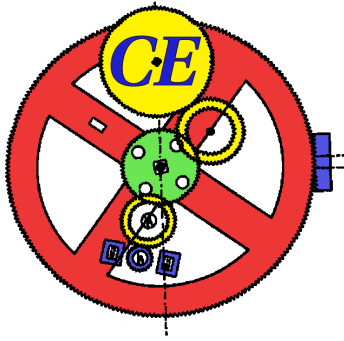


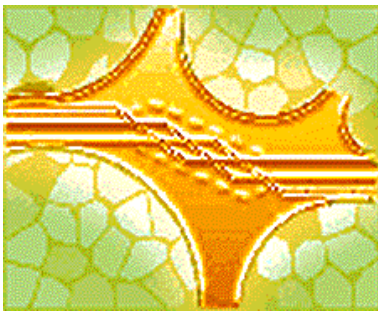


MSc THESIS

Architecture-Level Fault-Tolerance Techniques for Biomedical Implants

Robert Seepers

Abstract



CE-MS-2012-27

In this thesis the design and implementation of a new fault-tolerant architecture is described. The design targets both soft and hard faults by implementing a combination of known fault-tolerance techniques in an efficient way. The proposed architecture allows a trade-off to be made between performance and fault tolerance by means of instruction-level configurability. The design is evaluated in terms of fault coverage, area, average power consumption, total energy consumption and performance for various duplication policies and test-sequence schedules. It is shown that an area and power overhead of roughly 25% and 32%, respectively, are required to implement the techniques on the baseline processor. The main overheads of the architecture are performance (up to 106%) and energy consumption (up to 157%). It is observed that the average power consumption is often reduced when a higher degree of fault tolerance is set and therefore the energy consumption does not increase linearly with a higher execution time. It is shown that test sequences can effectively be scheduled during program stalls, and that nearly 100% of all soft faults are tolerated by using instruction duplication. The main advantages of using our techniques are the flexibility to make a trade-off between the overheads and the required degree of fault tolerance, the high portability of the used techniques and the small increase in area and power consumption.

Architecture-Level Fault-Tolerance Techniques for Biomedical Implants

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Robert Seepers
born in Rotterdam, The Netherlands

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

Architecture-Level Fault-Tolerance Techniques for Biomedical Implants

by Robert Seepers

Abstract

In this thesis the design and implementation of a new fault-tolerant architecture is described. The design targets both soft and hard faults by implementing a combination of known fault-tolerance techniques in an efficient way. The proposed architecture allows a trade-off to be made between performance and fault tolerance by means of instruction-level configurability. The design is evaluated in terms of fault coverage, area, average power consumption, total energy consumption and performance for various duplication policies and test-sequence schedules. It is shown that an area and power overhead of roughly 25% and 32%, respectively, are required to implement the techniques on the baseline processor. The main overheads of the architecture are performance (up to 106%) and energy consumption (up to 157%). It is observed that the average power consumption is often reduced when a higher degree of fault tolerance is set and therefore the energy consumption does not increase linearly with a higher execution time. It is shown that test sequences can effectively be scheduled during program stalls, and that nearly 100% of all soft faults are tolerated by using instruction duplication. The main advantages of using our techniques are the flexibility to make a trade-off between the overheads and the required degree of fault tolerance, the high portability of the used techniques and as the small increase in area and power consumption.

Laboratory : Computer Engineering
Codenummer : CE-MS-2012-27

Committee Members :

Advisor: Christos Strydis, Neuroscience, Erasmus MC

Advisor: Georgi Gaydadjiev, CE, TU Delft

Chairperson: Koen Bertels, CE, TU Delft

Member: Chris de Zeeuw, Neuroscience, Erasmus MC

Member: Ioannis Sourdis, CE, Chalmers Univ. of Tech., Sweden

*To my friends, family, and in particular my parents, for supporting
me in everything, always.*

Contents

List of Figures	ix
List of Tables	xi
Acknowledgements	xiii

1 Introduction	1
1.1 Problem Statement	1
1.2 Thesis Goals	2
1.3 Thesis Outline	2
2 Background	5
2.1 A short overview of faults and fault tolerance	5
2.2 The SiMS Project	7
2.3 The Target Architecture: SISC baseline	10
2.4 Fault Model	11
2.5 Related Work	13
2.5.1 Control and Data Flow	13
2.5.1.1 Control- and data-flow checking using signatures	14
2.5.1.2 Control-flow checking using duplicate instructions (pre-emptive)	16
2.5.2 Computation	18
2.5.2.1 Computation fault tolerance based on information redundancy	18
2.5.2.2 Computation fault tolerance based on time redundancy	18
2.5.3 Memory Access	20
2.5.4 Other solutions	20
2.5.4.1 *Self-checking circuits (TSC)	21
2.5.4.2 *N-Modular Redundancy (NMR)	22
2.5.4.3 Instruction-Level Configurability	22
2.5.4.4 Instruction-Based Self-Testing and BIST	23
2.6 Comparison between the discussed solutions	26
2.7 Summary	27
3 Proposed Design and Implementation	29
3.1 Introduction	29
3.2 Proposed Fault-Tolerant Design	29
3.2.1 Proposed Fault-Tolerant Design for Soft Faults	30
3.2.2 Proposed Fault-Tolerant Design for Hard Faults	31
3.3 Soft-Fault Tolerance: Instruction Duplication	32

3.3.1	Introduction	32
3.3.2	Methodology and General Overview	32
3.3.3	Hardware Additions	33
3.3.3.1	The Duplication Unit	34
3.3.3.2	Error Detection By Comparison	36
3.3.3.3	Error Correction by Re-fetching	37
3.3.4	Software Additions	39
3.3.4.1	The ft_set Instruction	39
3.3.4.2	Duplication Policies	39
3.3.5	Summary	41
3.4	Hard-Fault Detection: Instruction-Based Testing	41
3.4.1	Introduction	41
3.4.2	Test-Sequence-Based Testing	42
3.4.2.1	Methodology	42
3.4.2.2	Jump-And-Link Extension (jalo)	44
3.4.2.3	The branch-when-not-equal-to-zero (bneqz) instruction	45
3.4.2.4	Example	46
3.4.2.5	Scheduling	47
3.4.3	Self-Test Instructions	49
3.4.3.1	Control-Flow Testing	50
3.4.4	Summary	51
4	Evaluation	53
4.1	Experimental Setup	53
4.1.1	Testing Method (Soft Faults)	54
4.1.2	Testing Method (Hard Faults)	57
4.1.3	Determination of Overheads	58
4.1.4	Tool Flow	58
4.2	Experimental Results	61
4.2.1	Fault Coverage	61
4.2.1.1	Fault-Tolerance to Soft-Faults	61
4.2.1.2	Fault Tolerance to Hard-Faults	65
4.2.2	Overheads	65
4.2.2.1	Processor Area	66
4.2.2.2	Performance	66
4.2.2.3	Power Consumption	70
4.2.2.4	Energy Consumption	72
4.2.3	Compound Metrics	74
4.2.3.1	The PA metric	75
4.2.3.2	The EA metric	75
4.2.3.3	The FT / PA metric	76
4.2.3.4	The FT / EA metric	78
4.3	Summary and Discussion	79

5	Conclusions and Future Work	83
5.1	Summary	83
5.2	Concluding Remarks	84
5.3	Thesis Contributions	85
5.4	Future Work	87
A	SISC Errata	89
B	ISA	91
C	List of Test Sequences	93
D	Processor / Compiler Designer Review	103
E	Benchmarks	105
	Bibliography	117

List of Figures

2.1	Involved risk factors with (A) and without (B) a design platform.	8
2.2	Block diagram of the SiMS concept.	9
2.3	IMD energy efficient system architecture.[1]	10
2.4	Correct (black) and incorrect (red) ways of traveling the CFG.	14
2.5	Multiple input shift register for the 6-bit opcodes.[2]	15
2.6	Control flow checking architecture.[3]	17
2.7	The shadow program counter.[3]	18
2.8	Alu operations, with associated operations on the parity bits. [4]	19
2.9	Re-computing with shifted operands.[5]	19
2.10	Checking of functional units in Argus.[6]	21
2.11	The DIVA checker architecture. Source: [7]	22
2.12	The TMR scheme. [8]	23
2.13	The hardware-based approach (left) and software-based approach (right).[9]	25
3.1	Error detection by instruction duplication.[5]	33
3.2	The total hardware additions made for instruction duplication.	35
3.3	The generation of the re-fetch signal.	36
3.4	An error detection unit.	37
3.5	Error correction by re-fetching.	38
3.6	Post-processing the ft_set function call.	40
3.7	Block diagram representing instruction-based testing.	42
3.8	The tool flow for scheduling test sequences.	48
3.9	Self-Test additions for detecting errors in the control-flow logic.	52
4.1	A 16-bit maximal-length LFSR.	55
4.2	Replacing a data signal with the LFSR value.	56
4.3	The toolflow used for evaluating the design.	60
4.4	Manifested soft errors per test set.	62
4.5	The total amount of manifested soft errors.	64
4.6	The total processor area.	67
4.7	The number of cycles required to execute the benchmarks.	68
4.8	The critical path for the various designs.	70
4.9	The average power consumption for the various designs and duplication policies.	71
4.10	The energy consumption for the various designs and duplication policies.	74
4.11	The PA metric relative to Baseline.	75
4.12	The EA metric relative to Baseline.	76
4.13	The FT / PA metric relative to Baseline.	77
4.14	The FT / EA metric relative to Baseline.	79

List of Tables

2.1	Comparison of the different techniques	27
4.1	The various processor designs and features	54
4.2	Amount of soft-fault tolerance added over the baseline processor	65
4.3	Summarized overheads (relative to Baseline)	80
4.4	Area and critical path overheads (relative to Baseline)	81
4.5	Performance overhead (relative to baseline)	81
A.1	SISC specifications (fixed)	89
B.1	Instruction Formats (baseline and optimizations)	91
B.2	SISC ISA (baseline and optimizations)	92

Acknowledgements

I would like to thank Georgi Gaydadjiev and Christos Strydis for allowing me to work on this project, which was practically custom-tailored to my interests. Many thanks are also do to Dimitrios Siskos and Danny Riemens, for helping me learn the Processor Designer tool quickly and support for synthesis respectively. Furthermore I would like to thank Ghazaleh Nazarian, for helping me understand the CoSy compiler (even though we did not use it in the end). My gratitude also goes out to the system admins of the 15th floor, Erik and Eef, for fixing and installing several tools faster than the speed of light. Finally, I would like to thank Johan and Sander, my neighbours in the MSc room of the 15th floor. Not only have they shown me how to use some tools I've never used before, but also the endless mockery back-and forth, which has kept fun in coming to the room every day.

Robert Seepers
Delft, The Netherlands
February 13, 2012

Introduction

The first pacemaker, an external motor with electric cables going to the heart to keep it beating at a steady pace, was revolutionary in the 1950's. Thanks to technological advances, pacemakers these days are not only implanted into the body, but they also gather data for analysis, have a long lifetime and can even be configured to the patients' needs on the fly during a visit to a cardiologist.

The pacemaker is only one example of an implantable medical device. Many of these devices, used for both monitoring and treating a patients illness, are implanted in patients. Examples include treating Parkinson's disease by implanting deep-brain stimulators and giving the deaf the ability to hear using cochlear implants. Traditionally, these devices were designed and implemented from scratch to fulfill a single particular purpose. Due to the high development time and the high risk factor attached to developing such a system, the development costs are high. Developing medical devices is one of the main drivers behind the rapid increase in health-care costs, and as such it is highly desirable to minimize the development time and risk factor.

In order to reduce the development time and the risk factor attached to creating an implant from scratch, a design framework can be used. One such design framework currently being developed is SiMS (Smart implantable Medical Systems) [10]. The work presented in this thesis is part of the SiMS project. SiMS will contain multiple components, including batteries, sensors, actuators and processors. These components will be generic enough to allow many applications to run on them whilst complying with a number of standards for medical technology.

One of these standards is concerned with the reliability of medical devices. In case a fault in the system goes undetected, this can have an adverse effect such as storing an incorrect sensor value, or in the worst case, death. As such, medical devices need to have some form of fault tolerance to guarantee correct operation during the lifetime of the device.

1.1 Problem Statement

Many techniques exist to provide fault tolerance in a processor. Common examples include triple modular redundancy [11], self-checking circuits [2], built-in self-test [9] and software techniques [12], among others. However, many of the techniques are not suitable for a design framework targeting biomedical implants. For example, the average power consumption is tripled in triple-modular redundancy. This is not viable for biomedical implants as high power consumption is related to various medical conditions, such as tissue burns. Other techniques, such as built-in self-test, increase the processor area significantly and are not always able to detect soft faults, which are the main type of faults occurring in digital systems nowadays [11]. Finally, some of the techniques (such

as self-checking circuits) are too specific to be part of a design framework. Generic techniques are much more suitable for a design framework, as they allow a design to be tailored to a particular application more easily.

It is concluded that a number of commonly used fault-tolerance techniques are not in tune with the demands of a design framework targeting biomedical implants. As such, it is our desire to propose an architecture using techniques which are well-suited for this framework.

1.2 Thesis Goals

In order to provide generic yet highly effective fault-tolerance techniques, we are interested in using architecture-level techniques. Architecture-level fault-tolerance techniques focus on manipulating both the instruction-set architecture and the micro-architecture of a processor. While much research has been conducted into fault tolerance, not much research has been conducted into fault-tolerance techniques which optimize specifically for the constraints set in the biomedical environment. Therefore, this thesis will focus on the design of an architecture using fault-tolerance techniques at the architecture level. This design is then implemented and evaluated, in order to answer the questions “Is the proposed fault-tolerance architecture suitable for a biomedical implant?” and “How do the techniques used in the proposed fault-tolerance architecture compare (in terms of area, power, performance and fault-coverage) to other, well-known techniques?”.

