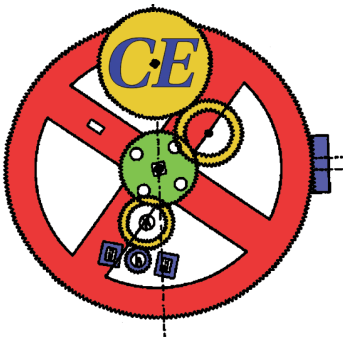

Computer Engineering Laboratory
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.ewi.tudelft.nl/

2016

MSc THESIS

Towards a fault tolerant RISC-V softcore

Wietse F. Heida

CE-MS-2016-09

Abstract

As predicted by Gordon E. Moore in 1975, the number of transistors has doubled every two years over the last decades. This technology scaling led to a higher performance of Integrated Circuits (ICs) like processors, but have also made these devices more susceptible to Single Event Effects (SEEs). SEEs are caused by transistors changing state unintended due to particles disposing significant amounts of energy when they hit an IC. In this thesis, the design, implementation, and verification of a fault tolerant RISC-V processor, which can detect two errors and correct one error, is presented. The RISC-V Instruction Set Architecture (ISA) is an open-source architecture which defines all interactions between the software and hardware. Two designs, the ECC-based and Hybrid design, are presented which both use Error Correction Codes (ECC) and N-Modular Redundancy (NMR) for adding fault tolerance to the softcore. Hamming Single Error Correction, Double Error Detection (SECDED) codes were selected as the ECC coding scheme. The hybrid design, which uses NMR in the pipeline and ECC for memory elements, is chosen for implementation, because it has a lower complexity, less Single Points of Failure (SPOFs) and a higher estimated clock frequency. The design's fault tolerance is verified with fault injection through saboteurs, which are placed at predefined locations in the design. The test architecture for the design consists of a verification suite, a simulation-based environment using ModelSim and an on-board test environment using a Spartan-6 Field Programmable Gate Array (FPGA). The verification tests showed that the hybrid design can mitigate all injected single and double faults, without having a single failure occurring. The design did not meet the clock frequency and resource utilization targets set by Technolution B.V. because of long delay paths and large decoders. The design, however, allows for improvements like the insertion of extra pipeline stages and parallel decoding and data processing.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Towards a fault tolerant RISC-V softcore

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Wietse F. Heida
born in Hoofddorp, The Netherlands

Computer Engineering Laboratory
Department of Software and Computer Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Towards a fault tolerant RISC-V softcore

by Wietse F. Heida

Abstract

As predicted by Gordon E. Moore in 1975, the number of transistors has doubled every two years over the last decades. This technology scaling led to a higher performance of Integrated Circuits (ICs) like processors, but have also made these devices more susceptible to Single Event Effects (SEEs). SEEs are caused by transistors changing state unintended due to particles disposing significant amounts of energy when they hit an IC. In this thesis, the design, implementation, and verification of a fault tolerant RISC-V processor, which can detect two errors and correct one error, is presented. The RISC-V Instruction Set Architecture (ISA) is an open-source architecture which defines all interactions between the software and hardware. Two designs, the ECC-based and Hybrid design, are presented which both use Error Correction Codes (ECC) and N-Modular Redundancy (NMR) for adding fault tolerance to the softcore. Hamming Single Error Correction, Double Error Detection (SECDED) codes were selected as the ECC coding scheme. The hybrid design, which uses NMR in the pipeline and ECC for memory elements, is chosen for implementation, because it has a lower complexity, less Single Points of Failure (SPOFs) and a higher estimated clock frequency. The design's fault tolerance is verified with fault injection through saboteurs, which are placed at predefined locations in the design. The test architecture for the design consists of a verification suite, a simulation-based environment using ModelSim and an on-board test environment using a Spartan-6 Field Programmable Gate Array (FPGA). The verification tests showed that the hybrid design can mitigate all injected single and double faults, without having a single failure occurring. The design did not meet the clock frequency and resource utilization targets set by Technolution B.V. because of long delay paths and large decoders. The design, however, allows for improvements like the insertion of extra pipeline stages and parallel decoding and data processing.

Laboratory : Computer Engineering
Codenumber : CE-MS-2016-09

Committee Members :

Advisor:	dr. ir. J.S.S.M. Wong, CE, TU Delft
Chairperson:	dr. ir. J.S.S.M. Wong, CE, TU Delft
Member:	dr. ir. A.J. van Genderen, CE, TU Delft
Member:	dr. ir. T.G.R.M. van Leuken, CE, TU Delft
Member:	ir. A.D. Wiersma, Technolution B.V.

Dedicated to my parents

Contents

List of Figures

List of Tables

List of Acronyms

ALU Arithmetic Logic Unit

ASIC Application Specific Integrated Circuit

BFM Bus Functional Model

BRAM Block Random Access Memory

CP Check Prediction

CPI Cycles Per Instruction

CPU Central Processing Unit

CRAM Configuration Random Access Memory

CRC Cyclic Redundancy Check

DAED Double Adjacent Error Detection

DED Double Error Detection

DIVA Dynamic Implementation Verification Architecture

DRAM Dynamic Random Access Memory

DVF Device Vulnerability Factor

DMR Dual Modular Redundancy

ECC Error Correction Codes

EEMBC Embedded Microprocessor Benchmark Consortium

FF Flip Flop

FI Fault Injection

FIFO First In First Out

FIT Failure In Time

FPGA Field Programmable Gate Array

FT Fault Tolerant

HWFI Hardware-based Fault Injection

IC Integrated Circuit

ISA Instruction Set Architecture

LET Linear Energy Transfer

LUT Lookup Table

MBU Multiple Bit Upset

MLU Module Level Utilization

MUT Module Under Test

NMR N-Modular Redundancy

PLL Phase Locked Loop

RISC Reduced Instruction Set Computer

RDL Register Definition Language

SCTB Self Checking Test Bench

SEC Single Error Correction

SED Single Error Detection

SECDED Single Error Correction, Double Error Detection

SEE Single Event Effect

SEFI Single Event Functional Interrupt

SEL Single Event Latchup

SEMBU Single Event Multiple Bit Upset

SET Single Event Transient

SEU Single Event Upset

SFI Simulation-based Fault Injection

SoC System on Chip

SRAM Static Random Access Memory

SPOF Single Point of Failure

STD Standard

STMR Selective Triple Modular Redundancy

SWFI Software-based Fault Injection

TAED Triple Adjacent Error Detection

TED Triple Error Detection

TMR Triple Modular Redundancy

UART Universal Asynchronous Receiver/Transmitter

XST Xilinx® Synthesis Technology

VHDL Very High Speed Integrated Circuit Hardware Description Language

List of Symbols

Symbol	Description	Unit
d_H	Hamming distance	-
d_{min}	Minimum Hamming distance	-
f_{max}	Maximum attainable clock frequency	MHz
L	Number of Lookup Tables	-
m	Length of the check word	-
n	Length of the code word	-
k	Length of the data word	-
t_{comb}	Maximum combinational path delay	ns
t_{input}	Minimum path from an input pad to any synchronous element.	ns
t_{period}	Minimum period from any synchronous element to another.	ns
t_{output}	Maximum path from any synchronous element to an output pad.	ns

Acknowledgements

Performing my graduation project and writing this thesis has been a long journey. I would like to thank several people for their help and support during this process.

I would like to express my gratitude to Technolution B.V. for providing me with the opportunity to work on their RISC-V project. Without their support, this thesis would not have been possible. It was an interesting experience to work on the softcore and use different languages like VHDL, C, and Python in one project. I learned a lot and gained valuable experience for my future career.

I would like to thank my daily supervisor Ard Wiersma, for the useful discussions at our weekly meetings, helping me out whenever I was stuck at a difficult problem and for providing valuable feedback on the drafts for my thesis. I would also like to thank my other company supervisors Jonathan Hofman for offering me the project assignment and providing feedback on the designs and my progress in general, and Sjors Hettinga for letting me bother him whenever I had questions. Furthermore, I would like to thank my fellow interns Daan, Alex, and Maarten for their collaboration on the RISC-V project and Dennis, Kevin, and Wouter as well for the making the hours at the office enjoyable.

I would also like to express my gratitude to Stephan Wong for being my university supervisor, helping me out with the academic side of the project and providing guidelines and feedback for my thesis. I would also like to thank Arjan van Genderen and Rene van Leuken for participating in my graduation committee.

Last but not least, I would like to thank my parents for always supporting me in my journey through life.

Wietse F. Heida
Delft, The Netherlands
August 23, 2016

1 Introduction

In 1965 Gordon E. Moore, one of the co-founders of Intel, predicted that by 1975 65,000 components would be included in a single silicon chip [13]. Over the years the extrapolation of this prediction has become known as Moore's Law: the number of transistors on a single chip will double approximately every two years [14]. Although this may seem like a very bold statement, industry and science have proved that this law is a self-fulfilling prophecy. This increase resulted in massive advances in computational power of devices using integrated circuits, but also a large decrease in power consumption, as the size of each transistor became smaller and smaller and circuits could operate at lower supply voltages. These advances did, however, come with disadvantages: smaller transistors will fail sooner due to higher wear and tear and integrated circuits became susceptible to soft errors or Single Event Effects (SEEs).

Soft errors are caused by transistors changing state unintended due to particles disposing significant amounts of energy when they hit an Integrated Circuit (IC). Because the transistors became smaller and smaller the amount of energy required to induce such state changes also became smaller. These effects were first reported in 1975 for ICs used in satellites in space where large amounts of radiation are present [15]. Initially, these effects could be avoided by using older technology with larger transistors in space equipment. However, over the years the transistor sizes have shrunk so much, that nowadays soft errors are also experienced at normal flight altitudes for civilian aviation [16] and even at ground level [17]. Applications like respiratory support devices, heart rate monitors, and other live supporting equipment in hospitals can make life-threatening mistakes due to a single transistor accidentally changing its state. However, also equipment used for access control in buildings like smart cards could be influenced by soft errors, resulting in potential security breaches. Using older technology for such applications is not an option, as the demands for more computational power and lower power consumption are rising. Therefore, other solutions are required to solve this problem.

Traditionally, Central Processing Units (CPUs) are used for most applications as the development of software is relatively cheap. For most applications the performance of a modern CPU is sufficient, but for more computationally intensive applications like radar or data security applications, this is not enough. Such applications use a lot of Digital Signal Processing and cryptography algorithms, which cannot complete their tasks in real-time when executed on a CPU. These applications typically use a Field Programmable Gate Array (FPGA), a reconfigurable chip used for designing dedicated hardware for computationally intensive tasks. The dedicated hardware blocks used in FPGAs can perform these very intensive tasks much faster than a CPU, but are more expensive to develop than software. These specialized designs often include a softcore, a small CPU, which can perform simple tasks which do not require more expensive hardware blocks. Applications used in high safety or security critical environments could use

FPGAs instead of CPUs if these are better suited for the requirements of the application. If such an FPGA design includes a softcore, it should, however, be able to handle soft errors, as mistakes are not allowed. But how could a softcore be modified so it can handle soft errors?

Today, there are multiple softcores available on the market, developed by open source communities or (large) companies. The softcores developed by companies are often proprietary and can only be used when expensive licenses are acquired. Most of these softcores are not suited for handling soft errors, so dedicated solutions need to be added to the design. However, most companies do not provide the source files of the design and do not allow modification of their design. Open source softcores provide more possibilities for modification as the source files of the design are available, but often lack the support of a software ecosystem which severely limits the usability of these softcores in any application.

In the last few years, a new open source initiative emerged: the RISC-V Instruction Set Architecture (ISA). An ISA is the architecture which defines all interactions between the software and the hardware of a processor including available data types and instructions. The community provides a complete specification of the ISA along with an extensive software ecosystem, which can be used for software development. The only thing left to design for individual members of the community is the processor itself. This gives the freedom to design a softcore with all kinds of extra features required for specific applications. The open source RISC-V ISA provided Technolution B.V. with the opportunity to design a softcore which can handle soft errors for future opportunities in the high safety and security critical domain. This thesis will describe how this softcore was designed, implemented, and verified and which solutions were used to handle soft errors.

1.1 Problem statement, Thesis objectives & Methodology

The starting point of this thesis is the baseline implementation of the softcore made by Technolution B.V., which uses the classic five-stage Reduced Instruction Set Computer (RISC) pipeline [18] and is implemented in the Very High Speed Integrated Circuit Hardware Description Language (VHDL). The to be developed variant of this softcore is targeted at high safety and high-security applications, so the correct execution flow should not be influenced by an SEE without such errors being detected. The problem statement can be formulated as:

> *How can an RISC-V softcore using a classic five-stage RISC pipeline be extended such that a maximum of two concurrent Single Event Effects in the pipeline or in data or address words cannot influence the correct execution flow of the processor without being detected in a modular and efficient way?*

Based on the problem statement the thesis objectives can be formulated as:

1. To design a modular and efficient solution for detecting and correcting errors in the pipeline and in data or address words in an RISC-V softcore.

2. To implement the design as a proof of concept on a Xilinx Spartan-6 while taking platform independence into account.

3. To verify the implemented solution and thus prove its correctness.

4. To provide possible options for improving the designed solution.

To find a solution for the proposed problem statement and to achieve the thesis objectives, a research methodology is defined. This methodology will be used to structure the work in this thesis:

- Study the possible causes of Single Event Effects in softcores.
- Study existing solutions for detection and correction of Single Event Effects in processors.
- Design possible solutions for the target system based on existing solutions found in literature.
- Implement the most suitable solution found for the existing RISC-V softcore in a modular and efficient manner.
- Test the implemented solution comprehensively and thus prove the correctness of the implemented solution.
- Analyze the results of the implementation and compare these with the baseline.
- Draw conclusions based on the results from both the implementation and verification phase
- Provide possible options for future development of the implemented solution.

1.2 Thesis Outline

This thesis is structured as follows:

In Chapter 2 the background for this thesis will be presented. Both natural and human-induced causes of Single Event Effects and their effects in softcores will be investigated. Based on these causes different solutions for mitigating these effects in softcores will be studied as these will provide the basis for the designs made in this thesis. These solutions will be investigated for the two major regions in a processor: the pipeline and the memory elements, as different solutions are available for these regions. Lastly, methods for verifying these solutions will be investigated as such methods need to be used to verify the implemented design.

In Chapter 3 the RISC-V softcore and Instruction Set Architecture, which form the basis for the project, will be introduced. Based on the problem statement, the targeted fault model will be defined along with the design criteria for which each design will be evaluated. The design options found in Chapter 2 will then be evaluated for these design criteria and a selection of the most suitable options for the design process will be made.

With the selected design options, two designs based on two different concepts will be made. These designs will be evaluated for each of the design criteria which will result in a design choice for the implementation. This evaluation will also include a design cost estimate based on the cost of the baseline implementation and some basic block implementations. Finally, the method for verifying the selected design will be presented based on the presented literature.

In Chapter 4 the implementation for selected parts of the design will be described. These selected parts are the most important parts of selected design, which will largely define the capabilities and the cost of the implemented design. The block implementation descriptions will include the block design, unit test, and the cost estimates. Furthermore, the changes to the System on Chip (SoC) like the addition of extra register blocks will be described. The memory mapping used by the SoC during verification will also be discussed.

In Chapter 5 the result of the verification of the design will be discussed. Firstly, the test architecture used will be presented, which includes the locations for fault injection, the test applications used and the test cases. The test results will include the instruction coverage of the test applications, which provide an insight into the comprehensiveness of the test suite, and the fault injection results. Besides the test results, the results from synthesis will also be presented as these provide an indication of the cost of the design and its efficiency.

In Chapter 6 the thesis will be summarized, the main contributions will be listed, and possible options for future improvements to the design will be presented.

2 Fault Tolerance in softcores

In the current IC technology Single Event Effects (SEEs) caused by strikes of particles are regularly occurring. As these SEEs can influence the functionality of a chip or the result of the computation, there is a need for fault tolerance in mission critical applications, like control in space and avionics or medical appliances at ground level. In this chapter, the source of SEEs will be investigated along with the resulting effects. Known techniques for dealing with these effects will be presented and evaluated for detection and correction capabilities, efficiency and overhead. This study will finish with a short overview of techniques which can be used for verification of the error mitigation capability. But first, the RISC-V ISA will be introduced.

2.1 RISC-V Instruction Set Architecture

The RISC-V Instruction Set Architecture was originally developed by researchers from the Computer Science Division at the Electrical Engineering and Computer Science department of the University of California, Berkeley. The initial purpose of the ISA was to support computer architecture research and education as proprietary ISAs are often not suitable for those purposes. Nowadays, the ISA is also aiming to become a standard open source architecture used by industry for their implementations and is therefore now governed by the RISC-V Foundation. Some of the goals of the current RISC-V ISA are [19]:

- A completely open ISA that is freely available.
- A real ISA suitable for direct native hardware implementation.
- An ISA that avoids "over-architecting" for a particular microarchitecture style or implementation technology, but always allows efficient implementation in any of these.
- An ISA separated into a small base integer ISA and optional standard extensions while also supporting extensive user-level ISA extensions and specialized variants.
- An ISA supporting both 32-bit and 64-bit address space variants.
- An ISA with support for highly parallel multicore or manycore implementations, including heterogeneous multiprocessors.

The original developers from Berkeley could have used a commercial ISA for their purposes, as these often have a large and widely supported software ecosystem and come with large amounts of documentation and tutorials. The developers, however, decided

against this, based on the following disadvantages [19] which outweigh the mentioned advantages:

- Commercial ISAs are proprietary.
- Commercial ISAs are only popular in certain market domains.
- Commercial ISAs come and go.
- Popular commercial ISAs are complex.
- Commercial ISAs are not enough to bring up applications.
- Popular commercial ISAs were not designed for extensibility.
- A modified commercial ISA is a new ISA.

As mentioned in the goals of the ISA, it contains a small base integer ISA, which can be extended with additional extension, both standard as defined in the user-level specification[19] or user-defined. The 32-bit base integer instruction set is called RV32I and its 64-bit brother is called RV64I. The standard extensions as defined by the specification are:

- **M:** integer multiplication and division
- **A:** atomic instructions
- **F:** single-precision floating-point
- **D:** double-precision floating-point
- **Q:** quad-precision floating-point

For stable software development additional targets, called RV32G and RV64G, are defined, which are a combination of the base instruction set (RV32I or RV64I) plus selected standard extensions (IMAFD[1]). Some other standard extensions are mentioned in the user-level specification but are still under development at the time of writing.

The RISC-V ISA in general uses a load/store architecture which divides instructions into two separate classes: memory access instructions and ALU operations. In the RV32I instruction set the memory access instructions are the load and store instructions which have versions for byte, half word, and word data accesses. The ALU operations for RV32I are basic logic, arithmetic, branch, jump and timing instructions.

[1] I stands for the base integer instruction set

2.2 Single Event Effects

SEEs are caused by particles disposing significant amounts of energy when they hit an IC. These particles can originate from different sources [20]:

- Cosmic radiation consisting of large amounts of high energy particles which can liberate nuclei when colliding with molecules in the IC.
- Atmospheric interactions with cosmic radiation creating highly energized particles.
- Particles (small traces of uranium and thorium) in the packaging materials emitting alpha particles while decaying.
- Boron-10 isotopes, used in polysilicon doping, substrate doping, and borophosphosilicate glass, creating alpha particles when struck by low energy neutrons.

There are two ways in which high-energy particles can cause SEEs in an IC: direct or indirect ionization [21]. When an inciding particle has enough energetic charge, the particle can free up electron-hole pairs in the semiconductor material along its path. When the particle has lost all of its energy, it will come to rest. The amount of energy deposited is denoted by the Linear Energy Transfer (LET) which is the energy loss per unit path length of a particle as it passes through a material [21]. If the charge of a group of freed electron-hole pairs is high enough in the right place (typically around PN junctions) it will cause unwanted effects in the device. These effects can be categorized as follows [20][21]:

- **Single Event Upset (SEU):** a change in the state of a memory element. The susceptibility to SEUs depends on the type of memory element (Static Random Access Memory (SRAM), Flash etc).
- **Single Event Transient (SET)** a voltage pulse in combinatorial circuits which can result in erroneous data values propagating through the circuit.
- **Single Event Functional Interrupt (SEFI)** an effect which triggers the FPGA to (temporarily) lose functionality.
- **Single Event Multiple Bit Upset (SEMBU):** a single particle strike which causes more than one error in the FPGA.
- **Single Event Latchup (SEL):** a short circuit which disrupts the functioning of the FPGA and could lead to the destruction of the FPGA. An SEL can only be solved by power cycling the device.

SEUs were first reported to have happened in a satellite in 1975 in [15]. These effects were initially tackled by using radiation hardened hardware, but during the 1990s the number of manufacturers who offered such devices rapidly decreased. This was due to the higher costs of such devices and the fact that such devices did not use state-of-the-art technology. The sensitivity to SEEs was increased significantly as the feature size decreased. This resulted in SEEs not only happening in space but also in avionics [16] and on ground level [17]. This created the necessity to implement fault tolerance in mission-critical devices on the ground level as well as in space.

2.3 Fault Attacks

Besides SEEs caused by physical causes SEEs can also be caused on purpose. Fault attacks are used by attackers to disturb the functioning of a processor or retrieve secret information. Fault attacks are mostly used for cryptography applications and can be induced by the following sources [22] [23]:

- **Glitch attacks:** Glitches in the supply voltage or the external clock can disturb the correct execution of the processor. Glitches in the supply voltage can cause the processor to misinterpret or skip instructions whereas glitches in the clock can cause data misreads or instruction misses.
- **Temperature based attacks:** By using an IC outside of the specified operating temperature range the functionality might be disabled.
- **Light attacks:** White light can induce currents because electric circuits are sensitive to photoelectric effects. Lasers can induce similar currents, but due to the directionality of a laser, these current can be targeted at specific areas of the circuit.
- **Radiation attacks:** X-rays and ion beams can also induce currents in the circuits and have the advantage that the packaging does not have to be removed for being effective, whereas this is needed for white light and lasers.
- **Magnetic attacks:** Magnetic pulses can create local currents on the surface of the component, which can generate a fault.

When launching a fault attack, there are two major issues the attacker needs to deal with: the time and place the fault occurs. The fault needs to happen at the right time in the execution flow for it to have the right effect. The execution flow of the processor can be monitored by using side channel information like the power consumption, and the magnetic radiations or events like writes to external memory.

When the fault occurs at the right time, it also has to occur at the right place in the chip. Glitch attacks, variations in the supply voltage or external clock, are coarse techniques because they perturb the whole chip. Temperature magnetic, white light and X-ray based attacks can be directed to specific areas of the circuit, but the directionality of these sources can be limited, so the surrounding areas in the circuit could also be affected by these attacks. Laser and ion beams sources have a higher degree of directionality and can, therefore, be targeted at small areas of the circuit and thus have the best result when a fault in a specific component is needed.

Fault attacks can target different parts of a softcore: the inputs, memory banks, the data path or the control path. Furthermore, the precision of the induced fault attacks can range from single bits to complete words [24]. For all these combinations of targets and precision, the effects of a fault on the execution of processor will be different. FPGAs mostly use SRAM or Flash cells to store data from the core and the configuration data. [25] describes a light attack on SRAM cells which can be used to flip a single transistor or reverse engineer the memory address map. [26] improves this type of attack by using a laser instead of white light from a flashlight, which enables the attacker to prevent memory updates by exposing the memory control logic.

2.4 Impact of faults

When an SEE occurs in a softcore, it could affect the execution flow of the processor. Depending on the location and time of the SEE, the impact of the SEE will differ ranging from no effects on the correct execution flow to a failing execution flow. In analyzing the impact of an SEE, the terms fault, error, and failure are often used [27]:

- **Fault:** the cause of a malfunction.
- **Error:** the system state due to a fault which may cause a failure.
- **Failure:** the undesired effect observed in the system's delivered service.

An SEE is classified as a fault and often creates an error, but not all errors will cause a failure. Four scenarios are possible when an error is present in the system:

- **Correctable error:** The error is detected and corrected and will not cause a failure.
- **Detectable error:** The error is detected, but cannot be corrected. Uncorrectable errors can cause a failure depending on the system handling of this type of errors.
- **Silent error:** The error is masked by logic. These errors are not detected nor corrected and do not cause a failure.
- **Undetectable error:** The error is not detected nor corrected and causes a failure.

2.5 Forms of Redundancy

Fault tolerance can be provided by adding redundancy to a system, so the system will be able to mitigate the effects caused by a fault. Redundancy is defined as *having more of a resource than is minimally necessary to do the job at hand* [28]. The different forms of redundancy in a system can be divided into four main categories [28]:

- **Hardware redundancy:** incorporating extra hardware into the design to either detect or correct the effects of a failure in the system.
  - Static hardware redundancy: the form of hardware redundancy aimed at immediate masking of a failure.
  - Dynamic redundancy: the activation of spare components for replacing a currently active component which has produced a failure
  - Hybrid hardware redundancy: the combination of static and dynamic hardware redundancy
- **Information redundancy:** application of error detection and correction codes for the protection of information in memory or data communication systems.

- **Time redundancy:** re-execution of the same program on the same hardware for detection of transient faults.

- **Software redundancy:** using several different independent versions of a program for dealing with faults in the software.

Software redundancy will not be used in this thesis because this form only provides protection against software faults whereas SEEs are hardware faults. The applicability of time redundancy for a softcore is also somewhat limited because it can only compensate for transient faults. SEUs in the processor memory or the configuration memory cannot be mitigated with this form of redundancy. Some form of scrubbing (checking of configuration memory) or reconfiguration can be used to compensate for configuration errors, but this will not solve all limitations. Both hardware and information redundancy can be used effectively in softcores because they can mitigate both SEUs and SETs. But just like with time redundancy, information redundancy cannot compensate for configuration errors. In some implementation options, hardware redundancy can mitigate the effects of configuration errors, because a component has multiple instantiations within the system.

Depending on the form and level of redundancy applied, faults can be detected or corrected. With detection, a fault is only detected and the processor has to be restarted or rolled back to a previous state and restart execution for mitigation of the fault. If correction is possible, the erroneous result can be corrected so the processor can continue with the correct result and no restart or rollback is required. For correction, a higher level of redundancy is required than for detection, because the processor needs to know what kind of error has occurred. Some levels of redundancy combine error correction and detection capabilities into one system which can correct x errors and detect y where $x < y$.

Using redundancy ensures that the systems will maintain its functionality, but it will induce a penalty or overhead. In Table 2.1 the impact of the different forms of redundancy on the area and performance of a softcore are listed.

Table 2.1: The impact of redundancy on the area and performance of a softcore.

Redundancy	Area	Performance
Hardware	high	low
Information	medium	medium
Time	low	high
Software	low	high

2.6 Redundancy in the Pipeline

In a processor, most of the data is manipulated in the pipeline. If an SEE occurs in the pipeline, it should be intercepted before the result is committed to the register file or memory or a jump or branch is taken.

2.6.1 Triple Modular Redundancy

Triple Modular Redundancy (TMR) is a technique providing hardware redundancy by executing the same process in three different modules [29]. Once the modules are finished a majority voter will decide which outcome is the correct result. In Table 2.2 an example of the single bit majority voting procedure for TMR is presented. TMR can correct one error for single bit signals and possible detect two errors if the for multi-bit signals if the majority voting procedure is not applied per bit but per word.

Table 2.2: The single bit majority voting procedure for TMR.

Input	Output
000	0
001	0
011	1
111	1

One of the main drawbacks of this method is that the pipeline is triplicated which results in an area overhead of more than three times the size of the original pipeline. Furthermore, the error correction and detection capabilities are somewhat limited. More pipelines could be added to increase these capabilities, but this will result in a larger area overhead.

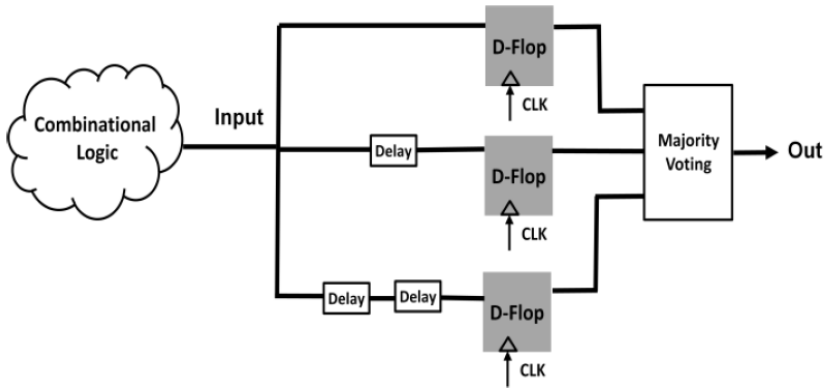

Figure 2.1: The TMR cell used in the modified ARM® Cortex®-R4 CPU [1].

There are techniques which optimize the overhead of TMR by applying it selectively in the pipeline. [1] describes a solution for the ARM® Cortex®-R4 CPU where all sequential cells are replaced by special TMR cells, consisting of three D-flip-flops and a majority voter to cancel the effects of an SEU (depicted in Fig. 2.1). To prevent SETs from propagating, delay elements are used to delay the input of Flip Flop (FF) two and three, so the SET will have faded out when these flip-flops sample the input. The choice of delay for these delay cells is critical for the robustness and efficiency of this solution. A short delay might result in a limited capability of detecting and canceling out SETs, but a large delay will result in a high clock frequency overhead. The frequency overhead

is caused by the fact that the minimum path delay of any combinatorial circuit should be larger than the delay of 2 delay elements. In the described implementation in 65 nm GP technology, the CPU clock frequency is 31%. The dynamic power overhead is 2.04x, and the area overhead is 2x which is considerably less than the overhead of a standard TMR implementation. The overhead could be further reduced if the cache sizes would be increased; the used cache size is 16 KB.

2.6.1.1 Selective TMR

For the implementation in [1] the designer has to implement the TMR himself, but there are also techniques available which can automatically identify the circuits which are sensitive to SEEs and implement the needed TMR for these circuits.

In [30] an algorithm for applying TMR to selected parts of the circuit, called Selective Triple Modular Redundancy (STMR), is described. The algorithm identifies the sensitive subcircuits based on the input signal probabilities which are propagated through the circuit. A subcircuit is marked sensitive if an SEU occurring in this circuit has a high probability of affecting one of the circuit outputs. For all sensitive circuits TMR is applied. If the output of a triplicated subcircuit is connected to only one other subcircuit a majority voter will be inserted to accumulate the results of the triplicated circuit. This algorithm ensures that only those gates which really need redundancy are triplicated, which results in a saving of about one third of area overhead compared to traditional TMR. However, STMR cannot guarantee that no single error will propagate through the circuit because the appliance of TMR is based on signal probabilities. It should also be noted that in some cases the result of the STMR algorithm will be the same as for traditional TMR, because of the choice of parameters and signal probabilities. But the largest limitation of the algorithm described is that it can only be applied to combinatorial circuits and not to sequential circuits which are also present in a processor pipeline.

In [31] another methodology for inserting STMR, which is applied to flip-flops instead of combinatorial circuits, is described. This methodology uses a graph-partitioning like algorithm, which are NP-complete, for partitioning the registers into two graphs: one for sensitive register, which are triplicated, and one for non-sensitive registers. These partitions are then optimized based on the cost function and reliability constraints, which are defined as the maximum percentage of wrong cycles allowed in each output of the target design. For reducing the runtime performance of the algorithm, optimizations are applied to the choice of the initial partition, so less candidate solutions in the search space have to be explored, before the optimal solution is found. The FFs for the initial partition are selected based on five parameters [31]:

- **Fan in:** the number of FFs which influence the state of the analyzed flip-flop.
- **Fan out:** the number of FFs which are affected by the state of the analyzed flip-flop.
- **Influence on output paths:** the number of primary outputs which are affected by the analyzed FF.
- **Proximity to outputs:** proximity of the evaluated FF to the outputs which are affected by it.

- **Feedback loops:** the output signal of a FF is fed back to its input, directly or through several previous stages.

The results show that the applied optimizations do result in a significant saving in run-time, because the number of explored partial solutions is reduced. The results also show that the algorithm can find a solution, which satisfies the reliability constraints. But if no wrong cycles are allowed the algorithm cannot find a solution which is better than TMR, so this algorithm is only useful when some errors are allowed.

2.6.2 Lockstep

So far only methods which triplicate hardware have been discussed, but duplication of hardware, called Dual Modular Redundancy (DMR), is also a feasible option for error detection. Error correction is not possible with DMR because the level of redundancy is too low.

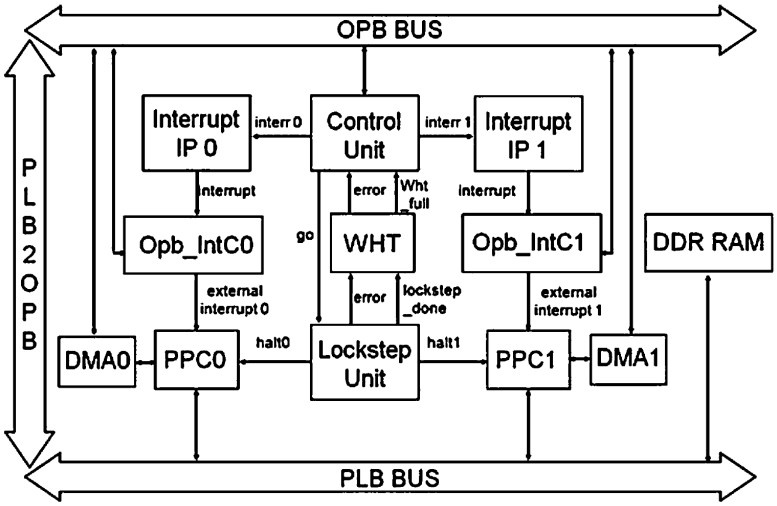

Figure 2.2: The DMR architecture using two embedded PowerPC cores as proposed in [2].

[2] describes a solution which uses DMR for providing fault tolerance for two embedded PowerPC cores in a Xilinx Virtex II Pro FPGA (depicted in Fig. 2.2). The two PowerPC cores (`PPC0` and `PPC1`) are both running the same program and are interrupted at predefined points for a consistency check. Locksteps can be defined by milestones in the program like data memory transactions. Large execution cycles, the time spanned by a lockstep, will reduce the performance penalty of the consistency checks, but can result in more damage caused by a fault. If a consistency check fails, the program needs to be brought back to a consistent state without faults. The processor can restart the program or load a previously saved checkpoint which did not contain faults from a safe (fault-immune) storage. The context is defined as the set of information needed to unequivocally define the state of a processor-based system. When the consistency check fails, a rollback operation is executed which should bring the processors back to a fault-free state. Initial results in [2] showed that the architecture is not affected by most

injected faults (around 97%). The faults that did result in wrong results were faults inserted in one execution cycle, but only manifested themselves in the next execution cycle and were thus saved in a context.

