intermediate signals of the integer pipeline available to the comparator, those signals were added as outputs in the integer pipeline port. The choice was to compare the inputs to each stage. Since most inputs to a stage are represented by registered signals in the pipeline, this choice was made to allow the coverage of the registered signals directly before any combinational logic. Another crucial reason for this choice is mentioned below.

3.3.4 Comparator

In order to detect faults within the pipeline, the intermediate signals of the two instances of the pipelines are compared bit-wise. The mechanism yields one fault signal per stage that would be asserted when a difference is seen between the two pipelines.

The comparing logic is illustrated in *Figure 3.3* though on a much smaller scale. In the figure, only two signals (A and A'), each two bits wide (.1 and .2), are compared. The resulting fault signal is the output of the OR gate in the figure. As in the figure, the fault signals are combinationally computed.

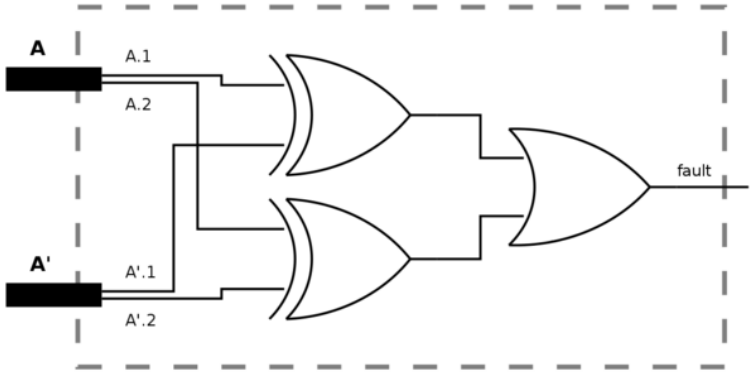


Figure 3.3: *Comparison logic*

Below, the two possible scenarios for the place of the comparator are investigated: one, with the comparators operating on the stage outputs and the other, with them operating on the stage inputs.

As can be seen in *Figure 3.4*, the combinational logic of the pipeline stage has to stabilize its results first, then the comparing logic has to do its computation before the resulting signals can be registered. Assuming the stages' combinational logic critical path delay is T_c and the delay for the comparator is T_d, the total critical path delay would be $T_c + T_d$. While this approach is easy, especially when there is a need to take a decision whether there has been a fault or not in the last stage, it would directly impact the performance of the processor.

On the other hand, as seen in *Figure 3.5*, the addition of the combinational logic in parallel to the pipeline stage logic would not cause a longer critical path. This holds true only if the pipeline combinational logic is larger than the comparison tree. The comparison tree depth is $log_m(n)$ where m is the number of inputs to a CLB/LUT and n is the number of signals to be compared. The comparison tree in this case should be only a few levels deep and for most pipeline implementations this condition should hold true. In other words, the critical path would be $max(T_c, T_d)$ in this case.

As mentioned before, the fault signals are computed after the inter-stage registers. This allows for a very important feature of the design: it does not affect the instruction cycle length.