In order to find the answers to these questions, the following methodology is followed during this thesis:

1. A large number of fault-tolerance schemes and solutions have been considered, all with their own overheads (both in area, performance, power consumption and energy consumption), the first step has been to study as number of fault-tolerance solutions which target the architecture-level;
2. From all available techniques, the most interesting techniques (with respect to the added fault tolerance and overheads) are selected, and a new architecture is proposed based on both combining and innovating on these techniques;
3. The design is implemented, and evaluated in terms of the added overheads and fault tolerance;
4. The design is compared to the solutions found under step 1 using aforementioned criteria.

1.3 Thesis Outline

This thesis is divided as follows: In Chapter 2, the background information required to understand the work in this thesis is presented. Furthermore, this chapter will provide information related to the SiMS project, our target processor and the fault model we assume. Finally, the various fault-tolerance techniques will be described and compared to each other. In Chapter 3, the proposed design is described in detail. The evaluation,

consisting of the experimental setup and experimental results is given in Chapter 4. The toolflow employed throughout this thesis is also briefly presented in this chapter. Finally, concluding remarks and future work are given in Chapter 5.

Background

This chapter will present the background information necessary to understand the concepts and choices made in this thesis. In section 2.1, some basic knowledge and terminology related to faults and fault tolerance are given. The framework in which this thesis is made, SiMS, is presented in section 2.2. The five-stage RISC architecture on which our design is implemented, SISC Baseline, is presented in section 2.3. In section 2.4, the fault models used for the different types of faults considered is discussed. Section 2.5 will discuss a number of techniques for fault tolerance which have been suggested (and implemented) previously. While our focus will be on architecture-level fault tolerance, some other, well-known techniques are also discussed for clarity. Finally, the fault-tolerant techniques discussed in section 2.5 are compared in section 2.6.

2.1 A short overview of faults and fault tolerance

Faults are any defect or temporary upset which may affect the correct functioning of an application or device. While this definition includes the programming of an application or chip and any other parts of the system, we will restrict our definition to the processor and instruction / data memories for the rest of this thesis.

Faults may occur due to imperfections during chip realization, aging, radiation, fluctuations in the power supply and crosstalk to name a few. A physical defect (either due to imperfections during chip manufacturing or aging of the chip) is defined as a hard / permanent fault (as an example, a wire may be stuck-at logical value 1 (SA1) or stuck-at logical value 0 (SA0)). In the other cases, where the fault only exists for a short time, we speak of soft faults. Soft faults are classified in two types: Transient and intermittent faults. Intermittent faults are bursts of faults that occur repeatedly at the same location of the micro-architecture, and occur due to various factors, including process variations, in-progress wear-out and voltage fluctuations. Transient faults on the other hand are described as single faults, i.e. there is no burst of faults, and the faults do not always occur at the same location in the micro-architecture. The majority of system errors are transient in nature, and can not be reproduced [11].

Strictly speaking, faults do not always result in errors, even when no fault tolerance is added to the platform: For example, we may not care for a bit flip in the shifter-unit of an ALU, when we are running an addition. In such a case the fault is masked. However, faults which occur on the active datapath without any form of fault tolerance will most likely result in an error. Errors may occur anywhere in a processor and can cause, for example, an incorrect result from an arithmetic instruction. In particular, it should be noted that a processor is operating correctly if the following fundamental tasks are working as intended [6]:

1. **Control flow**, which makes sure the program follows the control flow graph specified in the binary. This includes the fetch unit, the computation of the branch address, and the updating of the program counter.
2. **Data flow**, to make sure the program adheres to the data-flow graph specified in the binary. This includes register access (data dependencies), register renaming and instruction scheduling activities. This is of particular interest in processors which support out-of-order execution.
3. **Memory (access)**, in order to safely store the program's variables. This includes the instruction memory, the data memory, the registers and reading from / writing to these units.
4. **Computation**, to guarantee the computed values are correct. This includes the ALU and all other units related to computation (bit-wise logic units, the setting of control signals, etc.).

Fault tolerance is the property of a system to continue normal operation in case a fault occurs. This can be achieved by either masking the fault, or by detecting a fault after which corrections can be made. This implies that one or more of the following overheads will be added [8]:

1. Hardware redundancy, by adding extra hardware components. One example of such additions is duplicating the ALU stage of a processor, and comparing the outputs. In this case, a fault is detectable due to comparison. The faulty component may be isolated, after which its results will be ignored. In general, hardware redundancy has adverse affects on area and power consumption.
2. Time redundancy, by allowing the program execution to use more time. One example of such a scheme is a simple "execute the code twice, and then compare the output". In this case, a fault will have to be corrected by executing a third instruction to verify the correct result. In general this affects performance and power consumption.
3. Information redundancy, by adding extra information to (on-chip) signals, for example adding a parity bit to each memory element. More complex encodings (for example ECC) also have ways to recover the information, in case a fault is detected. Information redundancy generally affects area, and power consumption in a negative way.
4. Software redundancy, by adding software techniques to avoid software-induced errors (wrongly designed or generated code). One example is n-version programming, in which a program is written n times (but using different styles or by different programmers). These are then run separately, with the majority being assumed correct after finishing. This also affects performance and power. It should be noted that software redundancy is only reported here for completeness, as the research conducted in this thesis is into architecture level-techniques.

It is highly dependent on the executed application and constraints set by the environment what approach(es) is (are) viable. For example, if we are designing a processor to run very time-critical operations, hardware redundancy might be the only option, as it induces next to no performance loss. On the other hand, if we are designing a processor in a low-power, low-performance setting and area being a key constraint, we may go for adding time redundancy over hardware.

Fault tolerance is typically targeting single faults (one fault present at any given time). The reasoning for this is two-fold: First off, single faults occur much more frequently than multiple faults. Secondly, designing for single faults is considerably easier than for multiple faults, and techniques which have a high coverage for single faults generally also have a reasonable coverage for the case that two or even three faults are present at any given time. [2][13].

Fault tolerance can be added at run-time or offline. In the first case, soft faults may be detectable, while in the latter case only hard faults can be detected. Furthermore, testing may be done functional (using the processor or a component as a black box), or when enough details are available, testing can be done structural (a white box model).[14] Faults may be prevented to corrupt an output (preemptive), or the system may be recovered to the state it was in before the fault occurred. In the latter case, a roll-back mechanism has to be added for the recovering process.

Finally, as reported earlier, it is worth mentioning that faults may be masked (the fault it will not affect the output) or non-masked (the fault will affect the output). In particular, the property of a test mechanism to reproduce a fault is called the controllability, and the property to observe the fault the observability of a fault. This makes it important to realize that not every test will detect every fault, thus multiple tests may (and most often, will have to) be used to detect all possible faults in a circuit. [9].

2.2 The SiMS Project

The work presented in this thesis is part of the Smart Implantable Medical Systems (SiMS) project [10] which takes place at the Delft University of Technology. The main goal of the SiMS project is to construct a framework that provides medical researchers with the modules that are required to build variable implantable medical devices.

Nowadays, a typical implantable medical device (IMD) consists of a central unit microcontroller, a microprocessor, sensors, actuators, a wireless transceiver and a power source. The majority of IMDs today are used to measure, process and regulate a single parameter of the body. A problem with today's implants is the followed procedure when developing them. Usually such systems are made by researchers with electrical or mechanical engineering background collaborating with medical researchers. As it is not always possible to work together with specialists from the involved fields, and because these implants are generally built from scratch, a high implementation time and risk (due to various parameters and challenges during the design, development and normal operation of the device) is involved when designing such a product.

The existence of a platform can decrease this implementation time and risk factor of medical implantable devices, as the platform provides the designer with pre-developed

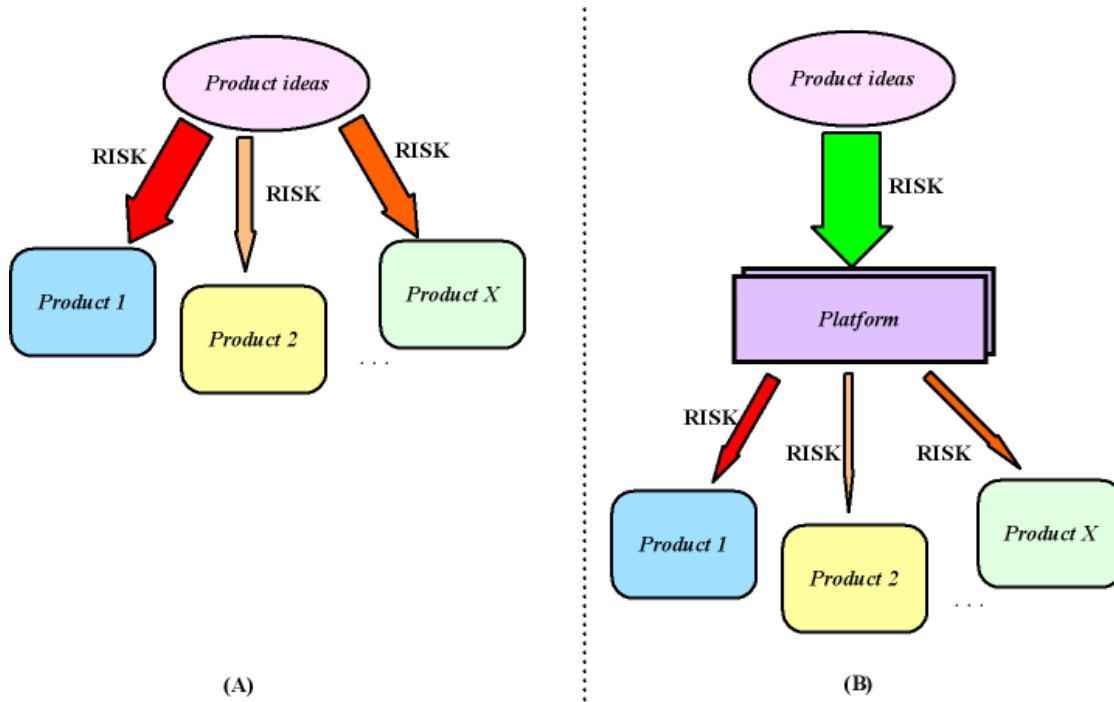


Figure 2.1: Involved risk factors with (A) and without (B) a design platform.

modules which have already been thoroughly tested. Such a framework must be designed according to the following requirements:

- high dependability,
- reliability and safety,
- miniature size,
- ultra-low power (ULP) consumption.

The concept of using a platform is depicted in figure 2.1. It should be realized that by reducing the time spent on designing the product, the total cost involved in designing the product is reduced, which consequently allows the product to be cheaper and thus more affordable. Furthermore it will provide a more standardized way of building new medical devices.

Monitoring (and/or controlling) a single parameter only, as is done in the majority of today's examples of similar systems does not meet the goals of the project. Given that such devices are directly related to human life, they will be characterized by a very high dependability, some degree of autonomy and self-awareness (within extremely demanding low power and size constraints). Our goal is silicon, multi-sensor/-actuator, single-chip, wireless medical systems. Such systems will be produced using fully integrated CMOS processes. In addition, they will be capable of context-sensitive behaviour (thus, smart), due to their multi-parameter awareness and communication abilities. The combination

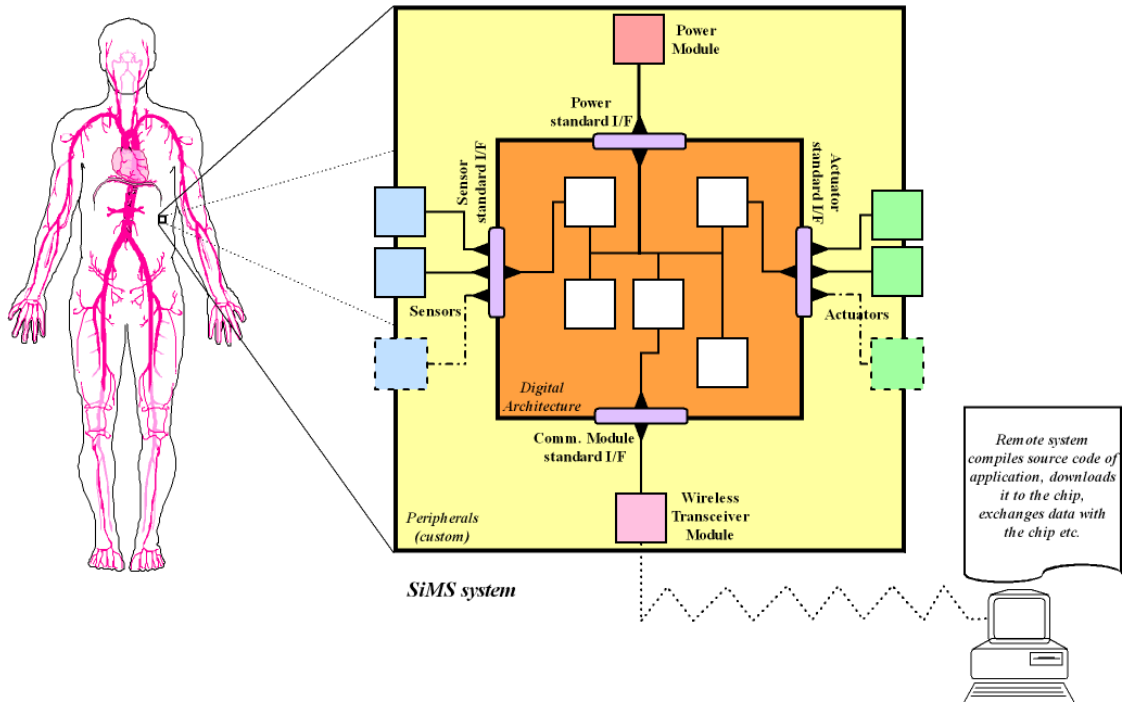


Figure 2.2: Block diagram of the SiMS concept.

of the aforementioned issues with the envisioned modular system approach, introduces new research challenges. A conceptual diagram of the SiMS platform is depicted in figure 2.2.