To eliminate the effects of faults which are saved in a context, further improvements were made in [32]. The main improvement is that several contexts are saved in the safe storage, so multiple rollback operations can be performed to bring the processor back to a fault-free state. However, saving more contexts is expensive, because safe storage blocks will typically consume more area than normal storage blocks. Therefore, a trade-off should be made, between the number of contexts to be saved and how often checkpoints are inserted. Checkpoints do not necessarily have to be inserted after every lockstep, but could also have multiple locksteps in between, so an execution cycle will span a larger portion of the execution flow. The first results in [32] showed that none of the faults resulted in not being detected and causing erroneous results, so the previously noticed problem was solved. The only faults remaining undetected are faults which cause both processors to produce the same erroneous result. The percentage of these faults varies with the interval between context savings: the smaller the interval, the higher the error rate. For 100.0% context savings the overhead is 8.6 times and for 16.7% context savings the overhead is still 5.6.

The large execution time overhead is caused mostly by the saving of the context which includes the whole data segments of an application. To reduce this overhead, the Write History Table (`WHT` in Fig. 2.2) was inserted in the architecture. The WHT contains all address-value pairs changed in an execution cycle. When the WHT is full, a consistency check is performed, and the data in the WHT is saved in a data segment mirror area instead of copying the whole data segment to the safe storage with the context. When the consistency check fails, the WHT is flushed, the context is rolled back, and the data segment is restored from the data segment mirror area. The results in [32] showed that for applications with large data segments the usage of a WHT increases the number of cycles per write compared to the architecture without WHT. For applications with small data segments, this improvement does not work.

2.6.3 DIVA

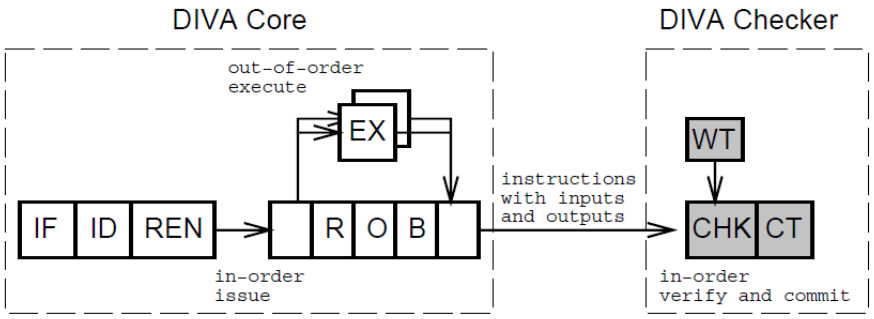

Figure 2.3: The dynamic implementation verification architecture for an out-of-order superscalar processor. [3]

Apart from TMR and lockstep, there are other techniques which can provide hardware redundancy without replicating the pipeline or elements of the pipeline. In [4] a processor architecture with a built-in checker unit is proposed based on the Dynamic Implementation Verification Architecture (DIVA) concept which was proposed in [3]. DIVA is an architecture which uses a dedicated checker for checking the results of a superscalar, out-of-order processor before the commit stage (depicted in Fig. 2.3). In [3] a DIVA checker is inserted to provide functional and electrical verification for the DIVA core by performing the calculation or fetch for the second time. The improved reliability of the checker is achieved by using a much simpler design for the DIVA checker than for the DIVA core and by using larger transistors for the checker so that it will be more resistant to SEEs. This should guarantee that the checker is 100% reliable and thus will always be able to correct all faults of the core and introduce no other faults. This approach is feasible for a complex processor implemented in an Application Specific Integrated Circuit (ASIC), but for a lightweight processor implemented on an FPGA, this approach does not provide the needed fault tolerance.

Therefore, a modified DIVA architecture was proposed in [4] which does not assume that the checker is fully reliable and uses a standard RISC pipelined processor instead of a complex out-of-order superscalar processor. When an error occurs, the pipeline is flushed and restarted at the instruction causing the fault instead of committing the result of the checker. As the memory is now protected by Error Correction Codes (ECC), there is no need to re-execute memory loads and stores, which eliminates memory contention between the core and the checker. In the provided example based on the LEON3 processor, the checker is inserted in the pipeline, by adding an extra ALU in the Exception stage and an extra stage for comparing the results of the Execute stage and the extra ALU (depicted in Fig. 2.4). This implementation provides hardware redundancy and a limited form of time redundancy. The results show that for an error rate of 10% the modified DIVA architecture has a 70% execution time overhead. In error free operation the performance overhead is 0% due to the forwarding paths added from the checker unit.

2.6.4 Error Correction Codes

In memories Error Correction Codes (ECC) are often used to provide protection against SEEs. But these codes can also be used to provide fault tolerance in processor pipelines as shown in this survey [33].

2.6.4.1 Fault Tolerant RISC (FT-RISC)

In [5] an implementation of a fault tolerant RISC processor using Single Error Correction, Double Error Detection (SECDED) Hamming codes is discussed (depicted in Fig. 2.5). Hamming codes are often used for protecting memories, but the extra bits are discarded once the data or instruction is used in the pipeline. This processor uses this data to perform error detection and correction in parallel to the computations in the normal datapath. The data fetched from the register file is first processed by the checker/correction units `CCA` and `CCB` to check if no faults are present in the data. The main Arithmetic Logic Unit (ALU) calculates the result of the instruction, while the

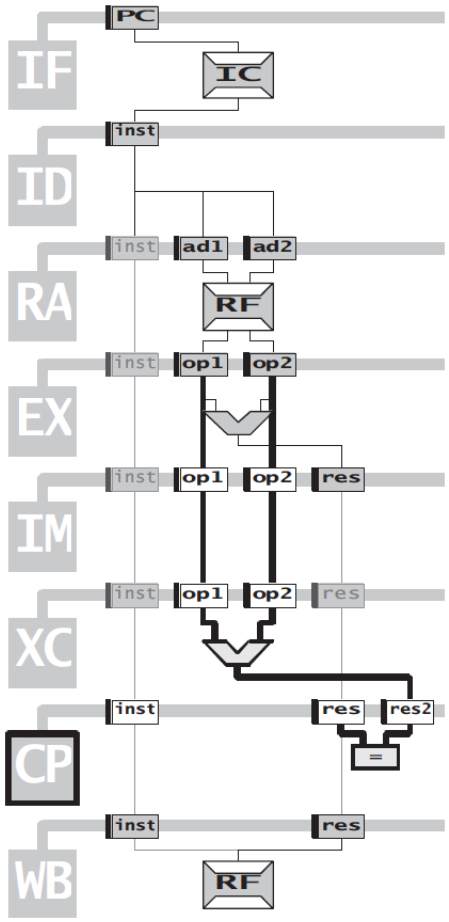

Figure 2.4: The modified DIVA architecture based on the LEON3 processor with the extra ALU and Compare stage. [4] RA = Register Access stage, XC = Exception stage, IM = Memory stage, CP = Compare stage

auxiliary ALU calculates the parity of the carry information from the input, P_{CY}, and the parity of the AND operation on the two input operands, P_{AB}. The parity ALU then calculates the parity information for the ALU result, P_R, based on the ALU inputs and not the ALU result. If the ALU result does not comply with the parity information, the result will be corrected in case of a single error or the processor will be interrupted in case of a double error.

This architecture has the advantage that it can detect two errors and correct one error, whereas some of the other solutions presented here do not have this capability. The results in [5] show that the fault coverage of faults occurring in the information hardware is very high (mostly 100% for single error correction and double error detection). Only faults occurring in the checker hardware might escape detection, but this could be solved by using modular redundancy. The required area for this architecture is most likely less than the area required for TMR, but [5] does not provide any data to support this claim.

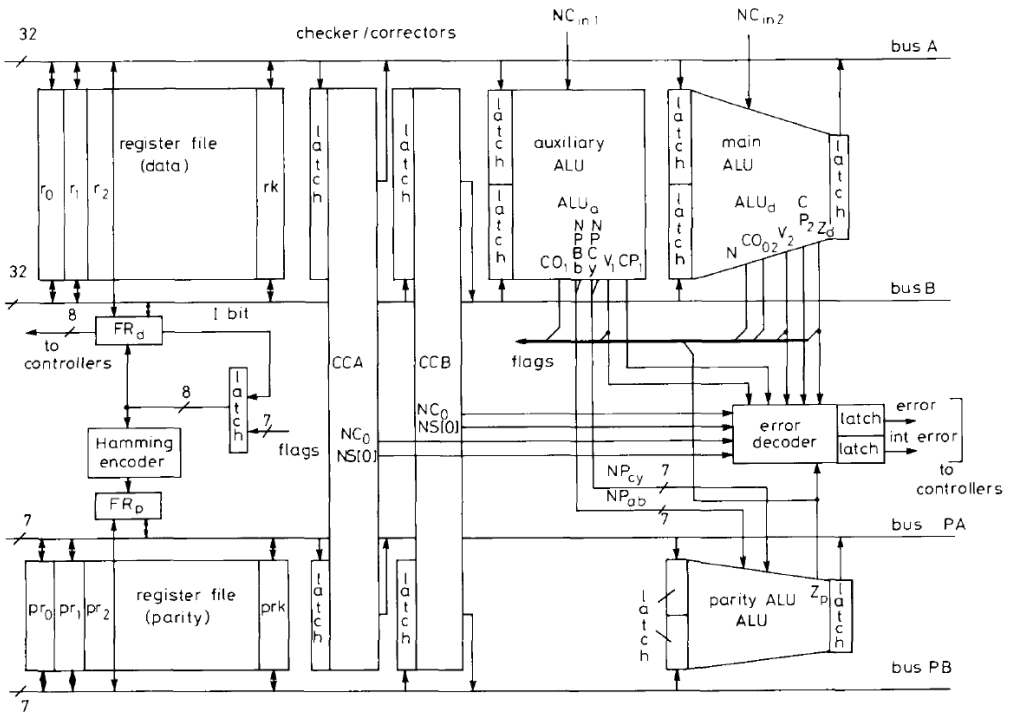

Figure 2.5: The self-checking and correcting 32 bit datapath of the fault tolerant RISC processor architecture [5].

2.6.4.2 Dependable embedded processor core

[6] describes another core which uses a (39,32) Hamming code to protect the data in the core. This core is MIPS32 compatible and has a five stage pipeline (depicted in Fig. 2.6). The pipeline registers are protected by an ECC encoder placed before the register bank to generate the parity information and an ECC decoder placed after the register bank to decode the data in the register bank (depicted in Fig. 2.7). If the calculation of this syndrome is too expensive, a byte parity check is used, where the encoder and decoder are placed at the same location as the ECC encoder and decoder. The register file is also protected by encoders and decoders. These encoders and decoders could be used per word, but this would double the size of the register file. To reduce the area overhead only one encoder for the write port and three decoders for the read port could be used. This can only be done if the probability of two consecutive soft errors in the same word is very small (depicted in Fig. 2.8) This solution does only provide protection against SEUs in the register banks between the pipeline stages and in the register file. SEFIs will not be mitigated because the parity information is calculated based on the input of the register bank or register file. For the target device, a ProASIC3E 1500-Std, an area overhead of 40% and a frequency overhead of 33% were measured. No results on the verification of the proposed architecture were presented.

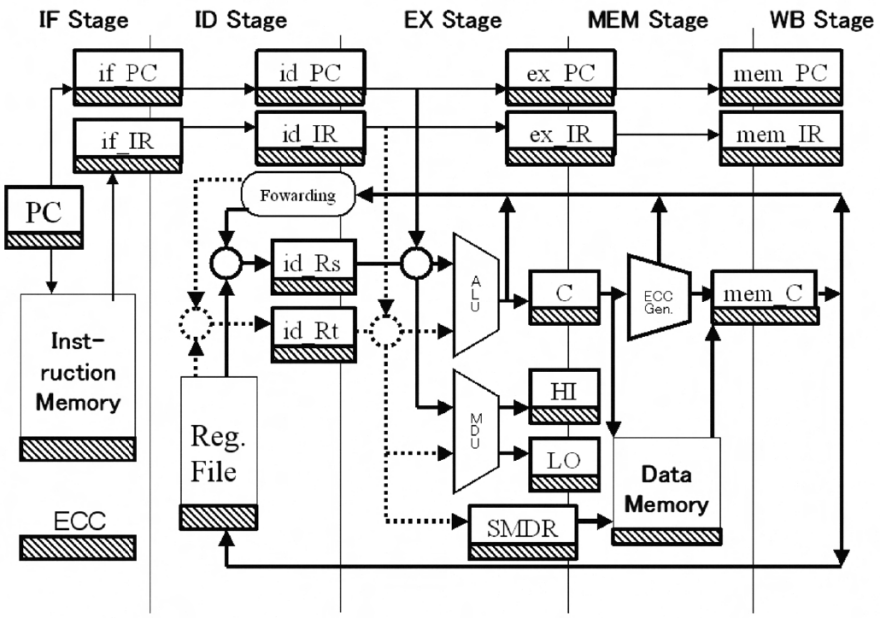

Figure 2.6: The five stage dependable embedded processor core architecture. [6]

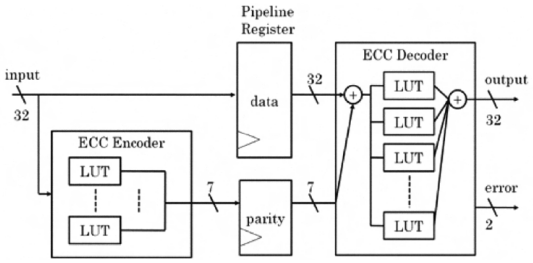

Figure 2.7: A pipeline register with encoder and decoder for the dependable embedded processor core. [6]

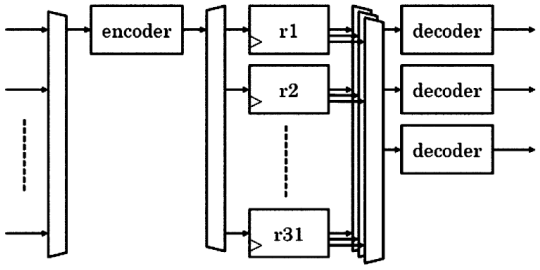

Figure 2.8: The register file of the dependable embedded core with one encoder for the write port and three encoders for the three read ports. [6]

2.6.4.3 Multi-Residue Codes

The FT RISC core [5] and the Dependable Embedded Processor core [6] both use linear parity codes for protecting the memory elements and the ALU. [7] describes a solution which uses multi-residue codes instead of linear parity codes for protecting the ALU. Linear parity codes often only detect single-bit or double-bit faults in one of the operands and not a double bit fault occurring in both operands which can be induced by a fault attack. Multi-residue codes provide several advantages over linear parity codes because the smallest operational distance (minimum weight for an undetected error for a given code under a given operation) is three instead of two. Furthermore, they are not affected by the carry corruption problem for instance (residue codes are described

in Section 2.7.2.6).

The presented solution in [7] uses a (48, 32) multi-residue code for protecting the ALU and the register file (depicted in Fig. 2.9). The encoder and checker unit is implemented in two cycles for reducing the area overhead, although this will induce extra performance overhead. The extra combinatorial circuits needed for the Residue ALU and the encoder and checker introduce more area (74%) and timing (46%) overhead compared to a DMR ALU. But when the register file is included, the use of multi-residue codes gives an area saving of 16 % compared to the DMR solution. Linear parity codes do not support multiplication natively, but multi-residue codes do. In this implementation, the multiplication is implemented in two cycles. When comparing the DMR ALU with multiplier against the full multi-residue code ALU, the last solution saves 24% area and has an only 9% longer critical path.

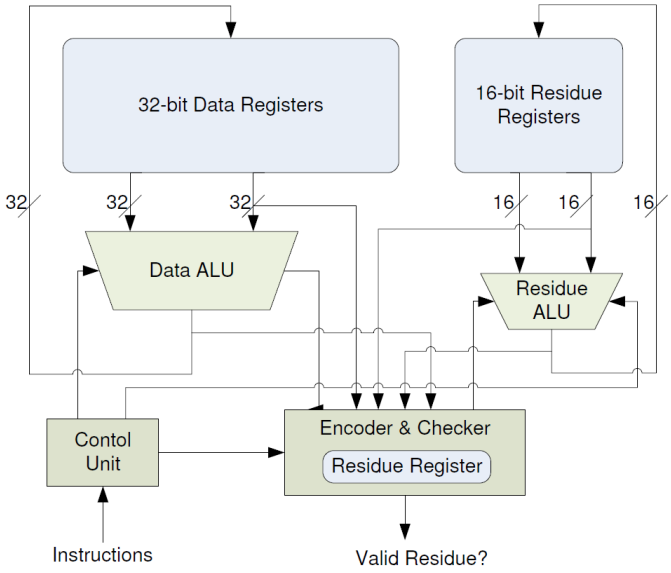

Figure 2.9: The optimized multi-residue code ALU. [7]

2.6.5 Watchdog

Another type of solution for providing fault tolerance in processors is the watchdog processor. A watchdog processor is a small and simple coprocessor used to perform concurrent system-level error detection by monitoring the behavior of the main processor [8] (depicted in Fig. 2.10). This error detection is performed in two phases: the setup phase and the checking phase. In the setup phase, the watchdog processor receives information about the processor and the process(es) running on the processor that need to be checked. In the check phase, the watchdog monitors the execution of the processor by collecting information and comparing it to the information received during the setup phase. If the collected and provided information do not match, the watchdog will notify the processor that an error has occurred and that it should restart the execution at a

point where the fault did not yet occur. The provided information for checking can be about memory access behavior, the control flow, or signals and the reasonableness of results. The architecture of the original processor does not have to be modified, the area overhead is relatively small compared to replication, and the error detection can be performed in parallel with the processor execution, which are clear advantages of this solution.

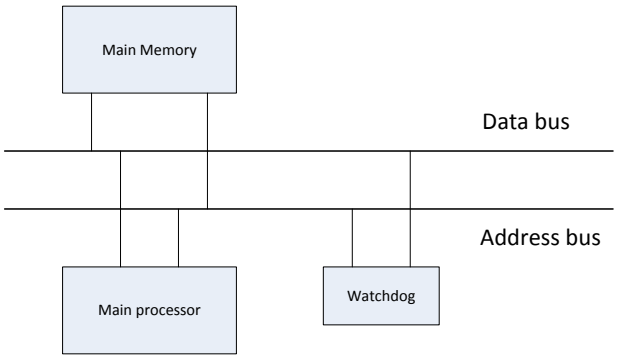

Figure 2.10: The watchdog processor architecture [8].

In [34] an implementation of a watchdog processor for detecting transient faults in the Motorola M6804 is presented. The watchdog processor and the processor are connected to each other through the system bus. The memory is protected by the internal memory of the watchdog processor which stores shadow variables of the most critical variables. A variable is defined as critical when it can produce wrong results when affected by a fault without causing the program to terminate incorrectly. The control flow is protected by comparing signatures of the executed Branch Free Blocks with a golden signature which is calculated off-line. In this particular implementation, a signature scheme based on regular expressions is used. The bus is protected by performing Automatic Repeat Requests (ARR) to check if the transmitted value is the same in different transmissions.

Although this implementation has several advantages like no changes to the processor architecture and concurrent error detection, there are also some disadvantages. The internal memory of the watchdog is limited so not all variables can be shadowed and thus not all variables can be monitored precisely. More internal memory will mean more susceptibility to memory faults, and a higher area overhead whereas less memory will mean less susceptibility and less overhead. Furthermore, the ARR will induce a significant performance penalty, because the time-consuming transactions have to be performed several times. The results also show that for large durations of the fault (more than 150 ps) the percentage of undetected errors rises significantly, so this solution is limited in the fault tolerance it can deliver.

The referred implementation showed that there are some disadvantages of using that particular watchdog processor, but there are more disadvantages for watchdog processors in general. In the setup phase information about the processor and the process

have to be provided for the checking phase. This information will have to be generated at compile time because the monitored processor cannot be used for this, as it is unreliable. So compiler support for watchdog architectures is required. Furthermore, the process's execution flow needs to be known beforehand, because the compiler needs to generate signatures. For some processes, this will be possible, but for processes which are controlling other components and thus are I/O dependent this can become complicated.

2.6.6 Time redundancy

So far all techniques described provided some form of hardware redundancy, but time redundancy can also provide a feasible solution for mitigating SETs because these effects are temporary.

2.6.6.1 DMT

In [35] the DMT architecture is presented which uses time redundancy. The DMT architecture uses two successive executions of the task code, called virtual channels, on a single physical channel to detect transient errors. Two dissociated datasets are used to prevent SEUs in the memory from affecting the execution of both virtual channels. Only the main results of the virtual channels are compared to keep the overhead as small as possible. This architecture is mainly implemented in software, so the only hardware support is the CESAM component, which provides memory protection. As this architecture only provides for duplication of the task execution, it can only detect faults and not correct any faults. If a fault is detected, the processor can skip the iteration and continue to the next iteration if this does not affect the operation of the system. The system is rolled back if the application cannot miss the result of the current iteration. The recovery phase is also executed in duplex because it is not known which virtual channel contains the fault. The reported protection against transient faults is above 98% when the kernel is also protected. But this protection comes at a cost: for a single-task application the performance penalty is four times and for multi-task applications, the penalty is three times.

2.6.6.2 Time shifted latching

[36] describes another time redundant solution which uses different moments in time for latching data. Two possible options for implementing the transient fault detection are depicted in Fig. 2.11. In the first option the output signal of the combinatorial circuit is delayed by a factor δ and in the second option, two different clock domains are used. All transient effects which are smaller than δ will be detected in which case the processor can be stalled to deal with the problem. This solution does not induce a performance penalty because the output of the output latch can be forwarded to the next circuit as soon as the data is latched. This will not interfere with the latching of data by the extra latch, as the output signal of the combinatorial circuit will not change for some time, because of propagation delays. This does, however, impose a constraint on the value of δ, which should be smaller than the minimum propagation delay of the combinatorial circuit. This solution can also be used for controlling the clock frequency because it can

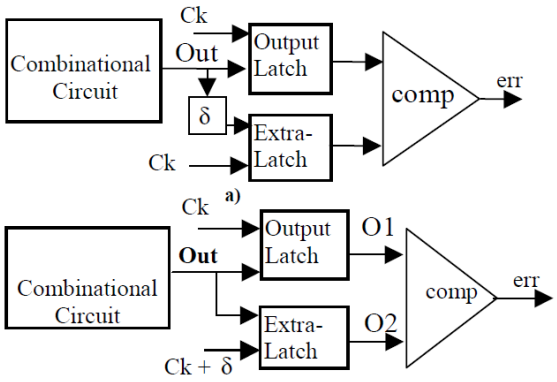

Figure 2.11: Transient fault detection implementation using a latch and a comparator [9].

detect timing issues. If the longest path in the combinatorial circuit is not frequently exercised, the frequency can be increased to speed up the circuit. If the longest path is used, the frequency can be temporarily lowered to allow the path to complete after which the frequency can be increased again. The implementation made in [9] shows that this solution can be implemented with a small area overhead and that indeed no performance penalty is induced. Furthermore, the detector was able to detect all timing issues and almost all soft errors.

In [37] another implementation of the time redundant solution is presented. This paper also shows that the proposed implementation is effective at detecting errors as this implementation introduces no more errors. Furthermore, it also shows some interesting results about the pulse width at different energy levels of the particles. In the used 0.25 micron technology the pulse width is below 540 ps for 5 MeV of LET, between 540 and 800 ps for 14 MeV and between 800 and 1000 ps for 30 MeV.

2.6.7 Comparison

In Table 2.3 the different solutions found in literature are listed and compared with each other. Based on the found solutions and their comparison five implementation options for providing redundancy in the pipeline can be defined:

- **N-Modular Redundancy (NMR):** N cores running in parallel, where checks will be performed on every data transaction.
- **Lockstep:** two cores running in parallel, where checks are performed at every clock cycle between the corresponding stages of the processor.
- **Error Correction Codes (ECC):** single core using ECC to protect FF banks and registers and prediction of the check bits to protect the ALU.
- **Watchdog:** a separate core which checks the execution of the main core based on information provided by the compiler.

- **Time redundancy:** code is executed N-times on a single core to check for faults and if possible correct them.

Table 2.3: Comparison of the solutions for adding redundancy to the processor pipeline. If no result is listed, the corresponding literature did not report these numbers.

	Types				Tolerance			Overhead		
Solution	SEU	SET	SEMBU	SEL	EC	ED	Det.	Area	Freq.	Perf.
TMR	✓	✓			1	1	-	3x	-	-
FF only TMR	✓	✓			1	1	-	2x	31%	-
STMR	✓	✓			1	1	-	$\leq$ 3x	-	-
Lockstep	✓	✓			0	1	97%	-	-	-
Improved Lockstep	✓	✓			0	1	100%	>5	-	
DIVA	✓	✓			0	1	-	-	-	<14%
Modified DIVA	✓	✓			0	1	100%	-	-	70%
FT-RISC	✓	✓	✓		1	2	100%	-	-	-
Dependable embedded core	✓		✓		1	2	-	-	-	-
Multi-residue ALU	✓	✓	✓		-	3	-	-	-	-
Watchdog	✓	✓		✓	0	1	100%	-	-	-
DMT	✓	✓	✓		0	1	98%	-	-	3-4
Time shifted latching	✓	✓			0	1	100%	-	-	-

2.7 Redundancy in Memory cells

Besides the pipeline, the processor also includes several memory cells, like the register file, instruction memory, and data memory. These cells, which are often implemented in Dynamic Random Access Memory (DRAM), SRAM or FFs, can also be affected by SEEs and therefore need to be protected. There are two basic methods for providing fault tolerance for memories: modular redundancy or Error Correction Codes (ECC).

2.7.1 Modular Redundancy

Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR), which were introduced in Section 2.6, can also be applied to memory cells. In this case, the individual FF or register is duplicated or triplicated and provided with some comparison logic. For DMR only error detection can be implemented, because the memory cells are only duplicated, and the provided redundancy is thus limited. TMR can correct n upsets per n-bit word if the voting logic is implemented per bit. But TMR memory needs three times more storage cells than a regular memory plus some extra logic cells for the majority voter, so the area penalty is considerable. [38] showed that for single registers TMR is a feasible solution regarding the area overhead compared to a Hamming code implementation. But for register files and embedded memories, Hamming code is the better option regarding the area consumption although the encoder and decoder introduce a performance penalty due to the extra delay on the critical path.

2.7.2 Error Correction Codes

Error Correction Codes (ECC) are all based on the principal of adding extra symbols to a data word to form a code word. These symbols are based on the symbols in the data word and thus provide redundancy such that errors in the data word can be detected or corrected [39]. There are two main categories of error correction codes:

- **Block codes:** These codes are applied to information on a block-by-block basis. The different blocks are independent of each other.
- **Convolutional codes:** These codes use the current input information as well as the previous inputs and outputs.

For the memory cells in a softcore, the used symbols are bits and only block codes can be applied. Convolutional codes are difficult/impossible to use because the values in the register file and data memory are (mostly) independent of each other and instructions can also be considered independent from each other although there is some form of dependency between consecutive instructions.

In this section the (n, k, d_{min}) notation for ECC properties will be used. The needed variables are defined as:

- n: the length of the code word
- k: the length of the data word

- m: the length of the check word
- d_{min}: the minimum Hamming distance

The Hamming distance d_H and the minimum Hamming distance d_{min} are defined as [39]:

$$d_H(\mathbf{x_1}, \mathbf{x_2}) = |i : x_{1,i} \neq x_{2,i}, 0 \leq i \leq n| = \sum_{i=0}^{n-1} x_{1,i} \oplus x_{2,i} \tag{2.1}$$

$$d_{min} = \min_{\mathbf{x_1}, \mathbf{x_2} \in C} (d_H(\mathbf{x_1}, \mathbf{x_2}) | \mathbf{x_1} \neq \mathbf{x_2}) \tag{2.2}$$

where $\mathbf{x_1}$ and $\mathbf{x_2}$ are two binary vectors representing a code word. An ECC with a minimum Hamming distance of d_{min} can correct $\lfloor \frac{d_{min}-1}{2} \rfloor$ and detect $d_{min} - 1$ errors. If words are treated as vectors, the code vector can be calculated by multiplying the data vector with the generator matrix G, where the rows of G form the basis of the k-dimensional vector subspace of the binary linear (n, k, d_{min}) code. There also exists a parity check matrix H such that $GH^T = 0$, which can be used to check received code words. If the code word $\mathbf{v}$ is valid, Eq. (2.3) should hold.

$$\mathbf{v}H^T = \mathbf{0} \tag{2.3}$$

A received code word $\mathbf{r}$ can be defined as:

$$\mathbf{r} = \mathbf{v} + \mathbf{e} \tag{2.4}$$

where $\mathbf{e}$ is the error vector. The error syndrome $\mathbf{s}$ can be used as an indication of the location of the error in some ECCs and is a linear transformation of the error vector:

$$\mathbf{s} = \mathbf{r}H^T = \mathbf{e}H^T \tag{2.5}$$

2.7.2.1 Hamming codes

One of the most well known and also one of the first error correction codes are Hamming codes introduced in 1950 by Richard Hamming [40]. In his paper Hamming introduced three different error correction codes:

- **Single Error Detection (SED) codes:** A single parity bit is added at the end of the data word, which makes the number of 1's in the code word even. If the number of 1's is uneven, an error has occurred.
- **Single Error Correction (SEC) Codes:** An m-bit data word is mixed with a k-bit parity word into an n-bit code word, where m and n satisfy the inequality:

$$2^m \leq \frac{2^n}{n+1} \tag{2.6}$$

The parity bits are placed at the positions with an index equal to a power of two, and the rest of the positions are used for the symbols of the data word. Parity bit x contains the result of the parity check of all positions in the code which have a 1 at the position x of the binary index. At the decoder, the parity checks for every parity bit are repeated and checked against the value of the parity bit. If the result of the parity check and the parity bit do not match, a 1 is written. Else a 0 is written to the corresponding position of the checking number. A non-zero checking number indicates the position of the error.

- **Single Error Correction, Double Error Detection (SECDED) Codes:** An extension of the SEC code with an extra parity bit for an even parity check. The SECDED code check can indicate three different cases:
  - The parity checks are satisfied, so no error has occurred.
  - The checking number is not zero, and the last parity check fails, so a single error has occurred.
  - The checking number is not zero, and the last parity check is satisfied, so a double error has occurred.

All these codes are optimal linear block codes because the redundancy needed to correct a single error for a given block length is minimal. For a 32-bit data word the size of a SEC code word has to be 38 to satisfy Eq. (2.6), so a SECDED code word has to be 39 bits long. As described in [5] and [7] the parity bits of the input operators of arithmetic operations can also be used to predict the parity bits of the arithmetic result when using Hamming codes.

Besides single error correction and double error detection, Hamming codes can also detect a limited amount of double and triple errors. This is because a (39,32) Hamming code is a shortened ECC because not all syndromes are needed for SECDED capabilities. In [41] a technique for reordering the bits in the code word is proposed in order to increase the probability of detecting double and triple adjacent errors. This technique maps as much double and triple adjacent errors as possible to the remaining error syndromes. For a (39,32) Hamming code a double and triple adjacent error detection probability of 57% can be achieved. Applying this technique will help protect the memory elements against Multiple Bit Upsets (MBUs) which can be caused by a single inciding particle due to the small feature sizes used today [42] [43].

2.7.2.2 Hsiao codes

Although Hamming codes are optimal, the encoding and decoding procedures are not optimal. A more optimal implementation of Hamming codes exists: Hsiao codes. These codes use the same principles as Hamming codes, but the generator and parity check matrices are constructed in a different way. For the parity check matrix the following constraints should be met [44]:

- There are no all-0 columns.
- Every column is distinct.
- Every column contains an odd number of 1's (odd-weight)

These constraints ensure that the number of 1's in the matrices is minimal, and thus the number of logic levels in the encoder and decoder is reduced.

2.7.2.3 Golay codes

Another optimal linear block code is the Golay code [45], which can correct up to three errors. There are two versions of binary Golay codes: the perfect (23, 12, 7) code and the extended (24, 12, 8) code. A (23, 12, 7) Golay code can be converted to a (24, 12, 8) Golay code by appending an overall parity bit at the end of the code word. In [45] the generator matrix is given which can be used for encoding and for generating the parity check matrix. The encoder and decoder can be easily implemented as Lookup Tables (LUTs). The encoder contains a LUT with 4096 entries containing the code words for all possible data words. The decoder contains a LUT with all correctable error vectors which uses the error syndrome as it's index. The error syndrome is calculated by multiplying the read code word with the parity check matrix. The correct code word can be calculated by bit-wise XORing the read code word with the error vector.

Although the Golay code has a good error correction capability and can be implemented easily, the applicability for softcores is limited, because a data word size of 12 is not used very often. Word sizes of 8, 16, 32 or 64 are more common in processor architectures, but these sizes cannot be easily encoded with a Golay code. Therefore Golay code is not a suitable option for providing fault tolerance in processors.

2.7.2.4 Cyclic Redundancy Check codes

Cycles codes are a special class of linear codes, which can be efficiently implemented in hardware using shift registers and combinatorial logic elements, based on their polynomial representation. A linear block code is cyclic if and only if every cyclic shift of a code word is another code word [39]. The code words are represented by polynomials of the form in Eq. (2.7) in modulo 2 arithmetic.

$$\mathbf{v}(x) = \sum v_i * x^i \tag{2.7}$$

Cyclic Redundancy Check (CRC) codes are well known cyclic codes used for error detection in data storage. A data word is encoded into a code word by performing a polynomial division with a generator polynomial $g(x)$ and attaching the remainder of this division to the data word. Decoding involves the same polynomial division, but now with the remainder attached to the data word. If the remainder of this division is zero no detectable errors have occurred and otherwise, an error is detected. The number of check bits m should satisfy $n \leq 2^m - 1$, where m is also the order of $g(x)$. The choice of the generator polynomial $g(x)$ will define the capabilities of the CRC code. If error correction is also possible with the chosen $g(x)$, the remainder will indicate what error has occurred. The corresponding error function $e(x)$ can be found in a LUT where the index is the remainder of the division in the decoder [46].