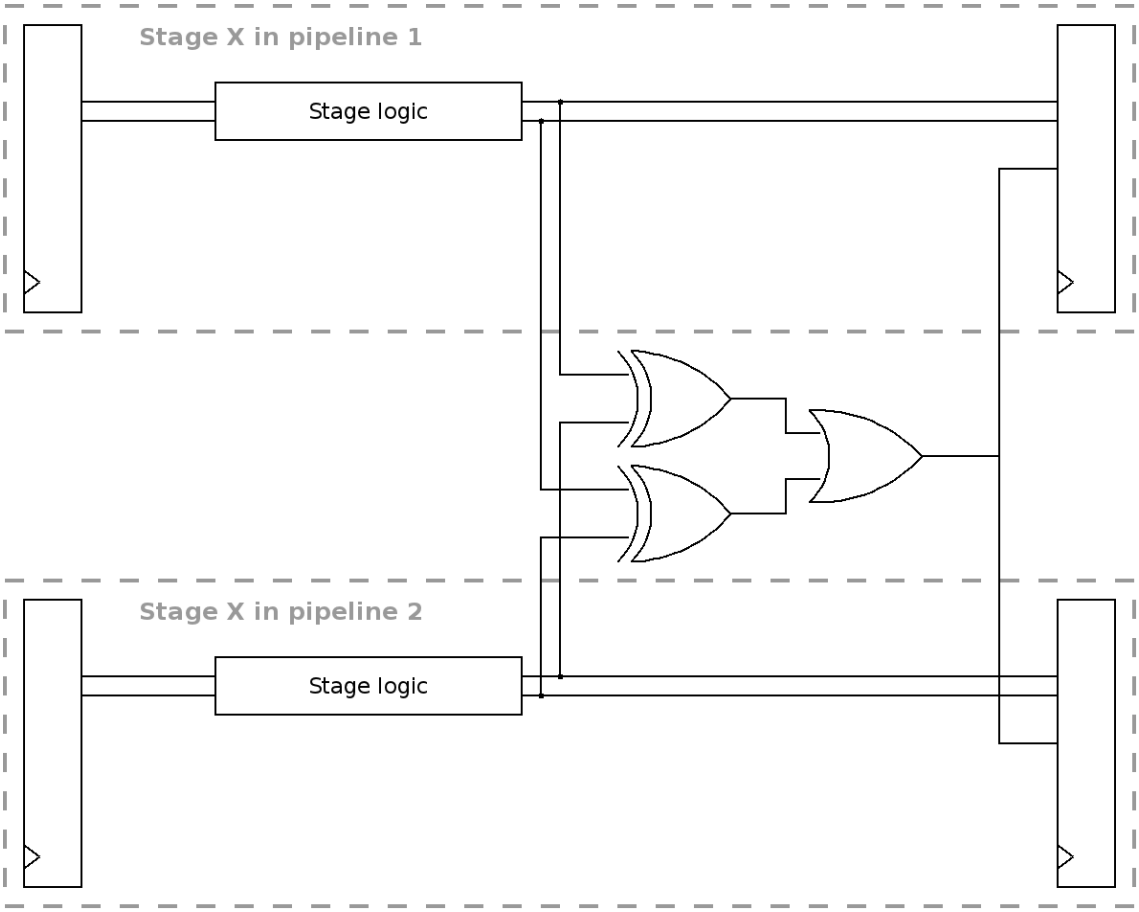


Figure 3.4: *Comparator in series with stage logic*

In the presented comparator entity, the intermediate signals from the two different pipeline instances are compared and a fault signal is created for each stage. The fault signal is asserted when there is a mismatch and deasserted otherwise. If a fault occurs in any of the gates in the detection tree, this will cause the assertion of the fault signal as well. Thus, still causing the fault detection mechanism to detect the fault. However, the fault detection mechanism might fail in case of more than one fault in the same clock cycle. Faults in user bits, that occur within the comparator, assert a fault signal when there is actually no faults in the pipeline. In this case, the only side effect would be the restart of the pipeline. However, if a configuration fault that forces a fault signal to '0' occurs, the design will not operate anymore and the system will potentially fail due to fault accumulation.

3.3.5 Fault Signals

Since there is a need to localize which instruction in the pipeline was affected by the fault, one fault signal is computed for each stage. This design has two main advantages:

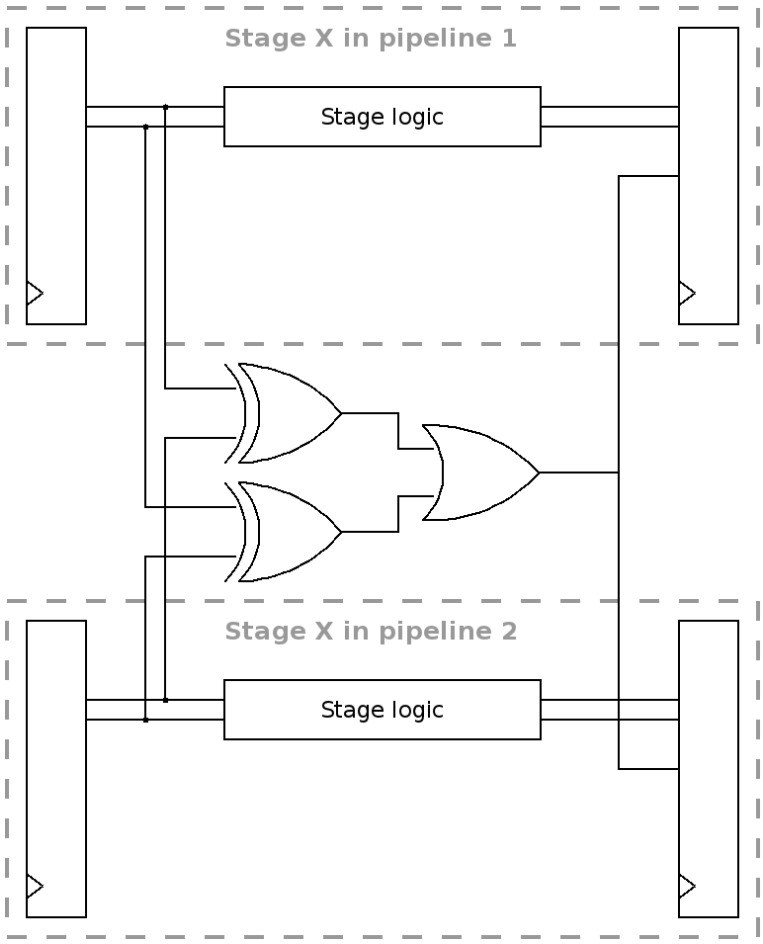


Figure 3.5: *Comparator in parallel with stage logic*

- The parallel comparison trees (one for each stage) will be able to perform the computations faster and thus not introducing unnecessary delays in the pipeline.
- The presence of a fault signal for each stage will allow the localization of exactly which instruction was affected by the fault.