In particular, our work is part of the SiMS digital architecture and the compiler tools. The digital architecture forms the main processing/controlling unit characterized by extremely low power consumption and fault tolerance. The SiMS digital architecture is responsible for:

1. Collecting and processing data from the sensors and/or regulating the functionality of the actuators,
2. Forwarding data to the radio transceiver for telemetering information externally and for accepting commands received from the external host (e.g. the treating doctors computer),
3. Controlling the functionality of the various device subsystems; for instance, it is able to turn subsystems off when idling for long periods of time and back on when their operation is required.

As previously mentioned, implantable devices are liable to a set of strict (often extreme) specifications due to the sensitive and demanding nature of their living environment, and SiMS have to conform to them. In this context, the digital architecture will be crucial in delivering highly dependable implantable devices. In principle, it will achieve this by displaying attributes of error detection and error correction as well as self-testing

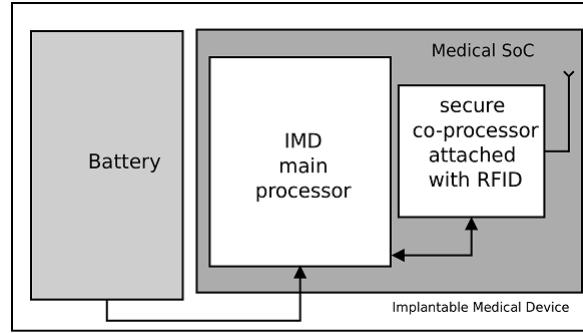


Figure 2.3: IMD energy efficient system architecture.[1]

and self-repairing properties. To this end, reconfigurable standard cells will be used. In addition to the hardware, all SiMS software modules have to be error-resistant and self-correcting, as well. The above features will give implantable systems the much-desired properties of robustness and safety, which are essential ones, given the medical nature of the applications.

The digital architecture has to be small and very basic, in order to have minimum power consumption. The size of the device is restricted as a large device may deem a device unusable in the case of e.g. intra-cortical implantation, whereas high power consumption a) drains the battery of an implantable device rapidly, and b) causes potential damage (e.g. burns) to its surrounding tissue due to heat dissipation. A less obvious reason for a tiny architecture is that it allows for achieving and maintaining the reliability of the system more easily.

2.3 The Target Architecture: SISC baseline

The majority of the work presented in this thesis consists of improvements and enhancements related to fault tolerance on the Smart-Implant Security Co-processor (SISC)[1] developed for the SiMS framework by Dimitrios Siskos, as discussed in this section.

IMDs equipped with wireless-communication capabilities are becoming increasingly popular, as they reduce the amounts of visits required to the doctor (as data can now wirelessly be downloaded to a base station at home) and can be used to keep the device functionality up-to-date (for example, reprogramming the implant *in vivo*). However, wireless communication with the outside world opens the door to potential unauthorized device access that can either lead to theft of sensitive personal data or, worse, render an implant inert or even hazardous to its host.

For this reason, SISC has been developed as part of the SiMS project, and is intended to be a co-processor preventing numerous possible attacks. SISC is intended to operate exclusively based on inductive power scavenged from the wireless link, to make sure the battery is not drained at all by the co-processor as depicted in figure 2.3.

In particular, the contributions made in SISC are:

- Proposing a new system-level low-power architecture,

- Identifying numerous attack scenarios and proposing a secure protocol to protect communication between the implant and the external reader,
- Providing the initial hardware design of a low-power ASIP co-processor and as-sorted compiler.

To this end, the SISC-architecture consists of:

- A 5-stage RISC processor with a stall-detection unit,
- A BaseLine 16-bit ISA with 24 instructions,
- An optimized processor version with two additional instructions, designed by profiling the security algorithm running on the co-processor. As this project intends to design the basis of the main processor, the additional instructions are not considered any further.

The stall-detection unit stalls the processor when data-dependencies are found within the pipeline. Note that this is in general to be considered part of a compiler. However, due to the high amount of stalls in the algorithm, due to the native data dependencies and the 16-bit ISA present in SISC, the choice has been made to move this task to the hardware so as to decrease the size of the binary. While this is an additional source of power consumption, it does allow the program binary to shrink and as such the system area is reduced. The necessity of having a stall-detection unit when duplicating instructions at run-time will be shown in section 3.3.3.1.

Benchmarking the processor has been done by running the target algorithm on the processor, and extracting the area, critical path, power and energy figures from the Synopsis Synthesis and Primetime tools. The results from this effort are used in a later stage to compare the fault-tolerant architecture to the SISC Baseline.

In this work some changes have been made to improve upon the basic SISC architecture. These changes are presented in Appendix A. The instruction set architecture for both the basic SISC architecture and our proposed architecture are presented in Appendix B.

In the remainder of the thesis, the names “SISC”, “SISC baseline” and “Baseline” all refer to the baseline version of the SISC processor (i.e. the version without optimizations for security protocols).

2.4 Fault Model

In this section the fault model used for designing and testing the design are introduced.

There are many ways to model the various faults in a digital system. Among others, there are models for single stuck-at faults, multiple stuck-at faults, path-delay faults, intermittent faults, memory faults and functional faults[13]. Each of the models have particular advantages: For example, the functional-fault model is very coarse grain and somewhat easy to design for, while path-delay faults are hard to model but cover more in-depth faults. A system designed for a more accurate fault model (with respect to

the actual faults occurring in a realization) will generally result in more reliable systems. However, designing for a more accurate fault model will generally also increase the engineering effort spent on designing the system.

Due to the fact our technique is split into two sub-techniques (As will be discussed in chapter 3, two fault models are used: a transient-fault model for soft faults, and a behavioral-fault model for hard faults. In both of the fault models we assume only a single fault can present in the system at a time. The reason for this assumption is that statistically the most faults are single in nature, and tests assuming a single fault cover roughly 99.6% of the cases with multiple faults[13]. This is also the main reason why more elaborate fault models are generally not considered in practice. Furthermore, we assume that the output will not corrupt after it has passed our checker. For example, the result of the beqz instruction (taken or not taken) will not corrupt between the checker and the control unit. While it is possible this may occur, the fan-in, fan-out and the size of the circuits which follow our check are small in general. While not negligible, the impact of these units on the fault coverage is small.

Soft faults are modeled by a temporary upset of a signal, which appears and disappears within ΔT . The following assumptions are made[15]:

1. Only a single fault is present in the system (single fault assumption),
2. The faults are well-behaved, i.e. during execution time the fault is permanent during the interval $[T, T+\Delta T]$, and the signal is free of faults outside this interval,
3. The faults are signal independent, i.e. the fault is not depending on the signal's value.

The design is based on the assumption the fault *may be* masked (even without any support for fault tolerance). This is somewhat different from the model originally described in [15]. However, this assumption allows us to simplify testing, as on a per-test basis, no checking needs to be done if the fault has been detected by the added fault-tolerance technique, or if it was simply masked. However, as the fault-coverage results of the fault-tolerant processor will be compared to those of the SISC baseline, masked faults are effectively canceled out, since the chance these occur is the same in both designs.

In our model, we assume a fault is only present in one cycle, and is disappeared in the next. This is justifiable as the processor is aimed for low frequencies (10-20 MHz), and as such the chance of a fault occurring in both cycles is very small (similar to the single fault assumption).

For hard faults, we model the faults with a behavioral fault model. In this model, the component-level design of the processor is used where components or signals are modeled to either fail or not. While single stuck-at fault modeling is more common and accurate for fault modeling and test-pattern generation, it requires gate-level details. As these details are not available in our case, a more coarse-grain fault model has to be used. As such, modeling the faults with the component-level design is the most accurate model we can use using the available information.

2.5 Related Work

In this section we will discuss the different fault tolerance schemes used to make sure certain tasks of the processors are functioning correctly. First, each of the fundamental tasks discussed under section 2.1 (control/data flow, memory access and computation) are discussed in sections 2.5.1 - 2.5.3. Generally speaking, the techniques are not strictly applied to only one of the fundamental tasks. For this reason section 2.5.4 contains a number of solutions which do not fall under a single one of these categories in particular, but are nevertheless related to architecture-level fault tolerance. In particular, three fault-tolerant platforms (COSAFE, Argus and DIVA) are discussed, which cover each of the four fundamental tasks. Finally, the discussed techniques are summarized and compared in terms of power, area, performance, fault coverage, and implementation time in section 2.6.

Certain solutions, which are not of particular interest to the field of biomedical implants due to the high overheads involved, yet are frequently used in modern systems, are briefly discussed for completeness. These are marked by a *-sign in front of the section or paragraph.

2.5.1 Control and Data Flow

Almost every program has to deal with control flow. Whether it is a simple function call or a large switch statement, the code will have to skip to another part of the instruction memory at some point. The blocks of code which we skip from / to are called basic blocks. As an example, an if-statement will consist of three blocks: The “then”, “else” and “goto” blocks. The “goto” block is the block where both the “then” and “else” blocks will return to after finishing.

A control-flow graph (CFG) connects the different basic blocks together, in a way specified by the program. An example CFG is depicted in Figure 2.4. By adhering to the control-flow graph, the program maintains the order in which basic blocks (or rather said: instructions) are executed. However, a fault, be it hard or soft, can lead to a mismatch between the actually executed program and the CFG, which is called a control-flow error (CFE). It is self-evident that a program, which is not executing its instructions in the right order, will generally not generate the correct output.

There are two ways the control flow may be corrupted. The first way is having an opcode corrupt in the middle of a basic block, for example turning an add instruction into a jump, causing a branch out of the basic block before the end. The second is a corrupted destination when traveling from one basic block to another. This corruption may either be due to a wrongly calculated address (e.g. an if-condition decides it needs to go to the else-block instead of the then-block), or due to a corruption of the destination in the branch instruction. Scenarios which may be caused by faults are depicted as a dashed line in Figure 2.4.

Software-based control-flow checkers and error recovery are typically hard to implement without any hardware additions. As such, a number of solutions have been

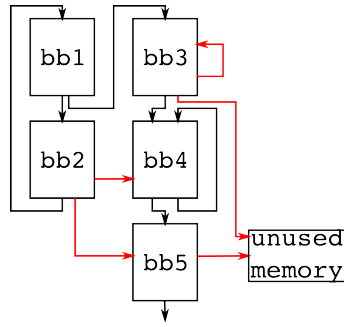


Figure 2.4: Correct (black) and incorrect (red) ways of traveling the CFG.

implemented combining soft- and hardware to prevent a fault from affecting control flow. A number of these are discussed in the following subsections.

2.5.1.1 Control- and data-flow checking using signatures

One of the more common ways of doing control flow checking, is by using signatures. These signatures summarize the information present in a piece of code (e.g. a basic block) in a pre-determined way. The compiler creates the signatures for the entire program, and adds signature-checking instructions to the program. The signature is reconstructed on the processor at runtime by means of additional hardware.

When a signature-checking instruction is executed, the processor will compare the signature pre-stored in an instruction to the reconstructed signature. In case of a match, the code has been traversed correctly (assuming a unique signature exists in case of a fault). In case of a mismatch, either the code has been traversed incorrectly or a fault has occurred related to the signature generation / checking. Either way, a basic solution is to re-execute all code from the the last time the static and dynamic signature matched (the checkpoint). This requires a roll-back mechanism (to roll the processor-state back to what it was before the first code line was executed).

COSAFE [2] is a processor designed for safety-critical applications with a high degree of fault tolerance. In COSAFE, a checksum is generated based on the instructions' opcodes in each basic block. COSAFE has three instructions dedicated to this mechanism: ESC / DSC (enable / disable signature check), and CSI (check signature immediate). The checksum is generated by a MISR (multiple input shift register, Figure 2.5). When ESC is executed, all inputs are reset and the flip-flops are enabled to be clocked (i.e. the circuit starts functioning). DSC has the inverse functionality (i.e. the circuit stops functioning). CSI contains an immediate value as operand. When CSI is executed (which is only allowed when signature checking is enabled), the signature derived by the MISR is compared to the immediate value. When the comparison is successful, the MISR is reset and continues to derive the signature for the next block. A faulty comparison results in a flag for the program, and an alarm bell is activated (given that COSAFE has been designed primarily for an infusion pump). It is up to software what to do or not with this flag.

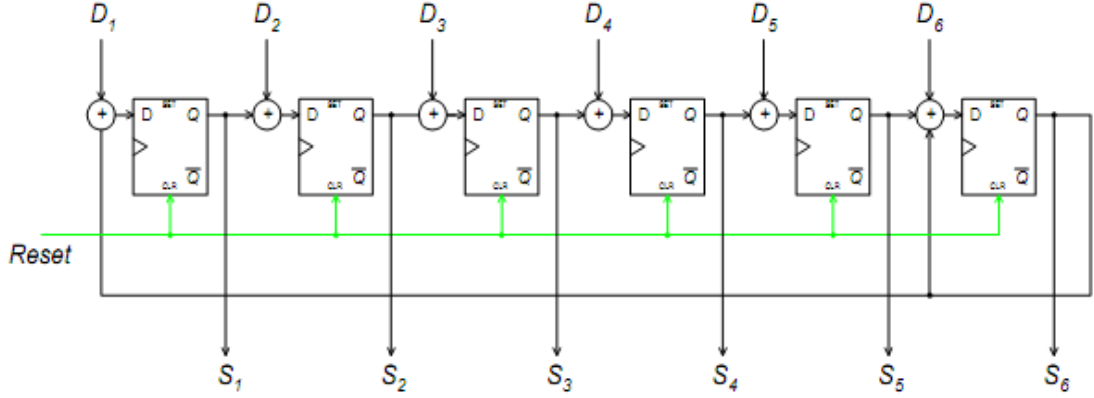


Figure 2.5: Multiple input shift register for the 6-bit opcodes.[2]

This addition in COSAFE only guarantees instructions within a block to be executed in the correct order (and thus, do not exit the basic block prematurely). To guarantee the control flow is also correct when exiting a block, too, COSAFE relies on total self-check circuitry (as will be discussed in section 2.5.4.1).