2.7.2.5 BCH codes

In Section 2.7.2.1 Hamming codes where discussed, which are also cyclic codes with the generator polynomial $g(X) = 1 + X + X^3$. BCH codes are a more generalized form of Hamming codes which can correct any error pattern t as long as the order of $g(X)$ satisfies $t < 2^{m-1}$ where m is the length of the data word [47]. The generator polynomial $g(X)$ of a BCH code is defined as:

$$g(X) = LCM\left(\Phi_1(X), \Phi_3(X), ..., \Phi_{2t-1}(X)\right) \tag{2.8}$$

where LCM is the least common multiple and $\Phi_i(X)$ is the minimal polynomial of some elements α_j, which are primitive elements of the Galois Field $GF(2^m)$ [47]. BCH codes can be encoded by multiplying the code polynomial with the generator polynomial and decoded by solving the key equation:

$$\sigma(X)S(X) = -W(X) + \mu(X)X^{2t} \tag{2.9}$$

where:

- $\sigma(X)$: the error-location polynomial
- $S(X)$: the error syndrome polynomial
- $W(X)$: the error evaluation polynomial (for binary BCH codes the error values are always one)

The key equation can be solved by several computational intensive algorithms like the Berklekamp-Massey algorithm and the Euclidian algorithm which often result in multi-cycle decoding procedures. Therefore, these algorithms are not very suitable for implementation in a memory element or pipeline. In [48] a parallel BCH code design is presented. In this design, single cycle encoders and decoders using the generator and parity check matrices for BCH codes are studied. This approach will induce an area and latency penalty, but these penalties are still acceptable for processors compared to multi-cycle algorithms.

2.7.2.6 Residue codes

Apart from linear codes, there are also non-linear ECC like residue codes. In residue codes, the check word consists of the residue of a modulo operations which is performed on the data word [49]. In multi-residue codes. several moduli are used to generate a concatenated check word. The number of residues and the chosen moduli define the error detection and correction capability of the code. In [50] an error detection variant of residue codes is proposed with three as the used modulo. This code can detect all single errors and about two-thirds of all other types of double errors. In [51] a bi-residue code is proposed which can correct all single errors. If an error occurs in the code word, a syndrome $(|e|_A, |e|_B)$ is generated which can be used for finding and correcting the error. If an error occurs in only one of the residues, this will have no effect on the correct operation of the processor and can be corrected by regenerating the residue from the data word. The cost of the processor proposed in [51] is not greater than duplicating the original processor. The proposed codes in [50] and [51] do however use 1's complement

logic, which limits the applicability in today's processor which mostly use 2's complement logic.

In [7] multi-residue codes are introduced. These codes use multiple moduli and thus the check word consists of multiple residues. The most suitable moduli with the desired d_{min} can be found by performing an exhaustive search. In the provided example in the paper a (48,32) multi-residue code is proposed with moduli {5,7,17,31} which gives $d_{min} = 4$.

2.7.3 Comparison

Table 2.4: Comparison of the different techniques used for adding fault tolerance to memory elements.

Type		n	k	d_{min}	ED	EC
NMR		2k	32		1	0
		3k	32		1	1
		nk	32		$\lfloor \frac{n}{2} \rfloor$	$\lceil \frac{n}{2} \rceil - 1$
Hamming	SED	33	32	2	1	0
	SEC	38	32	3	1	1
	SECDED	39	32	4	2	1
Hsiao	SECDED	39	32	4	2	1
Golay		23	12	7	6	3
		24	12	8	7	3
CRC		$n \leq 2^m - 1$	32			
BCH	SEC	38	32	3	1	1
	SECDED	39	32	4	2	1
	DEC	44	32	5	2	2
	DECTED	45	32	6	3	2
	TEC	49	32	7	3	3
	TECQED	50	32	8	4	3
	QED	56	32	9	4	4
	QEDPED	57	32	10	5	4
	PED	59	32	11	5	5
	PEDSED	60	32	12	6	5
Residue codes		48	32	4	3	1

In Table 2.4 the different solutions found in literature are listed and compared with each other. Because the word size of 32-bits used in the RISC-V core, Golay codes will not further be evaluated. Based on the found solutions and their comparison five implementation options for providing redundancy in memory elements can be defined:

- **NMR:** N-times replication of the data in memory elements.
- **Hamming:** Addition of several parity bits to the data word for error checking up to SECDED

- **Hsiao:** Addition of several parity bits to the data word for error checking up to SECDED in a more efficient order as Hamming.
- **CRC:** Addition of the remainder of a polynomial division with a generator polynomial.
- **BCH:** Linear transformation of a data word to a code word with a generator matrix based on a generator polynomial.
- **Residue:** Addition of the remainders after division operations to the data word.

2.8 Fault injection

When fault tolerance is added to a softcore, it should be tested and verified that the core can detect and handle faults in the intended manner. Fault injection is one of the techniques that can be used to perform this verification. According to [52] fault injection is defined as:

> *Fault injection is the validation of the dependability of Fault Tolerant Systems which consists in the accomplishment of controlled experiments where the observation of the system's behavior in presence of faults is induced explicitly by the written introduction (injection) of faults in the systems.*

There are three main categories of fault injection techniques, which can be used for experiments with a softcore [52][53]:

- **Hardware-based Fault Injection (HWFI)** inducing faults by means of disturbing the systems with physical phenomena, such as heavy ion radiation or electromagnetic interference, or external system parameters through pin-level injection.
- **Software-based Fault Injection (SWFI)** emulating faults at the software level by modifying data or by changing the software's functionality.
- **Simulation-based Fault Injection (SFI)** inducing faults by applying logical values to signals or memory elements in the system through simulator commands.

HWFI is the most realistic fault injection category, as some of these techniques are accelerated variants of phenomena which could occur during normal operation. However, these techniques are often very expensive to use and can have poor controllability as it is very difficult to induce a specific fault at a specific location in the system with for instance a radiation source. SWFI is less expensive than HWFI because there is no need for additional hardware and modifying software can be done at low cost. There are however limitations to SWFI techniques as the software can only use software accessible resources for fault emulation. Resources, where faults can occur which are not accessible for software, can therefore not be emulated, which limits the coverage of these techniques. SFI techniques provide a lower-cost alternative for HWFI. Simulator commands, such as the `force` command in ModelSim, can be used to induce a faulty value on a particular

signal, but the simulated model can also be modified. Modification of the simulated model can, however, come at a considerable cost and simulation times are typically very large, which limits the usability of such tests.

The techniques investigated in this thesis are all SFI techniques as such techniques provide the best possibilities for verification of the fault tolerance of the core at acceptable costs. Automatic tools also exist for fault injection experiments [53], but most of these tools are proprietary or have a limited scoop which might not make them suitable for fault tolerance techniques other than their target. One example of such a tool is described in [54] which can be used to evaluate systems which use TMR for fault tolerance. This tool analyzes the design in the early design phase and identifies which resources are susceptible to SEUs. This tool is however not a replacement for fault injection or radiation testing as pointed out by the authors.

2.8.1 Saboteur

A saboteur is a module, which can be added to the simulated model, which can modify the value or timing characteristics of a signal or signal vector [55]. The saboteur will only introduce faults when activated and lets signals pass unaltered during normal operation of the system. This technique can be used in simulation, but with the right provisions, it could also be used on an FPGA in order to decrease the test execution times. Different types of saboteurs can be identified [52][55]:

- **Serial Simple Saboteur:** placed between the driver and receiver of a signal which will modify the value of the signal.
- **Serial Complex Saboteur:** placed between the drivers and receivers of two signals which will modify the value of the signal.
- **Parallel Saboteur:** added as an additional driver to a resolved signal. A modified resolution function could also be used for this purpose.

2.8.2 Mutant

Another type of module which can be used for fault injection through modification of the simulated model is a mutant. A mutant is modification or replacement of an original component, which will behave normal while being in-active and will exhibit different behavior when activated. A mutant can be created in different ways [55]:

- Adding one or more saboteurs to the components description
- Mutating a component's structural description by replacing or modifying subcomponents
- Changing the component's behavioral description by modifying operator resolution functions, exchanging variables or other more complex modification for more complex fault models

2.9 Conclusion

In this chapter, the open-source RISC-V ISA used by the softcore made by Technolution B.V. was introduced. SEEs can be introduced by both natural causes due to the interaction between particles and human-induced causes, which are targeted at disrupting system functionality or retrieval of secret information. The effects of SEEs can be mitigated by adding redundancy to a system. Different forms of redundancy can be divided into four categories: hardware, information, time, and software.

Based on the information found in literature, five implementation options for adding fault tolerance to the pipeline were identified: NMR, lockstep, ECC, the watchdog and time redundancy. For adding fault tolerance to memory elements, six implementation options were identified in literature: NMR, Hamming codes, Hsiao codes, CRC codes, BCH codes and Residue codes.

For verification of the added fault tolerance, fault injection can be used. By injecting faults, the behavior of the system in the presence of faults can be observed and verified. Fault injection can be hardware-based, software-based or simulation-based. A saboteur or a mutant can be used for simulation-based fault injection.

Design of a fault tolerant RISC-V softcore

3

In Chapter 2 possible solutions for mitigating errors due to SEEs in both the pipeline and memory elements were studied. From this study, five implementations options for the pipeline and six implementation options for the memory elements were defined, which will be evaluated in Section 3.3. This evaluation will be done based on the fault model and design criteria which will be defined in Section 3.2. Based on this evaluation two designs were made: the ECC-based design discussed in Section 3.4 and the hybrid design discussed in Section 3.5. For both designs some optional design features will be presented in Section 3.6. In Section 3.7 the two designs will be evaluated against the design criteria and a choice for implementation will be made. Finally, the design of the test architecture will be discussed in Section 3.8. But first, the baseline core used for this thesis will be introduced.

3.1 Baseline softcore

The starting point for this thesis is the baseline implementation of the RISC-V softcore developed by Technolution B.V., which implements the user-level RV32I instruction set. This version of the softcore does not include any privileged instruction set features like interrupts.

3.1.1 Pipeline

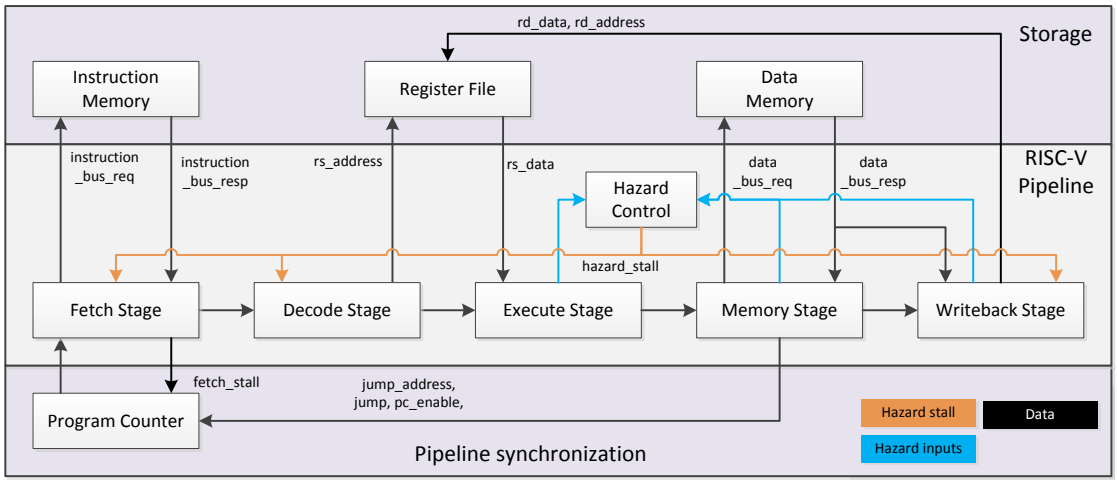

Figure 3.1: The block diagram of the RISC-V core and instruction and data memory.

In Fig. 3.1 the block diagram of the RISC-V pipeline is depicted. The softcore uses the classic five stage RISC pipeline[18]:

- **Fetch stage:** fetches the next instruction and stalls the pipeline, when the instruction memory is not able to provide the requested instruction in the clock cycle after the request.
- **Decode stage:** decodes the instruction received from the instruction memory and sends the register addresses to the register file. The register addresses are placed at a fixed position in all instruction formats, so these addresses do not have to be decoded, but can be routed to the register file immediately.
- **Execute stage:** calculates the result for all ALU operations based on the register values specified and/or the immediate value provided by the instruction. The execute stage also calculates the jump address in case of a jump or branch instruction.
- **Memory stage:** performs memory accesses in case of a store or load instruction. The absolute address to be accessed is calculated by the ALU in the execute stage. The memory stage will also stall the pipeline if the data memory does not complete the transactions within one clock cycle.
- **Writeback stage:** writes the value calculated by the ALU or the received data from the data memory to the register file.

The program counter is incremented by four, or set to the jump address if the execute stage signals that a jump should be performed. The address is incremented with four because the address is a byte address and not a word address. If a jump or branch instruction is performed, the program counter will be updated when this instruction is in the memory stage. The program counter could also be updated in the writeback stage, but the current implementation reduces the number of flushes required for a jump. The pipeline also includes a hazard detection unit for the detection of data and control hazards. If a hazard is detected the hazard detection unit will stall or flush one or more stages depending on the type and location of the hazard.

3.1.2 System on Chip

Fig. 3.2 shows the block diagram of the RISC-V System on Chip (SoC), a system containing the RISC-V core and other peripherals required for operation of the core on an FPGA. The SoC includes a separate instruction and data memory so the system can therefore be classified as a Harvard architecture. Using the Harvard architecture gives the ability to fetch instructions in parallel with accessing data in the data memory. The SoC also includes a Universal Asynchronous Receiver/Transmitter (UART) module for basic communication with systems outside the SoC and control registers which can be used to control and monitor the core.

All SoC peripherals can be accessed through Ethernet and the `tl_reg_bus`, a proprietary bus developed by Technolution B.V. The softcore can access the instruction and data memory through the `tl_data_bus`, a modified version of the `tl_reg_bus`, which

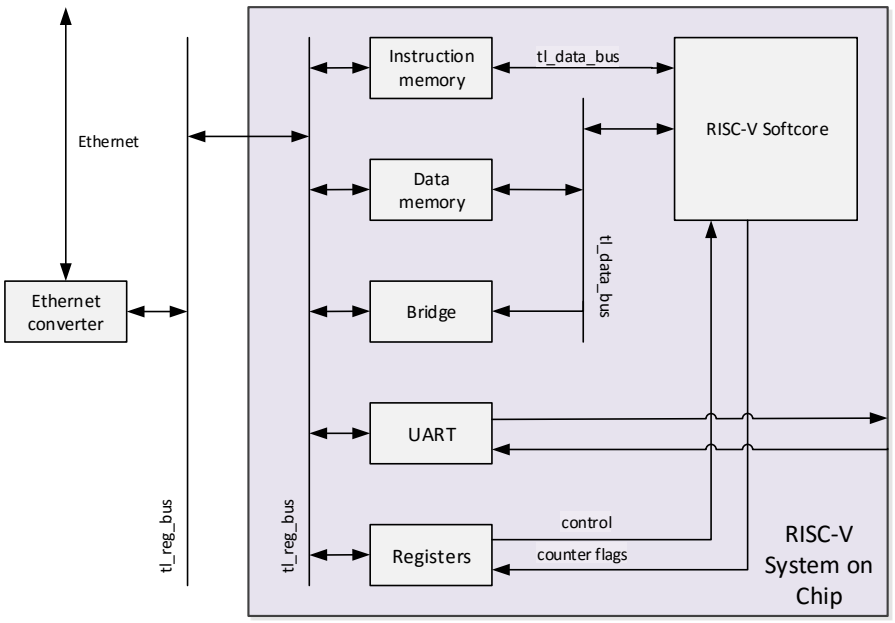

Figure 3.2: The block diagram of the RISC-V System on Chip.

allows the next transaction to be started before the acknowledgment of the previous transaction is received. The `tl_reg_bus` does not allow such overlapping transactions and is therefore not suitable to serve as a bus between a memory and the core as this will always induce a minimal stall of one clock cycle. If the memory sends the acknowledgment of the transactions in the clock cycle after the request was received, the softcore will not stall. The waveforms in Fig. 3.3 show the timing behavior of both the `tl_data_bus` and `tl_reg_bus`.

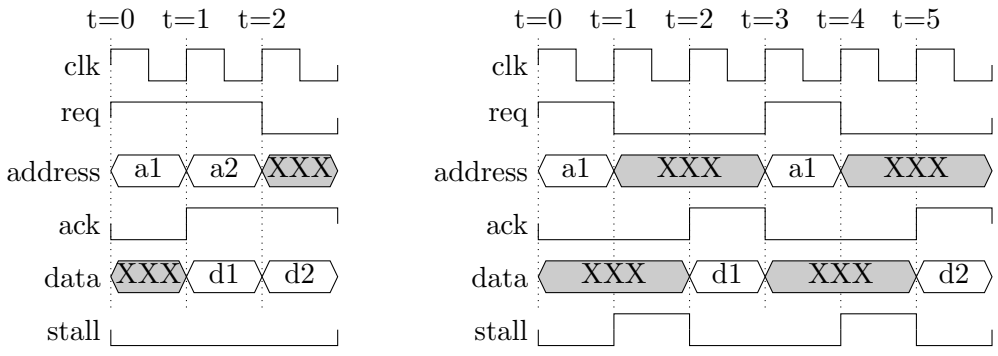

Figure 3.3: Timing behavior for transactions on the `tl_data_bus` (left) and `tl_reg_bus` (right).

3.1.3 Design cost estimate

For both designs, a design cost estimate will be made, based on the synthesis result for the baseline softcore as reported by Xilinx® Synthesis Technology (XST) for a Xilinx Spartan-6 FPGA. These results were retrieved by running XST for the complete softcore and for individual components with the project settings as reported in Table A.2. Along with these results, the synthesis results of some preliminary block implementations will be used in the design cost estimates. In Table 3.1 the synthesis result for the baseline softcore are presented (for the more detailed synthesis results, see Table A.3). These results only provide an indication of the actual resource usage, path delays of different components and the maximum attainable clock frequency for the softcore. The synthesis process might apply additional optimizations due to the interaction between components in the final design, which are not applied during synthesis of individual components. The mapping process might also apply further optimizations which will change the resource usage, corresponding path delays, and maximum clock frequency.

Table 3.1: The synthesis results for the baseline softcore for the Xilinx Spartan-6 FPGA.

Resource	Value
LUT	1732
Registers	835
f_{max}	87.7 (MHz)

3.1.4 Benchmark

Table 3.2: The Dhrystone benchmark result for 2,000,000 runs for the baseline softcore.

	Baseline
f (MHz)	83.3
CPI	2.24
Dhrystones/s	82590
Dhrystones/(s*MHz)	991.48
DMIPS/MHz	0.56

In Table 3.2 the Dhrystone benchmark for the baseline softcore is presented. Dhrystone is a small synthetic benchmark mimicking computational loads with mathematical and other operations using only integer operations [56]. The execution flow of the benchmark is checked by performing 20 tests after all runs are executed, which also makes it suitable for testing during fault injection as a correct execution flow can be verified. It is a small benchmark which can be easily ported to other platforms and architectures because it fits in L1 cache amongst other things. Although the small size is considered as a weakness, in the case of the RISC-V softcore it is an advantage as the memories in the SoC are effectively L1 caches. Another major weakness of the Dhrystone benchmark

is that compilers can heavily optimize its execution and thus improve the benchmark results. Due to the ease of implementation of the Dhrystone benchmark and the large number of checks, it was decided to use the Dhrystone as the basic benchmark for the RISC-V softcore.

`-O2`, `-ffreestanding`, and `-nostdlib` were used as specific compiler flags for compiling the Dhrystone benchmark. For some basic functions like `printf` and `memcpy` a dedicated library `stdlib.c` was used, which is targeted at the RISC-V SoC. The Dhrystone benchmark will perform 2,000,000 runs, as the execution time should be at least 20 seconds for the result to be reproducible.

3.2 Fault Model & Design criteria

Building a framework which can mitigate all types of SEEs, would require an immense effort and not be cost efficient. Therefore a fault model based on the problem statement in Section 1.1 is defined which should be targeted by the fault tolerant design. This fault model defines what type of faults can occur in specified locations and at which moments in time. In Fig. 3.4 the location and types of faults which will be targeted by the design are defined. These faults can occur at any moment in time when the core is active i.e. when the both the soft and hard reset signals are low and the enable signal is high.

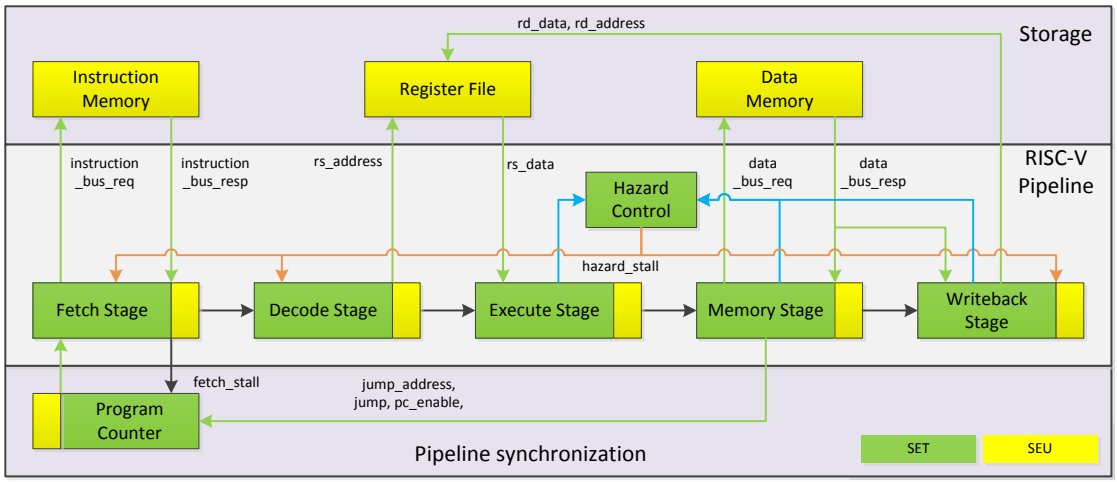

Figure 3.4: The block diagram including the fault model of the RISC-V core and instruction and data memory.

Besides the fault model, several design criteria are defined which will be used to evaluate the implementation options found in Chapter 2 and the designs made in this chapter. The design criteria along with their targets are listed in Table 3.3.

3.3 Design Options

In Tables 3.4 and 3.5 the score matrices for the implementation options for both the pipeline and the memory are given. The implementation options are evaluated against

Table 3.3: List of design criteria along with their intended targets.

Design criterion		Description	Target
Error Detection	ED	Number of concurrent detectable errors in a 32-bit word	2
Error Correction	EC	Number of concurrent correctable errors in a 32-bit word	1
Clock frequency	CLK	Clock frequency reduction	50 %
Resource Utilization	RU	Increase in the utilization of resources	4.5x
Cycles Per Instruction	CPI	Cycles Per Instruction increase	20%
Software transparency	ST	Number of modifications required to run software applications on the fault tolerant core	Minimal
Complexity	COM	Increased complexity of the fault tolerant core compared to the baseline	= baseline
Single Points of Failure	SPOF	Points in the pipeline or in address or data words where a fault will cause a failure	0
RISC-V compliance	RISC-V	Compliance with the RISC-V ISA specification	RV32I

each other and the baseline design of the softcore.

Table 3.4: Score matrix for the implementation options for the pipeline

Solution	ED	EC	CLK	RU	CPI	ST	COM	SPOF	RISC-V
NMR	++	+	+	−−	+	+	+	+	+
Lockstep	+	−	+	−	+	+	−	+	+
ECC	++	+	−	−	+	+	−	+	+
Watchdog	o	−	+	o	−	−	−	−−	+
TR	o	o	+	+	−−	−	+	−	+

Table 3.4 shows that NMR is the best option (for the pipeline) based on the design criteria. However, it does have one main drawback: the resource utilization will rise very significantly compared to the baseline design because the baseline is replicated N times. ECC provides a reasonable alternative to NMR as it has the same error detection and correction capabilities and uses fewer resources. This does, however, come at the price of reduced clock frequency, as encoders and decoders will be added to the critical path. The other implementation options in this comparison are less favorable as these have more drawbacks than positives.

Table 3.5 shows that NMR again is a suitable option for memory elements, but this time it has a few more drawbacks other than the resource utilization. For mitigation of

Table 3.5: Score matrix for the implementation options for the memory elements

Solution	ED	EC	CLK	RU	CPI	ST	COM	SPOF	RISC-V
NMR	++	+	+	−−	−	+	−−	−	+
Hamming	+	+	−	++	+	+	++	+	+
Hsiao	+	+	−	++	+	+	++	+	+
CRC	++	−	−−	+	+	+	+	+	+
BCH	++	++	−−	+	+	+	−	+	+
Residue	+	+	−	o	+	+	−	+	+

errors in the replicated memories, an extensive memory synchronization mechanisms is required, which will cause the Cycles Per Instruction (CPI) to decrease and will introduce a Single Point of Failure (SPOF) as it will be difficult to protect this mechanism. The ECC options do not require memory synchronization, as they use only a single memory for storing code words. This reduces the complexity of the memory architecture and saves a significant amount of area.

The score matrix in Table 3.5 also shows that Hamming and Hsiao codes are (almost) the same, the only difference being the lack of check prediction for Hsiao codes. It should, however, be noted that Hsiao code comes with slightly more efficient encoders and decoders, as this is a more optimized version of Hamming codes. BCH codes provide more extensive error detection and correction capabilities than Hamming and Hsiao codes, but this comes at the cost of a higher resource utilization and most importantly a larger reduction in clock frequency due to larger encoders and decoders. When more error detection and correction capabilities are requested, BCH codes can be used, but with the current design criteria, Hamming or Hsiao codes will suffice.

Based on these design options, two design were made: the ECC-based design and the hybrid design. Both designs use NMR and ECC to add redundancy to the core, but the major difference between these designs is how and where they apply these techniques.

3.4 ECC-based design

The ECC-based design uses Error Correction Codes as much as possible, as this option is the most viable for implementation in memory elements and is also a reasonable option for implementation in the pipeline. By using ECC as much as possible, there is a good possibility of eliminating the drawbacks of using ECC in the pipeline. Furthermore, it is expected that using ECC in the pipeline will result in a lower resource usage than NMR. NMR would result in a resource usage increase of more than 400% for SECDED capabilities, as the complete pipeline has to be replicated four times and majority voters have to be added. If ECC cannot be used, the design will use NMR to provide the required redundancy. This means that the border of the protection domains will be at a low level in the processor. A protection domain is an area of the core, which uses one particular technology for providing redundancy.

3.4.1 Design features

Hamming codes will be used as ECC, as these codes can provide the required SECDED functionality and also provide the option of using check prediction for the ALU. Hsiao codes do provide more optimal encoders and decoders, but the possibility of using check prediction was not mentioned in the studied literature. The Hamming codes will be used to protect all memory elements (instruction & data memory, register file and FF banks) and buses. This will reduce the added cost of the extended memories and eliminates the need for memory synchronization while still providing the required redundancy. In the implementation, Hamming SECDED codes will be used, but they can be easily replaced by other variants of Hamming codes or other ECC codes.

Faults on the address field of buses/connections between the pipeline and memory elements will be detected and corrected by protecting the addresses with ECC. After decoding of the address in a memory element, no checks will be available to check if the correct word was read from or written to the register file or the memories. The implications of providing protection for such operations are complicated and not taken into account in this design. Possible solutions for this problem will be presented as future work.

Whenever an arithmetic operation has to be performed on one or more code words, check prediction should be used for predicting the check bits of the results of the operation. The check prediction should only use the two check words and the type of operation to be performed as its inputs and thus calculates the new check word with an operation independent of the operation on the data words. The predicted check word and data word will be checked by a decoder before storing to avoid fault propagation. Check prediction is available for the following instructions:

- basic arithmetic: add, addi, auipc, sub
- bitwise operations: and, andi, or, ori, xor, xori
- Unconditional jumps: j, jal, jalr
- Load/store: lb, lbu, lh, lhu, lw, sb, sh, sw

Operations for which no check prediction is available should be executed N times in parallel and the final result should be determined through a majority voting procedure. In principal, N should be equal to the Hamming distance (for SECDED: four) in order to provide the required redundancy. If the extra resource utilization is too large for N equal to the Hamming distance, this may be reduced, but this decision should be made on a per-application basis and should therefore be separately configurable.

All non-arithmetic operations/components, which cannot be protected with ECC or Check Prediction (CP) will be protected with NMR. Majority voters will determine the correct outcome based on the replicated results and signal if an error was detected and if possible corrected. Again, N should be equal to the Hamming distance, but if resource utilization becomes too large, this can be adjusted on a per-application basis. If multiple arithmetic components are cascaded without FF banks in between, no majority voter is required between these components, as this will introduce SPOFs and increases the

critical path latency unnecessarily. Single bit control signals should also be replicated N-times whenever possible to reduce SPOFs.

3.4.2 Design diagrams

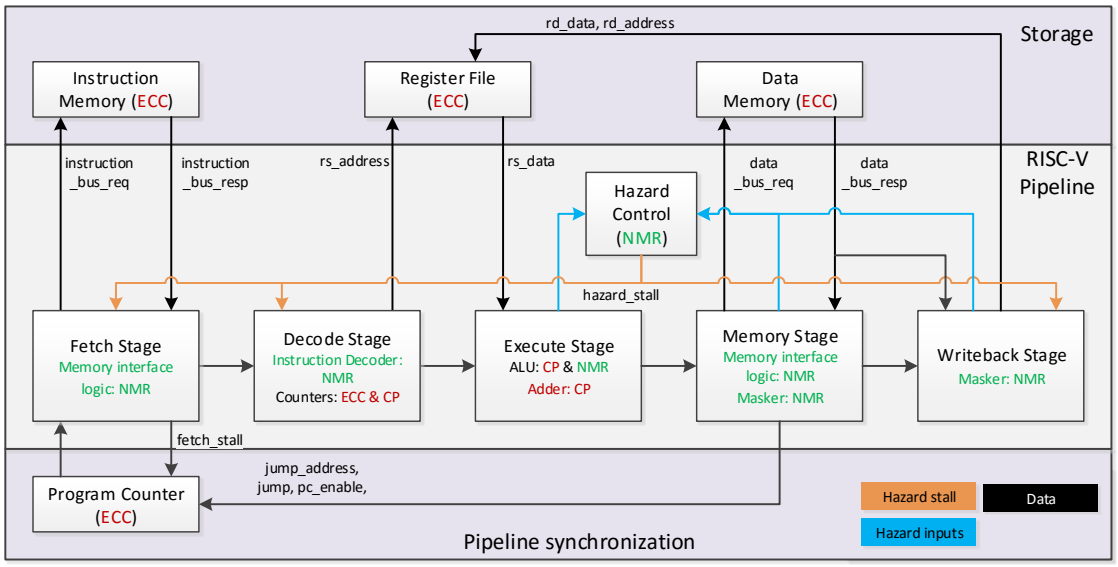

Figure 3.5: The block diagram of the fault tolerant ECC-based RISC-V core.

In Fig. 3.5 the block diagram of the ECC-based design is shown. The complete block diagrams of the ECC-based design can be found in Appendix B. On the top-level there are not many changes to the design, but for each stage/component the protection techniques are listed in this overview.

Execute stage example

As an example of the internal changes in this design, Figures 3.7 and 3.6 show the execute stages of the baseline and ECC-based design. The baseline execute stage is a relatively straightforward stage, with an ALU, an adder for the jump address, two multiplexers for selection of the ALU inputs and a FF bank. The execute stage of the ECC-based design is much more complex than the baseline execute stage, as it involves several encoders, decoders, a voter and check prediction circuitry. The ALU inputs supplied by the register file, instruction decoder and the stored program counter value, have to be decoded for avoiding fault propagation. The register values might have been stored for a longer time frame and have a higher risk of containing errors than a value stored in a FF bank. These values could be checked after the operation, but this will make it more difficult to correct the error, as it has been processed by an arithmetic unit. The check predictor will predict the check word for selected ALU operations, which are then checked by a checker, which is effectively a decoder. For ALU operations which do not have check prediction, the ALU replicator, which contains N-1 reduced ALUs, will calculate the

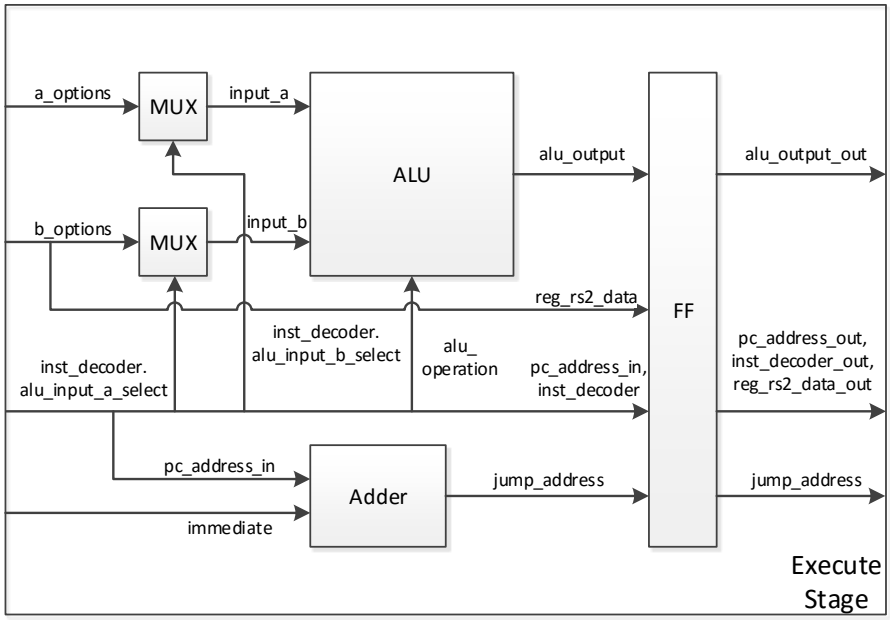

Figure 3.6: The detailed block diagram of the execute stage of the baseline RISC-V core.