Non-Pipelined Fault Signals

The straight-forward approach to implement the fault signals is as follows: whenever a fault is observed, all instructions younger than the erroneous instruction and the erroneous instruction itself are annulled and the pipeline is branched immediately to the erroneous instruction's address. This approach would definitely work, but there exists some complications with it. For instance, annulling all instructions that are younger than the erroneous one requires complicated logic thus occupying a large area. Moreover, having to select the branch target address from seven different possibilities

also requires larger logic. For those reasons, the choice was made to use the pipelined fault signals in the presented implementation.

Pipelined Fault Signals

In order to implement the branching and annulling of instructions in the simplest possible way, the branching is taken only when the erroneous instruction reaches the write-back stage. Thus, the fault signals are registered and propagated through stages along with the instructions. To this end, a fault signal register was added at the output of each stage. Each stage computes the value for this register as an OR between the previous stage's registered fault signal and the fault signal for the current stage.

The resulting architecture operates as follows:

- The fetch stage generates no fault since it is assumed that the inputs coming from the instruction cache unit are correct.
- Each stage forwards the fault signal unchanged to the next one, or asserts it if there is an error detected in that particular stage.
- The write-back stage branches to correct the fault once it sees an asserted fault signal or if there is a fault detected in the write-back stage itself.

Based on the pipeline fault signals, the register file write enable signal is disabled when the fault reaches the write-back stage to prevent the writing of erroneous data. A diagram of the pipelined fault signals implemented can be seen in *Figure 3.6*.

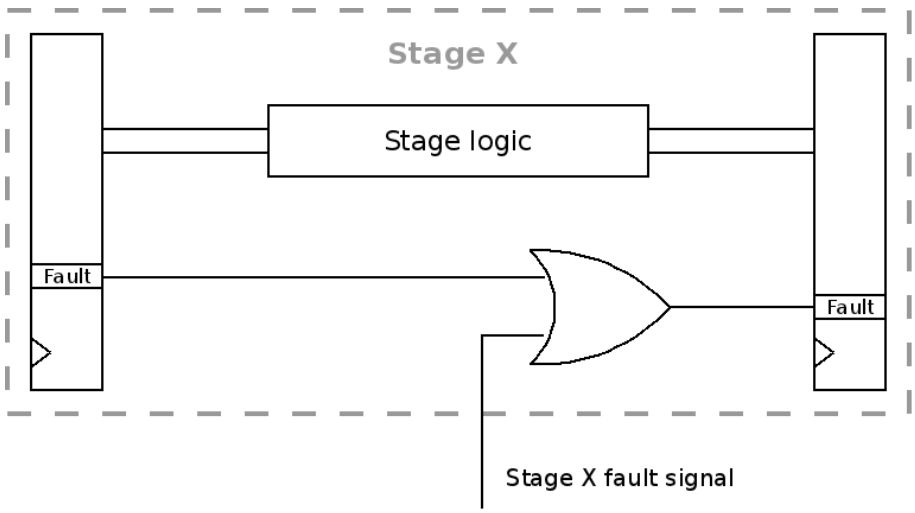


Figure 3.6: *Pipelined Fault Signals*

3.3.6 Register File Inputs

In addition to the intermediate signals that are monitored, the register file needs to be protected from erroneous writes. To accomplish that, a comparison tree that computes a fault signal for the register file write operation is constructed. The comparison tree does not compare all register file inputs, instead, only the ones involved in a write operation (write address, write data and write enable). This signal is then included in the write-back stage comparison tree so that a fault in the write-back stage will be detected in case of a fault in the write to the register file. This will cause the pipeline to restart to that instruction and thus repeat the erroneous write operation correctly.

On the other hand, the read operations from the register file are a potential place for faults as well. A fault on the signals specifying the read address will cause a wrong value to be read and thus, potentially, a wrong value to be computed and written. The register file read operation signals are also watched. The generated fault signal is included in the decode stage comparison tree so that a fault in the decode stage will be declared when a fault is detected in the reading of the register file. This is done because the computations responsible for the calculation of the register file read signals are done in the decode stage.