Similar to the control-flow graph is the data-flow graph specifying, among others, the data-dependencies between instructions. While some control-flow checkers [16] strictly check the control-flow graph, others [17] [18] [3] check some of the activities a data-flow checker is supposed to check, such as signals related to register access. However, these checkers are designed for in-order processors, and thus do not check all the activities a data flow checkers considers, such as instruction reordering.

One example of “pure” data-flow checking is Dynamic DataFlow Verification [19]. DDFV is employed by means of a static data-flow graph, specified in a program binary. In DDFV [19], static signatures (derived from the data-flow graph, known at compile time) are generated for each basic block. These signatures are calculated using vertex history signatures (VHS), which keep track of the inputs / outputs of a vertex (registers, PC, memory) and are stored in the program memory. Communicating this static signature is done by means of an additional DGS (data-flow-graph signature) instruction, and the processor is modified in such a way that the signature is stored. At run-time, the DDFV hardware re-computes the signature, based on the VHS observed (at run-time). It should be noted that in [19] only fault detection is implemented: recovering from a fault in the data flow graph would expectedly require a roll-back mechanism / checkpointing. Furthermore, communicating the signatures by executing an additional instruction will affect performance.

Argus is a fault-tolerant processor, designed for low-cost applications. In Argus [6], DDFV is employed for data-flow checking. Moreover, DDFV is also employed in order to guard against control-flow errors. This functionality is achieved by embedding the DCS instructions (which set the signatures for the basic block) in the calling blocks rather than inside the blocks. By comparing these signatures to the calculated

signatures in the called blocks, the transition from the calling block to the called block is also checked. However, by using this technique it can still not be checked if the branch itself was correct.

Similar to the VHS in DDFV, Argus keeps track of the history by SHS (State History Signature). When the signatures do not match, we know that either a problem has occurred regarding the data flow or the control flow. To hide the latency imposed by adding the signatures at run-time, the signatures are as much as possible hidden in unused bits in other instructions. Argus requires a fixed limit on the size of basic blocks, in order to upper-bound the time between checks.

Furthermore, Argus employs a parity bit in each register and part of the datapath to check correctness of an instruction or result. Finally, a 6-bit watchdog timer is present to check the liveness of the processor, with the timer being started when the pipeline stalls, and reset when the pipeline resumes normal execution.

The most simple watchdog timers are timers which give off an error signal in case no reset signal has been sent to the watchdog for a certain pre-set amount of time. While more elaborate watchdog timers exist, all have this basic functionality in common.

These timers can be used in order to detect control-flow errors [20]. First, an assertion is made in the program software. This assertion specifies a certain signature (not depending on the contents of the block per se), and the time required for the program to verify the correct functioning by sending out this signature. If the watchdog is not serviced in the specified time or an incorrect signature is received, an error signal is generated, which can be used to roll the system back to the original checkpoint (i.e. where we made the assertion).

In [20], two instructions (to set / service the watchdog) are added. The set instruction stores the signature in the processor, and starts the watchdog timer with the specified time. This signature is later compared to the signature provided by the service instruction. If the timer runs out before the service instruction is executed, or an incorrect signature is provided the watchdog will raise an error signal. In order to not only detect but also be tolerant to the detected fault, checkpointing and a rollback mechanism and have to be employed.

By using an FPGA for testing, a performance overhead of 40-80% has been measured. The average fault detection, measured by testing a number of benchmarks, is around 90%.

2.5.1.2 Control-flow checking using duplicate instructions (preemptive)

Another method is duplicating all instructions which can change the control flow in software[3]. In general, these instructions are basic branches and jumps, although specialized instructions may also cause intended changes to control flow. In the rest of this section however we will not consider such instructions.

In this method, the jumps and branches which are executed first do not commit their results: Instead, the resulting destination is stored. The result of the original instruction is then compared to the result of the duplicate. If both agree on the same

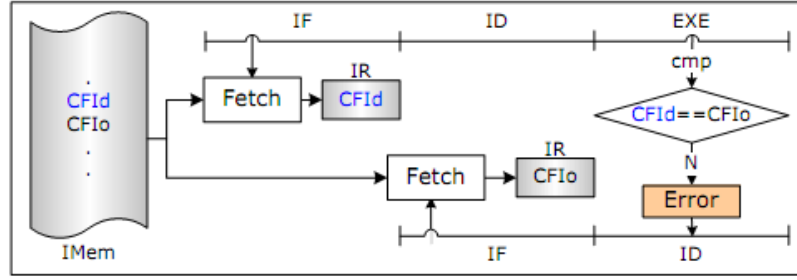


Figure 2.6: Control flow checking architecture.[3]

destination, the jump is made. Otherwise, if they do not agree (i.e. the destination of one of the two has corrupted), or the processor does not detect two jumps in a row (i.e. either the original / duplicate has corrupted from a control flow instruction into another instruction or vice versa), the pipeline is flushed and execution starts over from the last safe instruction. It should be noted that in [3], a limitation is set that non-control-flow instructions cannot change into control flow instructions. It will be shown in section 3.3.3.1 that our proposed architecture overcomes this limitation. This further simplifies the recovery process, as this limitation implies any instruction which is executed before its duplicate can always be considered safe. In [3] the duplicated instruction is executed before the original. The detection method is depicted in Figure 2.6. In this figure, CFId and CFId are the duplicated and original instruction, respectively. It is shown that when the result CFId is known, it is compared in the decode stage to the result of CFId. If it is not the same, an error signal is raised.

Another cause of control-flow errors are bit flips / upsets in the program counter (PC). For this, a shadow PC has been added, which acts as an identical twin to the PC [3]. In case of a mismatch between the shadow PC and normal PC, the program is terminated, since we are not sure which of the program counters is correct. This is depicted in Figure 2.7.

In this method, additional time and hardware are utilized to gain balance in the design. By allowing only control-flow instructions to execute twice, but executing other instructions a single time, a trade-off is made between performance and fault coverage. Furthermore, hardware cost is minimal, as only a register, a comparator and a flag for comparing / storing the value are added to the processor. Benchmarking has determined that the code size increased by 3-11% with a performance decrease of a similar magnitude when both control-flow-instruction duplication and a shadow PC are employed. Furthermore, the hardware increase has been around 3.59%.

This method will detect transient faults for a single-cycle upset. However, permanent faults in the combinational circuitry may not always be detected, e.g. if there is a permanent fault in a data line connecting the destination to the program counter, both jumps will generate the same (wrong) address. Due to its preemptive nature, no roll-back mechanism is required.

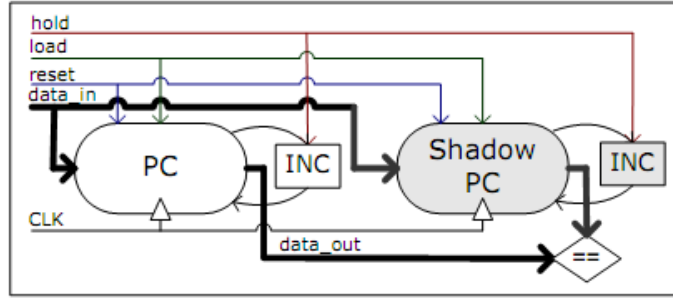


Figure 2.7: The shadow program counter.[3]

2.5.2 Computation

Providing a correct computation, even in the case of faults, has been applied to a very wide scale of processors. Computation is performed in the ALU, but other circuitry (such as setting control signals, or bit-wise operations) are also to be considered.

The following subsections will discuss a number of fault-tolerant schemes applied in order to strictly provide correct computation. All techniques introduced here are based on preemptive detection and correction, i.e. no value is considered safe until it is verified to be so.

2.5.2.1 Computation fault tolerance based on information redundancy

As presented in section 2.1, information redundancy is using extra information in order to verify correct behavior. Such a scheme can also be used to verify correct computation. This is the approach in [4]. In [4], each 32-bit register is encoded into a 7-bit code word using a SEC/DED hamming code. These parity bits are run through their own circuitry, which produce their own result. Figure 2.8 shows the operations performed by the regular operation, and the associated operation on the encoding. Communication between the two circuits is done by means of a bus. The circuit has the potential to correct the result in case an error is detected. The solution has an area overhead of 112%.

2.5.2.2 Computation fault tolerance based on time redundancy

One very basic way of detecting faulty hardware, is executing an instruction twice, and comparing the result. However, permanent faults will remain undetected this way, and such a technique comes at a performance overhead of 200% or more (execute, execute, compare). The performance overhead depends on the available registers (for holding the two values) and the available parallelism in the program (to re-order the instructions in such a way that we avoid unnecessary stalls. If a fault is detected, a third execution can be used to determine the correct result.

In [5], the problem with undetectable permanent faults is avoided by shifting the operands and re-computing the result. The inputs are first shifted by a factor k , after which computation is executed and the results are shifted back. It should be noted that for an ALU of size n , the size increases to the size $n+k$. The entire cycle is depicted in

Mnemonic	Operation (information bits)	Operation (parity bits)
ADD	$A + B \rightarrow R$	$P_A \oplus P_B \oplus P_{CY} \rightarrow P_R$
ADDC	$A + B + C_{IN} \rightarrow R$	$P_A \oplus P_B \oplus P_{CY} \rightarrow P_R$
SUB	$A - B \rightarrow R$	$(P_A' \oplus P_B \oplus P_{CY})' \rightarrow P_R$
SUBC	$A - B - C_{IN} \rightarrow R$	$(P_A' \oplus P_B \oplus P_{CY})' \rightarrow P_R$
SUBR	$B - A \rightarrow R$	$(P_A' \oplus P_B' \oplus P_{CY})' \rightarrow P_R$
SUBRC	$B - A - C_{IN} \rightarrow R$	$(P_A' \oplus P_B' \oplus P_{CY})' \rightarrow P_R$
COMPA	$A' \rightarrow R$	$P_A' \rightarrow P_R$
COMPB	$B' \rightarrow R$	$P_B' \rightarrow P_R$
NEGA	$A' + 1 \rightarrow R$	$P_A' \oplus P_{CY} \rightarrow P_R$
NEGB	$B' + 1 \rightarrow R$	$P_B' \oplus P_{CY} \rightarrow P_R$
INCA	$A + 1 \rightarrow R$	$P_A \oplus P_{CY} \rightarrow P_R$
INCB	$B + 1 \rightarrow R$	$P_B \oplus P_{CY} \rightarrow P_R$
DCRA	$A - 1 \rightarrow R$	$(P_A' \oplus P_{CY})' \rightarrow P_R$
DCRB	$B - 1 \rightarrow R$	$(P_B' \oplus P_{CY})' \rightarrow P_R$
CLR	$0 \rightarrow R$	$0 \rightarrow P_R$
LITA	$A \rightarrow R$	$P_A \rightarrow P_R$
LITB	$B \rightarrow R$	$P_B \rightarrow P_R$
AND	$A \cap B \rightarrow R$	$P_{AB} \rightarrow P_R$
OR	$A \cup B \rightarrow R$	$P_A \oplus P_B \oplus P_{AB} \rightarrow P_R$
EOR	$A \oplus B \rightarrow R$	$P_A \oplus P_B \rightarrow P_R$

Figure 2.8: Alu operations, with associated operations on the parity bits. [4]

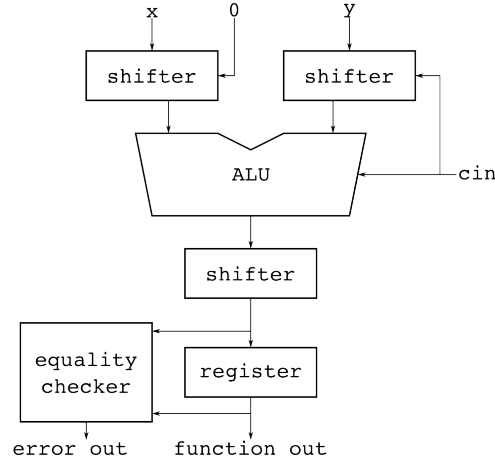


Figure 2.9: Re-computing with shifted operands.[5]

figure 2.9. Re-computing with shifted operands adds an additional area overhead (due to the shifters and increased ALU size).

In [21], a micro-architectural technique (Simultaneous Multithreading, SMT) is exploited to achieve fault tolerance. SMT is a technique for allowing several independent threads to use the multiple functional units on a superscalar processor, in one cycle. While this technique has originally been intended to achieve higher performance, it can also be used for increasing the fault tolerance of a system. By running multiple copies of the same thread, both soft and hard fault can be uncovered (by comparing the results of different functional units). Compared to regular execution, the performance overhead can be between 5-29%. It should be noted that there is also a hardware overhead, as SMT requires additional functional units.

2.5.3 Memory Access

A processor's memory elements, be it to hold variables or control signals, are essential to correct functioning. After all, even if we compute a value correctly, we need to guarantee this value will not be changed between now and the next time we use it. Due to the size induced by these storage elements, there is a relative large chance that an error will occur in the register file. It has been shown in [22] that in a simple processor which has both a floating point and integer unit, over 85% of all errors have been in the register file.