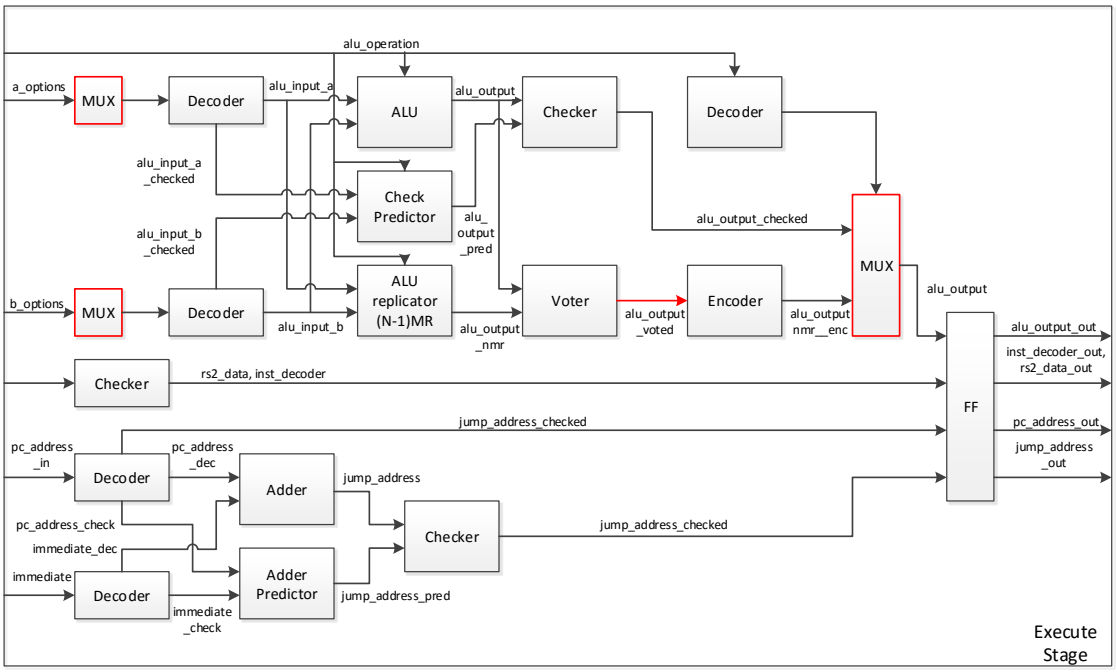

Figure 3.7: The detailed block diagram of the execute stage of the fault tolerant ECC-based RISC-V core. The components marked in red are SPOFs.

expected value N-1 times. The majority voter will vote the result of NMR protected operations, which is encoded by the encoder before latching. The jump address adder is accompanied by a address check predictor, as check prediction is available for addition. The combination of the calculated jump address and predicted check word for the new jump address is again checked by a checker before latching.

Fig. 3.7 also shows another problem of this design besides the increased complexity: the new Execute stage contains four SPOFs (marked in red in the diagram). These SPOFs are: the multiplexers for `alu_input_a` and `alu_input_b`, the multiplexer for `alu_output` and the signal `alu_output_voted` which is not replicated or protected with ECC.

3.4.3 Design cost estimate

Based on the design features and the block diagrams made, a design cost estimate was made for evaluation of the design. The design cost estimate for the ECC-based design is shown in Table 3.6 (for the more detailed estimate, see Table B.1). The design cost estimate shows that the increase in resource usage for this design is almost the same as for a simple NMR design (four times the baseline core). The goal of this design was to reduce the resource increase by using ECC instead of NMR in the pipeline. However, these results show that this effect is very minimal and comes with a major drawback: a tripled critical path latency.

Table 3.6: The design cost estimate for the ECC-based design using SECDED Hamming code and N = 4. The relative value is calculated versus the baseline softcore.

Resource	Absolute	Relative
LUT	7659	442%
Registers	1812	217%
f_{max}	29.3 (MHz)	33%

3.4.4 Design summary

Based on the design description and the cost estimate, the ECC-based design can be summarized as:

- A large number of encoders and decoders is required due to the large number of switches between protection domains.
- Every switch between a protection domain introduces a SPOF. At least 24 SPOFs in the pipeline and register file were identified in the block diagrams.
- The complexity of the design is high due to protection domain switches on a low level, i.e. inside the pipeline stage.
- The pipeline stages of the baseline softcore cannot be reused, which will reduce maintainability of the fault tolerant variant of the softcore.

- The resource utilization of the design is not much lower than for a simple NMR design. This is caused by the large number of encoders and especially decoders required for the design.
- The estimated clock frequency is three times smaller than the clock frequency of the baseline. This is caused by the large path delay of the decoders.

3.5 Hybrid design

The aim of the hybrid design is to minimize the number of changes between the protection domains and thus minimize the number of Single Point of Failure. This can be achieved by placing the complete pipeline in the NMR domain and not dividing the pipeline stages internally between both domains. The register file and memories will be placed in the ECC domain, as this is much more cost efficient and simple than using NMR memories. By creating an overlap between the protection domains, the number of SPOFs should also be significantly reduced. This new design might have a larger resource usage, but if the goals for this design are reached this can be acceptable.

3.5.1 Design features

The hybrid design will contain two protection domains: one protected with ECC and the other domain with NMR. The instruction memory, data memory, the register file and the program counter will be placed in the ECC domain, and the pipeline stages, and the hazard control are placed in the NMR domain. The ECC domain will again use a SECDED Hamming code for SECDED error capabilities. The instruction memory, register file, and program counter will be encoded per 32-bit word. The data memory will be encoded per byte because the RISC-V requires half word and byte word data accesses. In the implementation, Hamming SECDED codes will be used, but they can be easily replaced by other variants of Hamming codes or other ECC codes.

There will be five crossings from the NMR domain to the ECC domain and four crossings from the ECC to the NMR domain as listed in Table 3.7. On each crossing from the NMR domain to the ECC domain (except for the stats record), the data will first be encoded into the ECC domain, before the data is voted out of the NMR domain. By placing the encoder before the majority voter and thus replicating it N times, the chance of an error occurring in the encoder will not remain undetected. On each crossing from the ECC domain to the NMR domain, all pipelines will be provided with the same encoded data, which each pipeline then decodes. By providing each pipeline with a decoder, the chance of a fault in the decoder remaining undetected is reduced. Such faults might still not be noticed if it results in an undetectable error or is hidden by the operations applied to it. A silent fault is a fault which does not result in an error when used as an input for a logic operation. This concept of overlapping domains, illustrated in Fig. 3.8, will eliminate possible SPOFs.

By placing the memories and the register file in the ECC domain the need for memory synchronization between duplicated memories or register files is eliminated. Such synchronization would be needed when a fault would occur in the register file or one

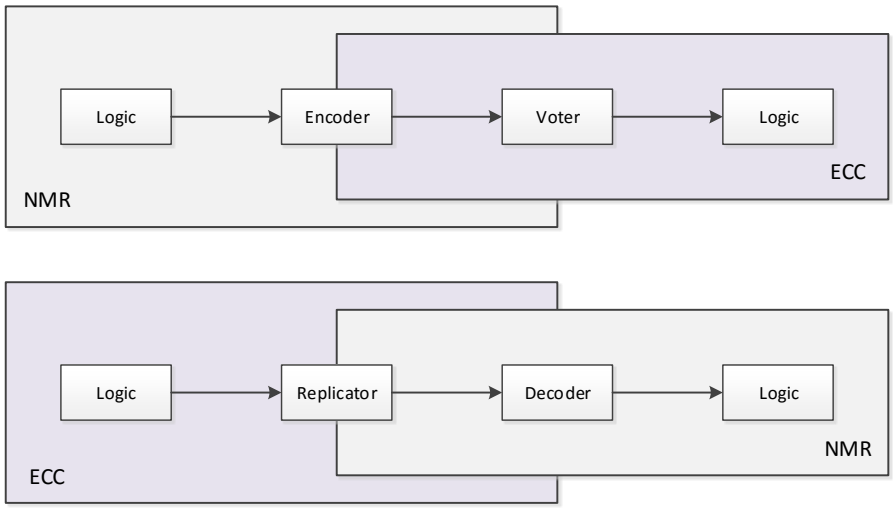

Figure 3.8: Illustration of the overlapping protection domains used for protecting domain crossings in the hybrid design.

Table 3.7: The crossing between the NMR and ECC domain in the hybrid design.

NMR to ECC	ECC to NMR
Program counter in	Program counter out
Instruction bus request	Instruction bus response
Registers in	Registers out
Data bus request	Data bus response
Stats	

of the memories. This choice also allows for a fine-grained comparison of results being committed by the pipelines and thus early error detection. Furthermore, the resource usage is decreased by not replicating these elements, but only extending them with a few extra bits.

Faults can remain undetected and also accumulate in the register file for a long period if the register is not used. This can be avoided by periodically launching a trap, which will induce the scrubbing of the register file. This scrubbing is automatically induced when a trap is launched, as the register file needs to be saved to the stack for usage after the trap finishes and thus each register value will pass through two decoders. The baseline softcore does not support traps yet, but this could be a useful feature once traps are implemented.

Faults on the address field of buses/connections between the pipeline and memory elements will be detected and corrected by protecting the addresses with ECC. After decoding of the address in a memory element, no checks will be available to check if the correct word was read from or written to the register file or the memories. The implications of providing protection for such operations are complicated and not taken into account in this design. Possible solutions for this problem will be presented as future

work.

Scrubbing of words in the data memory could also be done by loading these words into the register file and immediately afterwards writing them back to the data memory. If large sections of the data memory need to be scrubbed this can become very costly and a more dedicated scrubbing unit, which works in parallel to the core, might be required. It should be noted that scrubbing of memory elements is only required when the fault rate of a particular application is very high as it is rather costly and only gives improved performance if enough faults can accumulate in a relatively short amount of time.

3.5.2 Program Counter

By placing the program counter in the ECC domain and not in the NMR domain, the replicated pipelines can be kept synchronized at all times and a single pipeline cannot get completely out of sync because its program counter is incorrect. The program counter block also contains logic for calculating the new program counter value. For adding fault tolerance to this design, two solutions are available: using Check Prediction (CP) or N-Modular Redundancy (NMR).

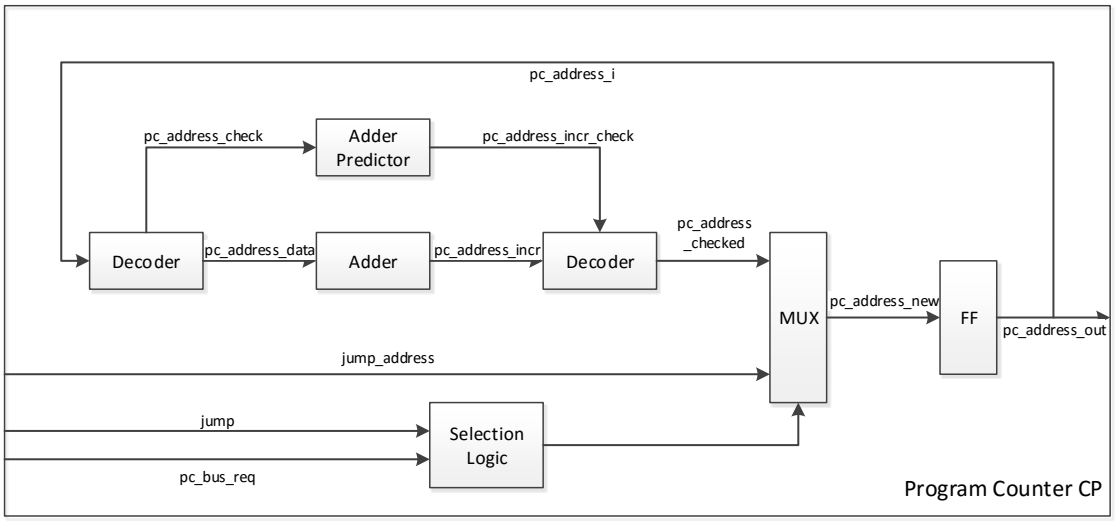

Figure 3.9: Block diagram of the program counter using check prediction.

$$P_R = P_A \oplus P_B \oplus P_{CY} \tag{3.1}$$

In Fig. 3.9 the block diagram for the program counter using CP is depicted. The check word for addition can be calculated with Eq. (3.1), where P_R is the check word for the result, P_A the check word for input A, P_B the check word for input B and P_{CY} the check word generated from the carries of the addition operation [5]. P_{CY} could be generated by the adder used for the addition operation, but the check prediction has to be independent of the operation generating the data word, so a separate carry chain should be used for generating these parity bits.

Fig. 3.10 depicts the block diagram for the NMR program counter. This program counter replicates the increment calculation and the selection of the new program counter

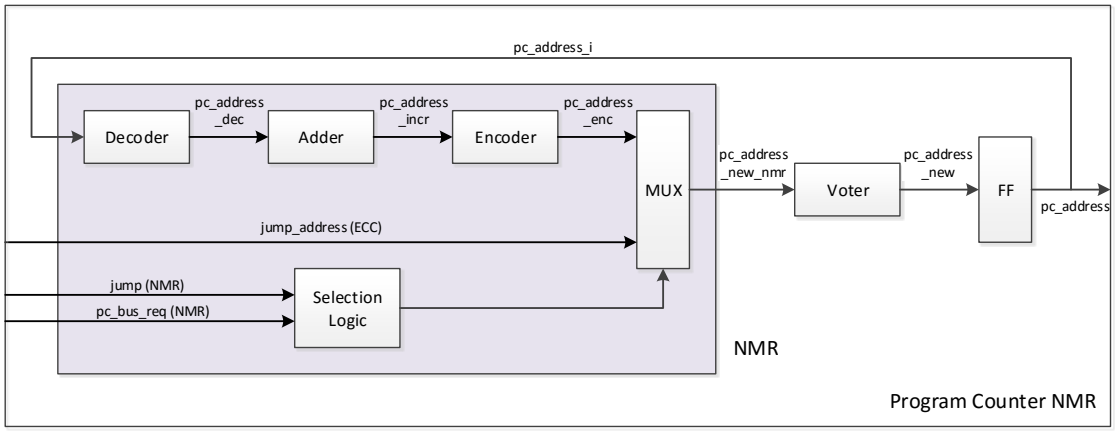

Figure 3.10: Block diagram for the NMR Program Counter.

value N times. The new program counter value is then encoded into the ECC domain and voted before being latched. The FF storing the program counter value is not replicated, as this would give synchronization problems if one of the FFs would latch a different value. The inputs to the new program counter calculation are not voted by the core when leaving the pipeline, as it would not make sense to vote a signal first and then replicate it. It might take longer for an error to be detected, but it is not expected that this will cause problems, as the majority voter is more resilient against errors than a decoder due to per bit voting.

Based on the block diagrams presented it is expected that the CP program counter uses fewer resources than the NMR program counter. The CP program counter, however, contains a SPOF at the multiplexer switching between the old program counter, the incremented program counter, and the jump address. This SPOF could be removed by placing the multiplexer in the NMR domain, but this would reduce the resource utilization advantage of the CP program counter over the NMR program counter. Furthermore, without the SPOF fix, the CP program counter would require a majority voter on its input, which would diminish the resource utilization advantage of this version even more. Because of the SPOF in the CP program counter and the resource utilization advantage which is not so clear, it was decided to select the NMR program counter for the hybrid design. The NMR version is expected to be more resilient to errors induced by faults in the pipeline.

3.5.3 Design diagrams

In Fig. 3.11 the block diagram of the hybrid design is shown, which illustrates the design features described. In the ECC-based design, the pipeline stages were modified as the redundancy was added at a low level. In the hybrid design the pipeline stages only required minimal modifications like some default values for additional bus signals, but in general, the pipeline stages remained the same. So for the execute stage of the hybrid design, the design from the baseline softcore as shown in Fig. 3.6 could be reused.

Besides the advantage of reusing already existing designs, this also prevents an in-

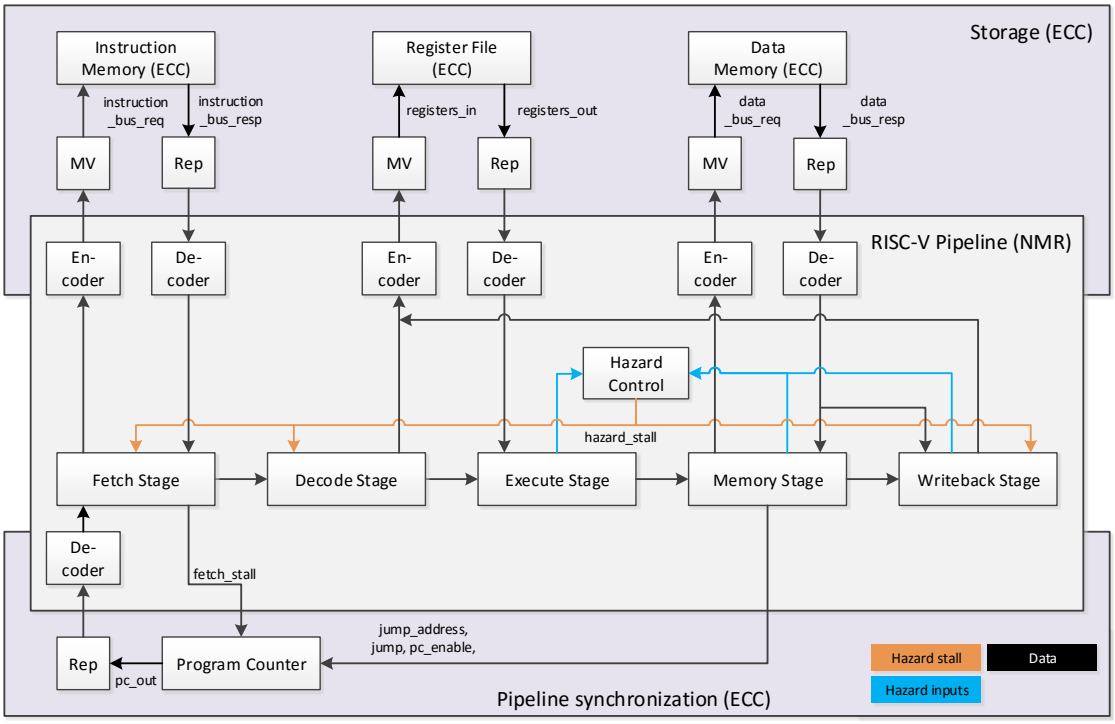

Figure 3.11: The block diagram of the fault tolerant hybrid RISC-V core.

crease of the complexity of the design. The critical paths in the hybrid design will be larger than for the baseline design, as most stages have encoders and majority voters on some of the outputs and decoders on some of the inputs. These critical paths can, however, be split by placing FF banks between the stage and the checking circuitry. Using extra FF banks will mean more stages being used in the pipeline and thus more stalls being induced for hazards and more flushes being induced on jumps, but this could be a good solution for solving the reduced clock frequency problem.

3.5.4 Design cost estimate

Based on the design features and the block diagrams made a design cost estimate was made for evaluation of the hybrid design. The design cost estimate for the hybrid design is shown in Table 3.8 (for the more detailed estimate, see Table C.1). The design cost estimate shows that the resource utilization is more than five times larger than for the baseline softcore. This large increase is caused by the pipeline being replicated four times and by the addition of encoders, decoders, and majority voters. The estimated maximum clock frequency is also much smaller than the baseline, but at 44% it is not too far off from the target for this design criterion. As mentioned before, the maximum clock frequency could also be increased by adding additional stages to the pipeline for decoding, but this will come at the cost of a higher CPI.

Table 3.8: The design cost estimate for the hybrid design using SECDED Hamming code and N = 4. The relative value is calculated versus the baseline softcore.

Resource	Absolute	Relative
LUT	9391	542%
Registers	3826	458%
f_{max}	38.6(MHz)	44%

3.5.5 Design summary

Based on the design description and the cost estimate, the hybrid design can be summarized as:

- The number of domain crossings is limited by only switching between protection domains while transferring data to and from the pipeline.
- The number of required encoders, decoders and majority voters is limited by only having nine protection domain crossings.
- The complexity of the design is relatively low. The original stages can be reused and only a few basic building blocks have to be added to the structural description of the core.
- Resource utilization of the design is higher than four times the baseline design plus majority voters, due to the encoders and decoders. However, the overall increase is still acceptable as this eliminates a lot of SPOFs.
- The estimated clock frequency is only slightly lower than the target clock frequency.

3.6 Optional design features

Besides the main features of the ECC-based and hybrid design, there are also some optional features, which can be implemented, but are not instrumental to the designs:

- For identification of fault attacks a watchdog can be added to the design. The watchdog monitors all checking circuitry which can indicate the occurrence of a fault. When these checkers indicate too many errors within a short time frame, the watchdog can stop the core and thus make it impossible for the attacker to force certain behavior of the core. The designer of the core can define the checking conditions and other parameters used by the watchdog. These conditions include the maximum number of concurrent errors, the maximum number of consecutive errors allowed and the minimum interval between errors. If these checking conditions are violated, the watchdog stops the processor. The watchdog is protected with NMR, so attackers cannot disable the core by purposely attacking the watchdog.

- For partial Double Adjacent Error Detection (DAED) and Triple Adjacent Error Detection (TAED) capabilities the implementation of the used Hamming codes can be optimized by adjusting the bit placement of the code. [41]
- For more extended error detection and correction capabilities, the used Hamming code can be replaced by BCH codes.
- For the mitigation of SEFIs configuration memory scrubbing can be implemented. This does, however, require a per-device-type implementation as the implementation of configuration memory differs between devices and some manufacturers already provide solutions for this problem with some devices.
- If the watchdog has stopped the processor because of a violation of the checking conditions and configuration memory scrubbing is implemented, the processor will be restarted after the wait interval has expired. This will enable the scrubber to solve any configuration errors caused by the attack.

3.7 Design selection

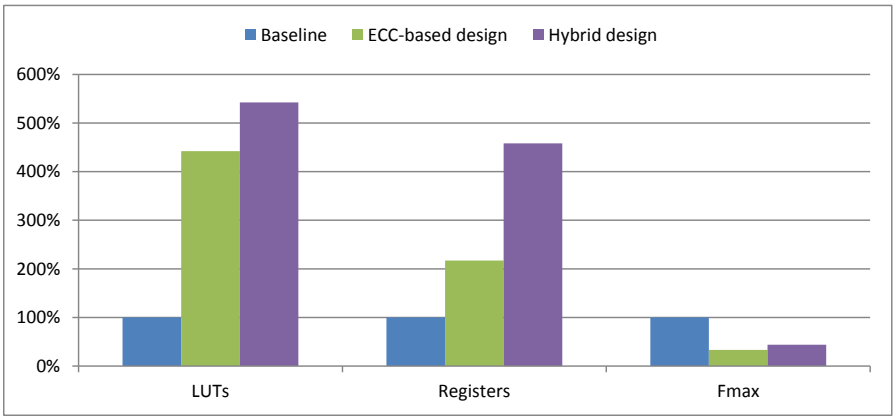

Figure 3.12: Design cost estimate comparison for SECDED ECC and NMR (Hamming distance = 4).

In Section 3.4 and Section 3.5 two designs for a fault tolerant RISC-V softcore were presented. For both designs, design cost estimates based on synthesis results from the baseline core and some preliminary block implementations were made. In Fig. 3.12 these design cost estimates for SECDED error capabilities are compared to the design cost of the baseline design.

For this thesis only one design should be implemented, so a choice has to be made between the two presented designs. Table 3.9 depicts the comparison of the ECC-based and hybrid design for each of the design criteria. The CPI of the hybrid design could remain the same as for the ECC-based design, but if extra pipeline stages are added to the design, the CPI might be lowered.

Table 3.9: Design criteria evaluation for both designs

Criterion	ECC-based	Hybrid
ED	+	+
EC	+	+
CLK	−−	−
RU	−	−−
CPI	+	+/−
ST	+	+
COM	−	+
SPOF	−	+
RISC-V	+	+

The ECC-based design performs worse than the hybrid design for complexity and SPOFs, because of the low level domain switches and the non-overlapping protection domains. This SPOF problem could be solved by letting the protection domains overlap by moving the encoders into the NMR domain, so the majority voters will vote data, which is already encoded. This solution will increase the resource utilization even more, so this will reduce the slight advantage of the ECC-based design for this design criterion.

Based on this design comparison, it was decided to implement the hybrid design for this thesis, as it performs better for most design criteria. The only criteria for which it performs worse than the ECC-based design is resource utilization, but this is partly caused by solutions for problems involving other design criteria like the overlapping protection domains. Furthermore, this design also gives more possibilities for future improvement and has better maintainability, which are also important factors for the usability of the fault tolerant softcore for Technolution B.V.

3.8 Verification design

The goal of the fault tolerant design is to prevent faults from influencing the correct execution flow of the design without being detected by adding redundancy to the design. The hybrid design should contain enough redundancy to mitigate most errors, but this has to be verified by testing the design. This verification can be performed by injecting faults into the design and check if the processor is still able to successfully execute its assigned task and if it detected the error at the expected location.

3.8.1 Test design criteria

In Section 2.8 some techniques for fault injection were investigated, which can be used for the fault injection architecture for this design. Several design criteria can be defined for evaluating these techniques:

- **Design modifications:** some techniques require modifications of the design, whereas other techniques do not. Modifying the design might introduce new bugs

or hide bugs in the design, so fault injection techniques requiring no modifications are preferred.

- **Modification level:** fault injection techniques can require modifications at different levels of the design. In this evaluation a difference will be made between low level (inside basic building blocks like pipeline stages) and high level (in structural descriptions of components).
- **Simulation time:** if fault injection is used in simulation this can result in larger simulation times. As fault injection testing requires a comprehensive test suite, long simulation times are not preferred.
- **On board:** some techniques are not suited for usage on a real FPGA, as these can only be applied in for instance a simulator. On board testing can be executed much faster, than testing in simulation, so for large test suites this can save large amounts of time. Furthermore, simulation based testing might not catch all problems in the design, as it is an approximation of real life. On board testing will provide better verification of the design, as this gives a much better approximation of real life.
- **Resource coverage:** a comprehensive validation of the added redundancy requires all resource being tested during fault injection testing. Therefore techniques covering more resources are preferred.

3.8.2 Test design options

In Table 3.10 the applicable fault injection techniques are evaluated for each of the design criteria for fault injection. Hardware based fault injection is not considered, as the required resources for this technique are not available for this project.

Table 3.10: Evaluation of the fault injection techniques for each of the design criteria for fault injection.

Criterion	Software based	Simulation based	Saboteur	Mutant
Design modifications	No	No	Yes	Yes
Modification level	n/a	n/a	High & Low	Low
Simulation time	Medium	Low	High	Low
On board	Yes	No	Yes	Yes
Resource coverage	Low	High	High	High

Saboteurs provide a versatile solutions for fault injection, as it can be used in all places in the design and depending on the location of the saboteur does not require major modifications to the design. Simulation times for saboteur based fault injection testing will, however, increase, as the simulator needs to simulate more components. Testing times can be reduced by only performing basic fault injection testing in simulation and performing the full fault injection testing on the FPGA where execution times are much lower. Mutants provide the same versatility as saboteurs, but also have the

same problem as saboteur regarding testing times. Mutants, however, require modifications of components and basic building blocks, which might introduce new bugs or hide existing bugs. Saboteurs can achieve the same effects as the result of fault injection should be a fault propagating through the design. Whether the fault was created inside a building block or on its output, is of less importance.

Based on this evaluation the saboteur was selected as the basic technique for the fault injection architecture. The saboteur provides a versatile solution requiring minimal modifications for fault injection which can be used both in simulation and on an FPGA. In simulation, only basic testing of the fault injection architecture will be performed, and the comprehensive test suite will only be run on the FPGA.

3.8.3 Fault injection locations

The locations for fault injection should be chosen such that all faults mentioned in the fault model are tested. But testing all possible locations where faults can occur according to the fault model, might result in too many locations to be tested and very long test execution times. This can be solved by randomly selecting the locations for fault injection, but this can create concerns over the coverage of such a framework, which is as good as the random number generator. Fixed injection locations can result in more confidence in the test coverage, but could be more limited than random fault injection. So, the fault injection architecture should cover as many faults as possible with a minimum number of locations. Another advantage of using fixed locations is that test can be reproduced if they fail as the locations are not randomly generated. The saboteurs in the hybrid design can be divided into two categories: saboteur mimicking faults just before decoders and majority voters and thus mimicking the propagation of faults to these components and saboteurs injecting faults at the outputs of large components where faults are likely to occur.

Faults injected by saboteurs placed before the decoders and majority voters mimic faults originating from other locations where according to the fault model a fault could have occurred. In Table 3.11 these saboteurs are listed along with the type of fault and location from which these originate.

Besides these basic fault injection locations, saboteurs have to be placed at the outputs of those components in the pipelines where a fault is most likely to occur. All the saboteurs listed in Table 3.11 are outside or on the edges of the pipelines and placed in front of a decoder or majority voter (except for the program counter input saboteur). Therefore, the injected faults will be detected immediately by one or more detectors depending on the location of the inserted fault. In real life operation induced faults can also remain silent and thus not be detected by a decoder or majority voter. Moreover, a single fault can also trigger more decoders and voters at once than the already defined saboteurs can. Therefore, extra saboteurs should be inserted into the pipeline, for fault injection on the outputs of components which are very susceptible to faults or can have a major impact on the softcore execution.

The susceptibility of components to faults can be determined by calculating the Failure In Time (FIT) rate for each component. The effective FIT rate, defined as the number of failures to be expected in 10^9 hours (approximately 114,155 years) of

Table 3.11: The basic locations of the saboteurs mimicking faults from the fault model.

	Fault type			
Saboteur location	SEU	SEMBU	SET	Fault origin
Instruction memory input			✓	`tl_data_bus_req` majority voter or interconnect
Instruction memory output	✓	✓		Instruction memory
Data memory input			✓	`tl_data_bus_req` majority voter or interconnect
Data memory output	✓	✓		Data memory
Register file input			✓	`risc_v_registers_in` majority voter or interconnect
Register file output	✓	✓		Register file
Program counter input	✓	✓	✓	Pipeline logic or FF
Program counter output	✓	✓		Program counter register
Stats output	✓	✓	✓	Pipeline logic or FF

service, is the measure used to indicate the likeliness of failures occurring in a system. For Xilinx FPGAs the FIT rate can be calculated with Eq. (3.2) where s is the size of either Configuration Random Access Memory (CRAM) or Block Random Access Memory (BRAM) in Megabits and DVF is the Device Vulnerability Factor (DVF).

$$FIT_{eff} = DVF * (s_{CRAM} * FIT_{CRAM}) + s_{BRAM} * FIT_{BRAM} \tag{3.2}$$

The DVF is the number of faults in the configuration bits actually creating a soft functional error, which according to [10] is 5% for a typical design and 10% in a worst case scenario. In the Device Reliability Report from Xilinx [10] the FIT rates per Megabit of CRAM and BRAM for the Spartan-6 are provided (listed in Table 3.12). The CRAM usage can be calculated with the estimated resource usage per resource type which are provided by [11] and listed in Table 3.13. With the mapping results for the baseline softcore listed in Table A.4 and the data from Xilinx, the FIT rates can be calculated. The results of this calculation are presented in Fig. 3.13. A DVF of 10% was used for these calculations.

Table 3.12: The FIT rates per Megabit of CRAM and BRAM for the Xilinx Spartan-6[10]

	FIT/Mb	
Type	CRAM	BRAM
Thermal Neutrons	21	83
Alpha particles	88	172
Real Time Soft Error Rate per Event	177	370

Table 3.13: Estimated CRAM usage per resource type for the Spartan-6 [11].

Resource type	CRAM (b)
Logic Slice	1166
BRAM (18 Kb)	4698
BRAM (9 Kb)	2349
I/O Block	2850

Based on the results presented in Table A.4 and Fig. 3.13 it was decided to place

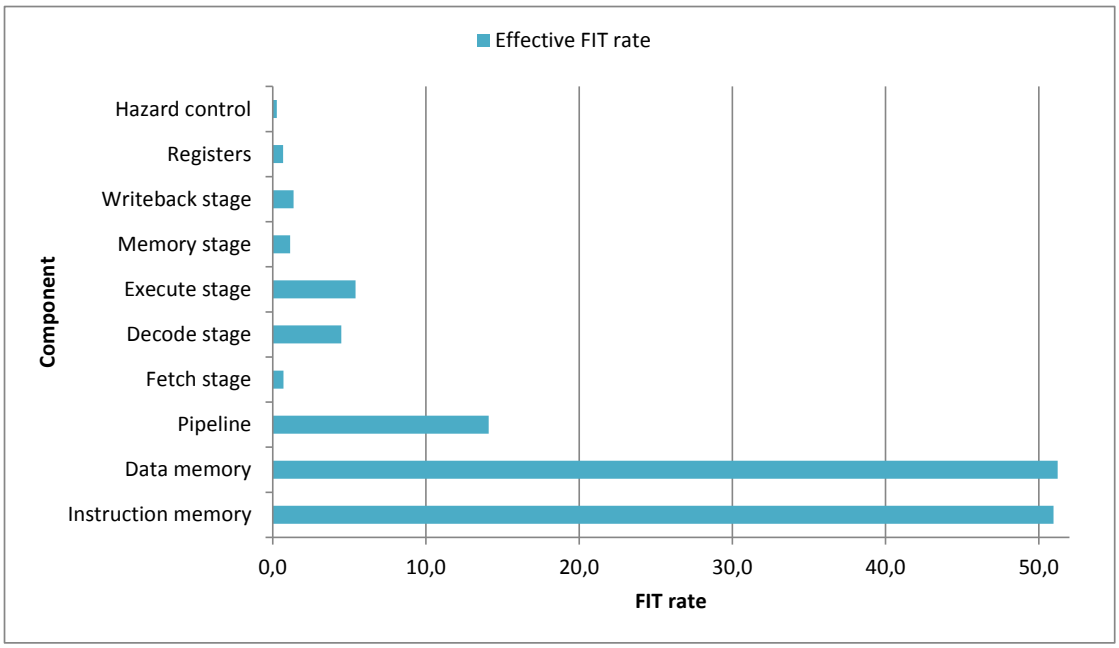

Figure 3.13: Results of the effective FIT rate calculations for selected components.

saboteurs on the outputs of the Decode and Execute stages and Hazard Control. The results show that the Decode and Execute stages are the biggest components and thus are the most susceptible to faults. On top of this, these stages also have an important role in the functioning of the processor, as faults in the Decode stage will cause the processor to execute erroneous instructions and faults in the Execute stage will cause erroneous calculation results to be used and stored. Hazard Control is also included as a fault injection location, as this component will stall the pipeline unnecessarily when a fault occurs. This will cause a pipeline to get out of sync and will create a lot of errors. By applying fault injection on the Hazard Control, it can be verified if the design can withstand such a large disturbance in its system. All the fault injection locations are included in the top-level diagram in Fig. 3.14 as red interconnections.