3.3.7 Annulling Instructions

As the aim is to correct faults by flushing the pipeline and re-executing all instructions that were present at the moment a fault occurred, there was a need to assure that those instructions will not change the state of the pipeline so that the execution will proceed as expected.

When a fault occurs, the registered fault signal for that stage will be asserted and then propagated along the pipeline. When the write-back stage sees the fault signal asserted while the instruction is not annulled, it will assert an annul-all signal which will cause the annul bit for all the instructions in the pipeline to be set and those instructions to cause no effect. The instruction at the write-back stage at this moment is effectively annulled. This holds true because the branch information is overwritten by the branching and the register file write enable signal is deasserted when the instruction in write-back stage is erroneous.

3.3.8 Branching

When the fault signal reaches the write-back stage, the fetch stage branches to the address of the instruction currently present at the write-back stage. After this branch, the execution will begin at the previously erroneous instruction.

3.3.9 Configuration Faults

As mentioned before, a configuration fault should be declared only when a fault at the same stage and same instruction is observed after the rollback. This functionality is implemented in the comparator entity.

In the comparator, a down counter is set to '7' once a fault is observed in any stage. Then, the counter is decremented at the end of every instruction cycle. This allows the counter to be in synchronization with the entry of new instructions to the pipeline. When the counter reaches '0', the previously erroneous instruction is back at the same place in the pipeline. That particular stage is observed for faults. If one exists then the permanent fault signal is asserted, otherwise, it is assumed that the fault was in the user bits and that it was effectively corrected by the rollback.

Chapter 4

Validation

This chapter introduces the methods used to validate the proposed design. *Section 4.1* explains the fault model used and how the set of faults to be simulated were generated. The details of the simulation and fault injection done to calculate the fault coverage are given in *Section 4.4* and *Section 4.5* respectively.

4.1 Fault Model

In the fault injection simulations for the implemented fault-tolerant version of the LEON3 processor, a fault model is needed. The fault model needs to represent two types of faults [AT05]:

faults in user bits, which would be corrected upon the writing of a new value to a flip-flop or signal.

faults in FPGA configuration memory, which cannot be corrected without a partial or complete reconfiguration of the FPGA.

Those two types of faults are explained in more details in *Section 2.2*. In order for the fault model to reflect the correct behavior, a closer look at the cell library used for synthesis is needed. The behavior of the different cells and the possible faults that could affect them need to be examined.

4.1.1 Cell Library

Xilinx Synthesis Technology (XST) was used to synthesize the design and produce a netlist representing the fault-tolerant design. XST uses the *unisim* and *simprim* cell libraries. The following entities are instantiated in the design: look-up tables (LUT), flip-flops (FDE), input buffers (IBUF), output buffers (OBUF), logic gates (e.g. XOR), and Multiplexers (MUX). Each of the entities has different fault locations that represent

different fault types. Only a random sample of the possible faults was simulated. In the simulations, faults are injected in the LUTs, FDEs, and I/OBUFs. This was done because logic gates and MUX's constituted only 2% of the entities within the design. *Subsection 4.4.4* and *Subsection 4.4.5* explain the details of how faults are injected into each of the entities.

4.1.2 Faults in User Bits

Cause

Faults in user bits are caused by bit flips in the user-defined design. As discussed in *Section 2.2*, those faults are correctable after a write to the affected signal.

Effect

Entities in the generated netlist correspond directly to the entities affected by the fault. Thus the effect of the faults and the model can be directly applied to those entities. Faults in user bits affect the entities used in the design in the following way:

LUTs, Logic Gates and MUX's: a bit flip in the inputs or outputs.

FDEs and I/OBUFs: a bit flip in the inputs, outputs or registered content.

Model

Since entities within the netlist take their inputs from other entities outputs, it is redundant to inject faults into both inputs and outputs. In the used model, faults are injected only at entities outputs. Moreover, in the simulation library, for FDEs and I/OBUFs, the registered signal is the same as the output. Thus, no separate injections are needed. User bit faults are injected as bit flips in the aforementioned locations.

4.1.3 Faults in Configuration Memory

Cause

Faults in the configuration memory are caused by bit flips in the FPGA's SRAM configuration memory. Those faults could be either in the CLB configuration or in the configuration memory of interconnects. Those fault are only correctable by partial/full reconfiguration.

Effect

CLB configuration faults can affect the memory of a LUT or the other configuration memory within a CLB. The effect of faults in LUT memory would be a bit flip in that memory content. The effect of faults in the rest of the memory in a CLB are manifested as altered functionality as seen in *Subsection 2.1.1*. Similarly, faults in the routing configuration are manifested as faulty connections.