Information redundancy is a very well-known tool for protecting memory. The information stored this way can make the variables non-restoring (e.g. a 1-bit parity bit which stores if there is an odd or even number of bits on the register) or restoring (e.g. using error correction codes). However, such a restoring scheme takes a high power and performance overhead (it has been reported a simple ECC can have the delay of up to three times that of an ALU-operation [23], and energy consumption can be an order of magnitude higher during register access).

*As mentioned in section 2.4, the majority of errors are single in nature. While an ECC scheme can help us detect faults in every register, the chance of more than one register being broken is already very low. This is the thought behind the proposals in [24].

In [24], a number of spare registers have been added to the register file. During operation, the register file determines what variables are "live" and "dead". The proposed technique is targeted for a superscalar processor, using register renaming. Two schemes are investigated:

1. Conservative, in which only registers not in use (for renaming) are replaced for duplicates (i.e. they are dead only when 100% not in use). This also incurs that no performance is lost,
2. Aggressive, in which registers can be used even before they are 100% dead. In particular, it has been noted that with an increasing amount of cycles past, the chance of re-access is significantly low enough to justify this.

As a result, the technique introduces a marginal increase in energy consumption (11.1-28.8% vs. 195.6% for ECC). As the energy consumed by the register file is typically small when compared to the rest of the processor (e.g. the ALU), the energy consumed by the processor will typically be low.

2.5.4 Other solutions

In this section, we will treat the use of several fault-tolerant schemes which do not directly fit into any of the above categories. This is because they can either be applied to a number of them (e.g. they check both computation and memory aspects) or are applied to a layer which does not specifically fit into architecture-level.

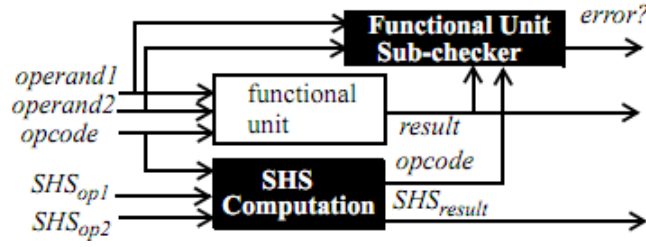


Figure 2.10: Checking of functional units in Argus.[6]

2.5.4.1 *Self-checking circuits (TSC)

A self-checking circuit consists of a functional unit, which produces encoded output vectors. These vectors are fed to a checker, which determines if an error has occurred. Encoding can be done using a variety of encodings, out of which parity, Hamming, Berger and CRC coding are common[2]. Self-checking circuits have the advantage of detecting a high amount of both hard and soft faults at a low cost in terms of performance, area and power. However, due to the need to design every component from scratch, implementation time is generally high.

In COSAFE [2], self-checking circuits are employed for the majority of the on-chip components, being the bus, memory, registers and register file, multiplexers, adders, counters and other logic units. Each of these components has been designed for single-faults and has been evaluated using a number of the above mentioned encodings. In general, it can be said that the area overhead of such an implementation is roughly a factor two, and an associated power increase of roughly a factor 1.5 - 2. Still, this result is considerably lower than in Triple Modular Redundancy (TMR), which will be discussed in the next section. In Argus [6], the SHS (state history signature, see section 2.1.1) is first calculated. The result of the functional unit and the SHS computation are then fed into a sub-checker, which has calculated the signature of the expected result. If a mismatch exists between the result of the sub-checker and the functional unit, an error flag is raised. This is depicted in figure 2.10.

In DIVA [7], a similar mechanism is employed. Each instruction is sent to the DIVA checker in addition to the regular pipeline, along with the operands (inputs) and computed results (outputs) of the operation. This information is then fed through two parallel verification pipelines: One verifies the correct computation (EX') using the operands and results, while the other verifies the operands themselves were correctly fetched (from memory, or the register file). If both pipelines agree, the result is committed. The DIVA checker architecture is depicted in figure 2.11. When implemented on a 4-issue out-of-order processor, the overhead of comparing the results after the DIVA core has finished execution was as little as 2.6% at rates of one fault per 1000 cycles. No area or power results were given.

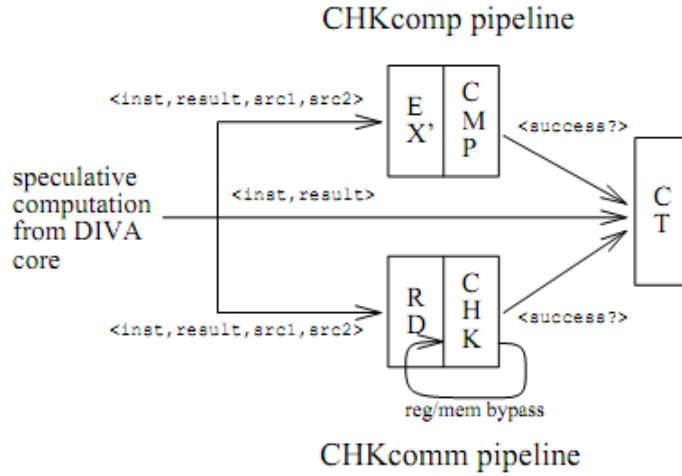


Figure 2.11: The DIVA checker architecture. Source: [7]

2.5.4.2 *N-Modular Redundancy (NMR)

NMR (n-modular redundancy) is an example of adding hardware redundancy in order to provide correct computation. By copying the same structure n-times, we can run the same program on all these n structures. By comparing the output of these structures, we can determine correct result. NMR can be applied on many levels, copying gates, functional units or even entire systems.

In particular, if the number is odd, we can check if the majority of processors agree on the same value. This is the idea behind triple modular redundancy (TMR), in which three processors use a majority voting system to determine which value is correct. If a processor makes the same fault multiple times, the hardware can be considered broken, and the processor can be taken offline. In this case a majority vote can no longer be achieved: Errors may be detected, but recovery relies on other methods. This is also true when a design starts out with only 1 copy, the dual modular redundancy (DMR) design. TMR is depicted in figure 2.12.

The overhead of such a scheme is obvious: We require a 100% increase in power and area per replication, plus the additional logic required for a comparator / voter. The upside of this scheme is the minimal (if any) performance overhead. Finally, it should be said that by using a TMR scheme multiple faults can be detected, and is very easy to implement, which is why it is found in a wide range of applications.

2.5.4.3 Instruction-Level Configurability

In [25], a method is proposed to allow a designer to make the trade-off between performance and fault-tolerance. It has been observed that for some applications (such as multimedia), fault-tolerance is less critical than others. The proposed technique, Instruction-Level Configurability of Fault Tolerance, allows the programmer to specify which instructions are critical (and thus be highly fault tolerant), and which are not. In particular, when a system supports multiple ways of fault tolerance (for example, TMR

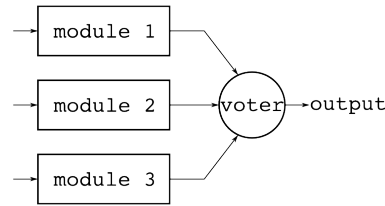


Figure 2.12: The TMR scheme. [8]

and re-executing and comparison), the degree of required fault tolerance can be set. Given as an example, the following code performs image addition.

```

for (i=0; i<N; i++){
    for (j=0; j<M; j++){
        sum = ImageX[i][j] + ImageY[i][j]
        if (sum>255) /* saturation */
            sum = 255;
        ImageX[i][j] = sum;
    }
}

```

The instructions controlling the loop may be considered more critical than others, and require a higher degree. The code inside the loop may be more optimized for performance and less for fault tolerance, as it can be considered acceptable if a single pixel is off by 1 bit.

2.5.4.4 Instruction-Based Self-Testing and BIST

Instruction-based self-testing is a method of using instructions to test for permanent faults. This can be either test sequences, which are a sequence of instructions native to the ISA to test for a defect, or test instructions which extend the ISA and have additional hardware to check specific circuits. Test sequences can be generated by treating the processor as a black box (functional testing) or as a white box (knowing for example, the Register-Transfer Level or netlist of the processor). A simple test sequence can, for example, be:

```

Li r3, 0
Bneqz r3, \#trap

```

In case a stuck-at fault occurs somewhere in the datapath the instructions traverse (for example, a SA-1 on a wire from the register file to the ALU), the branch will detect that r3 is not the intended 0, and will jump to a trap address to further deal with the fault.

Solutions basing themselves strictly on software have the advantage of not adding any additional hardware to the processor. In particular, when there are a lot of NOP instructions in the program, test sequences can be added with no performance penalty[9]. Tests may be compiled and executed at runtime, or program execution may be temporarily

stopped in order to run a number of tests. In the latter case, the tests can reside on an external, low cost memory and be loaded to the instruction memory [12].

It has been shown that by creating test sequences in a random way, the coverage is low despite the high amount of tests generated[26][27]. In [14], a deterministic approach for generating test sequences is presented, by means of considering the RT-Level. It is suggested that a pure software-based approach should not go into more depth than the RTL, in order to keep the methodology technology-independent. The method roughly follows the steps below:

1. For each instruction, the activated components (registers / combinational circuits / control signals etc.) are determined, along with the controllability and the observability of these components,
2. Instructions are grouped based on the operation they execute. Instructions which do not propagate any result to an internal processor register are discarded,
3. Instructions in these sets are ranked, based on how many instructions it takes generate a test result in a register (easier to control), and propagating the register value to an output (easier to observe),
4. Instructions are selected, based on the priority of the components they test. E.g. testing a large component such as the ALU may take priority over testing a smaller component such as a branch predictor.

For the PARWAN-CPU which is a simple eight-bit microprocessor, a fault coverage of 90% has been achieved, using 36 test sequences for the six ALU functional and 25 sequences for the three SHU functional units..

BIST (Built-In Self-Test) is a technique to test functional units for hard faults. In the most basic form, BIST consists of memory elements which contain the test vectors and the (compressed) results of these tests. The BIST controller can change the operation of a certain stage to test mode (for example, when a pipeline stall is detected), which will allow the BIST unit to feed the stage with test vectors, and compare the results with the (compressed) results. BIST comes with the price of an additional area overhead, due to the storing of the test vectors and results. Even in the case of a low amount of stored test vectors, the area overhead is still considerable[12].

In [9], it is suggested that adding sequences dedicated to improving the controllability and observability of processor cores can aid in achieving high fault tolerance with a minimum amount of instructions. NOP instructions are replaced by test instructions, to achieve fault tolerance without performance loss. These test instructions consist of a test vector, which modifies the internal pipeline registers and validates the correctness - however, no results are committed. Four test instructions exist, for the control unit, the branch unit, the adder which increments the PC and ALU. Two implementations have been proposed, which are depicted in figure 2.13:

1. A hardware oriented approach. In this approach, a BIST controller seeds a LFSR (linear feedback shift register) for each component. The results are then fed through

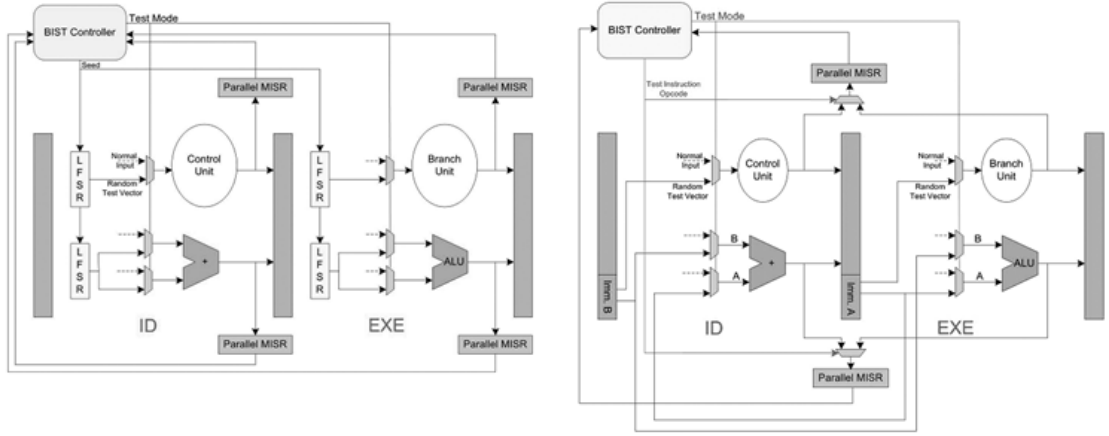


Figure 2.13: The hardware-based approach (left) and software-based approach (right).[9]

individual MISRs for compressing the results and are then compared by the BIST unit. A test instruction with a pseudo-random number is added by the assembler instead of a NOP, which specifies the value one of the four tests need to be run in,

2. A more software oriented approach, in which the test vectors (still pseudo-random) are part of the test instruction appended by the assembler. A BIST is still used for setting a given pipeline stage in test mode, and results are still compressed and compared in a similar way as in the hardware based approach. As this increases the required size of the instruction beyond the supported 16 bits, up to three consecutive instruction words are used per test (a dedicated controller takes care of this).

Both methods have been implemented on the PAYEH processor, a RISC processor with 16-bit registers which has a total of 24 instructions, the hardware-oriented method achieves 96.2% fault coverage, with 8192 randomly generated test vectors and at the cost of 13.1% additional area. The software oriented method achieves 90.3% fault coverage, with the same amount of random test vectors and at the cost of 5.6% additional area.

In [28], it is mentioned that there may be a large amount of signals and circuits which are functionally untestable, meaning many paths cannot be tested by means of the processor's own instructions. While this number is high for PARWAN (between 88-92%), it is noted that this number is most likely restricted to control lines for other, more pipelined designs. This paper further provides a methodology for testing path-delay faults, either by means of the processor's own instructions (in case they are functionally testable), or by adding hardware to structurally test these circuits (in case they are functionally untestable).