3.9 Conclusion

In this chapter, the baseline design of the RISC-V softcore, implementing the RV32I instruction set and a classic five-stage RISC pipeline, was presented. The presented fault model defines at which locations in the softcore and at which moments SEUs and SETs can occur in softcore. The set of design criteria used for evaluation of the design options and the fault tolerant designs include error detection/correction capabilities, resource utilization, and complexity amongst others.

Two fault tolerant designs were made, both using NMR and ECC. The ECC-based design uses ECC for protection whenever possible, both in the pipeline and in memory elements, in an effort to save resources. Only for combinatorial elements which ECC

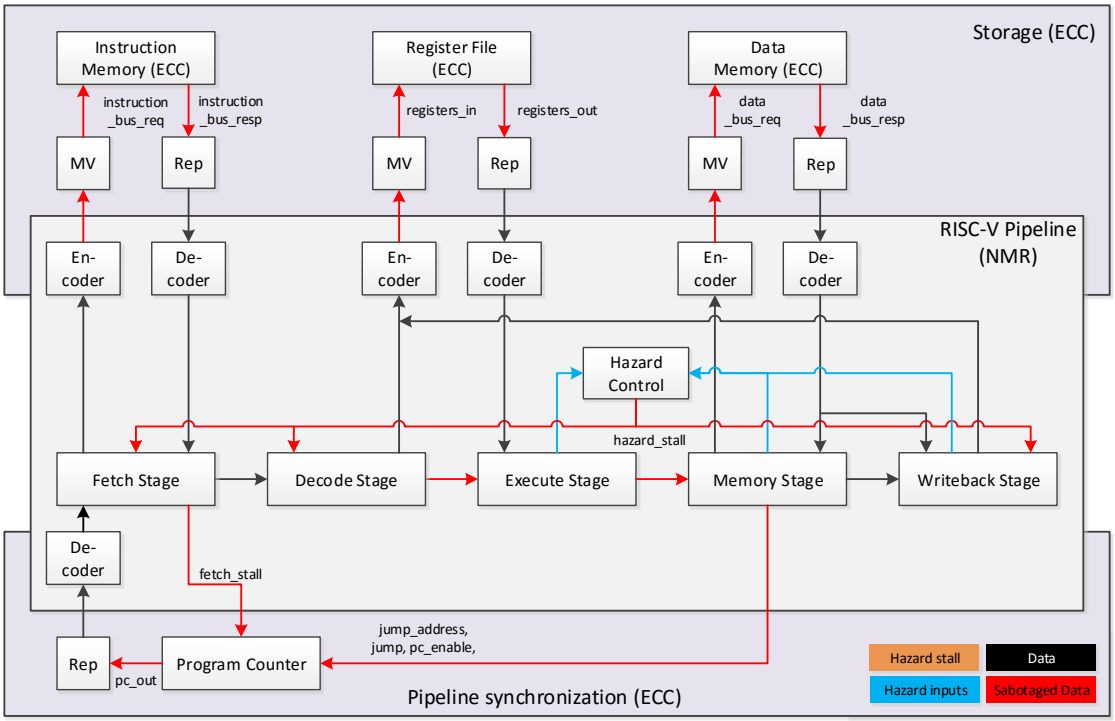

Figure 3.14: Top-level block diagram of the hybrid design, with all signals going through a saboteur marked red.

cannot protect NMR is used. The low-level switches between the ECC and NMR domain result in a significant number of SPOFs and a high complexity of the ECC-based design. The hybrid design uses NMR to protect the pipelines and ECC to protect memory elements. This design has a limited amount of switches between protection domains and by letting the domains overlap, possible SPOFs are avoided.

After evaluation of both designs for all the design criteria, the hybrid design was chosen for implementation, because of its better performance for all design criteria except resource utilization. The redundancy of the design will be tested by using fault injection through saboteurs, which will be inserted into the design. In simulation, the basic fault injection architecture will be tested and on the FPGA a more comprehensive test suite will be used to test the fault tolerance.

Implementation of a fault tolerant RISC-V softcore

4

In Chapter 3 two designs were presented: the ECC and hybrid design. The hybrid design was selected for implementation based on the evaluation of both designs for the design criteria. This design can be implemented in two steps. First, the missing building blocks need to be designed, implemented, and tested. These blocks are the Hamming encoder & decoder, the majority voter, and the saboteur Once, these blocks are available, the structural changes to the core and SoC level can be made.

In this chapter the design, implementation and unit test of the missing basic building blocks will be described in Sections 4.2 till 4.4. The implementation of the structural changes in the softcore itself will not be discussed extensively as these changes largely follow from the block diagram in Fig. 3.11. In Section 4.5 the changes to the SoC required for incorporation of the fault tolerant core and the memory mapping of this new SoC will be discussed.

4.1 ECC specifications

In the hybrid design, ECC will be used to protect the memory elements and most buses and interconnect records originating from the domain crossings as defined in Table 3.7. The `tl_data_bus` will be used for communication between the pipelines and the instruction memory, data memory, and other SoC peripherals. For communication between the pipelines and the register file two interconnect record types were defined: `risc_v_registers_in` and `risc_v_registers_out`. For communication between the pipelines and the program counter the interconnect record types `risc_v_pc_in` and `risc_v_pc_out` were defined. In Table 4.1 the fields of the bus and interconnect record types are defined along with their width and whether the field is protected with ECC or not. If a field is protected with ECC an extra field containing the check bits with the suffix `_check` is added to the record type. The ECC coding scheme uses a systematic form where data and check bits are placed in two separate parts of the code. In the interleaved format the check bits would be placed at the bits of the code word for which the index is a power of two.

The `tl_data_bus_req` and `tl_data_bus_resp` records are used for both the instruction and data memory, which use different encodings. The instruction memory uses standard word encoding, where the data memory uses byte encoding to enable the byte accesses required by the RISC-V ISA. The check words for the four bytes in the data word are stored in concatenated form. To accommodate the larger, concatenated check word for byte encoding an extra field, `data_check_byte`, was added to the `tl_data_bus` records.

Not all fields of the records mentioned in Table 4.1 will be protected with ECC. Most of these fields are control bits, which were not specifically targeted by this design. It should, however, be fairly easy to add protection for these bits by adding a check word

Table 4.1: Record types used in the RISC-V softcore, which are (partly) protected with ECC

Record	Field	k	ECC
`tl_data_bus_req`	write	1	
	read	1	
	byte_en	4	
	data	32	✓
	address	32	✓
`tl_data_bus_resp`	ack	1	
	nack	1	
	data	32	✓
`risc_v_pc_in`	jump	1	
	fetch_stall	1	
	memory_stall	1	
	hazard_stall	1	
	jump_address	32	✓
`risc_v_pc_out`	pc_address	32	✓
`risc_v_registers_in`	r_enable	1	
	w_enable	1	
	rs1_address	5	✓
	rs2_address	5	✓
	rd_address	5	✓
	rd_data	32	✓
`risc_v_registers_out`	rs1_data	32	✓
	rs2_data	32	✓

for a single bit data word. Checking whether the correct operation was executed, based on the control bits value, however, is much more difficult and is not considered in this thesis.

In Table 4.2 the specifications for the Hamming ECC coding for different data widths is presented. The main requirement for the FT softcore is the SECDED error capability, which corresponds to a Hamming distance of four. The check widths for lower Hamming distances are also mentioned as these encodings should be easily interchangeable with the SECDED encoding.

Table 4.2: The specifications for Hamming-based ECC coding for different word sizes and d_{min}.

			data width					
	Redundancy		5		8		32	
d_{min}	ED	EC	n	m	n	m	n	m
1	0	0	5	0	8	0	32	0
2	1	0	6	5	9	1	33	1
3	1	1	9	4	12	4	38	6
4	2	1	10	5	13	5	39	7

4.2 Hamming encoder & decoder

The basic components for the ECC protection domain are the encoder and decoder, which can transfer data to and from the domain. In this section, the generic block design of the encoder and decoder will be described. This basic block will be used to construct record specific encoders and decoders, which are structural architectures containing encoders/decoders for all fields in the record. In the case of the `tl_data_bus` encoders, and decoders, a generic will be available to switch between word and byte encoding for the data field. This section will also describe the unit test and timing/area estimates for these components.

4.2.1 Block design

In Fig. 4.1 the block design for the Hamming encoder is presented. This encoder provides a Hamming code word in systematic from on its two outputs `data_out` and `check_out` based on the input `data_in`. The check generator uses an algorithm which is based on the interleaved form of the Hamming code word for generating the check bits. This is a relatively simple algorithm which could be implemented for all possible bit widths of `data_in`. This algorithm could also be replaced by a table indicating which data bits should be XORed for which check bits [57]. Using a table might enable the synthesizer to generate a more efficient implementation of the encoder, but it limits the usability of the encoder to the length of the table.

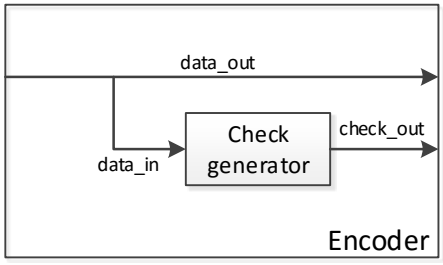

Figure 4.1: The block diagram for the Hamming encoder.

In Fig. 4.2 the block diagram for the detection only Hamming decoder is depicted. This decoder can detect errors for a specified Hamming distance for any combination of data and check words of valid length. If the decoder detects an error, it will set the output data and check word to the `std_logic` value X and will assert the `error_detected` flag.

In Fig. 4.3 the block diagram for the full Hamming decoder is depicted. This decoder can detect and if possible correct errors for a specified Hamming distance for any combination of data and check words of valid length. When a correctable error is detected, the error will be corrected and the corrected data or check word will be placed on the output along with the valid data or check word and the `error_detected` and `error_corrected` flags will be asserted. When a non-correctable error is detected, the decoder will behave the same as the detection only decoder and the `error_corrected` flag will be de-asserted.

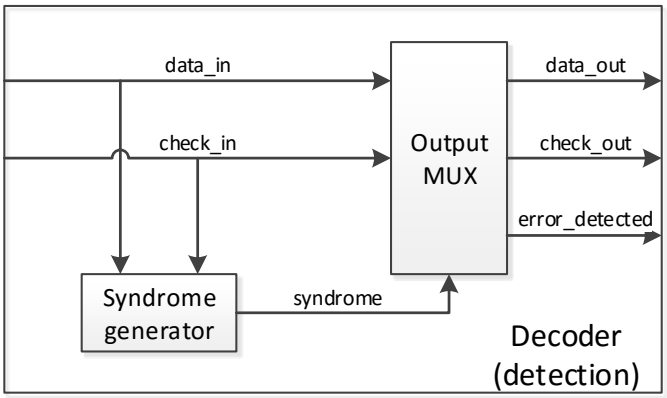

Figure 4.2: The block diagram for the detection only Hamming decoder.

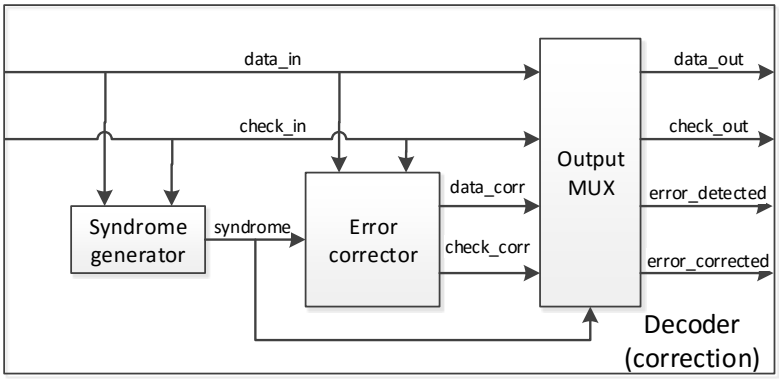

Figure 4.3: The block diagram for the correction Hamming decoder.

4.2.2 Unit test

The implementations of the Hamming encoder and decoder were tested with a unit test bench. The unit test bench uses a test harness (depicted in Fig. 4.4), which can be configured for fault tolerance level and bit width. The source provides the encoder with input data along with a mask used for fault injection through the XOR port. Sink 1 sends the code word produced by the encoder along with the input data back to the test bench for checking. The test bench sends all possible input values for the given bit width to the test harness through the source. When output data is received, the test bench checks if the values received are in the right order and for selected values, it checks whether the calculated values are correct. These selected values were calculated by hand to ensure correctness of these values. Sink 2 sends the decoded data back to the test bench for checking along with input data. The test bench also checks the data from sink 2 for the right order of receipt and against the selected values. The test bench will also take into account if faults were injected and if the decoder detected or corrected the faults depending on the number of faults injected. In Table 4.3 the tests performed by the unit test bench along with the configuration and the result are listed.

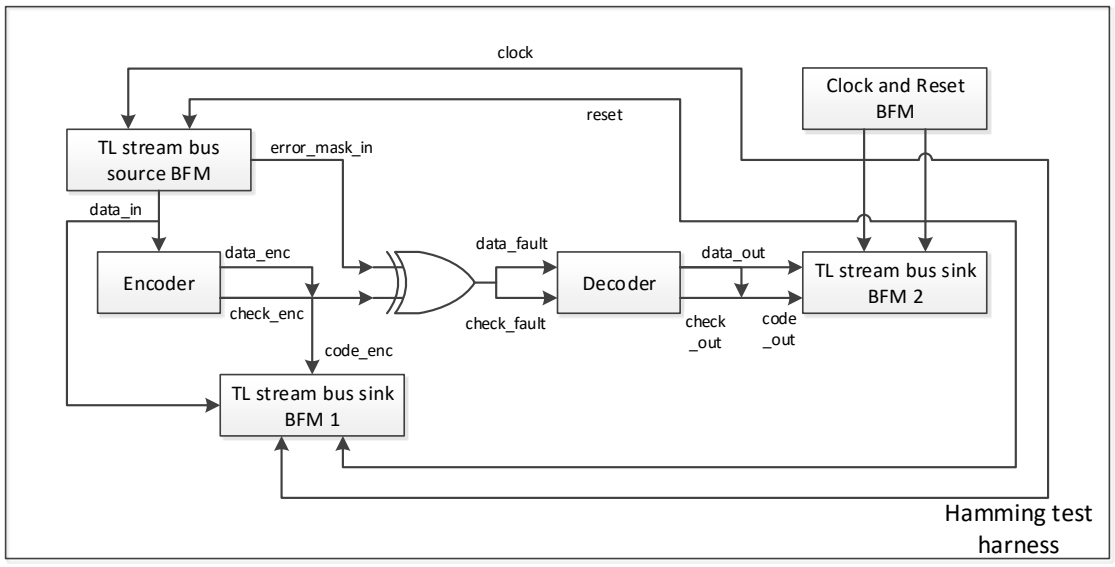

Figure 4.4: The simplified block diagram for the test harness used by the unit test bench for the Hamming encoder and decoder.

Table 4.3: List of the tests performed in the unit test bench for the Hamming encoder and decoder.

test	Redundancy	k	Faults	Result
1	SED	8	1	pass
2	SED	10	0	pass
3	SEC	8	1	pass
4	SEC	10	0	pass
5	DED	8	2	pass
6	DED	10	0	pass
7	SECDED	8	2	pass
8	SECDED	10	0	pass
9	TED	8	3	pass
10	TED	10	0	pass

4.2.3 Design cost estimate

In Tables 4.4 and 4.5 the design cost estimates for the Hamming encoder and decoder as used in the implementation of the hybrid design are listed. These design cost estimates differ from the estimates used for the design cost estimates made during design, as the implementation of the encoder and decoder was refactored during implementation from an interleaved format to a systematic format. These design cost estimates show that especially the decoder consumes a lot of resources for error correction and has a long critical path. This supports the observation made during the design phase that especially the decoders in both designs are responsible for the large decrease in clock

frequency. If only error detection is required, the resource usage and the critical path can be significantly reduced.

Table 4.4: Design cost estimate for the Hamming encoder with L the number of LUTs and and t_{comb} the maximum combinational path delay.

	SED		SEC/DED		SECDED/TED	
k	L	t_{comb} (ns)	L	t_{comb} (ns)	L	t_{comb} (ns)
5	1	0.7	3	0.5	4	0.5
8	2	1.4	4	0.7	5	0.7
32	7	2.1	19	2.1	23	2.1

Table 4.5: Design cost estimate for the Hamming decoder with L the number of LUTs and t_{comb} the maximum combinational path delay.

	SED		SEC		DED		SECDED		TED	
k	L	t_{comb} (ns)	L	t_{comb} (ns)	L	t_{comb} (ns)	L	t_{comb} (ns)	L	t_{comb} (ns)
5	1	0.7	14	2.4	3	2.1	29	4.8	5	1.9
8	2	1.5	18	2.4	5	1.9	38	5.0	8	2.9
32	7	3.3	90	5.8	21	3.5	129	10.3	26	4.5

4.3 Majority voter

The pipeline will be replicated several times based on the requested redundancy level (four times for SECDED). All the data which leaves the NMR domain to the ECC domain needs to be accumulated before it can be processed by components in the ECC domain. This accumulation is done by the majority voter, which will determine which value, zero or one, has the majority in the provided signals. If no majority can be determined, the output values will be set to zero and the voter will signal an uncorrectable error being detected. In this section, the block design will be discussed, along with the unit test and the design cost estimate.

4.3.1 Block design

Fig. 4.5 depicts the block diagram for the majority voter for an N-bit vector. The majority voter can handle any number of M vectors of N bits each, but these vectors need to be provided to the majority voter as one concatenated `std_logic_vector` due to limitations of VHDL-93. The router will split the input vector into separate vectors for each bit position in the original input vectors. The Data logic will calculate the output value for each of these separate vectors, the ED logic will determine if any errors are present in the vector and the EC logic will determine if these errors can be corrected by the voter. If an uncorrectable error is detected in any of the bits, `data_out` will be set to

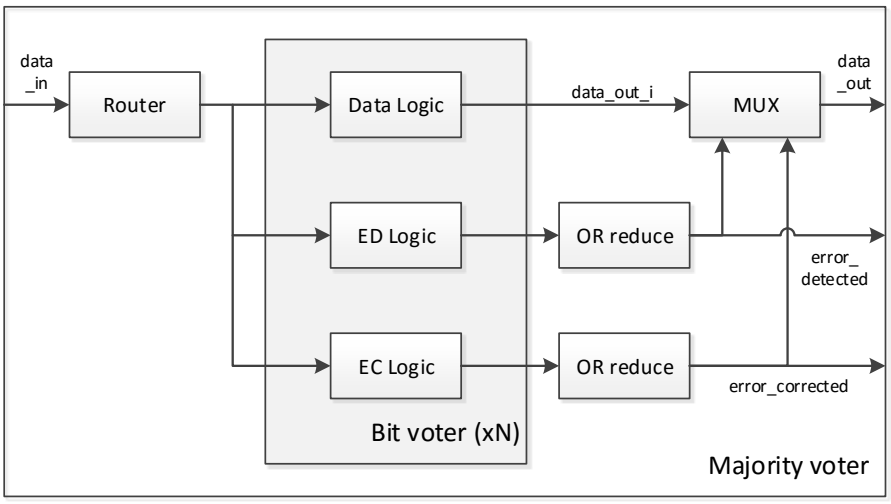

Figure 4.5: The block diagram for the majority voter.

the default value of 0 by the output multiplexer. The majority voter can also be used in a detection-only mode in which the voter will set `data_out` to 0 for every detected error. Table 4.6 list some binary examples for the bitwise majority voting procedure applied in the majority voter.

Table 4.6: Binary examples for the bitwise majority voting procedure for SECDED applied in the majority voter.

Example	Inputs				Output	ED	EC
No errors	101	101	101	101	101	0	0
One single bit error	101	101	101	100	101	1	1
Two single bit errors	111	101	101	100	101	1	1
One double bit error	101	101	100	100	000	0	1

Applying the majority voting procedure on a per bit basis means a significantly higher level of redundancy is used compared to the redundancy in ECC protected code words. For SEC the input vectors may contain multiple errors as long as these errors are not in the same bit position. A SEC code word decoded by a decoder may only contain one error if it is to be correctable. However, using extra redundancy for the majority voter is justified, as a pipeline containing a fault is likely to produce a completely different outcome than the other pipelines. This is especially the case when the fault occurred in a control signal in the pipeline which forced the pipeline to manifest different behavior than the other pipelines.

4.3.2 Unit test

The majority voter was also tested with a unit test bench using the test harness depicted in Fig. 4.6. The test bench uses different harnesses with different settings for the

redundancy and the bit width. Table 4.7 contains a complete listing of the tests along with the test results. Each test harness is fed with all possible combinations of input values for the used bit width and the number of faults injected by the test bench. The number of injected faults is changed by setting one or more of the original input vectors to 0, simulating a pipeline which produces completely erroneous results. The test bench checks if the received results are in order and are matching with what was expected.

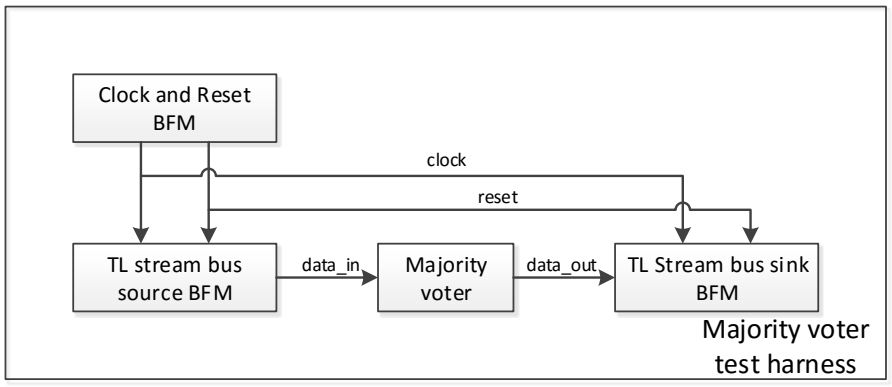

Figure 4.6: The block diagram for test harness used by the unit test bench for the majority voter.

Table 4.7: List of the tests performed in the unit test bench for the majority voter with n as the bit width.

Test	Redundancy	d_{min}	n	Result
1	SECDED	4	1	pass
2	SED	2	6	pass
3	SEC	3	6	pass
4	DED	3	6	pass
5	SECDED	4	6	pass
6	TED	4	6	pass
7	SECDED	4	8	pass

4.3.3 Design cost estimate

In Table 4.8 the design cost estimate for the final implementation of the majority voter is listed. This estimate was not used for the design cost estimates of the ECC-based and hybrid design in Chapter 3. The design cost estimate shows that some configurations of the majority voter can be mapped better to the FPGA fabric than other configurations. The resource usage will increase if the bit width is increased, but a smaller majority voter might have a higher latency than a larger majority voter. It might be advantageous to combine several signals into one `std_logic_vector` for voting, for saving resources and reducing the critical path. But in the final implementation of the hybrid design, the

synthesizer might apply other optimizations due to the cascade of the majority voter with other components, which eliminate these resource mapping effects. Furthermore, other devices might not be affected by these effects, so such optimizations should only be applied when they are required for achieving certain performance requirements. As this design should be platform independent, such optimizations were not used in this implementation.

Table 4.8: Design cost estimate for the majority voter with L the number of LUTs and t_{comb} the maximum combinational path delay.

	SED		SEC		DED		SECDED		TED	
k	L	t_{comb} (ns)	L	t_{comb} (ns)	L	t_{comb} (ns)	L	t_{comb} (ns)	L	t_{comb} (ns)
1	2	0.4	2	0.5	2	0.5	3	0.5	2	0.5
4	6	2.7	7	2.6	7	3.9	15	4.5	10	4.8
5	7	3.0	9	2.7	9	4.0	19	5.6	14	6.0
8	12	4.6	18	2.9	17	4.6	49	5.5	22	7.9
20	27	3.0	40	1.6	40	3.4	113	6.1	40	3.4
32	43	3.3	64	1.9	64	3.9	184	7.3	64	3.9

4.4 Saboteur

In Section 3.8 fault injection was selected as the verification method for the hybrid design. Saboteurs will be used to inject faults into signals in the design. The locations were saboteurs will be inserted into the design were defined in Section 3.8.3. In this section, the block design, the unit test, and the design cost estimate for the saboteur will be discussed.

4.4.1 Block design

Fig. 4.7 depicts the block diagram of the basic design for the saboteur, which can be configured for any possible bit width of the `input` signal. The faulty signal will be created by XORing the input signal with a mask with the same width as the input. The saboteur is fitted with a 16-bit counter, which is compared with the condition. If there is a match between the counter and the condition, the faulty signal will be placed on the output by the multiplexer and the counter will be reset. The saboteur will only inject faults when the core is enabled as indicated by the `core_enable` signal and when the saboteur is enabled as indicated by the `enable` signal. By using a counter for periodic injection, the saboteur can insert multiple faults in a test run, without the need for interaction with the tester during the run itself. Along with the fixed injection locations, this makes the fault injection test completely reproducible as the fault injection pattern is based on only a few parameters in a deterministic manner.

The mask, condition and enable signals are coming from an external register block, the saboteur registers, used for controlling all saboteurs in the core. The `fault_inserted`

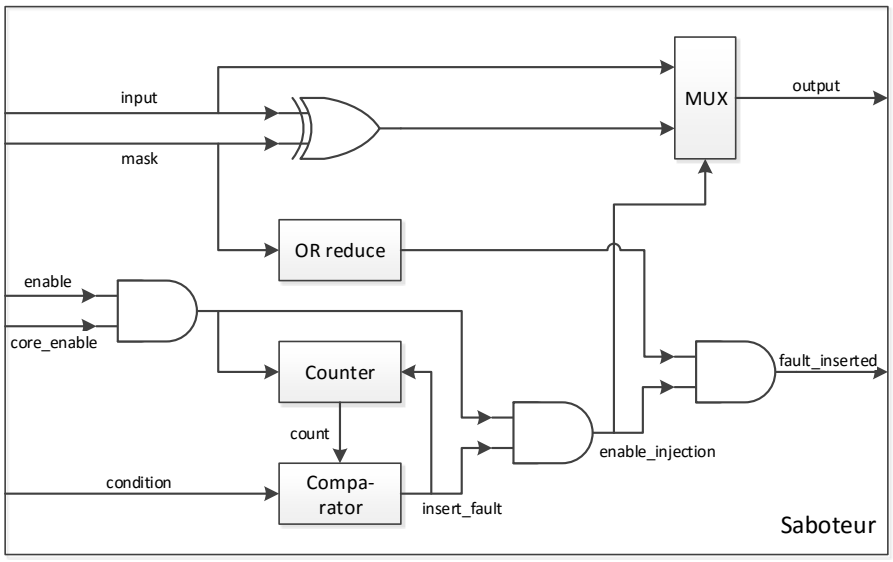

Figure 4.7: The block diagram of the saboteur.

flag is connected to a counter in the saboteur registers and will be asserted each time a fault is injected. The counter will count the number of times a fault is inserted during a test run. The counter cannot differentiate between the injection of a single, double or multiple fault injection into a single word, because `fault_inserted` is a 1-bit signal. The information stored in the saboteur registers can be used after the test run for verifying the results of the test run. The saboteur registers can be accessed (both read and write) by the test environment running on an external system through the Ethernet connection. Using these saboteur registers gives the test environment complete control over the fault injection pattern used for each test.

The basic saboteur described before will be used in larger record saboteurs which can be inserted on the `tl_data_bus` and other record types used in the softcore. These record saboteurs contain saboteurs for all fields of the record, which can be completely enabled or disabled with one generic `g_fault_injection_enable`. Most record types contain different kind of fields (control, address, data), for which fault injection can be enabled or disabled as well through separate generics. In the `tl_data_bus` request and response saboteur, there is also a generic available for switching between word and byte encoding for the data field. These generics are used on top of the enable signals provided by the saboteur registers and the `core_enable` signal, to speed up simulation by completely bypassing the saboteur if requested. These generics will also help in reducing the size of the generated core by not including some saboteurs in the design or completely disabling fault injection.

4.4.2 Unit test

The saboteur was also tested with a unit test bench using the test harness depicted in Fig. 4.8. Table 4.9 lists all test performed by the unit test bench along with the test results. Each test harness is fed with all possible combinations of input values for the

used bit width. Test 3 is also fed with all possible combinations of input masks containing a single error for each possible input value. The testbench checks if the received results are in order and are matching with what was expected.

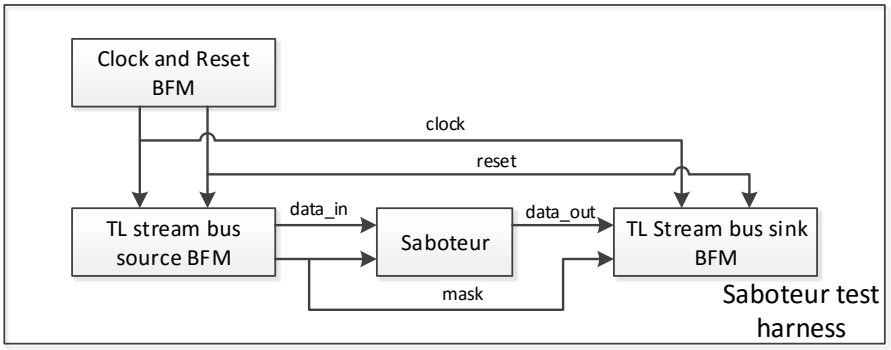

Figure 4.8: The block diagram for test harness used by the unit test bench for the saboteur.

Table 4.9: List of the tests performed in the unit test bench for the saboteur with n as the bit width.

Test	n	Condition	`enable`	Results
1	6	0x0000	0	pass
2	8	0x0000	0	pass
3	6	0x0004	1	pass

4.4.3 Design cost estimate

In Table 4.10 the design cost estimate for the saboteur is listed. The LUT resource utilization for a single saboteur might be not so large, but there are 14 record saboteurs present in the hybrid design, each containing several saboteurs for the different fields in the record. Therefore, it should be expected that the fault injection core uses much more LUTs than the fault tolerant core. The period (and thus the f_{max}) of the saboteur is small enough not to affect the overall clock frequency of the design. The path delays of the in- and output, however, are considerable when taking into account that these delays will be added to already long paths. Therefore, it can be expected that the fault injection core will also run at a lower frequency than the fault tolerant core.

4.5 System on Chip

In Fig. 4.9 the block diagram for the fault tolerant SoC incorporating the fault tolerant softcore based on the hybrid design is depicted. Compared to the baseline SoC, three components were added: an encoder for the `tl_data_bus_resp` from the bridge between the `tl_reg_bus` and the `tl_data_bus`, the error counters, and the saboteur registers.

Table 4.10: Design cost estimate for the saboteur.

			Path delays (ns)				
k	LUTs	Registers	t_{period}	t_{input}	t_{output}	t_{comb}	f_{max}(MHz)
1	40	16	3.6	2.6	3.9	2.8	274.1
4	44	16	3.7	2.7	4.1	3.0	268.4
5	45	16	3.7	2.7	4.1	3.1	266.9
7	47	16	3.8	2.7	4.1	3.1	263.6
8	48	16	3.8	2.8	4.2	3.1	262.1
20	63	16	4.0	3.0	4.2	3.2	248.9
32	78	16	4.2	3.2	4.4	3.4	237.1

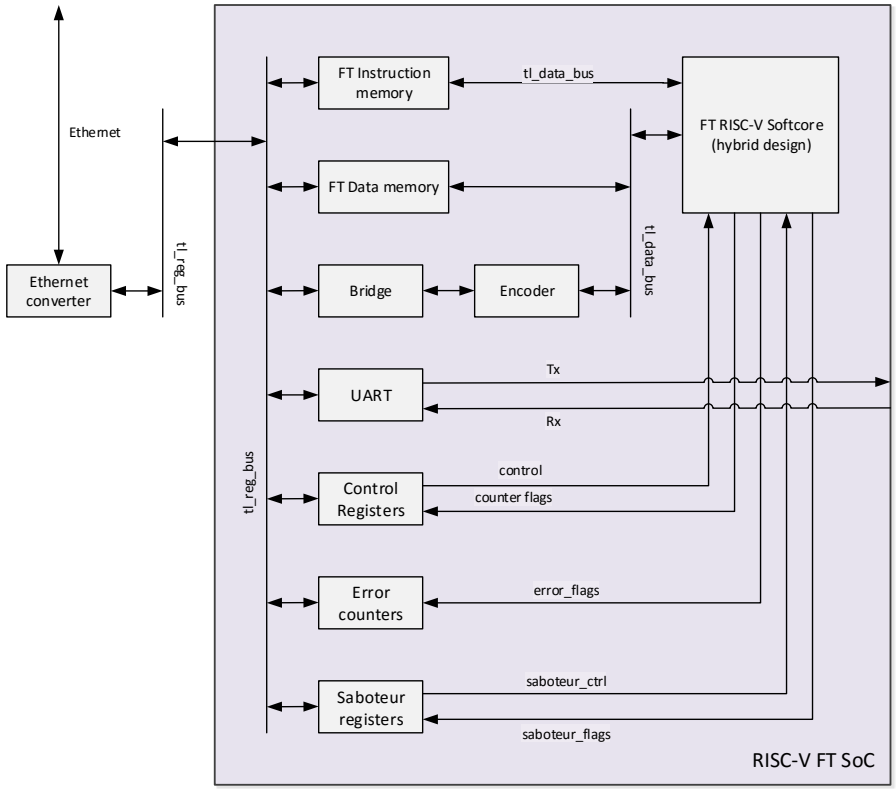

Figure 4.9: The block diagram of the fault tolerant RISC-V System on Chip.

Furthermore, the instruction and data memory were replaced with the fault tolerant variants.