Model

Faults in the LUT memory can be directly modeled by injecting bit flips in the LUT table memory. However, the simulation model of the cell library does not directly represent the programmable switches and the configurable CLBs. For this reason, an approximation of the behavior of those faults is needed. This approximation does not affect the results of the simulation as long as it represents faults that are not correctable without reconfiguration.

CLB configuration and routing configuration faults are modeled by a stuck-at model for the entities' outputs.

4.2 Fault Simulation

4.2.1 Software Functional Test

To detect faults, a software functional test runs on the processor and its results are used. Three different software were used: coremark, a jpeg decoder and the grlib system test. The different tests required different amounts of time to finish the simulations:

grlib system test: about one minute real time

coremark: about 17-20 minutes real time

jpeg decoder: 30+ minutes real time

During an initial run, the grlib system test was found to have the highest fault coverage and the lowest execution time. Due to the relatively short time available to complete the thesis, grlib system test was chosen to run all the simulations.

4.2.2 Fault Injection

In the simulation, configuration faults and use bit faults are injected differently. User bit faults are injected after $400\mu s$ of processor time, and the injection repeated every $100\mu s$. On the other hand, configuration faults are injected at $400\mu s$ of processor time and no further action is taken by the simulation environment.

4.3 Failure Modes

The fault injection simulation returns a single character for each fault injected independent of its type. Each character has a specific meaning. Those are presented below:

'0' This return value indicates that the test software finished correctly and did not detect any faults.

'1' This return value indicates that the test software finished correctly and detected a fault.

'C' This return value indicates that the test software never returned any value. In this case, the most probable scenario is that the pipeline halted for some reason because of the injected fault. It could be also returned in the case that the test software is taking too long to finish, in which case, it is considered as faulty behavior as well. The simulation is terminated after roughly 125% of the original test software runtime. The original runtime is based on a simulation done without any faults injected.

'P' This return value indicates that the configuration fault detection mechanism has detected a fault in the configuration memory of the FPGA.

4.4 Implementation of the Environment

The fault injection environment was reused from another experiment performed by the department in the past. Some modifications had to be done in order to adapt it to the needs of this thesis. This section explains how the used environment works.

4.4.1 Generating the Fault List

To generate the set of faults to be injected into the design during the fault injection, a programming script was used. The script reads the generated netlist in VHDL and distinguishes between the different entities used. The script generates a separate file for each of LUT, FDE, IBUF and OBUF. The files include the simulation path to each entity and in case of the LUTs, they also include the original value of SRAM inside the LUT. This value is used to compute a modified value which is only one bit different from the original value. The simulation environment takes those files as input and runs the simulations.

Another script was used to parse the files. It outputs one unified file that contains all the faults to be injected in the simulation. Details about the different types of faults that are simulated are given in *Subsection 4.4.4* and *Subsection 4.4.5*.

4.4.2 Cluster Simulation

The injection environment runs on a cluster of computers in a client-server model. It is basically a set of Tool command language (Tcl) and bash scripts that run on all

machines where the simulations are performed. In addition to that, there is a set of scripts that runs on the server.

The server scripts are responsible for keeping track of the machines participating in the simulation, distributing the tasks to them and collecting and archiving the results. The server scripts write the results to a text file indicating the fault and the symbol representing the result of the simulation.

The scripts running on the individual machines wait on the server's input, perform the computation and reply with a return value to the server.

4.4.3 ModelSim

ModelSim is the tool chosen to perform the simulations. ModelSim is a powerful HDL simulation tool from Mentor Graphics.

4.4.4 User Bits Faults

User bits faults are presented in the fault injection environment by a line in the faults file. The line starts with the character 'F', followed by the simulation path of a signal where the fault should be injected. Flips are applied to FDEs, LUTs and I/OBUFs.

Injection of Faults into LUTs, FDEs and I/OBUFs

Faults are injected into the LUT, FDE and I/OBUFs by depositing a value on the output signal. The value to deposit can be either a '1' or a '0'. To eliminate redundant injections, the state of the registered signal is checked and the inverted value is injected. This reduces the injected faults to only one injection per entity and eliminates the unnecessary injection cases which would consume a lot of time in simulation.

4.4.5 FPGA Configuration Faults

Faults in FPGA configuration memory are represented by two different models: 'P' type faults and 'I' type faults.

The 'I' type faults exactly match faults that would change a LUT memory bit. They represent faults that would cause a LUT to perform a different function other than the intended one. In the faults file, 'I' type faults are represented by 'I', the path in the simulation, followed by a value that should be written to the SRAM. This value is one bit different from the original value. Thus, it represents a single SEU.