2.6 Comparison between the discussed solutions

In order to get an insight into what fault-tolerance solutions are most attractive, a comparison is to be made. By comparing area, power and energy consumption, performance, fault coverage and expected implementation time, we can get a good insight in what architecture-level technique could provide a solution with respect to the set constraints. While implementation time is not relevant in terms of overheads or usability, it is an important figure due to the limited time span a MSc. thesis is expected to take.

As has been shown in section 2.1, a processor is guaranteed to function correctly if the four fundamental tasks (*control / data flow*, *computation* and *memory access*) are working as intended. It is possible to use a technique which is applicable to the processor as a whole (e.g. TMR or TSC). Alternatively, it is possible to build a fault-tolerant processor by combining techniques which cover these tasks individually. If a number of techniques is selected, the overhead of all techniques combined will depend on the overheads of each individual technique.

As a very rough estimation, table 2.1 specifies these constraints for the solutions brought forth in the previous sections. As in most cases no real numbers are given, a rating is given on what the expected overheads are. The rating varies between none (0) to high (10). For overheads the rating indicates the expected overhead, and as such a higher rating indicates higher overhead. For the fault coverage the rating indicates the expected coverage, and as such a higher rating indicates a better performing technique.

Ratings are based on the level of the processor, meaning that if, for example, a method adds 100% fault tolerance to the register file, the fault tolerance added to the processor as a whole is not 100%. As the total coverage is the chance for faults in the decode stage divided by the processor-wide chance for faults, the added fault tolerance is related to the total processor area. As a very coarse approach, the ratings for each pipeline stage are based on the area determined during the synthesis of the baseline processor. These are:

FE: 2581 units
 DE: 3130 units
 EX: 8069 units
 MEM: 1232 units
 WB: 1364 units

As shown in table 2.1, both methods of control-flow checking (“Control flow (signatures)” and “Control flow (duplication)”) both have similar overheads, with mild variations. Similarly, the methods for preventing memory errors (“Memory (ECC)” and “Memory (live / dead)”) have similar overheads. It is shown that most techniques for computation checking add a high overhead to area and power consumption, except for “Computation (2x execute)”. However, a high performance penalty exists in “Computation (2x execute)”. The techniques which are applicable to the entire processor (TSC, TMR and BIST) have a high area and power consumption overhead.

Recall from section 2.2 that the main constraints of a biomedical implant are area

Table 2.1: Comparison of the different techniques

Technique	Area	Power	Performance	Fault Coverage	Impl. time
Control flow (signatures)	4	3	1	3	6
Control flow (duplication)	2	2	2	2	2
Computation (information red.)	8 (112%)	8	2	10 (100%)	8
Computation (2x execute)	1	5	10 (200%)	6	3
Computation (shifter op.)	7	7	5 (100%)	8	6
Computation (AR-SMT)	9	9	3 (5-29%)	9	3
Memory (ECC)	3	3	2	3	4
Memory (live / dead)	2	1	1	3	6
TSC	8 (100%)	8 (100%)	2	10 (100%)	8
TMR	10 (200%+)	10 (200%)	1	10 (100%)	3
BIST*	1-8	3	1	1-10	1-10

and power consumption. Due to the overheads introduced by some of the techniques discussed in table 2.1, it can be said that out of all discussed techniques only a subset of them is viable for biomedical implants. In particular, we conclude that time-based techniques such as “Computation (2x execute)” will be most viable, as it is shown that the area and power consumption overhead is lower when compared to other techniques. Note that some of the techniques here only cover soft faults; as such, if these techniques are employed a technique dedicated to hard faults (as, for example, one of the software based techniques described in section 2.5.4.4) will have to be employed as well.

The following chapter will describe the proposed architecture which has originated from the comparison in this section.

2.7 Summary

In this chapter, the background information has been presented necessary to understand the rest of this thesis. First, the difference between faults, fault tolerance and errors has been made apparent, after which some basic knowledge on faults has been given. Furthermore, the SiMS platform within which this project has taken place is presented. Also, the SISC processor, on which our design is implemented is presented. Finally, a survey is presented on a wide scale of fault-tolerant techniques, and these have been compared based on the expected overheads, the degree of fault tolerance, and expected implementation time.

Proposed Design and Implementation

3

3.1 Introduction

It is obvious that an ideal chip would be 100% tolerant to both hard and soft faults. However, as shown in section 2.6, achieving such a high degree of fault tolerance comes at a high overhead; for example area and power consumption (through TMR) or performance (through software duplication). In all cases, a high power or energy overhead is added, putting high loads on the battery. Furthermore, the techniques mentioned in section 2.6 generally have a high implementation time. For example, self-check circuits may be applied to all components the processor consists of, but applying this technique on the scale of a processor requires numerous man-years (as demonstrated in the COSAFE project)[2].

As such, combining architecture-level fault-tolerance techniques are expected to be more viable. Solutions on the architecture level typically require less time to implement than gate-level techniques, yet can still provide a high degree of fault tolerance. By combining a number of the techniques presented in section 2.5 in a smart way, a balanced design with respect to area, power / energy consumption and performance can be achieved.

This chapter is structured as follows. First, the proposed design is presented, explaining what the design consists of and the reasoning behind its proposal. In sections 3.3 and 3.4, the techniques to prevent soft- and hard-errors are explained in detail.

3.2 Proposed Fault-Tolerant Design

To achieve a high degree of fault tolerance for a minimum overhead, the pros and cons of the found solutions in section 2.5 have been evaluated. The aim is to create a balanced design, attempting to reduce the area-power-performance product compared to the previously discussed solutions. Such a design will have reduced energy consumption, increasing battery lifetime. At the same time, by not allowing the area or power consumption to get too high, the design remains suitable for biomedical implants, where physical constraints are put on the area / power consumption as presented in section 2.2.

By combining the previously discussed techniques in a smart way, overheads may be reduced due to resource sharing. Furthermore, combining existing techniques lowers the total design and implementation time. In particular, the proposed design consists of a combination of the works presented in sections 2.5.1.2, 2.5.2.2, 2.5.4.3 and 2.5.4.4. From a broad point of view, the design is split in two techniques: one to detect and correct soft-faults, and one to detect hard-faults.

3.2.1 Proposed Fault-Tolerant Design for Soft Faults

The technique for providing fault tolerance to soft faults is similar to the works presented in section 2.5.2.2, in the sense that an instruction is run twice and the results are then compared to the results at the output. However, key differences are:

- Instruction duplication is done by the processor at run-time rather than at compile-time. as proposed in [29]. This allows us to make a trade-off between fault tolerance and overheads at runtime;
- A parity bit guards against opcode corruption. This allows the “original” instruction to corrupt as well as the duplicated instruction, which is a limitation in the technique proposed in [3];
- A shadow program counter guards against both soft and hard faults in the program counter. This idea is a straightforward implementation of what has been proposed in [3];
- Additional hardware is added to each stage for the storing and comparing of the committing results. This is similar to what has been done in [3] for the target-address of control-flow operations. However, we apply the same technique to all stages which commit results;
- Error correction is done by re-fetching from the last correctly executed instruction onward, and flushing the pipeline. As this is a preemptive technique this method requires no mechanism to roll the processor back to the last known correct state, which is a key difference when compared to checkpointing (section 2.5.1.1); and
- The design allows a trade-off to be made between performance and fault tolerance by means of instruction-level configurability, depending on the needs of the application. The required degree of fault tolerance is set for a subset of instructions in a block of code. Using a custom instruction is one of the two ways of setting the degree of fault tolerance as proposed in [25];

The advantages of performing the comparisons in each individual stage are as follows: First, by adding extra hardware for the storing and comparing to each stage, the high performance drop in software duplication is partially avoided, as an additional compare instruction is not required. Moreover, it should be realized that hardware additions are already required in the case of control-flow operations, as demonstrated in [3]. Secondly, as the registers in the register file are not used for holding the intermediate values, register pressure is not increased and thus the scheduler puts similar effort scheduling the instructions. Thirdly, having registers dedicated to a particular stage allows the registers to have a custom size in order to hold the expected result of a given size.

Duplication is done by the processor at run-time rather than at compile-time, due to a two-fold of reasons: First, the timing overhead of re-fetching the same instruction in case it needs to be duplicated is expected to be small as it only requires a comparison between the results of the original instruction to the result of duplicated instruction. It is even possible no overhead is added in case the critical path is not depending on this

comparison. Secondly, and more importantly, a stall-detection unit is already present in the SISC baseline processor, which is the most crucial part when doing the duplication in the processor, as it prevents data-dependencies from being violated. As such, minimal overheads are added in comparison and a minimal design effort is spent on implementing the duplication method.

Finally, as the processor is designed to be part of a framework, customization is an important factor in designing any solution, including fault-tolerance. By allowing the programmer to make a trade-off between fault-tolerance and performance, as presented in [25], the processor is viable for more applications. It should be noted that such applications also exist in the biomedical field: For example, more performance may be required for image or sound processing (artificial eye or ear) in which a fault in the processed data (e.g. a pixel value) might not be much of a problem, while other applications (artificial pancreas, encryption) may require more fault-tolerance due to the criticality of the application.

In addition to the key differences mentioned above, two improvements are made to the works related to this technique: To overcome the limitation presented in [3] which states that the original instruction may not be corrupted, a parity bit is added to each instruction's opcode, so as to detect a corrupted opcode. After all, if only a sub-set of instructions needs to be duplicated, an error can only be detected if the original instruction (which needs to be duplicated) is not corrupted. Otherwise, the instruction is not duplicated, and the fault is still not detected. Note that this limitation exists regardless of whether duplication is done in hardware or software. Checking with the parity bit is possible under the assumption that both the parity bit and opcode will arrive at the memory location the instruction is read from without corrupting (i.e. no double faults will occur from the main memory to the fetched instruction).

3.2.2 Proposed Fault-Tolerant Design for Hard Faults

Duplication and re-execution is a powerful technique for preventing soft faults from causing errors. However, hard faults are hardly covered by this technique, as the result of both the original and the duplicated instruction will be the same, yet incorrect. While soft faults are by far the most common faults [11], covering hard faults is also required due to the safety-critical nature of biomedical implants. To guard against hard faults, an approach is taken using a combination of the methodologies presented in [12], [30]. The derived methodology is as follows:

- Test sequences are created to test the functionally testable paths in the processor. The methodology used to derive the test sequences is the same as in [12];
- Scheduling these sequences is done in a round-robin style, taking the place of stall (nop) instructions. This is similar to what has been proposed in [9], where it is stated that replacing the stalls with test sequences hides the performance overhead of executing these test sequences. Testing for hard faults in this way is very suitable to the baseline architecture, due to the high amount of stalls present in the programs compiled for the baseline. These stalls exist due to the inherent

data-dependencies in the programs, and the dependencies occurring due to the 16-bit ISA; and

- Specific hardware structures are added to the processor to test the functionally untestable paths. An example of a functionally untestable path is the logic involved in updating the program counter. This is similar to what has been proposed in [30] and is similar to design for testability.

The following sections will discuss the aforementioned techniques in more detail.

3.3 Soft-Fault Tolerance: Instruction Duplication

3.3.1 Introduction

In this section, the details related to the instruction-duplication technique are discussed. The section is organized as follows: First, the basic methodology followed is given in section 3.3.2. In section 3.3.3 the necessary additions made to the processor are presented, after which the additions to the ISA and duplication policies are introduced in section 3.3.4.

3.3.2 Methodology and General Overview

As discussed in section 3.2, the technique for detecting and correcting soft faults uses instruction duplication. The basic scheme for such a solution is depicted in figure 3.1. In this scheme, the input x is first run through the function f and stored in a register. On the second run, x is again fed through the function f , and the result is compared to the result of the first execution. In case the results are not the same, an error is detected. It should be noted the error needs to be corrected after it has been detected (not shown in the figure). While the hard- and software additions required to do this are discussed at length in the following sections, they can be summarized as follows:

1. An instruction-duplication unit,
2. A register / comparator unit, to compare the result of the second execution to the result of the first execution. In particular, a register / comparator unit is added for each committing signal (i.e. the signals that write to a register (including the PC) or memory),
3. A way to correct the error. In this design, this is done by re-fetching from the address the corrupted instruction originated from.
4. A way to set which instructions need to be duplicated and which not.

In the current design, there are three sets of instructions which can be replicated: *none*, control-flow instructions, and *all instructions*. The reason for specifying these sets is as follows: First, in case no fault tolerance is required yet performance is highly desired, the designer should be free to set the duplication to zero. Similarly, in the case full fault tolerance is required, duplicating and comparing every instruction is the safe

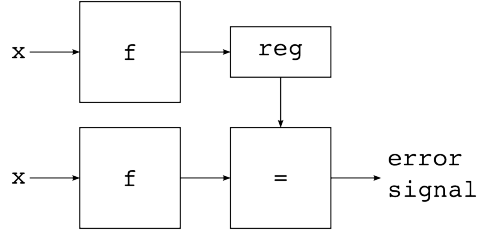


Figure 3.1: Error detection by instruction duplication.[5]

way to go. In case some form of fault tolerance is already present in the program (e.g. software-level fault tolerance), the duplication technique merely acts as a support to the already implemented software-level technique. As checking control-flow operations in software proves to be hard (requiring both check-pointing and roll-back mechanisms), the designer is free to duplicate only the control-flow instructions to support software techniques.

Two approaches have been taken when an error is detected:

1. Re-fetching in the cycle after the error has been detected (design name FT/SF),
2. Re-fetching as soon as the error has been detected (design name FT/SF/FP).