4.5.1 Memory mapping

In Table 4.11 the memory mapping of the `tl_reg_bus` of the fault tolerant RISC-V SoC used for testing is listed. This memory mapping describes the regions which can be used by the core and the test environment to run programs and check the results of

the programs after execution. The memory mapping can be easily adjusted with the Register Definition Language (RDL), a proprietary tool developed by Technolution B.V. to define memory mappings and register blocks. The RDL structure of the fault tolerant RISC-V SoC contains several map files, which define the different memory regions used and multiple register files defining the register blocks. Based on these files the RDL compiler will generate VHDL, C and Python files, with constants, records, objects, etc. which can be used to access components connected to the `tl_reg_bus`. This enables the user to adjust the memory mapping by changing one value in one RDL file, without having to adjust addresses and mappings in several locations in the SoC and the test environment.

Table 4.11: The memory mapping of the fault tolerant RISC-V core.

Region	Start	Size
Instruction memory	0x00000000	0x4000
Data memory	0x08000000	0x4000
UART	0x10000000	0x1000
Control registers	0x10001000	0x1000
Error counters	0x10002000	0x2000
Saboteur registers	0x10004000	0x2000
External registers	0x10100000	0x100000

Every region in the memory mapping is connected to a module with a different function:

- **Instruction memory:** memory containing all executable instructions.
- **Data memory:** memory containing all data elements used by the core to execute a program.
- **UART:** component for asynchronous serial communication with the outside world through `printf` and `scanf` function calls.
- **Control registers:** register block containing the soft reset and enable registers amongst others used for controlling the softcore. The control registers also contain soft reset and enable status registers, which indicate if the core is executing or has stopped.
- **Error counters:** register block containing counters for the number of errors detected by the decoders and majority voters. Every checker has a counter, so the exact location and number of errors are available after an execution run. The error counters can be reset with the soft reset.
- **Saboteur registers:** register block containing control registers and counters for all saboteurs in the softcore. The control registers include the condition, mask and enable signals for every saboteur. Every saboteur has a counter, so the exact

location and number of injected faults are available after an execution run. The saboteur registers can be reset with the soft reset.

- **External registers:** register block for components outside the SoC environment, which need to communicate with the core.

All regions listed in Table 4.11 can be accessed by the test environment through the Ethernet connection and the `tl_reg_bus`. The softcore has two `tl_data_busses`: one connected to the instruction memory and one connected to the data memory. The second `tl_data_bus` used for accessing the data memory is also connected to the bridge between the `tl_data_bus` and the `tl_reg_bus`. Through this bridge the softcore can access all peripherals connected to the `tl_reg_bus` like the UART which is used by `printf` and `scanf` function calls. The bridge also allows the softcore to disable itself by clearing the enable flag in the control registers. This can create a deadlock situation, but it is a very useful feature when testing. The enable flag from the control registers is also connected to the enable status flag in the same register block. Once the enable flag is cleared by the softcore because it has finished execution, the test environment can see this if it is polling the enable status flag in a busy waiting loop. The test environment can then proceed with checking the test results, clear the softcore and peripherals through the soft reset, and start a new run.

The saboteur registers are only used when the core is tested and do not have to be included in a final implementation. Therefore, the saboteur registers are only added to the design when the `g_fault_injection_enable` generic is true. When this generic is false, the saboteur register will be replaced with a `tl_reg_bus` dummy, which will provide a positive or negative acknowledgment for every read or write request to this memory region depending on the configuration of the dummy.

4.5.2 Control registers

The control register block contains the following registers:

- `reset_control`: contains the soft reset and core enable control flags, for resetting and enabling the core.
- `core_status`: contains the soft reset, core enable, and core stopped status flags.
- Several counter registers for gathering statistics on the software execution like the number of cycles, instructions, stalls, and hazards.

The control flags are read/write registers which can be accessed by every master on the `tl_reg_bus`, in this case, the test environment and the softcore itself. The core itself also has write permissions on these control flags so it can disable itself once it has finished execution of a test application. The status flags indicate to the test environment what the actual value of the reset and enable flags seen by the core are. This is necessary, as the reset control flag is ORed with the hard reset and the enable control flag is ANDed with the hard enable and the core stopped flag. The core stopped flag is asserted if the core detects an uncorrectable error and cannot continue execution. The core stopped flag

is also included a status flag in the control register block so that the test environment can check the cause of a failed execution. The status flags are read-only.

4.5.3 Error Counters

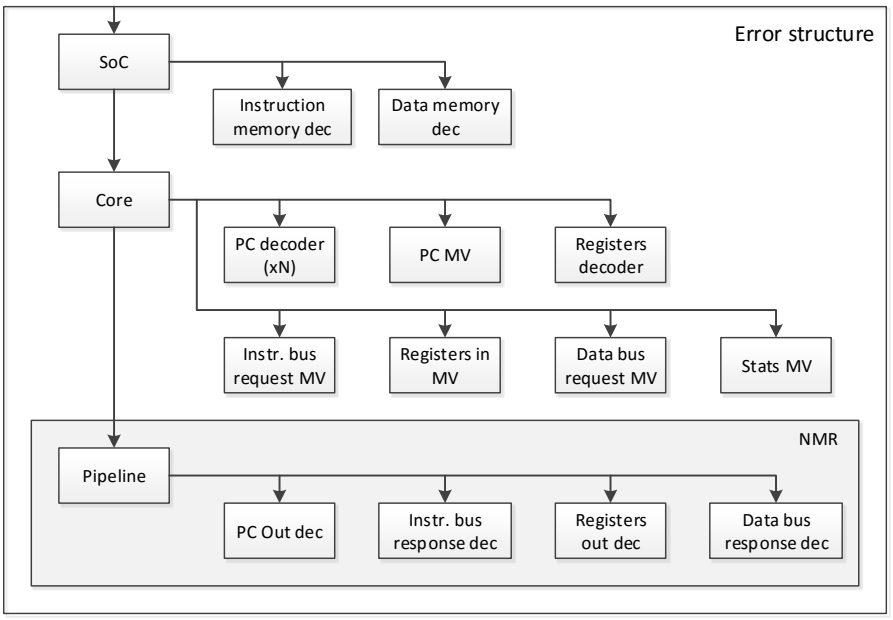

Figure 4.10: The structure of the Error counters register block.

The error counters will count the number of detected errors by each decoder and majority voter. These error counts will be compared with the number of injected faults to see if all faults were detected or if faults remained silent depending on the scenario. These error counters can also be accessed (both read and write) by the test environment through the Ethernet connection. When a counter is written by the test environment, the counter will be reset. The structure of the error counters is depicted in Fig. 4.10.

4.5.4 Saboteur registers

The saboteur registers will control the saboteurs in the softcore and count the number of times a fault is injected. For each record saboteur, a register set is included in the register block. Each register set contains:

- `condition`: condition used by the saboteur timer to count the interval between consecutive fault injections.
- `mask`: mask used for fault injection for each field of a record.
- `enable`: enable register containing an enable bit for each of the fields of a record.

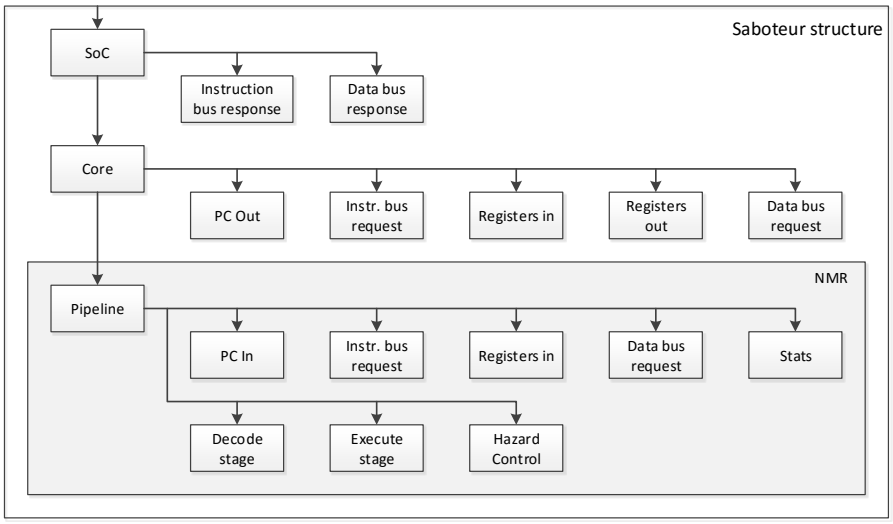

Figure 4.11: The structure of the saboteur register block.

- `field`: x counters (one for each field in the record) which count the number of injected faults for each field. `field` is the name of the field of the record.

All fields can be read and written through the `tl_reg_bus`. If one of the `field` counters is written, the counter will be reset. The structure of the saboteur registers is depicted in Fig. 4.11.

4.6 Conclusion

In this chapter, the implementation details of the hybrid design were presented. The specification of the used Hamming code was presented along with a list of all the signals which will be protected with ECC. Furthermore, the design, unit test and cost estimates for the basic building blocks, the Hamming encoder, and decoder, and the majority voter were presented. The cost estimates showed that the latency of these basic building blocks are very high compared to the latency of the different pipeline stages. This is one of the main causes why the clock frequency of the fault tolerant design is reduced. Finally, the fault tolerant RISC-V SoC and the memory mapping used for testing were presented. The SoC contains a register block for controlling the softcore, a saboteur register block used for controlling and monitoring the saboteurs, and error counters used for monitoring the checking circuitry. The data from these register blocks can be used to check if all inserted faults were detected during fault injection verification.

Verification of a fault tolerant RISC-V softcore

5

In Chapter 3 and Chapter 4 the design and implementation of a fault tolerant RISC-V softcore were discussed. This core should be able to correct the effects of any single fault and detect any double fault occurring in the pipeline, a data word, or an address word. It must be verified that the implemented solutions provide the softcore with these capabilities. For this purpose, a verification design was made in Section 3.8, which defines at which locations faults should be injected into the design to verify the fault tolerance. With the verification design in mind, a test architecture was build to control the verification design and perform automated verification. In this chapter the test architecture used will be explained in Section 5.1 and the results of this verification process will be presented in Section 5.2. Furthermore, the benchmarks of the different SoCs will be presented in Section 5.3. The resource utilization of the softcore on the Xilinx Spartan-6 will be presented in Section 5.4. The timing analysis performed for the proof of concept will be presented in Section 5.5. The resource utilization and timing results are then compared with the design estimates in Section 5.6 and with the design criteria in Section 5.7.

5.1 Test architecture

For performing the verification of the fault tolerance, a test architecture is required. The test architecture for this design contains the following components:

- Test applications to be executed by the test architecture during verification.
- A simulation-based test environment using ModelSim for verification of the basic functionality.
- An on-board test environment using Python to control the test execution on the Xilinx Spartan-6 FPGA for verification of the fault tolerance.

In this section each of these components will be discussed.

5.1.1 Test applications

In each test, a number of test applications should be executed on the softcore, while specific properties of the softcore are tested. These properties can be the basic functionality of the softcore or mitigation capabilities of the softcore during fault injection. For the suite of test applications the following requirements can be defined:

- Verify if each instruction of the RV32I instruction set is properly implemented and executed.

- Execute all implemented instructions multiple times.
- Verify if the execution flow of test program was correct by checking results in data memory or checking the received data on the UART transmission line.

Based on these requirements, four test applications were selected for the verification suite:

- **Basic SoC test:** a small test program executing some often used instructions (addi, jal, sw), which tests basic functionality of the softcore. The test stores two values in memory, which can be checked for correct execution after the test finished. This test is used for quick debugging, as the execution time of this program is very small.
- **RISC-V ISA verification set:** a verification set provided by the RISC-V community [58] which tests the compliance of the hardware implementation with the specification with unit tests (one per instruction). After each unit test, a result code is stored in memory. If this result code is one, the test was successful. Any other code indicates a failure, where the result code points to the specific test causing the failure. The test strategy used by these unit test was developed at UC Berkeley and ICSI during the Torrent-0 vector microprocessor project [59].
- **Hello World:** a unit test for the UART transmission capability. When successfully executed, the test environment should receive two strings on the UART transmission line. The UART receipt capability was not tested as this required a busy waiting loop or interrupt.
- **Dhrystone:** a synthetic benchmark [56] used for executing a long program on the softcore while performing fault injection. The Dhrystone benchmark contains 20 checks on both integer and string variables which will indicate successful execution.

For each of these tests checking conditions are defined, which indicate success or failure after test execution is finished. For each test application except the basic SoC test, the instruction coverage was determined by counting the number of instruction reaching the write back stage with an extra register block in the standard SoC. The instruction coverages for the individual applications and the verification suite are listed in Table E.1. The instruction coverage for most instructions is reasonably well to very good. Some instructions are executed more than others, but the ISA verification set ensures most instructions are executed. The only exceptions to this are the timing instructions and sltu. Writing a good test for the timing instructions is difficult, as this is very much dependent on the handling of data and control hazards by the softcore. For the sltu (and the sltiu) instruction, a unit test is missing in the ISA verification set.

Besides the applications of the verification suite, one other application will be used for benchmarking the processor: CoreMark [60]. This benchmark is maintained by the Embedded Microprocessor Benchmark Consortium (EEMBC) and is aimed at replacing the Dhrystone benchmark. Like Dhrystone, CoreMark tests how much iterations the processor can perform within one second and presents a single number which can be

used to compare processors. Where Dhrystone is a synthetic benchmark, CoreMark uses real algorithms like list processing, matrix multiplication, and CRC. The instruction coverage of CoreMark is also listed in Table E.1.

5.1.2 Simulation-based test environment

The basic functionality of the softcore will be tested in the simulation-based test environment using ModelSim as the simulator. The simulation-based test environment also uses the Self Checking Test Bench (SCTB) test environment of Technolution B.V. for automated verification. The SCTB consists of:

- Test harnesses containing the Module Under Test (MUT) and the required Bus Functional Models (BFMs).
- Test cases using one of the available harnesses.
- A makefile containing a list of all available test cases plus a definition of the regression tests.

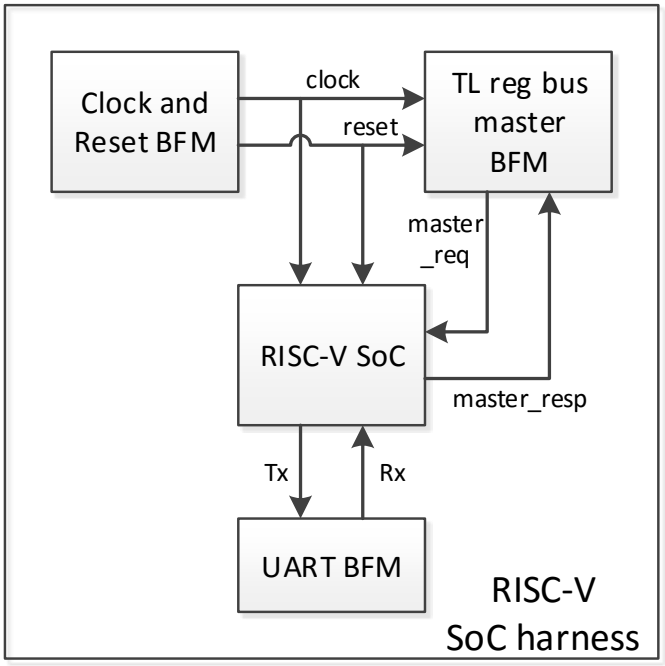

Figure 5.1: The block diagram of the test harness used for verification of the fault tolerant RISC-V softcore.

A BFM is a non-synthesizable model written in VHDL, which simulates the behavior of other components on a bus or any other functionality normally performed by other components on which the MUT relies. Each test case tests a part of the functionality of the MUT by issuing commands to the BFM to apply certain stimuli to the MUT and checking the results produced by the MUT. Based on the performed checks the test case

will return a pass or fail result to the test environment. If all test cases in the regression test pass, the regression test will also pass. Fig. 5.1 depicts a block diagram of the test harness for the fault tolerant RISC-V softcore containing three BFMs for the clock and reset, the `tl_reg_bus` and the UART connection.

For all test applications except the Dhrystone benchmark, a test case was made and included in the regression test. This regression test can be used for functional verification of the softcore in simulation and allows the designer to eliminate most bugs before starting on-board testing. A basic SoC test SCTB is also used for testing the fault injection architecture described in Section 3.8. This does not include all possible fault injection tests, but only one fault injection test on the address line of the instruction bus request in the softcore. During on-board testing the basic SoC test SCTB was also used to replicate failures of on-board test cases, as the ModelSim simulator provides more information for debugging than the on-board test environment.

5.1.3 On-board test environment

The on-board test environment will be used to perform functional and fault tolerance verification on an FPGA. A Xilinx Spartan-6 XC6SLX100 FPGA on the Trenz Electronic Gigabee TE0600 micro module will be used in the on-board test setup. The test execution on the softcore will be managed by the test controller running on a computer. There are three connections available between the computer and the Gigabee (depicted in Fig. 5.2):

- JTAG programming cable used to program the FPGA.
- An Ethernet connection for accessing the memories and register blocks connected to the `tl_reg_bus` in the SoC.
- A UART connection for sending application print output to the computer.

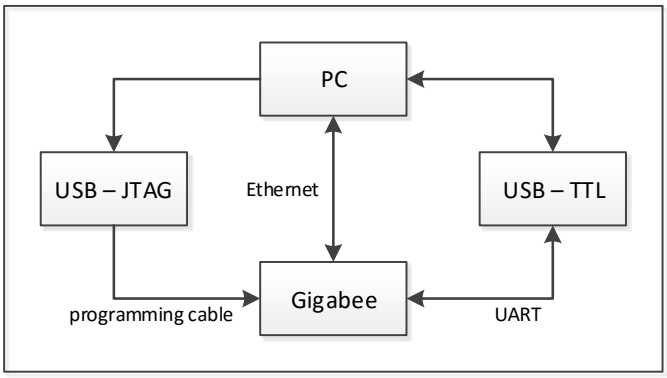

Figure 5.2: The on-board test environment.

Functional verification is performed in both test environments, as it is an important step in the verification process. The simulation-based functional verification is used to find most bugs, where the on-board functional verification is used to find any bugs not

found in simulation and other bugs which were introduced by the Xilinx toolchain. Fault tolerance verification is performed in two steps: first, the basic fault injection test case used in simulation is replicated on the FPGA after which all fault injection tests are executed. By performing the basic fault injection test case first, it is possible to verify if the results produced on-board match the results in simulation. If these results match, the fault injection architecture is working properly. Some specific bugs might remain undetected, but the basic functionality is as expected.

The test controller is implemented in Python and uses the PyART module of Technolution. The PyART module is used to perform automated verification of the MUT on the FPGA. The PyART module consists of the following objects:

- An initialization script: all required objects for a test.
- A test runner: searches the specified directory for test cases, expands the parameter lists in the test cases, and executes the test cases.
- A reporter: generates a report and an error log for each test run based on the information passed by the test runner.
- Multiple test cases: each test case tests certain functionality of the MUT by applying stimuli to the MUT and checking the results produced by the MUT.

Each test case should contain the following methods which are called by the test runner:

- `setup`: initialization function defining parameters lists, used for iteration expansion of the testcase, and other parameters.
- `startup`: initialization function called at the start of each iteration of the test case.
- `shutdown`: function called at the end of each iteration of the test case for cleanup after the test case run.
- `testx_y`: test function defining a test, where `x` is the three digit number of the test and `y` is the name of the test.

For each parameter in a parameters list, an iteration of the test case will be created by the test runner so that the test cases can be executed multiple times for different parameters. New test cases can be defined by inheriting the class `PyARTCase`, which defines these methods except the test methods. The new test cases should define the required tests to be executed and should overload the standard methods if the test case requires initialization or cleanup. Besides these standard methods, a test case class may also include other helper methods, which implement functionality used by all tests.

Each test case uses the `GigabeeFT` class for all interactions with the FGPA. `GigabeeFT` contains the `tl_reg_bus` Ethernet client object used for sending and receiving information to and from the SoC and several helper methods for:

- Clearing memories and register blocks.

- Writing applications to the instruction memory.
- Starting the softcore and wait until the softcore has finished execution.
- Reading (and if requested dump to log file) memories and register blocks. For register blocks, the data is returned in a dictionary.
- Generating register dictionaries with default values.
- Comparing register dictionaries.

The dictionary compare function is used to compare the read register values with the expected register values, which are generated by setting elements of a register dictionary with default values to the expected values. The `GigabeeFT` class uses an object of the `RiscVFtSocMapping` class, which can be used to access the memories and register blocks without knowing the absolute address of these components. The `RiscVFtSocMapping` class and all its subclasses are generated from the RDL source files by the RDL compiler.

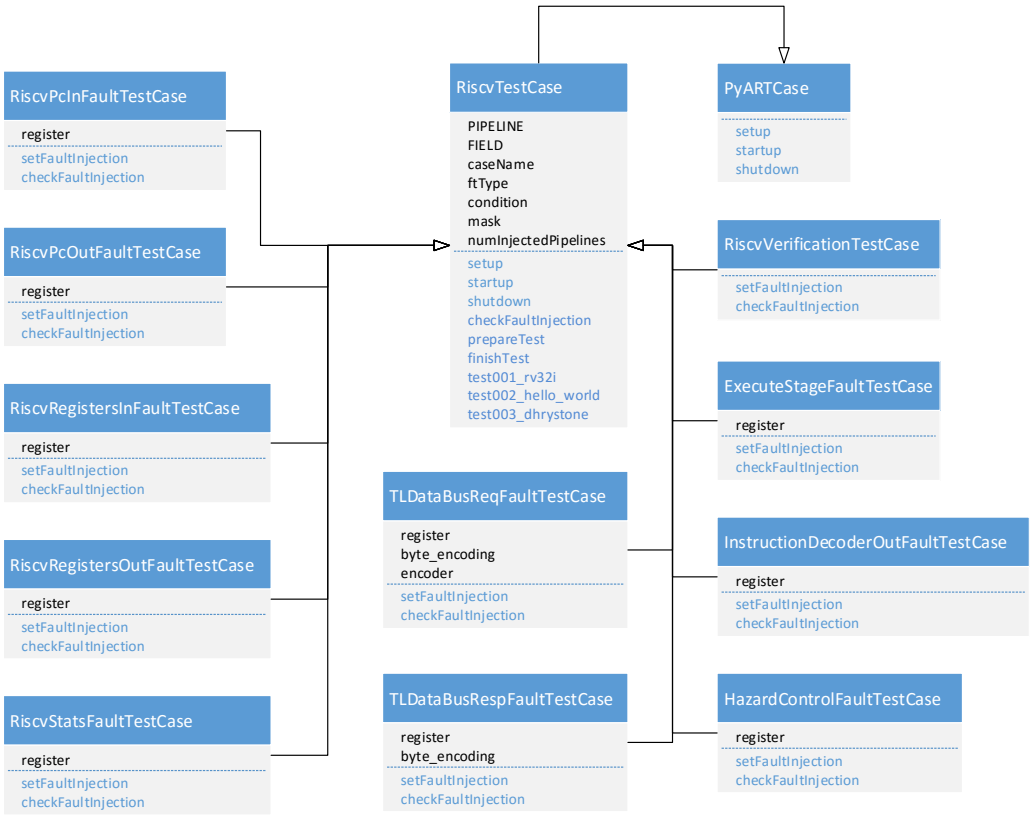

Figure 5.3: UML diagram of the test classes used by the on-board verification suite.

In Fig. 5.3 the UML diagram of the test classes used by the functional and fault tolerance verification suite is depicted. The `RiscvTestCase` implements the tests performed by every test case and all common methods for the different test classes like test case initialization and shutdown. Most subclasses of `RiscvTestCase` are fault injection test cases for record types on which fault injection should be performed according to the verification design in Section 3.8. Fault injection test cases will configure the fault injection through the saboteur registers and also indicate where the errors are expected to be detected/corrected. Test cases for record types are executed multiple times, one time for each record field, by adding a `PyARTParameterList` as attribute to the test case class. The parameter lists will be expanded by the PyART module before the test case is executed. If a fault injection test case is performed at the pipeline level, an additional parameters list is added to the test case, so every pipeline is tested as well. Besides the functional verification test case and the fault injection test cases for record types, there is one separate test case, `RiscvBaseFaultInjectionTestCase`, which performs basic fault injection testing on the instruction bus request for both single and double faults with the base SoC test application. In Table 5.1 the test cases included in the RISC-V on-board test suite are listed.

Table 5.1: List of all test cases from the on-board test environment

Basic verification			
	Faults	Case	Description
	0	100	Functional verification of the softcore
	1/2	200	Fault injection verification
Fault injection test cases			
Level	Faults	Case	Injection location
SoC	1	201	Instruction memory output
		202	Data memory output
Core	1	210	Instruction memory input
		211	Data memory input
		212	Program counter output
		213	Registers input
		214	Registers output
Pipeline interface	1	220	Program counter input
		221	Instruction bus request output
		222	Register majority voter input
		223	Data bus request output
		224	Statistics output
Pipeline	1	230	Execute stage output
		231	Decode stage output
		232	Hazard control output
SoC	2	301	Instruction memory output
		302	Data memory output
Core	2	310	Instruction memory input
		311	Data memory input

		312	Program counter output
		313	Registers input
		314	Register output
Pipeline interface	2	320	Program counter input
		321	Instruction bus request output
		322	Register majority voter input
		323	Data bus request output
		324	Statistics output

5.2 Verification results

In Section 5.1 the simulation-based and on-board test environments were described. In this section, the results of the verification suite from the on-board test environment will be presented. The results of the verification suite from the simulation-based environment will not be presented explicitly because all test cases in this suite passed before the verification from the on-board test environment was used. Furthermore, the benchmark results of the Dhrystone benchmark will be presented.

5.2.1 Verification suite

When presenting the results from verification suite, the terms fault, error, and failure will be used, which are defined in Section 2.4. When presenting the results, the errors will be categorized based on the scenarios presented in Section 2.4. In literature like [2] and [55], comparable definitions are used, so this should make it possible to compare the presented design with other designs in the future.

During fault injection, both single and double faults were injected. When a single error is detected, the error will be corrected, and the softcore continues execution and successfully completes the test application. In the case of a double error, the error cannot be corrected, and the the softcore will be stopped. The condition was set to inject one fault every four cycles (= 200 (ns)), which is a very high fault rate and can be considered as a stress condition. This stress condition might not be very representative for SEEs due to natural causes, but a disruptive fault attack might cause such a high fault rate. If the fault tolerant softcore can survive this high fault rate, it can be expected that it will also survive much lower fault rates. The Dhrystone benchmark used only 200 runs to reduce test execution times. For benchmarking the number of runs should be higher, as the Dhrystone should run for at least 20 seconds for the results to be reproducible.

5.2.1.1 Single fault injection

In Fig. 5.4 the results from the single fault injection test cases 200 till 224 are presented. While running these test cases about 32 million single faults were injected by the saboteurs, but no fault was able to cause a failure. Almost all errors were detected and corrected by either a decoder or a majority voter. Most of the faults were injected

Figure 5.4: The results from the single fault injection test cases 200-224

directly in front of a majority voter or decoder, so all resulting errors were detected immediately. The only exception to this are faults injected on the program counter input, where the error can be masked by the program counter logic.

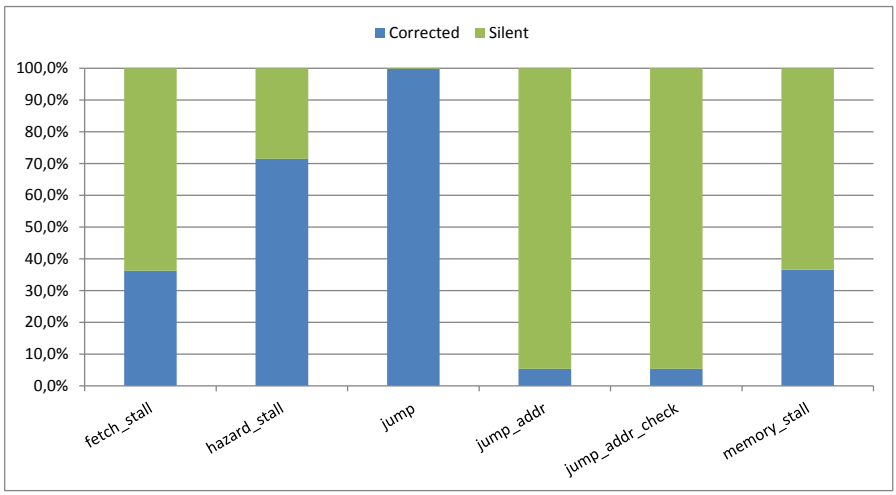

Figure 5.5: The error distribution for fault injection on the program counter input.

In Fig. 5.5 the fault injection results of the program counter input test case are presented. The percentage of silent errors depends on which input signal of the program counter the fault is injected. A fault on the jump signal almost never remains silent, because this signal determines if the jump address is forwarded to the majority voter by the multiplexer or not. A fault in the jump address on the other hand often remains silent, because it has to coincide with a jump for it to be detected or corrected by the majority voter.

5.2.1.2 Pipeline fault injection

The results from single fault injection in the pipeline are presented separately because the pipeline under injection is almost completely disabled due to the stress condition used. Faults inserted on the injection locations inside the pipeline are likely to cause multiple errors due to the importance of the signals being injected for the overall functionality of the softcore. In combination with the stress condition, this results in a massive amount of errors being corrected by the majority voters (a ratio of 32 between errors and faults was observed for the hazard control output test case). In the used fault injection architecture, it is not possible to trace every injected fault and determine which errors were caused by which faults. Furthermore, it is also not possible to measure the error manifestation latency, the latency between fault injection and error detection/correction, as only aggregated data is available after an execution run. Therefore, it is difficult to present percentages for detected, corrected and silent errors. Instead, the distribution of the corrected errors over the different majority voters can be presented, as this gives a clear insight into the impact of a fault in different signals.

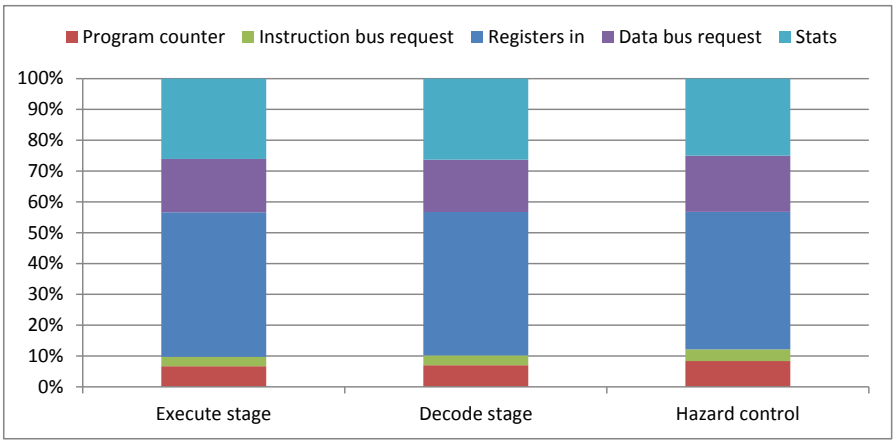

Figure 5.6: The error distribution for single fault injection inside the pipeline.

In Fig. 5.6 the distribution of errors for single fault injection inside the pipeline is presented. In this chart, the error distributions for all fields tested in a test case are combined into a single distribution. Although there are some minor differences between the different test cases, the overall trend is pretty much the same: the registers are the most likely place where errors will occur when faults are injected into the pipeline. This is not surprising, as the majority of instructions use only register operands. The data bus is also influenced by faults inside the pipeline, as the address and data fields in the data bus request are provided by the execute stage and the read and write flags are provided by the decode stage. The program counter and the instruction bus are not very susceptible to the faults injected into the pipeline during verification, as only around 10% of the errors are detected here. In the case of the program counter errors remaining silent are also influencing its share in the error distribution. The stats category in the error distribution is an output which contains several flags used inside the pipeline like

hazard stall, fetch stall, memory stall, and jump. This output can be used for analysis of the program execution flow, but also provides some insight into the effects of faults inside the pipeline, as these flags indicate the behavior of the pipeline. Lastly, it is important to note, that the error distribution presented here, is dependent on the test applications and fault injection locations used. Other test applications and fault injection locations might result in another distribution of the errors over the majority voters.

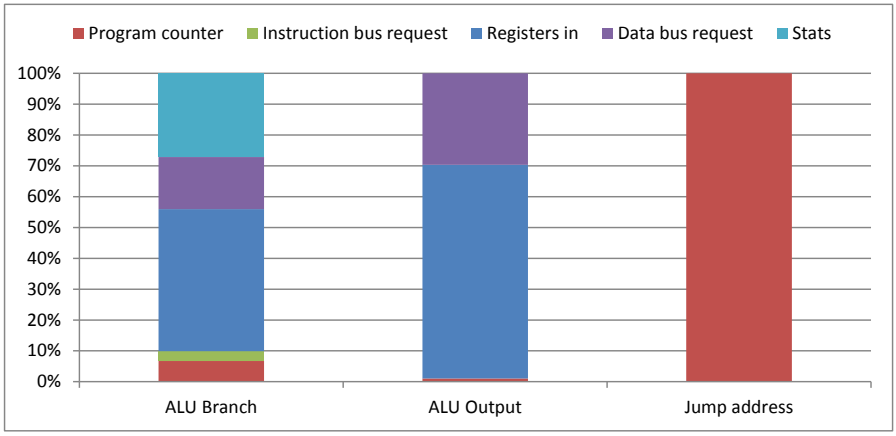

Figure 5.7: Distribution of errors for single fault injection on the Execute Stage output for each signal.

Fig. 5.7 presents the error distribution for fault injection in output signals of the execute stage. This distribution gives more insight into the effects of faults in different signals. Faults in the ALU output will cause errors on the register data input, as most ALU results are stored in the register file, and on the data bus address and byte enable signal. For load and store instructions the address is calculated by the ALU, so 5% of the errors are detected at the address and address check word each. Furthermore, 13% of the errors is detected at the byte enable field of the data bus request. One would expect the share of the byte enable field to be the same as for the address, but the address is only placed on the data bus by a multiplexer when a request has to be performed. For the byte enable such a multiplexer is not used, causing a larger share of errors being corrected in the byte enable signal. As expected, faults in the jump address are only detected and corrected by the program counter, as this is the only component using this address. Faults on the ALU branch signal are spread over different majority voters and show the same trend as identified in Fig. 5.6.