'P' type faults on the other hand are an approximation for routing configuration faults. In the simulations, they are represented by a stuck-at fault for some signal. As mentioned before, they are not an exact model of the behavior of a fault in a routing switch, but for the scope of this thesis they are detected in the same way. They are injected at the outputs of FDEs, LUTs and I/OBUFs.

'P' type faults are represented in the faults file by a 'P', the path in the simulation, followed by the value to be injected. For each signal, two faults are generated, one with a value of '0' and the other with a value of '1'.

4.4.6 Injecting F Type Faults

'F' type faults are injected in the environment by using the ModelSim command "force". The "-deposit" argument is used to guarantee that the signal will be free to change after the fault was injected.

4.4.7 Injecting P Type Faults

The injection of 'P' type faults is done using the ModelSim command "force". The "-freeze" argument is used to guarantee the permanent effect of a configuration fault.

4.4.8 Injecting I Type Faults

Injecting 'I' type faults required some preparations before the experiment. The simulation library used was modified to allow for the writing to a LUT memory during the simulation by using a variable for the memory. During the simulation, the fault is injected by using the ModelSim "change" command followed by the variable to be changed and the new value. The new value is calculated during the generation of the fault list and it has only one bit different from the original value guaranteeing the injection of a single fault.

4.5 Calculating Coverage

Two results are given for the experiments:

- Coverage for faults in user bits
- Coverage for faults in FPGA configuration memory

The coverage of faults in user bits is calculated using 'F' type faults only and the FPGA configuration memory coverage is calculated using 'P' and 'I' type faults.

Each of the two results given is divided further into smaller categories. Those categories are based on the pair given by "Original design simulation":"Fault-tolerant design simulation". For instance, if a fault causes a result of '1' in the original design and a result of '0' in the fault-tolerant design, it will be under the category 1:0.

To summarize this and give a better understanding, a description of a good behavior of the system is given. A 100% fault coverage for both user bits and configuration memory would mean that a 'P' should be seen for all faults of type 'P' and 'I' that caused misbehavior in the original design. This implies that categories 1:0, 1:1, 1:C, C:0, C:1 and C:C should be completely empty. A 100% fault coverage would also mean that only '0' should be seen for all faults of type 'F'. This implies that categories 0:1, 0:C, 0:P, 1:1, 1:C, 1:P, C:1, C:C and C:P should be completely empty.

Some categories might seem redundant at first. For instance, 1:1 and 1:C, since they both represent misbehavior with the original design and the fault-tolerant design. Despite that, the distinction between those categories is necessary to understand how the fault-tolerant design modified the behavior of the processor. This understanding is necessary for debugging and improving the proposed design.

Chapter 5

Evaluation

Chapter 4 introduced the methodology used to validate the proposed design. This chapter will present the results gathered from the experiments.

5.1 Fault Coverage for User Bits

5.1.1 Proposed Mechanism

Table 5.1 lists the percentages of user bits faults in each category for the proposed mechanism.

Original\FT	P	0	1 and C
1 and C	14.7%	1.7%	1.2%
0	48.2%	30.6%	3.3%

Table 5.1: *Fault coverage of the proposed mechanism for faults in user bits*

Around 83% of faults that caused misbehavior in the original design were either detected as configuration fault or recovered from. However, only 12% of those were recovered from. As can be seen in *Table 5.1*, many of the faults in user bits were detected as configuration faults. The reasons behind this are discussed further in *Section 5.5*.

5.1.2 Alternative Mechanism

Table 5.2 lists the percentages of user bit faults in each category for the alternative mechanism.

Original\FT	P	0	1 and C
1 and C	28.2%	0.0%	0.0%
0	49.5%	22.2%	0.0%

Table 5.2: *Fault coverage of the alternative mechanism for faults in user bits*

The alternative mechanism detected 100% of all faults in user bits that caused misbehavior in the original design.

5.2 Fault Coverage for FPGA Configuration Memory

5.2.1 Proposed Mechanism

Table 5.3 lists the percentages of configuration faults in each category for the proposed mechanism.

Original\FT	P	0	1 and C
1 and C	22.1%	0.1%	2.8%
0	24.3%	50.1%	0.3%

Table 5.3: *Fault coverage of the proposed mechanism for faults in configuration memory*

In other words, 88% of all configuration faults that caused misbehavior in the original design were detected by the detection mechanism.

5.2.2 Alternative Mechanism

Table 5.4 lists the percentages of configuration faults in each category for the alternative mechanism.

Original\FT	P	0	1 and C
1 and C	20.5%	0.0%	0.0%
0	10.3%	69.2%	0.0%

Table 5.4: *Fault coverage of the alternative mechanism for faults in configuration memory*

The alternative mechanism detected 100% of all faults in configuration memory that caused misbehavior in the original design.