These two approaches have different characteristics in terms of area, power / energy consumption, and critical path. As will be shown, FT/SF/FP performs better in terms of area, performance (critical path), power consumption and energy consumption. Throughout the rest of the thesis, remaining designs are based on the re-fetching policy of FT/SF. The designs are not based on FT/SF/FP, as the benefits of changing the refetching scheme have been found at a later stage. However, minor changes to the presented designs can reduce the overheads in these designs with similar reductions.

The following sections will present in-depth information of the instruction duplication technique.

3.4 Soft-Fault Tolerance: Instruction Duplication

3.4.1 Introduction

In this section, the details related to the instruction-duplication technique are discussed. The section is organized as follows: First, the basic methodology followed is given in section 3.3.2. In section 3.3.3 the necessary additions made to the processor are presented, after which the additions to the ISA and duplication policies are introduced in section 3.3.4.

3.4.2 Methodology and General Overview

As discussed in section 3.2, the technique for detecting and correcting soft faults uses instruction duplication. The basic scheme for such a solution is depicted in figure 3.1. In

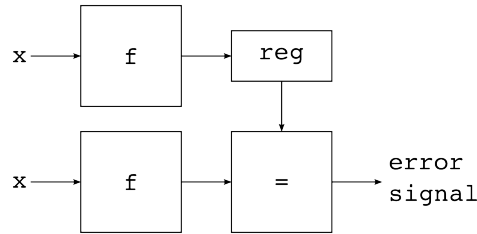


Figure 3.2: Error detection by instruction duplication.[5]

this scheme, the input x is first run through the function f and stored in a register. On the second run, x is again fed through the function f , and the result is compared to the result of the first execution. In case the results are not the same, an error is detected. It should be noted the error needs to be corrected after it has been detected (not shown in the figure). While the hard- and software additions required to do this are discussed at length in the following sections, they can be summarized as follows:

1. An instruction-duplication unit,
2. A register / comparator unit, to compare the result of the second execution to the result of the first execution. In particular, a register / comparator unit is added for each committing signal (i.e. the signals that write to a register (including the PC) or memory),
3. A way to correct the error. In this design, this is done by re-fetching from the address the corrupted instruction originated from.
4. A way to set which instructions need to be duplicated and which not.

3.4.3 Hardware Additions

The additions made to the processor are as follows:

1. **A duplication unit**, which determines if an instruction needs to be duplicated or not. This unit also has a unit to determine if the opcode has been corrupted by checking the parity bit (as discussed in the previous section);
2. **An error-detection unit** for each result, to determine if a soft fault has occurred
3. **Additions to the fetch stage** to correctly re-fetch in case an error has occurred or an instruction needs to be duplicated.

The following subsections will explain the modules, being the duplication unit, the error-detection unit(s) and the error-correction unit in more detail. However, without going into detail, the general layout will be described here for reference.

The total processor design for the FT/SF processor is depicted in figure 3.2. The signals and modules in black correspond to signals and modules which already existed

in the Baseline processor. The dashed signals and modules have been added to the processor for the instruction-duplication technique. It should be noted that the figure is not drawn to scale in order to make the additions clearly visible.

First, at the bottom of figure 3.2, the duplication module is shown. The FTmode register is set by the software, and is used by the duplication module to determine if the instruction needs to be duplicated. This module generates the signals for re-fetching in case the instruction needs to be duplicated, and also a parity error in case the opcode is corrupted. Note that the parity comparator is not shown separately, but rather as part of the duplication unit.

On the left side of figure 3.2, the changes made to the fetching unit are visible. It monitors the error signals (generated in the subsequent stages) and re-fetches from the corrupted instruction's location onward an error has occurred.

Finally, a combination of a register and a comparator is shown in numerous locations throughout the processor. These circuits are the main error-detection units, and are used to store (register) and compare (comparator) the results, in order to generate an error. It will be shown later that resource sharing is possible - for this reason, several MUXes allow separate stages to write to the same register / comparator.

3.4.3.1 The Duplication Unit

As can be seen in figure 3.2, the duplication unit resides in the decode stage. The main outgoing signals are the “fetch_same” signal, the “alu_mode” signal and the “parity_error” signal.

The duplication unit has a small state machine, which keeps track of whether the last instruction has been an original / duplicated instruction. As the current supported duplication modes are *none*, *control flow* and *full* (as described in section 3.3.4.1), a signal is set whenever a control-flow operation is detected. Based on the internal state of the duplication unit, the control-flow-detected signal and the mode set in the FTmode register, the re-fetch signal is set to let the fetch stage know it needs to re-fetch (i.e. the current instruction needs to be duplicated) or not. Other than setting the re-fetch signal, the duplication unit is also responsible for letting the subsequent stages know if they need to store / compare the result (in case the instruction requires duplication), or simply commit the result (in case the instruction is not duplicated, i.e. the instruction is not to have fault-tolerance).

A mealy-machine representation of the duplication unit is depicted in 3.3. In this figure “dup_ins” is the signal set when an instruction needs to be duplicated. “ref” Is the signal sent to the fetch-stage in order to re-fetch from the same PC and “mode” is the alu_mode signal, which is used to tell subsequent stages know the mode of operation (i.e. store, compare or commit).

As shown, two states exist: “orig” and “dup”. These states correspond to whether the currently decoded instruction is expected to be an original or duplicated instruction respectively. When the decoded instruction does not need to be duplicated the state will remain orig, the ref signal is not set (as we do not need to re-fetch) and the mode signal is set to exec (as the subsequent stages simply need to execute the instruction). Duplication is achieved by making a transition from orig to dup, which happens when

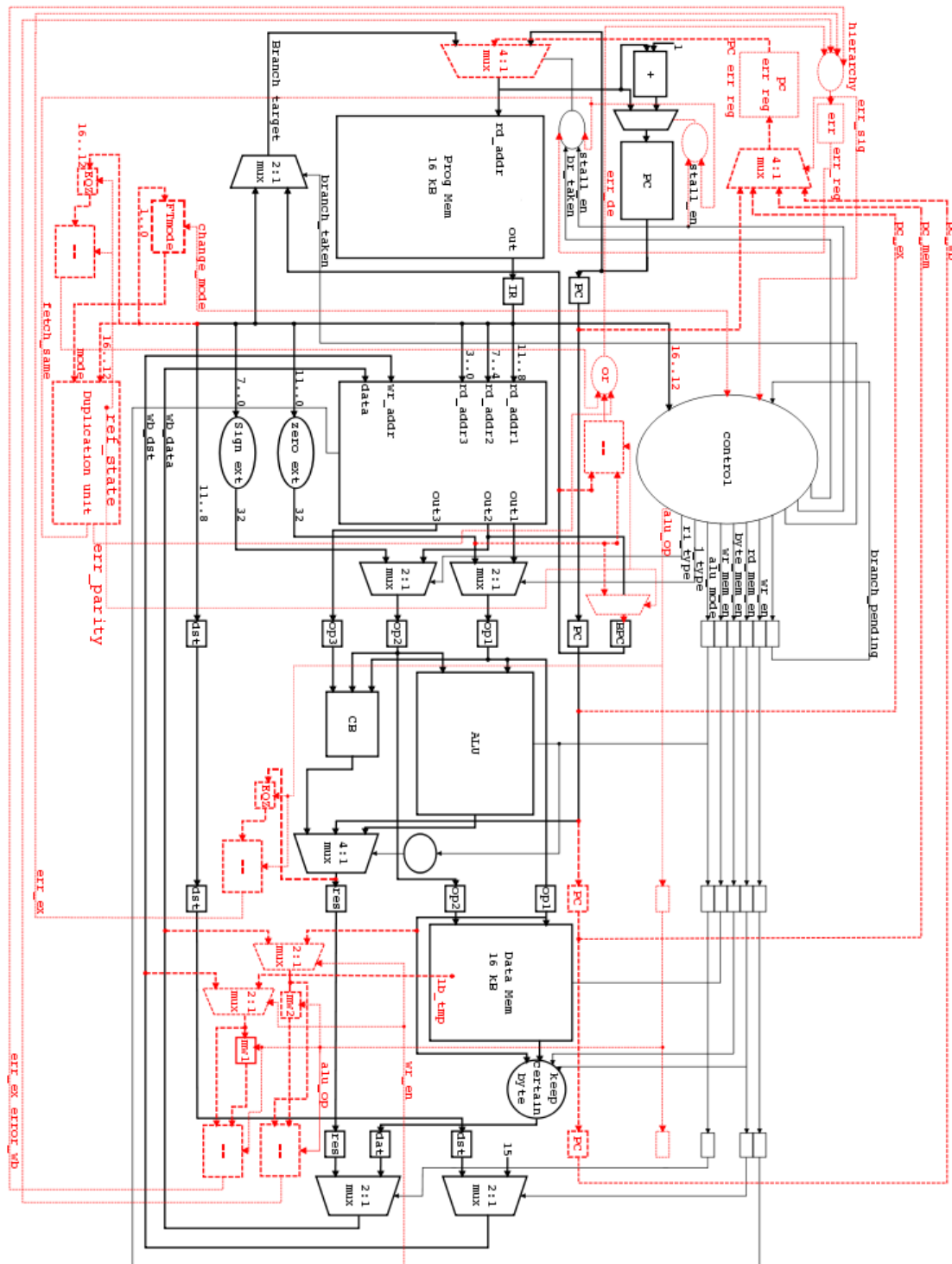


Figure 3.3: The total hardware additions made for instruction duplication.

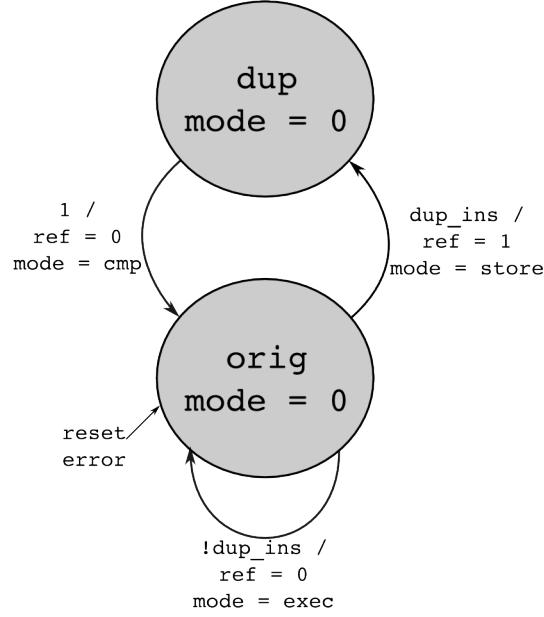


Figure 3.4: The generation of the re-fetch signal.

the `dup_ins` signal is set. When this transition occurs, the `ref` signal is set (as the same instruction needs to be re-fetched) and the `mode` signal is set to `store`. As the instruction following a duplicated instruction will always be an original, the state will always fall back to `orig` with the appropriate signals being set. Note that the `parity_error` signal is not shown in this state machine for simplicity. Every error will have the same behavior as the `reset` signal, that is, the state is reset to `orig`.

The module can be configured by using the `ft_set` instruction, which is discussed in section 3.3.4.1. Doing the duplication at run-time rather than at compile-time requires the presence of a stall-detection unit. This stall-detection unit, which is already present in the baseline, prevents data-dependencies from being violated. These data-dependencies are introduced by the duplication unit as the dataflow will not follow the dataflow graph specified at static time.

3.4.3.2 Error Detection By Comparison

Depending on the signal originating from the duplication unit, the result is stored, compared or simply executed (i.e. the result is neither stored nor compared). The error-detection unit(s) consist of a register to hold the result of the original instruction, and a comparator to compare this result to the result of the duplicated instruction. The layout of such a unit is depicted in figure 3.4. In this figure, the result (in this case, the branch/jump address) of the original instruction stored in the BPC register and is compared to the result from the duplicated instruction in the next cycle.

All committing results with associated stage and size are as follows:

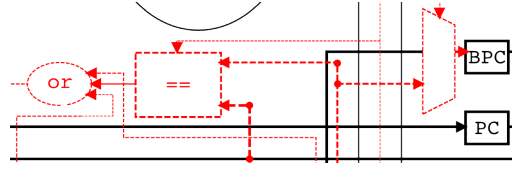


Figure 3.5: An error detection unit.

1. last executed operation (DE, 5 bits)
2. jump- and branch target address (DE, 32 bits)
3. branch result (taken or not taken) (EX, 1 bit)
4. store-target address (MEM, 32 bits)
5. store data (MEM, 32 bits)
6. write-back data (WB, 32 bits)
7. write-back destination (WB, 5 bits)

Rather than adding seven registers / comparators to the processor, it is noted that two adjacent stages (e.g. MEM / WB) cannot both be processing an original / duplicate instruction at the same time. This allows the resources between two adjacent stages to be shared. In particular, the error detection unit between the store target destination and write-back destination, and the store data and write-back data signals are shared. Since the branch result (taken or not taken) is only a single bit, sharing the hardware with the decode stage would not be too beneficial (considering only a single flip-flop and an and-gate are needed to check the branch result). As such, the error-detection units for comparing “store-target address” and “write-back data” (4 and 6) can be combined, as well as the error-detection units for comparing “store data” and “write-back destination” (5 and 7).

Finally, it should be noted that each individual stage also has a flip-flop added to it. In case the operation corrupts (i.e. it is not known if the stage has received an original / duplicate instruction), there is no way of telling if only the current stage needs to be flushed (as with an original) or also the next stage. The flip-flop keeps track of whether the previous executed instruction was an original or a duplicated instruction, and as such guarantees the pipeline is properly flushed.

3.4.3.3 Error Correction by Re-fetching

With modules in place for duplication and associated error detection, the last hardware addition for treating soft faults is the error-correction unit. Correction is done by means of re-fetching from the address associated with the instruction which generated the corrupted result. As such, changes are made to the fetching stage, as depicted in figure 3.5. The operations performed by the module will be presented using this figure.