Fig. 5.8 presents the error distribution for fault injection into output signals of the decode stage. Only the output of the instruction decoder was targeted in this test case, as this decoder provides all control signals for the consecutive stages. The fields of the instruction decoder output mentioned are:

- Funct3: second operation selector provided by the instruction
- DmemWrite: flag indicating write access to memory
- DmemRead: flag indicating load access to memory

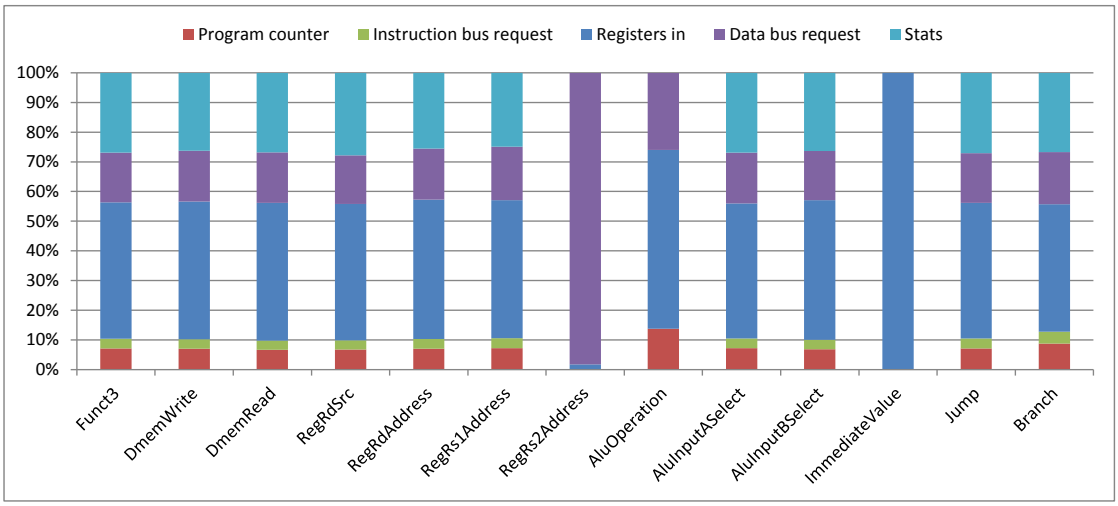

Figure 5.8: The error distribution for single fault injection on the Decode Stage output for each signal.

- RegRdSrc: selector signal for the value written back to register file
- RegRdAddress: address of the destination register
- RegRs1Address: address of the first source register
- RegRs2Address: address of the second source register
- AluOperation: type of operation performed by the ALU
- AluInputASelect: input selector signal for the first input of the ALU
- AluInputBSelect: input selector signal for the second input of the ALU
- ImmediateValue: immediate value provided by instruction
- Jump: flag for selection of jump address send to program counter (ALU output or jump address calculated separately)
- Branch: flag indicating if a branch instruction is performed or not

Most fields show the same pattern identified in Fig. 5.6, the only exceptions being Funct3, RegRdSrc and ImmediateValue. In the RISC-V instruction format, the Funct3 field is used as the second indicator for the operation type. If a fault occurred in the Funct3 field before the instruction decoder, this would have much larger consequences. After the decode stage, the Funct3 field is only used by the data maskers in the memory stage and writeback. These maskers make half word, and byte accesses possible, so these errors are mostly identified at the data bus majority voter and for a small amount at the registers. The registers' share is much smaller because most faults in the Funct3 field are masked by the destination register data multiplexer, which only selects the erroneous data from the masker for load instructions. The RegRdSrc field is the selector used by the destination register data multiplexer, so faults injected on this field are only detected at the registers input majority voter. The ImmediateValue contains the immediate value generated by the instruction decoder based on the immediate value provided by the instruction. This value is used for calculating the jump address or address offsets and as input for arithmetic operations. Therefore, faults in this value are mostly detected at

the registers, but also at the data bus majority voter and the program counter.

5.2.1.3 Double fault injection

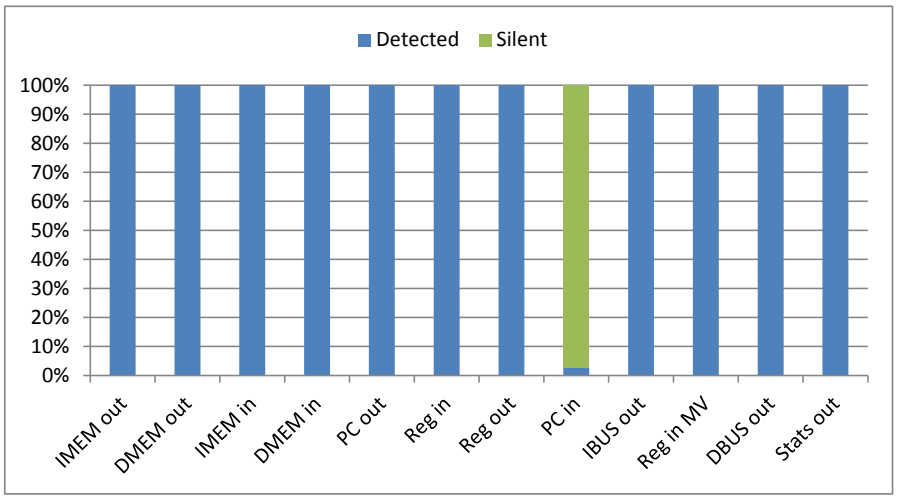

Figure 5.9: The results of the double fault injection test cases 200-224

In Fig. 5.9 the results from the double fault injection test cases 300 till 324 are presented. While running these test cases around 48,000 double faults were injected by the saboteurs, but no fault was able to cause a failure. The number of faults injected for double faults is much lower than for single fault injection because for most tests only one fault is injected. As soon as this fault is detected the softcore is stopped, so no more faults can be injected. Again the only exception to this is case 220 which injects faults on the program counter input.

Because faults can remain silent in the program counter, the condition needed to be increased to one fault every two cycles. With a lower condition all injected faults could remain silent, which in principal is not much of a problem if this only happens in a few test cases. But due to implementation details of the verification framework, it was decided to increase the condition so in all test cases a double error would be detected. Again, there is a difference between the injected signals. For most input signals, the double error is detected immediately after injection, but for the jump address, it can take a while before a double error is detected. Therefore, a lot more faults need to be injected into these signals, which causes the high number of silent errors in the accumulated results for double fault injection on the program counter input.

5.2.1.4 Result overview

In Table 5.2 the full results (pass/fail) of the verification suite are listed. In this table, the number of saboteurs activated for each test case and the maximum number of detectors triggered is also listed. If a test passes, no failures were detected and the number of injected faults and detected/corrected errors are as expected.

Table 5.2: The full results of the verification suite along with the number of saboteurs activated per test, S, and the maximum number of detectors triggered per test case, D.

Case	Case name	S	D	Result
100	Functional	0	0	pass
200	Fault injection	1	1	pass
201	IMEM out	1	4	pass
202	DMEM out	1	4	pass
210	IMEM in	1	1	pass
211	DMEM in	1	1	pass
212	PC out	1	4	pass
213	Reg in	1	1	pass
214	Reg out	1	4	pass
220	PC in	1	1/2	pass
221	IBUS out	1	1	pass
222	Reg in MV	1	1	pass
223	DBUS out	1	1	pass
224	Stats out	1	1	pass
230	Execute	1	31	pass
231	Decode	1	31	pass
232	Hazard	1	31	pass

Case	Case name	S	D	Result
301	IMEM out	1	4	pass
302	DMEM out	1	4	pass
310	IMEM in	1	1	pass
311	DMEM in	1	1	pass
312	PC out	1	4	pass
313	Reg in	1	1	pass
314	Reg out	1	4	pass
320	PC in	2	1/2	pass
321	IBUS out	2	1	pass
322	Reg in MV	2	1	pass
323	DBUS out	2	1	pass
324	Stats out	2	1	pass

Although no SPOFs were found during verification, there could be SPOFs remaining which were not found by the current verification suite. During design and implementation, two SPOFs were noticed, but not solved as these SPOFs were not covered by the problem statement.

One of the objectives of this thesis was to prevent any fault in the pipeline or in data or address words from causing a failure. This was achieved by using NMR in the pipeline and ECC in memory elements, but there are also control signals connecting the pipeline with the memories and the register file. These control signals are not protected, so a SET on the interconnect used by these signals could cause a failure. The risk could be reduced by replicating the control signals and applying a majority vote as close as possible to the location where it is used. This does however not guarantee that the correct operation will be executed because a fault could still occur after the majority vote. Therefore, other techniques for checking the type of operation performed should be applied.

Besides the control signal SPOF, another SPOF is still remaining in the instruction and data memory. Due to the implementation of the `tl_data_bus`, the address passed to the memories are not absolute, but relative to the base address of the memory on the bus. The check word is calculated for the absolute address, so it will not match with the relative address and cause the decoder to fail. To solve this, the address offset is added the relative address before decoding in the memory and subtracted after decoding. If a fault is introduced in the addition, the decoder could detect this if the total number of faults is still lower than the maximum detection capacity. But if a fault is introduced

in the subtraction, it will not be detected and cause a failure. Implementations using another bus protocol (or caches) might not have to deal with this problem, so therefore no structural solution was implemented for this problem.

The coverage of the current verification suite and thus its ability to find other unnoticed SPOFs could be improved by adding additional test applications to it, which can improve the overall coverage by either exercising all instructions or by exercising a particular instruction which is not already exercised enough by the verification suite. The fault injection coverage could also be improved by adding a random fault injection method to the verification suite, which can inject faults into locations not already covered by the saboteurs.

5.3 Benchmark

In this section the benchmark results for both the Dhrystone and CoreMark benchmarks will be discussed.

5.3.1 Dhrystone

Table 5.3: The Dhrystone benchmark results for the three different cores.

SoC	f (MHz)	Time (s)	CPI	Dhrystones/s	DMIPS/MHz
STD	83.3	24.2	2.23	83084	0.57
FT	35.7	56.2	2.23	35607	0.57
FI	20.0	100.3	2.23	19940	0.57

In Table 5.3 the results from running the Dhrystone benchmark on three different versions of the softcore are presented. The three versions are:

- **Standard (STD) softcore:** the baseline softcore introduced in Section 3.1
- **Fault Tolerant (FT) softcore:** the fault tolerant softcore using the hybrid design described in Section 3.5 with SECDED error capabilities
- **Fault Injection (FI) softcore:** the fault injection softcore using the hybrid design with SECDED error capabilities and the verification design described in Section 3.8

The Dhrystone benchmark was executed for 2,000,000 runs so that the execution time would be at least 20 seconds. The results show that the performance of the benchmark per MHz of clock frequency did not decrease due to the addition of fault tolerance. This performance number is largely based on the handling of data and control hazards by the softcore. This behavior was not changed by the addition of fault tolerance. But as the clock frequency decreased the execution time increased, so the number of Dhrystones executed per second also decreased significantly.

The CPI of 2.24 is relatively high for a softcore, as for each data or control hazard three pipeline stall are induced. The cost of data hazards could be reduced by implementing register forwarding, where the result of a calculation is forwarded to the next instruction using it, without first committing it to the register file. The cost of control hazards could be reduced by predicting the outcome of a branch operation and continue with fetching instructions based on the prediction. If the prediction is incorrect, several stages of the pipeline need to be flushed, but if the prediction rate is sufficient, this will result in a lower CPI. Although these solutions can significantly improve the core performance, the implementation of these solutions in the fault tolerant core should be done carefully, so no new SPOFs are introduced. Keeping the pipelines in sync might be a challenge when using these techniques.

5.3.2 CoreMark

Table 5.4: The parameters used for generating the CoreMark results.

Parameter	Value
Version	1.0
Compiler	GCC 5.2.0
Compiler flags	-m32 -marc=RV32I -O2
Data allocation	STACK
Seeds	0x0, 0x0, 0x66
Size	2000
Instruction memory size	64 kB
Data memory size	64 kB

Table 5.5: The CoreMark benchmark results for two cores.

SoC	f (MHz)	CPI	CoreMark	CoreMark/MHz
STD	62.5	2.46	34.29	0.55
FT	25.0	2.46	13.72	0.55

In Table 5.4 the settings used for running the CoreMark benchmark are listed. In Table 5.5 the results from running the CoreMark benchmark on the STD and FT SoC are presented. CoreMark was not run on the FI SoC as the Dhrystone benchmark showed that this SoC has roughly the same characteristics as the FT SoC when it comes to software execution.

As expected, the CoreMark can perform less iterations per second on the FT SoC than on the STD SoC, because of the lower clock frequency. The CPI is not affected by the added fault tolerance but is slightly higher than for the Dhrystone. This is caused by the higher number of data and control hazards induced by CoreMark. For lowering the CPI the same solutions as mentioned in Section 5.3.1 could be applied.

5.4 Resource utilization

Besides the results from the on-board verification suite, the on-board test environment also provided the resource utilization for the hybrid design, which will be presented in this section. These results will be presented for three different version of the softcore mentioned in Section 5.3. The parameters used to retrieve the results for theses softcores are listed in Table E.2. Besides the parameters for the SoCs, additional settings for the Xilinx ISE tooling were used, which are listed in Table E.3.

Table 5.6: The absolute resource utilization of the RISC-V SoCs after synthesis.

	Absolute RU		
Resource	STD	FT	FI
Slice Registers	4546	13223	29508
Slice LUTs	5844	21510	45123
block RAM	19	27	27

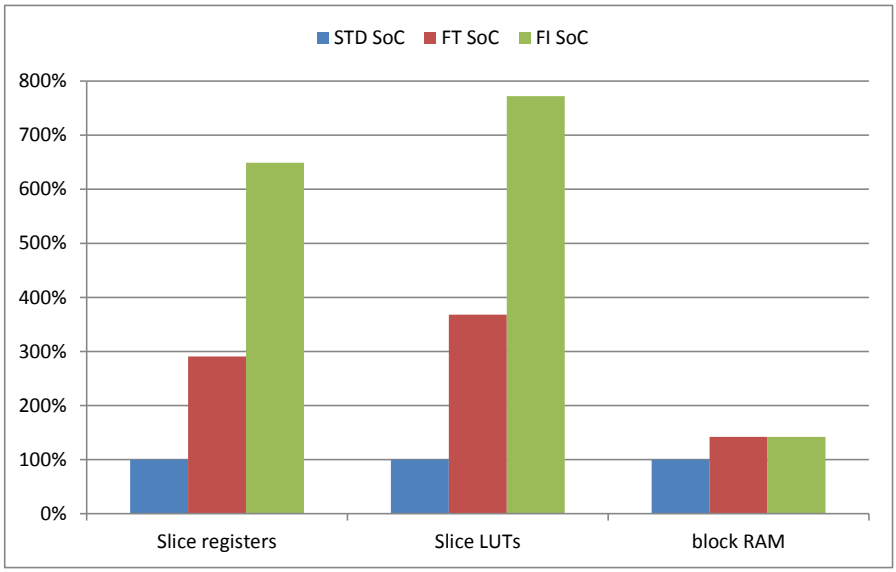

Figure 5.10: The relative resource utilization of the of the RISC-V SoCs after synthesis.

In Table 5.6 and Fig. 5.10 the absolute and relative resource utilization of the baseline, fault tolerant and fault injection RISC-V SoC after synthesis are presented. The FT SoC uses considerably more registers and LUTs than the STD Soc, which is as expected. The pipeline uses a lot of registers and LUTs and is replicated four times, so this accounts for this large increase. The block RAM increase is very limited because of the application of ECC instead of NMR for the memories. As FPGAs have a limited amount of BRAMs available, which are also distributed over several columns in the FPGA, this usage should be kept as low as possible. Overall, the resource utilization increase for the FT SoC

remains below 400%, so it remains below target.

The FI SoC uses about two times more registers and LUTs than the FT SoC. In this core, a lot of saboteurs are added to the core, which each contain some logic implemented in LUTs and a 16-bit counter using LUTs and registers. Furthermore, the large saboteur register block is added to the SoC, which consumes a lot of resources because of the high number of registers required for controlling the saboteurs. Overall, the fault injection core uses a lot more resources, but as this version will only be used for verification of the fault tolerant design, this is not so much of a problem. In applications, only the baseline or the fault tolerant SoC will be used, depending on the requirements for fault tolerance.

Table 5.7: The resource utilization of the RISC-V SoCs after mapping.

	Absolute RU		
Resource	STD	FT	FI
Slice Registers	4534	13145	23496
Slice LUTs	5155	19487	42059
Slices	1894	6459	12231
RAM16BWER	17	23	23
RAM8BWER	3	7	7

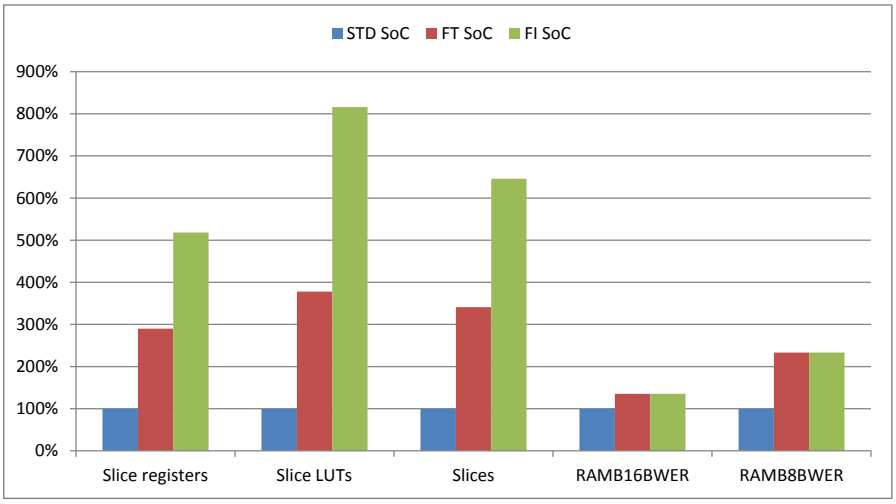

Figure 5.11: The relative resource utilization of the RISC-V SoCs after mapping.

In Table 5.7 and Fig. 5.11 the absolute and relative resource utilization of the three RISC-V SoCs after mapping are presented. These results show the same trends identified in the resource utilization after synthesis, which also applies for the slices. The increased usage of RAMB16BWER and RAMB8BWER is caused by the addition of ECC check words to the data words stored in the instruction and data memories. RAMB16BWER and RAMB8BWER are the block RAM library primitives of the Spartan-6, which are used as the basic building blocks for all block RAM configurations [61]. Some of the

check words can be stored in RAMB16BWER blocks, which are also used to store the data words. For storing the check words for the bytes in the data words in the data memory, the mapper also uses RAMB8BWER blocks. As these blocks are not used in the STD SoC, the increase of the utilization of these blocks is also larger.

5.5 Timing results

During the design phase, several long paths could already be identified based on the critical paths of the baseline design. The design cost estimate also showed that the expected clock frequency was much lower than the baseline design due to these long paths. The long paths identified during design are marked in red in the diagram in Fig. 5.12 as paths one till four.

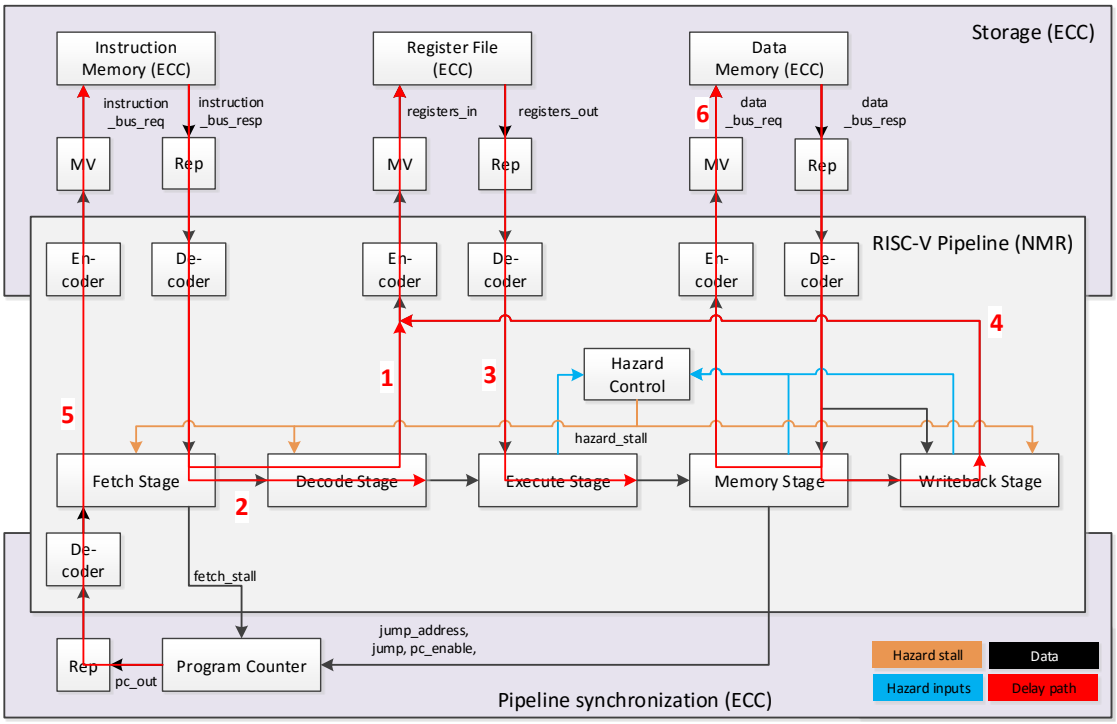

Figure 5.12: The long delay paths in the fault tolerant hybrid RISC-V core identified during design.

The length of the first two paths is only increased by the addition of an extra decoder, so these paths are likely to double. The length of the last two paths is increased much more dramatically, as one majority voter, one encoder, and two decoders are added to the path. These paths could be quadrupled by the addition of these components according to the design cost estimates for these components in Chapter 4. The actual path length might be lower because the tooling might be able to reduce the delay path of the logic. Nevertheless, it is not a very good indication when considering the clock frequency target of 50%.

For the implemented SoC design, three timing constraint were implemented: two for the Ethernet converter and one for the SoC. The SoC timing constraint is listed in Listing 5.1.

Listing 5.1: The SoC clock timing constraint

```
NET "CLOCK_125M" TNM_NET = "CLOCK_125M";
TIMESPEC TS_CLOCK_125M =
        PERIOD "CLOCK_125M" 125 MHz HIGH 50 % PRIORITY 1;
```

This timing constraint specifies an input clock of 125 MHz with a 50% duty cycle, for which the tooling should do a high effort to achieve it. The constraints for the Ethernet core have priorities 1 and 100, where a lower number means a higher effort. The input clock of 125 MHz is converted to the desired clock frequency by the Phase Locked Loop (PLL). The tooling will therefore also generate a new timing constraint for the SoC, based on the division and multiplication parameters set for the `PLL_ADV` module. It is important to note, that for this design no timing constraints were implemented for any specific component. One of the thesis objectives was to implement the design as a proof of concept on an FPGA and if possible optimize its performance without making the design platform dependent. Timing constraints are often platform dependent optimizations, so it was not an objective of this thesis to implement these.

Table 5.8: The timing results for the RISC-V softcores on the Xilinx Spartan-6.

	STD	FT	FI
t_{period} (ns)	11.6	27.7	48.8
f_{max} (MHz)	86.0	36.1	20.5
f_{max} comparison		42%	24%

Table 5.8 presents the timing results of the three different SoCs on the Xilinx Spartan-6. As expected from the design phase, the fault tolerant SoC runs at a lower frequency than the standard core. Furthermore, the target of the clock frequency design criterion was not met. This is largely due to the addition of encoders, decoders and majority voters to the critical paths of the design. By optimizing these components the clock frequency could be brought closer to the intended target. Furthermore, the critical paths can be split by the addition of extra stages to the pipeline if the component delay path remains too long. Paths one and two could be split by adding a stage between the Fetch and Decode stage for decoding the instruction bus response. Path three could be split by inserting a stage between the Decode and Execute stage for decoding the supplied register data. Path four could be split by inserting a stage between the Memory stage and the Write back stage for decoding the data bus response. This will, however, result in a lower CPI, as more stages have to be stalled for data hazards or flushed for control hazards.

During timing analysis additional long paths were found in the design, two of which are marked in the diagram in Fig. 5.12 as paths five and six. The other paths included SoC components, so they could not be marked in the diagram, but are listed in Table 5.9.

Table 5.9: Some additional long paths identified during the timing analysis.

	PC out ⇒ Reg block	DMEM ⇒ Reg block
1	Program counter FF	Data memory output
2	Program counter out decoder	Data bus response decoder
3	Instruction bus request encoder	Write back stage multiplexer
4	Instruction bus request voter	Registers in encoder
5	Instruction memory decoder	Registers in majority voter
6	Control register block	Registers in decoder
7		Control register block
	Reg bridge ⇒ Reg file	**Reg bridge ⇒ DMEM**
1	Reg2data bridge	Reg2data bridge
2	Data bus response encoder	Data bus response encoder
3	Data bus response decoder	Data bus response decoder
4	Data memory masker	Memory stage interface logic
5	Write back stage multiplexer	Data bus request voter
6	Registers in encoder	Data memory decoder
7	Registers in majority voter	Data memory input
8	Registers input decoder	
9	Register file input	

In this list two paths from the core to the register block are mentioned: the error and saboteur flags connecting the detectors and saboteurs to the counters in the register block. Furthermore, the error flags are also used to check for the detection of uncorrectable errors in which case the core should be stopped. The counters do not have to be updated in the same cycle, and if the core is stopped one cycle later, this should not be much of a problem. Therefore, a FF bank was placed between the core and the register block to delay these flags by one cycle so that this path can be reduced. The Reg2Data bridge to Data Memory path is also shortened by placing a FF bank after the Reg2Data bridge. This will induce an extra memory stall for each access to the SoC peripherals, but accesses to peripherals are rare in the current verification suite. If the core regularly accesses the SoC peripherals, it might be better to remove the FF bank especially considering that this insertion only reduces the path by a few nanoseconds.

5.6 Comparison with Design Estimates

In Chapter 3 the design cost estimates were made for the baseline softcore and the hybrid design. These design estimates only included the pipeline, so they cannot be compared with the resource utilizations presented in Section 5.4. Xilinx ISE provides the Module Level Utilization (MLU) view, which shows the resource utilization per component. From this view, the resource utilization of the core of the fault tolerant core, which is equivalent to the pipeline of the baseline, can be extracted. In Table 5.10 the design cost estimates and the resource utilization extracted from the MLU are presented. Table 5.10 also

Table 5.10: Absolute comparison of the cost estimates with the realization retrieved from the Module Level Utilization (MLU) view.

	Estimate		Realization		
	Baseline	Hybrid	STD	FT	FI
Lookup Tables	1732	9391	1353	8059	16415
Registers	835	3826	899	3329	7353
Slices			696	4437	8262
f_{max} (MHz)	87.7	38.6	86	36.1	20.5

includes the clock frequencies from the design cost estimates and the timing analysis reported in Section 5.5.

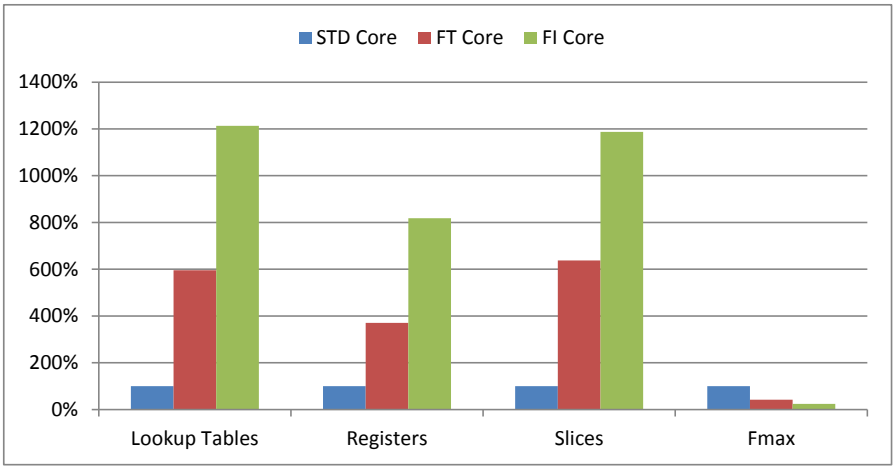

Figure 5.13: Relative comparison of the RISC-V cores retrieved from the Module Level Utilization (MLU) view with the STD core set at 100%.

In Fig. 5.13 the three RISC-V cores are compared against each other, where the STD core is set to 100%. The target for the resource utilization criterion is not met for both the LUTs and the slices. The utilization for both resource types is about 150% above the target. The target was set to the resource utilization expected for a standard NMR solution using SECDED which includes a four time replication plus a majority voter. The utilization above target is largely due to the addition of encoder and decoders to the pipeline and thus replicating these as well. If the encoders and decoders would not have been replicated, the utilization could be lowered, but this would introduce SPOFs on the domain crossings. With the current design, the resource utilization could be lowered by using more efficient encoders and decoders. However, it is not likely that the design target will be met by this improvement, as the extra resource utilization is inherent to the design.

In Fig. 5.14 the three RISC-V cores are compared with the design cost estimate of the hybrid design. As the synthesis tool does not report the slice utilization, the slices

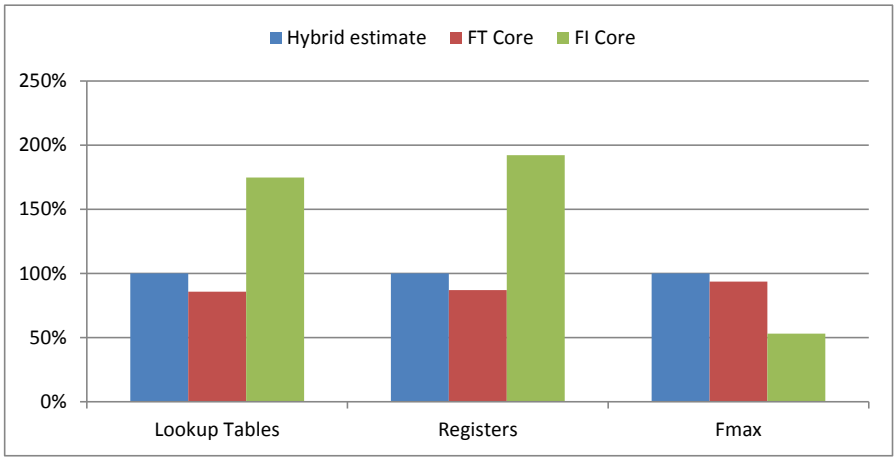

Figure 5.14: Relative comparison of the FT and FT RISC-V cores retrieved from the Module Level Utilization (MLU) view with the hybrid design cost estimate.

are not mentioned in this comparison. This comparison shows that the FT core uses less LUTs and Registers than estimated. This proves that the resource utilization estimates were accurate, and no unexpected utilization of resources was introduced during the implementation of the design. The FI core uses more resources than the estimates, as this core includes a large number of saboteurs which were not included in the design cost estimate. Although the resource utilization estimates are accurate, the clock frequency was lower than estimated. Estimating the clock frequency of a design is much more difficult than estimating the resource usage because there might be unexpected critical paths in the design which were not anticipated. During the timing analysis, several of such unanticipated paths were also identified and if possible solved. Although the clock frequency is lower than estimated, there are still possibilities left for improving the clock frequency, which will be listed in Section 6.3.

5.7 Design Criteria Evaluation

In Table 5.11 the design criteria are listed with their targets and what was achieved for these criteria.

In Section 5.2 it was proven that the hybrid design met the design criteria:

- **ED:** the FT SoC passed all double fault injection test cases.
- **EC:** the FT SoC passed all single fault injection test cases.
- **CPI:** the CPI FT SoC is the same as for the STD SoC.
- **SPOF:** the fault injection test cases all passed without failures, so no SPOFs were discovered in the targeted areas.
- **RISC-V:** the FT SoC passed the unit tests, provided by the RISC-V community, which test specification compliance.

Table 5.11: List of design criteria along with their intended targets.

Design criterion		Target	Achieved
Error Detection	ED	2	2
Error Correction	EC	1	1
Clock frequency	CLK	50%	42%
Resource Utilization	RU	4.5x	6.5x
Cycles Per Instruction	CPI	20%	0%
Software transparency	ST	Minimal	yes
Complexity	COM	= baseline	yes
Single Points of Failure	SPOF	0	yes
RISC-V compliance	RISC-V	RV32I	yes

In Sections 5.4 and 5.5 it was shown that the hybrid design did not meet the design criteria:

- **CLK:** the clock frequency is lower, due to long paths inherent to the design.
- **RU:** the resource utilization is higher, due to the replication of encoders and decoders for avoiding SPOFs.

In this chapter no prove was delivered that the design met the COM and ST design criteria. During the design phase described in Chapter 3 it was already shown that the hybrid design had a lower complexity than the ECC-based design, so therefore this criterion can be considered to be met. No modifications to the software were required to run applications on the FT SoC on the FPGA. For simulation of the FT SoC, an extra tool was required, which generated the Hamming check words for the data stored in the instruction and data memory for each application, which was implemented in a small Python script. Therefore, we can also list the ST design criterion as achieved.

5.8 Conclusion

In this chapter, the test architecture consisting of a set of test applications (called the verification suite), a simulation-based test environment using ModelSim, and an on-board test environment using a Spartan-6 FPGA were described. Both test environments are used to test the basic functionality of the design. The on-board test environment is also used to perform fault injection tests, which are controlled by a Python application running on a separate PC. During fault injection testing no failures were detected so that the design can mitigate all injected faults. In some test cases, some errors remained silent, due to hiding by the logic. Faults injected into the pipeline caused large numbers of errors on the outputs of the pipeline, most of which were detected/corrected by the registers in and data bus request majority voter.

The results of implementing the design on the FPGA were also presented and analyzed in this chapter. For the fault tolerant SoC the resource utilization target was

met, but for the fault tolerant core the targets were not met due to the replication of encoders and decoders. Furthermore, the clock frequency target was also not met, due to the long paths inherent to the design. Some of these paths were already identified during design, but some additional paths were also found during the timing analysis. The resource utilization and clock frequency targets were the only targets not met by the hybrid design.

6 Conclusion

In this chapter conclusions will be drawn based on the work presented in this thesis. First, the work in this thesis will be summarized in Section 6.1. Then, the main contributions will be presented. The problem statement and objectives of this thesis will also be re-examined to check if they were met by the work in this thesis. Lastly, the future work will be presented in Section 6.3.