5.3 Area Overhead

5.3.1 Proposed Mechanism

The integer unit uses 2835 slices. It is duplicated twice, thus yielding a total of 5670 slices. The comparator logic uses 1205 slices. Therefore, the total usage for the fault tolerant design is 6875 slices. This represents 142.5% of area overhead. Duplication with Comparison is associated with an area overhead of more than 100%. Triple Modular Redundancy is associated with an area overhead of more than 200%.

5.3.2 Alternative Mechanism

The comparator in the alternative mechanism occupies 544 slices. Thus the total area overhead is 119.1%.

5.4 Time Overhead

5.4.1 Proposed Mechanism

The critical path length has increased by only 2.1% compared to the original pipeline. The time overhead added by the design is zero in case of no faults. The time overhead for recovery from user bits faults is seven instruction cycles per SEU fault. The instruction cycle time varies depending on many factors, among which are cache misses. Thus a percentage value cannot be calculated.

5.4.2 Alternative Mechanism

The alternative mechanism does not require any changes to the pipeline and therefore does not change the critical path.

5.5 Problems with Proposed Design

The proposed design did not yield a high probability of recovering from faults in user bits. The reason for this is that most faults in user bits do not get overwritten by the rollback. For instance, a fault in the multiplier inputs does not get overwritten in the rollback unless there is a multiply instruction in the pipeline. This causes most faults in the user bits to be detected as configuration faults or the pipeline to be stuck in continuous rollbacks. However, the pipeline could be modified in such a way to do

a write to all signals, including the do-not-care ones. This should resolve the problem with the unrecovered faults.

The alternative design, on the other hand, yields an excellent fault coverage for both configuration faults and user bits faults.

Chapter 6

Conclusion

6.1 Summary

In Field-Programmable Gate Arrays (FPGAs), Single Event Upsets (SEUs) can happen in two locations:

- User bits
- Configuration memory.

The two locations have different effects. SEUs in user bits behave exactly like SEUs in traditional Application-Specific Integrated Circuit (ASIC) designs; they can be corrected upon a write to the faulty location. On the other hand, SEUs in the FPGA configuration memory have a semi-permanent behavior; they cannot be corrected until a reconfiguration of the FPGA is done.

In this thesis, fault-tolerance techniques targeted at ASIC [BQS07, BMS07, BW89], FPGAs [CCS00, AT05, KNCR04] and microprocessors [GPV02] have been investigated. This thesis proposed a technique that would be suitable for soft core microprocessors running on FPGAs.

A technique was developed using duplication with comparison for fault detection, rollback for fault correction and for configuration fault distinction. The technique provides an advantage over other techniques in that it reuses the fault recovery mechanism for the detection of configuration faults. In contrast, scrubbing and readback both require a fault-tolerant design to be running on the FPGA in order not to cause misbehavior until the fault is corrected. They also do not correct faults in user bits, which again requires a fault-tolerant design.

The proposed design was implemented on the LEON3 soft core processor. The final design was synthesized using Xilinx Synthesis Technology and a VHDL netlist was extracted. Fault injection experiments were performed on the netlist using simulation. A fault model was derived for the simulations. The model exactly matches SEU behavior in user bits. However, it only approximates the behavior of SEUs in FPGA configuration memory. The area overhead of the proposed design was found to be about 142%.

The proposed mechanism did not yield the expected fault coverage due to reasons specific to the LEON3 pipeline. However, an alternative mechanism was proposed. The mechanism detects faults independent of their location. The alternative mechanism has an area overhead of 119%. It was able to detect all faults that cause misbehavior of the pipeline, whether they are in the configuration memory or in the user bits.

6.2 Outlook/Future Work

This section will provide pointers to areas where further research and experiments are needed.

6.2.1 Adding Dynamic Reconfiguration

Due to the limited time span of this thesis, it was not possible to extend the work to implement dynamic reconfiguration for the FPGA. The presented techniques could be augmented with dynamic reconfiguration to recover from faults. This would allow an uninterrupted operation of the pipeline under the effect of SEUs in user bits and in FPGA configuration memory.

6.2.2 Implementing Recovery Mechanism in TMR

Currently, the recovery mechanism is not implemented in a fault-tolerant way. It has some points of failure that could cause it to malfunction if certain SEUs hit those points. The proposed mechanism can be hardened by employing selective TMR.

6.2.3 Optimizing Overhead

The design presented by this thesis compares all the signals that are operated on by the pipeline stages. The signals watched for mismatches could be augmented or decremented based on the required fault coverage. It could be beneficial to watch different sets of signals starting from all signals until watching only the pipeline outputs or even parts of the outputs. This would yield smaller comparators and thus smaller area overhead. However, this is likely to come at a cost of smaller fault coverage. An optimal point in this compromise between area overhead and fault coverage could be investigated.