6.1 Summary

In Chapter 2 the open-source RISC-V ISA used by the softcore made by Technolution B.V. was introduced. SEEs can be introduced by both natural causes due to the interaction between particles and human-induced causes, which are targeted at disrupting system functionality or retrieval of secret information. The effects of SEEs can be mitigated by adding redundancy to a system. Different forms of redundancy can be divided into four categories: hardware, information, time, and software.

Based on the information found in literature, five implementation options for adding fault tolerance to the pipeline were identified: NMR, lockstep, ECC, the watchdog and time redundancy. For adding fault tolerance to memory elements, six implementation options were identified in literature: NMR, Hamming codes, Hsiao codes, CRC codes, BCH codes and Residue codes.

For verification of the added fault tolerance, fault injection can be used. By injecting faults, the behavior of the system in the presence of faults can be observed and verified. Fault injection can be hardware-based, software-based or simulation-based. A saboteur or a mutant can be used for simulation-based fault injection.

In Chapter 3 the baseline design of the RISC-V softcore, implementing the RV32I instruction set and a classic five-stage RISC pipeline, was presented. The presented fault model defines at which locations in the softcore and at which moments SEUs and SETs can occur in softcore. The set of design criteria used for evaluation of the design options and the fault tolerant designs include error detection/correction capabilities, resource utilization, and complexity amongst others.

Two fault tolerant designs were made, both using NMR and ECC. The ECC-based design uses ECC for protection whenever possible, both in the pipeline and in memory elements, in an effort to save resources. Only for combinatorial elements which ECC cannot protect NMR is used. The low-level switches between the ECC and NMR domain result in a significant number of SPOFs and a high complexity of the ECC-based design. The hybrid design uses NMR to protect the pipelines and ECC to protect memory elements. This design has a limited amount of switches between protection domains and by letting the domains overlap, possible SPOFs are avoided.

After evaluation of both designs for all the design criteria, the hybrid design was chosen for implementation, because of its better performance for all design criteria except resource utilization. The redundancy of the design will be tested by using fault injection through saboteurs, which will be inserted into the design. In simulation, the basic fault injection architecture will be tested and on the FPGA a more comprehensive test suite will be used to test the fault tolerance.

In Chapter 4 the implementation details of the hybrid design were presented. The specification of the used Hamming code was presented along with a list of all the signals which will be protected with ECC. Furthermore, the design, unit test and cost estimates for the basic building blocks, the Hamming encoder, and decoder, and the majority voter were presented. The cost estimates showed that the latency of these basic building blocks are very high compared to the latency of the different pipeline stages. This is one of the main causes why the clock frequency of the fault tolerant design is reduced. Finally, the fault tolerant RISC-V SoC and the memory mapping used for testing were presented. The SoC contains a register block for controlling the softcore, a saboteur register block used for controlling and monitoring the saboteurs, and error counters used for monitoring the checking circuitry. The data from these register blocks can be used to check if all inserted faults were detected during fault injection verification.

In Chapter 5 the test architecture consisting of a set of test applications (called the verification suite), a simulation-based test environment using ModelSim, and an on-board test environment using a Spartan-6 FPGA were described. Both test environments are used to test the basic functionality of the design. The on-board test environment is also used to perform fault injection tests, which are controlled by a Python application running on a separate PC. During fault injection testing no failures were detected so that the design can mitigate all injected faults. In some test cases, some errors remained silent, due to hiding by the logic. Faults injected into the pipeline caused large numbers of errors on the outputs of the pipeline, most of which were detected/corrected by the registers in and data bus request majority voter.

The results of implementing the design on the FPGA were also presented and analyzed in this chapter. For the fault tolerant SoC the resource utilization target was met, but for the fault tolerant core the targets were not met due to the replication of encoders and decoders. Furthermore, the clock frequency target was also not met, due to the long paths inherent to the design. Some of these paths were already identified during design, but some additional paths were also found during the timing analysis. The resource utilization and clock frequency targets were the only targets not met by the hybrid design.

6.2 Main contributions

In Section 1.1 the problem statement for this thesis was formulated as:

> *How can an RISC-V softcore using a classic five-stage RISC pipeline be extended such that a maximum of two concurrent Single Event Effects in the pipeline or in data or address words cannot influence the correct execution flow of the processor without being detected in a modular and efficient way?*

Based on the evaluation of the design criteria in Section 5.7 we can conclude that the hybrid design proposed in Section 3.5 and implemented as a proof of concept can indeed mitigate the effects of one or two (concurrent) Single Event Effects in the pipeline, data words, or address words. During testing, none of the injected errors was able to cause a failure for the used test applications. The design allowed for a modular implementation which reused as much components from the baseline as possible. Only a few additional components, like the Hamming encoder and decoder and the majority voters, needed to be added to the design, but especially the Hamming encoders and decoders can be easily replaced by other encoders and decoders using a different ECC technique. It was tried to make the implementation as efficient as possible without losing platform independence out of sight, but the proof of concept failed to meet the resource utilization target. This was due to measures taken to reduce the number of Single Points of Failure (SPOFs), which resulted in a high number of encoders and decoders in the design. Another point where the design failed in terms of efficiency is the clock frequency of the design, which was also below the target due to long paths through decoders and voters. Possible solutions for these problems will be presented in Section 6.3.

In Section 1.1 several thesis objectives were proposed, which were also met by the work of this thesis. As mentioned before, the hybrid design is a modular design which can detect and correct errors, where the basic building blocks can be replaced easily. This design was implemented on the Xilinx Spartan-6, as described in Chapter 4, where platform independence was taken into account. Therefore, all basic components were implemented with behavioral descriptions and structural descriptions using no platform specific building blocks, except for some peripherals like the PLL in the top level entity containing the SoC. The efficiency of the implemented design did not meet the targets, but possible solutions to improve the efficiency of the design will be presented in Section 6.3. Besides the functional design, a verification design was also made to verify the functionality of the design. This verification design included several test applications and a fault injection method used to verify the fault tolerance of the design. No failures were detected during verification, which proves the correctness of the design. However, the verification design is always limited by the quality and coverage of the test method. In order to improve the confidence in the design, the verification design should be improved for which possible solutions will be presented in Section 6.3.

The main contributions of this thesis can be summarized as:

1. Created a design concept for a fault tolerant RISC-V processor using a classic five-stage RISC pipeline which can detect and correct a number of errors depending on the configuration.

2. Created an implementation of the design concept, which can detect two concurrent errors and correct a single error, thus proving the feasibility of the design concept.

3. Created a verification design for the design concept, which proves the correctness of the design and which can be extended to improve the confidence of the design.

4. Provided possible solutions for improving the proof of concept and the design concept itself.

6.3 Future work

One of the thesis objectives is to present possible solutions for improving the design concept and its proof of concept. In this section, these improvements will be presented. The improvements can be divided into four categories: resource utilization, clock frequency, fault tolerance, and verification improvements.

6.3.1 Resource utilization

In Section 3.2 a design target of 4.5 was set for the resource utilization increase. This target was not met by the implemented proof of concept, because of the large number of encoders and decoders present in the pipeline. The resource utilization could be improved by applying one or more of the improvements below:

- **Basic building blocks optimization:**
  The current implementation was compared to an implementation provided by Xilinx, which was comparable in terms of combinatorial path delay and resource utilization to the used implementation. Other algorithms might give better performance, so the current implementation should be reviewed for performance improvements. The majority voter block was not compared to other implementations. The logic functions for the majority voting procedure might be optimized by applying logic optimizations which can reduce the resource utilization and the combinatorial path delay.

- **Different ECC coding scheme:**
  The Hamming coding scheme is known for not being optimal in terms of resource utilization when implemented in hardware as Hsiao codes require a less complicated XOR tree structure. By using a less complicated XOR tree structure, the number of XORs required for the encoder and decoders can be reduced.

- **Error detection only:**
  The cost estimates in Section 4.2 showed that detection only decoders are much more efficient in terms of resource utilization. If error correction is not required, the resource utilization per decoder could be lowered by 80%, resulting in LUT savings of about 15% for Triple Error Detection (TED) or 60% for Double Error Detection (DED).

When applying resource utilization improvements, there is one important thing to keep in mind: a lower resource utilization could result in a longer critical path. Depending on the application a trade-off should be made between the functional, timing and resource utilization requirements, which suits the applications needs.

6.3.2 Clock frequency

In Section 3.2 a design target of 50% was set for the clock frequency reduction. This target was not met by the implemented proof of concept, because of the long paths through encoders and decoders present in the design. The clock frequency could be improved by applying one or more of the improvements below:

- **Basic building blocks optimization:**
  The current implementation of the Hamming encoders and decoders have been compared to an implementation provided by Xilinx. This comparison showed that the implementation is most likely not far off the most efficient implementation, but more detailed analysis might reveal possibilities for improvement. Another algorithm for calculating the Hamming codes or the majority vote might also result in better performance.

- **Different ECC coding scheme:**
  The Hamming coding scheme is known for not being optimal in terms of combinatorial path delay when implemented in hardware as Hsiao codes require a less complicated XOR tree structure. By using a less complicated XOR tree structure, the delay of the encoder and decoders can be reduced.

- **Error detection only:**
  The cost estimates in Section 4.2 showed that detection only decoders are much more efficient in terms of combinatorial path delay. If error correction is not required, the critical path delay could be lowered by at least 5 ns, resulting in a clock frequency increase of about 8 MHz.

- **Pipeline stage insertion:**
  Several long delay paths can be split up by adding extra stages specifically for decoding to the pipeline. The following stages could be added to the pipeline:
  - Between the Fetch and Decode stage for decoding of the instruction bus response.
  - Between the Decode and Execute stage for decoding of the source register values.
  - Between the Memory and Write back stage for decoding of the data bus response.

  Although these extra stages will increase the clock frequency, the CPI will be lowered by these insertions as for each hazard; two more stages need to be stalled or flushed.

- **Parallel decoding and processing of data:**
  Most long paths include one or more decoders. The hybrid design uses a systematic

format for the code word, so there is no real need for decoding the code word before processing besides error correction. Therefore, the processing of the data could be performed in parallel with the decoding operation. If no error is detected, which will be the case in the majority of operations, the processor can proceed. If a correctable error is detected, the pipeline needs to be stalled for one cycle so the data processing operation can be executed with the corrected data. This could reduce some long paths up to 50% and thus would result in a major improvement of the clock frequency. The only drawback of this improvement is that it will affect the CPI, as the pipeline needs to be stalled for one cycle every time a correctable error is detected. The effect of this performance penalty will be dependent on the environment in which the softcore will be used.

6.3.3 Fault Tolerance

Although the hybrid design can mitigate a lot of different errors, faults could still occur at some places where they can lead to a failure. Furthermore, the softcore will be killed once a non-correctable error is detected and the design is also not able to solve problems in the configuration memory. These fault tolerance problems can be solved by implementing one or more of the improvements listed below:

- **Control signal protection:**
  Control signals leaving the pipeline are not protected, so faults induced on these signals can cause failures. Additional protection in the form of ECC or replication of the control signals can be used to provide the required fault tolerance on the interconnect. By placing the decoder or majority voter as close as possible to the location where the control signal is used, the chance of inducing a fault on the remaining interconnect can be kept as small as possible. Checking if the right operation was performed is much more difficult as these control signals are mostly used to control memories or registers. NMR could be used to prevent incorrect operations, but NMR was not used for memories due to synchronization problems between replicated memories.

- **Read value protection:**
  In the memories components (instruction memory, data memory, and registers) the address value is first decoded before the value is read from the actual memory. Between the decoder and memory, a fault could be induced, which will remain undetected. This can be avoided by storing the check word of the address along with the data and check word of the data. After a value is read, a decoder can determine if the address check word, matches with the address data word used for the read operation. If the address data word used and the address check word read do not match, the read operation needs to be restarted by the pipelines. This can be indicated to the pipelines by sending a nack with a distinctive error code. The decoder does not need to perform correction, which saves resources and path delay. The address data word used needs to be latched for one cycle so that the decoder can produce false negatives due to faults induced in the FF.

- **Write value protection:**
  In the memory components, there is no check if the value to be written is actually written or written to the right address. Therefore failures could be induced by influencing the writing process. Fault tolerance can be added to this write process by reading the written value in the cycle after the write operation and performing checks on the read values. The written data and check word need to be latched for one cycle by a separate FF bank so that they will be available in the next cycle for these cycles. The following pairs of data and check words need to be decoded:
  - The latched data word and the read data check word
  - The read data word and the latched data check word
  - The latched address data word and the read address check word

  The last pair is used for providing read value protection, as write value protection can only be implemented with read value protection also being available. Faults might be induced in the latched data, so false negatives can occur when using write value protection.

- **Roll back on non-correctable error detection:**
  If a non-correctable error is detected in a correction able or detection only softcore, it will be killed as there is no possibility to do something about the error other than to stop and let the user handle the situation. In many applications like medical applications this might not be an option, so a mechanism needs to be implemented which can revert the core to an error-free state. Rollback is a well-documented method for reverting a system to an error-free state, by saving the context of a system at specified points in time as described in [2] and [32]. Saving contexts can be a very expensive method because a large storage which is not susceptible to faults, is required and the actual saving operation takes considerable time. If rollback is not required for uncorrectable errors occurring in the memories, a much simpler rollback mechanism for reversing the pipeline and recently committed values can be used. If a uncorrectable error is detected in the pipeline the previous pipeline stages need to be flushed, the program counter needs to be rolled back to the correct address, and the recently committed values in the registers and memories need to be reverted. Reverting recently committed values can be performed by delaying the actual commit by a few cycles and placing these values in a First In First Out (FIFO) buffer. The values in the buffer should be available for read and write operations, but reverting these commits can be done easily by flushing the buffer. Rolling back the program counter is a much more difficult task, as this might involve jumps. Therefore, a FIFO buffer containing the last six program counter values should be added so that the program counter can be restored to the right value.

- **Configuration memory scrubbing:**
  The hybrid design does not have any features for dealing with SEFIs, but depending on the type of the FPGA the softcore might still experience such effects. One method to handle SEFIs is implementing configuration memory scrubbing, where the configuration memory is periodically checked for errors and corrected if neces-

sary. Most FPGA manufacturers provide device specific solutions for configuration memory scrubbing, so it is best to implement this on a per-device basis using the solution provided by the manufacturer.

- **Higher detection/correction capability:**
  If more errors need to be detected or corrected, the fault tolerance of the design needs to be increased. Increasing the fault tolerance of the pipeline can be done, by increasing the number of replicated pipelines. Traditional Hamming codes are limited to SECDED, so another coding scheme, like BCH codes, is required to provide increased fault tolerance for the memory elements. Other coding schemes might result in a decreased clock frequency and increased resource utilization.

6.3.4 Verification

The verification tests described in Section 5.2 proved the correctness of the design, but the coverage might be too limited to give enough confidence to this claim of correctness. The confidence in the design could be improved by applying one or more improvements listed below:

- **Test application suite extension:**
  The coverage of some instructions by the test application suite is limited. This coverage should be increased by adding additional applications which test all instructions, or by adding applications targeted at those specific instructions.

- **Random fault injection:**
  In this thesis fault injection at fixed locations at fixed intervals was chosen because of the confidence which can be gained from this method of injection, besides the advantages of controllability and reproducibility. However, this fixation of fault injection might miss bugs, so fault injection at random locations and at random times is a good addition to the verification suite.

Bibliography

[1] M. M. Ghahroodi, E. Ozer, and D. Bull, "SEU and SET-tolerant ARM Cortex-R4 CPU for Space and Avionics Applications," in *Second Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale (MEDIAN)*. Southhampton University and ARM, May 2013.

[2] F. Abate, L. Sterpone, and M. Violante, "A New Mitigation Approach for Soft Errors in Embedded Processors," *IEEE Transactions on Nuclear Science*, vol. 55, no. 4, pp. 2063–2069, 2008.

[3] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Proceedings of the 32nd Annual International Symposium on Microarchitecture, MICRO-32*. IEEE, November 1999, pp. 196–207.

[4] A. Bouajila, T. Sommer, J. Zeppenfeld, W. Stechele, and A. Herkersdorf, "A Fault-Tolerant Processor Architecture," in *22nd International Conference on Architecture of Computing Systems (ARCS)*. VDE, 2009, pp. 1–6.

[5] I. Elliott and I. Sayers, "Implementation of 32-bit RISC Processor Incorporating Hardware Concurrent Error Detection and Correction," *IEEE Proceedings (Computers and Digital Techniques)*, vol. 137, no. 1, pp. 88–102, 1990.

[6] H. Kanbara, R. Kinjo, Y. Toda, H. Okuhata, and M. Ise, "Dependable Embedded Processor Core for Higher Reliability," in *13th IEEE International Symposium on Consumer Electronics (ISCE 2009)*. IEEE, 2009, pp. 819–822.

[7] M. Medwed and S. Mangard, "Arithmetic Logic Units with High Error Detection Rates to Counteract Fault Attacks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[8] A. Mahmood and E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processors-A Survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, 1988.

[9] L. Anghel and M. Nicolaidis, "Cost Reduction and Evaluation of Temporary Faults Detecting Technique," in *Proceedings of the Conference on Design, Automation and Test in Europe*. ACM, 2000, pp. 591–598.

[10] Xilinx Inc, "Device Reliability Report, Second Half 2015," Xilinx Inc, Tech. Rep., April 2016.

[11] ——, "LogiCORE IP Soft Error Mitigation Controller v3.4.1," Xilinx Inc, Tech. Rep., September 2015.

[12] ——, "AR 4313, 14.x Timing - What are the design statistics in the timing summary (i.e. maximum/minimum arrival input/output time)?" Online support answer database, December 2012. [Online]. Available: http://www.xilinx.com/support/answers/4313.html

[13] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965.

[14] ——, "Progress in Digital Integrated Electronics," in *IEEE International Electron Devices Meeting*, vol. 21, 1975, pp. 11–13.

[15] D. Binder, E. Smith, and A. Holman, "Satellite Anomalies from Galactic Cosmic Rays," *IEEE Transactions on Nuclear Science*, vol. 22, no. 6, pp. 2675–2680, 1975.

[16] E. Normand, "Single-Event Effects in Avionics," *IEEE Transactions on Nuclear Science*, vol. 43, no. 2, pp. 461–474, 1996.

[17] ——, "Single Event Upset at Ground Level," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[18] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*. Newnes, 2013.

[19] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0," DTIC Document, Tech. Rep., 2014.

[20] Microsemi, "Understanding Single Event Effects (SEEs) in FPGAs," Microsemi, White paper, August 2011.

[21] P. E. Dodd and L. W. Massengill, "Basic Mechanisms and Modeling of Single-Event Upset in Digital Microelectronics," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583–602, 2003.

[22] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[23] C. Giraud and H. Thiebeauld, "A Survey on Fault Attacks," in *Smart Card Research and Advanced Applications VI*. Springer, 2004, pp. 159–176.

[24] I. Verbauwhede, D. Karaklajić, and J.-M. Schmidt, "The Fault Attack Jungle-A Classification Model to Guide You," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2011, pp. 3–8.

[25] S. P. Skorobogatov and R. J. Anderson, "Optical Fault Induction Attacks," in *Cryptographic Hardware and Embedded Systems-CHES 2002*. Springer, 2003, pp. 2–12.

[26] S. Skorobogatov, "Optical Fault Masking Attacks," in *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2010, pp. 23–29.

[27] P. Bourque, R. E. Fairley *et al.*, *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[28] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann, 2010. [Online]. Available: https://books.google.nl/books?id=o_Pjbo4Wvp8C&printsec=frontcover&dq=Fault-Tolerant+Systems&hl=nl&sa=X&ved=0CCkQ6AEwAGoVChMI-Iu4q82PyAIV6RbbCh0fqg_d#v=onepage&q=Fault-Tolerant%20Systems&f=false

[29] J. Von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata studies*, vol. 34, pp. 43–98, 1956.

[30] P. K. Samudrala, J. Ramos, and S. Katkoori, "Selective Triple Modular Redundancy (STMR) Based Single-Event Upset (SEU) Tolerant Synthesis for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 5, pp. 2957–2969, 2004.

[31] O. Ruano, J. Maestro, and P. Reviriego, "A Methodology for Automatic Insertion of Selective TMR in Digital Circuits Affected by SEUs," *IEEE Transactions on Nuclear Science*, vol. 56, no. 4, pp. 2091–2102, 2009.

[32] F. Abate, L. Sterpone, C. Lisboa, L. Carro, and M. Violante, "New Techniques for Improving the Performance of the Lockstep Architecture for SEEs Mitigation in FPGA Embedded Processors," *IEEE Transactions on Nuclear Science*, vol. 56, no. 4, pp. 1992–2000, 2009.

[33] M. H. Sulaiman, M. Salim, S. Irwan, A. Jaafar, and M. M. Ibrahim, "A Survey of Fault-Tolerant Processor Based on Error Correction Code," in *2014 IEEE Student Conference on Research and Development (SCOReD)*. IEEE, 2014, pp. 1–6.

[34] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "A Watchdog Processor to Detect Data and Control Flow Errors," in *9th IEEE International On-Line Testing Symposium, IOLTS*. IEEE, july 2003, pp. 144–148.

[35] M. Pignol, "DMT and DT2: Two Fault-Tolerant Architectures developed by CNES for COTS-based Spacecraft Supercomputers," in *12th IEEE International On-Line Testing Symposium, IOLTS*. IEEE, 2006, pp. 10–pp.

[36] M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies," in *Proceedings of the 17th IEEE VLSI Test Symposium*. IEEE, 1999, pp. 86–94.

[37] E. Dupont, M. Nicolaidis, and P. Rohr, "Embedded Tobustness IPs for Transient-Error-Free ICs," *IEEE Design & Test of Computers*, no. 3, pp. 56–70, 2002.

[38] R. Hentschke, F. Marques, F. Lima, L. Carro, A. Susin, and R. Reis, "Analyzing Area and Performance Penalty of Protecting Different Digital Modules with Hamming code and Triple Modular Redundancy," in *Proceedings of the 15th Symposium on Integrated Circuits and Systems Design, 2002.* IEEE, 2002, pp. 95–100.

[39] R. H. Morelos-Zaragoza, *The Art of Error Correcting Coding*. John Wiley & Sons, 2006.

[40] R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[41] A. Sánchez-Macián, P. Reviriego, and J. A. Maestro, "Enhanced Detection of Double and Triple Adjacent Errors in Hamming Codes Through Selective Bit Placement," *IEEE Transactions on Device and Materials Reliability*, vol. 12, no. 2, pp. 357–362, 2012.

[42] S. Satoh, Y. Tosaka, and S. Wender, "Geometric Effect of Multiple-Bit Soft Errors Induced by Cosmic Ray Neutrons on DRAM's," *Electron Device letters, IEEE*, vol. 21, no. 6, pp. 310–312, 2000.

[43] S. Baeg, S. Wen, and R. Wong, "SRAM Interleaving Distance Selection with a Soft Error Failure Model," *IEEE Transactions on Nuclear Science,*, vol. 56, no. 4, pp. 2111–2118, 2009.

[44] M.-Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[45] M. J. Golay, "Notes on Digital Coding," in *Proceedings of the Institute of Radio Engineers*, vol. 37, no. 6, 1949, pp. 657–657.

[46] D. V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Look-Up," *Communications of the ACM*, vol. 31, no. 8, pp. 1008–1013, 1988.

[47] J. C. Moreira and P. G. Farrell, *Essentials of Error-Control Coding*. John Wiley & Sons, 2006.

[48] R. Naseer and J. Draper, "Parallel Double Error Correcting Code Design to Mitigate Multi-bit Upsets in SRAMs," in *Proceedings of the 34th European Solid-State Circuits Conference*. IEEE, 2008, pp. 222–225.

[49] D. Henderson, "Residue Class Error Checking Codes," in *Proceedings of the 1961 16th ACM national meeting*. ACM, 1961, pp. 132–101.

[50] T. Rao, "Error-Checking Logic for Arithmetic-Type Operations of a Processor," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 845–849, 1968.

[51] ——, "Biresidue Error-Correcting Codes for Computer Arithmetic," *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 398–402, 1970.

[52] J. Gracia, J. C. Baraza, D. Gil, and P. J. Gil, "Comparison and Application of different VHDL-based Fault Injection Techniques," in *Proceedings. 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE, 2001, pp. 233–241.

[53] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[54] L. Sterpone and M. Violante, "A New Analytical Approach to Estimate the Effects of SEUs in TMR Architectures Implemented Through SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2217–2223, 2005.

[55] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool," in *Digest of Papers. Twenty-Fourth International Symposium on Fault-Tolerant Computing (FTCS-24), 1994.* IEEE, 1994, pp. 66–75.

[56] A. R. Weiss, "Dhrystone Benchmark: History, Analysis, Scores and Recommendations," 2002.

[57] S. Tam, "Single Error Correction, Double Error Detection," Xilinx Inc, Application Note XAPP645 (v2.2), August 2006.

[58] RISC-V community, "riscv-tests," Github repository. [Online]. Available: https://github.com/riscv/riscv-tests

[59] K. Asanovic, "Vector Microprocessors," Ph.D. dissertation, University of California, Berkeley, 1998.

[60] S. Gal-On and M. Levy, "Exploring CoreMark–A Benchmark Maximizing Simplicity and Efficacy," *The Embedded Microprocessor Benchmark Consortium*, 2012.

[61] Xilinx Inc, "Spartan-6 FPGA Block RAM Resources User Guide," Xilinx Inc, Tech. Rep., July 2011. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug383.pdf

Baseline RISC-V softcore

A

A.1 Synthesis settings

In table Table A.2 the XST used for making the design cost estimates were used. The complete core was synthesized, but also most individual components like the stages as the resource usage, and the path delays of these components were required for making the estimates. The path delays and period are listed in Table A.1.

Table A.1: The definitions of the path delays as reported by XST [12]

Delay type	Description	Symbol
Minimum period	The minimum period from any synchronous element to another.	t_{period}
Minimum input arrival time before clock	The minimum path from an input pad to any synchronous element.	t_{input}
Maximum output required time after clock	The maximum path from any synchronous element to an output pad.	t_{output}
Maximum combinational path delay		t_{comb}

Table A.2: The project settings for the synthesis project used for gathering preliminary synthesis results.

	Setting	Value
ISE	Version	14
Design properties	Family	Spartan 6
	Device	XC6SLX100
	Package	FGG676
	Speed	-2
	VHDL standard	VHDL-93
XST process properties	Resource sharing	False
	Register duplication	False
	Equivalent Register Removal	False

A.2 Detailed synthesis and mapping results

Table A.3: The synthesis results for the baseline softcore.

	Resources		Path delays (ns)		
Component	LUT	Registers	t_{input}	t_{output}	t_{period}
Fetch stage	72	68	2.4	2.2	2.5
Decode stage	332	436	3.5	12.4	1.6
Execute stage	660	202	-	10.6	0.5
Memory stage	99	139	1.7	0.8	3.6
Writeback stage	173	32	3.5	2.3	0.5
Hazard Control	19	6	-	0.5	4.5
Registers	130	64	1.9	3.2	0.5
ALU	463	0	6.6	-	-
Instruction decoder	186	0	10.6	-	-
Total	1732	835	11.4	8.6	7.6

Table A.4: Mapping results for the baseline core

Component	Slices	Registers	LUTs	BRAM
RISC-V SoC	1323	1941	2803	17
Instruction memory	116	90	213	9
Data memory	129	88	219	8
Pipeline	683	898	1383	0
Fetch stage	34	68	29	0
Decode stage	217	510	418	0
Execute stage	262	142	560	0
Memory stage	55	76	68	0
Writeback stage	66	32	155	0
Registers	33	64	130	0
Hazard control	13	6	20	0

ECC-based design diagrams

B

B.1 Design diagrams

For reasons of company confidentiality the full design diagrams of the ECC-based design are not included in the public version of this thesis.

B.2 Full Design Cost Estimate

Table B.1: The detailed design cost estimate for the ECC-based design using SECDED Hamming code and N = 4.

	Resources		Path delays (ns)		
Component	LUT	Registers	t_{input}	t_{output}	t_{period}
Fetch stage	383	304	11.4	2.5	0.0
Decode stage	2391	443	27.9	1.6	34.2
Execute stage	1988	230	33.4	0.5	0.0
Memory stage	904	640	15.9	3.6	1.7
Writeback stage	776	39	17.4	0.5	26.3
Hazard control	282	6	14.8	4.5	0.0
Registers	532	78	14.6	0.5	1.9
Total ECC-based	7659	1812	33.4	4.5	34.2
Total baseline	1732	835	11.4	8.6	7.6
Comparison	442%	217%	293%	52%	448%

C Hybrid design diagrams

C.1 Design diagrams

For reasons of company confidentiality the full design diagrams of the hybrid design are not included in the public version of this thesis.

C.2 Full Design Cost Estimate

Table C.1: The detailed design cost estimate for the Hybrid design using SECDED Hamming code and N = 4.

	Resources		Path delays (ns)		
Component	LUT	Registers	t_{input}	t_{output}	t_{period}
Fetch stage	760	304	13.5	2.5	0.0
Decode stage	1740	1744	23.8	1.6	3.5
Execute stage	3400	808	22.0	0.5	0.0
Memory stage	752	640	2.9	0.0	1.7
Writeback stage	2116	156	10.5	0.0	0.0
Hazard control	76	24	0.5	4.5	0.0
Registers	218	78	9.4	0.5	1.9
Core	266	0	2.1	0.0	0.0
Total	9391	3826	23.8	4.5	3.5
Total baseline	1732	835	11.4	8.6	7.6
Comparison	542%	458%	209%	52%	46%

Implementation

D

D.1 Block Algorithms

For reasons of company confidentiality the algorithms used in the Hamming encoder, and decoder, and the majority voter are not included in the public version of this thesis.

Verification results

E

Table E.1: The instruction coverage for the applications used during verification.

Instruction	RISC-V tests	Hello World	Dhrystone	CoreMark	Verification suite	Total
nop	12091	22963	426522	1083664366	461576	1084125942
add	89	26	5943	58928305	6058	58934363
addi	5798	108	27650	93058670	33556	93092226
and	41	0	2648	4569	2689	7258
andi	19	2506	30821	23203660	33346	23237006
auipc	240	2	2	14	244	258
beq	39	742	5292	23341408	6073	23347481
bge	45	0	3090	72786322	3135	72789457
bgeu	45	0	482	353608	527	354135
blt	39	0	613	1711652	652	1712304
bltu	39	0	2058	5126165	2097	5128262
bne	1485	1818	35800	104453039	39103	104492142
jal	43	18	6552	13984550	6613	13991163
jalr	9	18	5710	11809651	5737	11815388
lb	57	0	0	183000	57	183057
lbu	24	28	2359	9236436	2411	9238847
lh	58	0	0	17271037	58	17271095
lhu	24	0	0	694884	24	694908
lui	1114	23	4926	1177419	6063	1183482
lw	59	15063	187040	30644672	202162	30846834
or	41	0	6382	2812797	6423	2819220
ori	19	0	0	28161	19	28180
rd_cycle	0	0	2	2	2	4
rd_cycleh	0	0	0	0	0	0
rd_instret	0	0	2	2	2	4
rd_instreth	0	0	0	0	0	0
rd_time	0	0	2	2	2	4
rd_timeh	0	0	0	0	0	0
sb	35	0	1047	93449	1082	94531
sh	35	0	0	1430662	35	1430697
sll	57	2	2	204	61	265
slli	67	3	6380	153246127	6450	153252577
slt	52	0	0	1296000	52	1296052
slti	29	0	0	0	29	29

sltiu	1	0	616	26	617	643
sltu	0	0	0	1863	0	1863
sra	57	0	0	0	57	57
srai	33	0	0	7606437	33	7606470
srl	57	0	0	251	57	308
srli	30	0	4340	88292266	4370	88296636
sub	53	0	886	649898	939	650837
sw	182	292	15998	14350586	16472	14367058
xor	41	0	0	7100005	41	7100046
xori	19	0	1616	685	1635	2320

E.1 Parameters

The clock multiply parameter in Table E.2 is used for the generic `CLKFBOUT_MULT` of the `PLL_ADV` module of the Xilinx Spartan-6, which is the Phased Locked Loop used by the top-level architecture for generating the clock signal. The clock divide parameters Table E.2 is used for the generic `CLKOUT0_DIVIDE` of the `PLL_ADV` module.

Table E.2: The parameters used by the three different RISC-V SoCs for generating the synthesis results on a Xilinx Spartan-6.

Parameter	STD	FT	FI
Clock multiply	8	8	8
Clock divide	12	28	50
UART frequency (MHz)	83.3	35.7	20.0
IMEM size (kB)	16	16	16
DMEM size (kB)	16	16	16
`g_fault_injection_enable`	false	false	true
`g_inst_count_enable`	false	false	false

Table E.3: The parameters used by Xilinx ISE for generating the synthesis results for the RISC-V softcores on a Xilinx Spartan-6.

Process	Setting	Value
Synthesis	Optimization Goal	Speed
	Optimization Effort	High
	Register Balancing	No
	Register Duplication	False
	Resource Sharing	False
Map	Global Optimization	Speed
	Combinatorial Logic Optimization	True
	Equivalent Register Removal	True
Place & Route	Place & Route Effort Level (Overall)	High

Biography

Wietse Heida is an Electrical Engineer with a keen interest in digital hardware design, computer architectures and the interaction between hardware and software. Besides his professional interests, he is a fan of motor racing (Formula One in particular), has a large appreciation for (orchestral) music, and likes to read books.

His interest in Electrical Engineering was first raised during his participation in the CanSat competition of the TU Delft in 2008. The team from his school managed to win a rocket launch to one kilometer altitude, during which the satellite successfully performed its mission.

After completing his gymnasium, he started with the Bachelor Electrical Engineering at the TU Delft in 2010. He combined his major with a minor in Computer Science because the courses on programming had raised another interest he wanted to investigate. In 2013 he obtained his Bachelors with distinction with a thesis on simulating a Self Deploying Sensor Swarm for the Delft Robotics Institute. During his Masters in Embedded Systems, he specialized in Computer Engineering, combined with courses on computing in high-performance environments. This resulted in performing his graduation project on fault tolerant softcores within the RISC-V project at Technolution B.V. in 2015.

Besides studying at the EEMCS faculty of the TU Delft, he was also very involved in the education programs of the faculty. As a student-assistant, he mentored students during projects, labs and their first semester at university. As a student representative, he participated in several committees including the Faculty Student Council. In order to gain experience outside academics, he also participated in the Summer Internship program of TOPdesk in 2013.