6.2.4 Improving User Bits Fault Recovery

The user bits fault recovery mechanism did not yield the optimal results. This is largely due to signals that are not re-written by the instructions. The pipeline could be modified to write a predetermined value (e.g. '0') to do-not-care signals. This could improve the recovery mechanism's coverage.

Bibliography

[AT05] Ghazanfar Asadi and Mehdi B. Tahoori. Soft error rate estimation and mitigation for SRAM-based FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 149–160, New York, NY, USA, 2005. ACM. 1, 8, 11, 20, 28, 39

[BBB+04] M. Bellato, P. Bernardi, D. Bortolato, A. Candelori, M. Ceschia, A. Paccagnella, M. Rebaudengo, M.S. Reorda, M. Violante, and P. Zambolin. Evaluating the effects of SEUs affecting the configuration memory of an SRAM-based FPGA. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 584 – 589 Vol.1, 2004. 1, 14

[BMS07] Cristiana Bolchini, Antonio Miele, and Marco D. Santambrogio. TMR and Partial Dynamic Reconfiguration to mitigate SEU faults in FPGAs. In *Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, pages 87–95, Washington, DC, USA, 2007. IEEE Computer Society. 10, 39

[BQS07] Cristiana Bolchini, Davide Quarta, and Marco D. Santambrogio. SEU mitigation for sram-based fpgas through dynamic partial reconfiguration. In *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, GLSVLSI '07, pages 55–60, New York, NY, USA, 2007. ACM. 10, 15, 39

[BW89] A.E. Barbour and A.S. Wojcik. A general constructive approach to fault-tolerant design using redundancy. *Computers, IEEE Transactions on*, 38(1):15 –29, January 1989. 10, 39

[CCS00] Carl Carmichael, Michael Caffrey, and Anthony Salazar. Correcting Single-Event Upsets through Virtex Partial Configuration: Application Note 216. on web: http://www.xilinx.com, 2000. 11, 12, 39

[FCS+00] Earl Fuller, Michael Caffrey, Anthony Salazar, Carl Carmichael, and Joe Fabula. Radiation Testing Update, SEU Mitigation, and Availability Analysis of the Virtex FPGA for Space Re-configurable Computing, presented at the IEEE Nuclear and Space Radiation Effects Conference. In *in Proc.*

International Conference on Military and Aerospace Programmable Logic Devices, 2000. 14

[GPV02] C. Galke, M. Pflanz, and H. T. Vierhaus. On-line Detection and Compensation of Transient Errors in Processor Pipeline-Structures. In *Proceedings of the Proceedings of The Eighth IEEE International On-Line Testing Workshop (IOLTW'02)*, IOLTW '02, pages 178–, Washington, DC, USA, 2002. IEEE Computer Society. 12, 13, 39

[JAR+94] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: the MEFISTO tool. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 66 –75, June 1994. 14

[KNCR04] Fernanda Lima Kastensmidt, Gustavo Neuberger, Luigi Carro, and Ricardo Reis. Designing and testing fault-tolerant techniques for SRAM-based FPGAs. In *Proceedings of the 1st conference on Computing frontiers*, CF '04, pages 419–432, New York, NY, USA, 2004. ACM. VI, 6, 12, 39

[Lev00] Régis Leveugle. Fault Injection in VHDL Descriptions and Emulation. In *Proceedings of the 15th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, DFT '00, pages 414–, Washington, DC, USA, 2000. IEEE Computer Society. 14

[QGK+05] H. Quinn, P. Graham, J. Krone, M. Caffrey, and S. Rezgui. Radiation-induced multi-bit upsets in SRAM-based FPGAs. *Nuclear Science, IEEE Transactions on*, 52(6):2455 – 2461, 2005. 14

[SKK+02] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389 – 398, 2002. 8

[VH98] J. Villasenor and B. Hutchings. The flexibility of configurable computing. *Signal Processing Magazine, IEEE*, 15(5):67 –84, September 1998. 4

[VSC+04] M. Violante, L. Sterpone, M. Ceschia, D. Bortolato, P. Bernardi, M.S. Reorda, and A. Paccagnella. Simulation-based analysis of SEU effects in SRAM-based FPGAs. *Nuclear Science, IEEE Transactions on*, 51(6):3354 – 3359, 2004. 1, 14

Statement/Declaration - Erklärung

Statement/Declaration

This is to certify that:

i. the thesis comprises only my original work towards the Master's Degree

ii. due acknowledgment has been made in the text to all other material used.

Andrew Boktor
19^{th} of April, 2011

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Andrew Boktor
19. April 2011