First, the left-most signals correspond to the incoming error signals coming from the

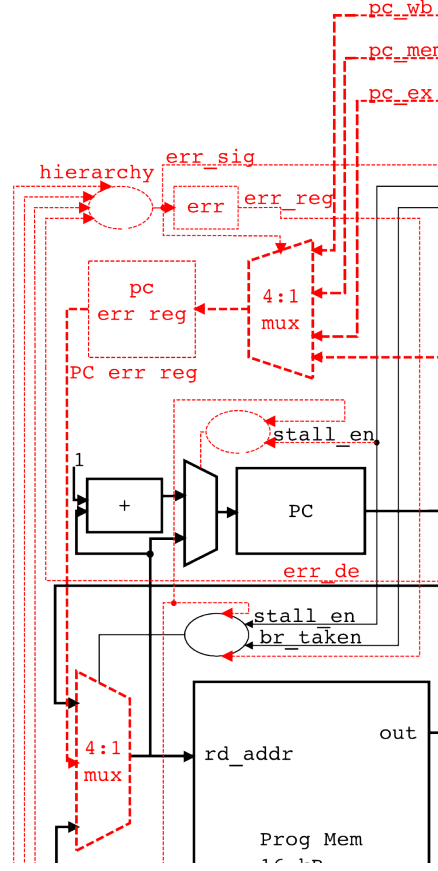


Figure 3.6: Error correction by re-fetching.

decode, execute, memory and write-back stages. The error signals are handled using a hierarchy: If more than one error signal occurs (e.g. both the decode and write-back stage report an erroneous result), the error signal originating from the latest stage in the pipeline (in the case of the example, this is the write-back stage) will take precedence. This ensures that no erroneous results can be committed. When one or multiple of the error signals are set, the “err_sig” signal is set to the latest occurring error signal. This, in turn, causes the following actions to occur:

1. The control unit is forced to flush the appropriate pipeline stage(s),
2. The “err_reg” register is set to “1” (meaning that an error has occurred),
3. The “PC_err_reg” register is set to the appropriate address. This register is an extra register dedicated to storing the target address. The address corresponding to each instruction is buffered throughout each pipeline stage (as shown in figure 3.2).

However, other than flushing the pipeline, the re-fetching is not done immediately in FT/SF.

Instead, re-fetching is done in the next cycle, meaning error recovery is now taking an additional cycle. However, as errors are only expected to occur every-so-often, this is not expected to be a major overhead. Re-fetching is done by both updating the PC to the value currently stored in the “PC_err_reg”, and similarly fetching the correct instruction from that location.

Note that while the FT/SF/FP re-fetching logic is slightly altered when compared to FT/SF, the error hierarchy and target-address selection are the same.

3.4.4 Software Additions

This section will describe both the addition to the architecture to allow us to set the duplication mode (section 3.3.4.1) and the duplication policies used to evaluate the design (section 3.3.4.2).

3.4.4.1 The `ft_set` Instruction

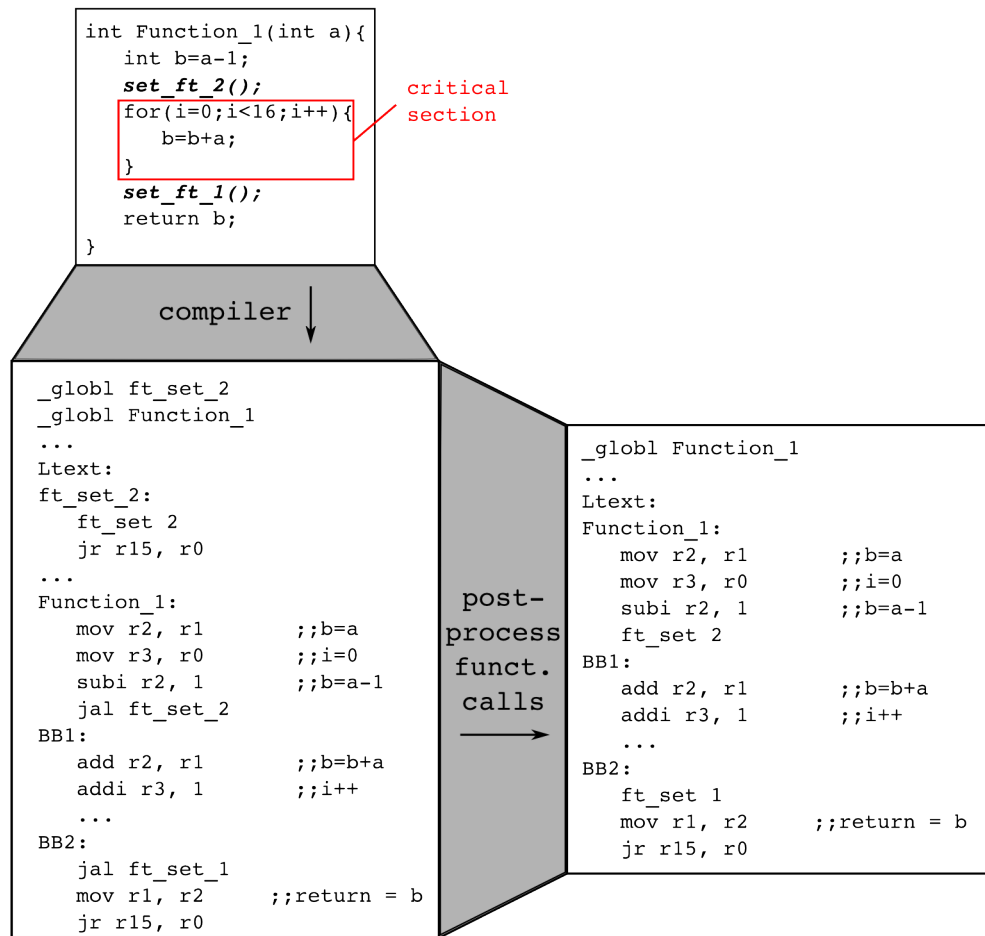
As can be read in section 3.3.2, three different duplication modes can be applied to the application: *no duplication*, *control-flow operations* and *full duplication*. To achieve this behavior, the FTmode register, residing next to the duplication module, has to be set to 0, 1 or 2 respectively. This functionality is achieved by the custom ISA instruction `ft_set`.

In Synopsis Compiler Designer (the compiler generator used for automatic c-compiler generation) there are two ways of scheduling such an instruction: Using inline assembly, or using a function call to a Compiler Known Function (CKF). While inline assembly is the straight-forward approach (as the designer can specify the instruction in assembly level), this requires an inconvenient block of code in the C-program. Instead, it is chosen to use a function call (`set_fault_tolerance#()`). However, the actual function call, and the to-be-called function are removed when post-processing the assembly file, in order to prevent the performance drop which would occur if it were actually implemented using a function call. While this requires more work than by simply using inline assembly, it provides more readable code for the programmer. The process is pictured in figure 3.6.

3.4.4.2 Duplication Policies

As both an example and a way to measure the overheads and added fault tolerance, four duplication policies are developed. While the designer is free to add the `ft_set` instruction in whatever desired way, these policies are very general and operate on a per-program basis. The policies are:

1. No duplication (`ft_set 0`). This mode is used for maximum performance, as no fault-tolerance is added. It is expected this scheme is only used in biomedical applications if full software-based solutions are already in the compiled program, or the program simply requires no fault tolerance. As such, the processor does not need to do any additional checks.
2. Control-flow operations (`ft_set 1`). In this mode, no control flow error can occur from inside the block to another block. Also, the branch target of the final in-

Figure 3.7: Post-processing the `ft_set` function call.

struction in a basic block cannot corrupt. However, the instructions involved in for example updating a loop variable are not duplicated, which may still cause control flow errors to occur. This mode can, for example be used to support a software level technique in which fault tolerance is already added to the program, except for the hard-to-implement control flow checking at the end of a block.

3. Full control flow (`ft_set 1`, and setting `ft_set` to 2 when updating loop variables and other variables related to control flow). In this mode, no control flow errors can occur. This is of interest to applications where the control flow needs to be guaranteed, yet performance is also a key metric.
4. Full duplication (`ft_set 2`). This mode is used for maximum fault tolerance, at the price of a decrease in performance. Applications which are safety-critical will generally require such an approach.

3.4.5 Summary

In this section, the implementation specifics of the technique to tolerate soft faults by instruction duplication has been presented. Error detection is done by duplicating an instruction and comparing the result generated by the original instruction to that of the duplicated instruction. When an error has occurred, the instruction at the address where the error has been found is re-fetched and re-executed.

A number of duplication modes are created (none, control-flow instructions only and full duplication) to allow a trade-off between performance and fault coverage. Four associated duplication policies have been created as both an example and to evaluate the overheads and added degree of fault tolerance.

3.5 Hard-Fault Detection: Instruction-Based Testing

3.5.1 Introduction

As mentioned in section 3.2, there are numerous stalls in programs compiled for SISC (and thus also for the fault-tolerance enhanced version). When applied to the benchmarks used in SISC, which will be presented in section 4, it is shown 5219 out of 14606 instructions are nops, divided as follows:

- Single nops: 657
- Double nops (two nops in a row): 415
- Triple nops (three nops in a row): 1244

It may seem that the majority of stalls are due to the triple nops. However, reported here are the total amount of nops in the program, not the time spent on executing these nops (i.e. a single nop running in a loop is only counted as one single nop). Each function starts with storing register contents on the stack and ends with reloading these contents in the appropriate registers. Due to the fact that this has to be done sequentially (as the stack pointer has to be updated before new data can be loaded to the address in the stack pointer) and each write back costs three cycles, three nops are scheduled between each of the required instructions. As such, the majority of nops are scheduled in the start-up and reloading segments of the function, and this explains why triple nops are occurring so often. It should be noted that the technique described in section 3.4.2 is not affected by the sequence size of the nops.

The majority of the execution time is spent on the actual operation a function provides, in which only a marginal portion of stalls are scheduled, as more parallelism is available (thus, other instructions can be executed when a result is not yet available).

For scheduling test sequences, it does not matter much where the nops are: as long as the test sequences are executed at some point during the program, a check for hard faults is executed. As such, above reports should be linked to the amount of test sequences which can be scheduled rather than the performance overhead added by these nops.

By scheduling instruction-based tests instead of running nops, additional fault detection can be gained at zero performance overhead. In Instruction-based testing two values

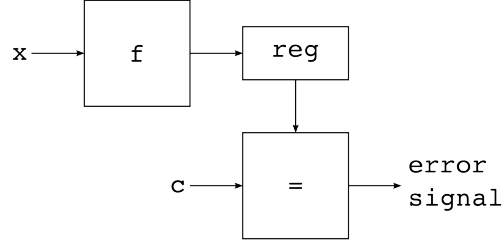


Figure 3.8: Block diagram representing instruction-based testing.

(the calculated result and the expected result) are compared in order to determine if an error has occurred or not. However, there are two key differences when compared to the aforementioned instruction-duplication technique:

1. Rather than comparing on a per-instruction basis, an entire sequence of instructions (the test sequence) is executed,
2. The test value used in the test sequence is known at compile-time, and is not recalculated by additional operations.

The block diagram representing this testing method is depicted in figure 3.7. It is shown that the input(s) x are processed by function f , which stores the result in a register. This value is then compared to a constant value to determine if an error has occurred.

In this section, the details related to this instruction-based testing technique are discussed. First, the designing of test sequences, the used test sequences, and the scheduler are discussed in section 3.4.2. In section 3.4.3 the methodology and design of self-test instructions is given, to cover the paths which are untestable by test sequences.

3.5.2 Test-Sequence-Based Testing

In this subsection, the details related to the test sequences are discussed. In section 3.4.2.1, the methodology followed to design **a minimal set** of test sequences covering all *testable paths* in the processor is presented. An additional instruction which has been added to the ISA to greatly simplify the test sequences, `bneqz`, is presented shortly in section 3.4.2.3. Section 3.4.2.4 will present an example of how this methodology is followed. As the list of test sequences is quite large, the full list is added as an appendix (Appendix C). Finally, section 3.4.2.5 will present the scheduling policies for the test sequences.

3.5.2.1 Methodology

Creating test sequences depends on the level of information available. In particular, three super-levels can be distinguished:

1. Knowledge of the ISA only. In this case, the processor is modeled as a black box,

2. Knowledge of the ISA and RTL of the processor. In this case, the processor is modeled as a gray box, as more detail is available, yet the internal functioning of for example the structures in the RTL schematics are not known.
3. Knowledge of the ISA and the gate-level details of the processor. In this case, the processor is modeled as a white box, as all information related to the functioning of the device is available.

As has been reported in section 2.5.4.4, more structured tests can be applied with more information available, generating smaller tests with a higher coverage. In particular, in case gate-level details are available full structured testing can be applied using, for example, the path sensitization method or the D-algorithm which are two well-known techniques for test pattern generation [13]. On the other hand, when this information is not available (e.g. only the ISA is known), functional testing has to be applied. In functional testing, all inputs have to be tested, as there is no way of telling whether or not a fault will propagate to the output under certain conditions.

The fault-tolerant processor under investigation is modeled as a gray box using the RTL model, as we are interested in architecture-level techniques for fault tolerance. The method for generating the test sequences is as follows:

1. The activated datapath is determined for each instruction;
2. For each of the instructions, the *controllability* and *observability* of the signals in the processor is determined;¹
3. Out of all possible combinations, a test sequence is generated containing the fewest possible instructions to observe an expected stuck-at fault, and determine its fault coverage;
4. Step (3) is repeated until all testable paths in the RTL-schematics of the processor are covered.

While the processor is strictly speaking not structurally testable, some parts of the processor are testable using pseudo-structural tests even without the gate-level details of the design due to the simplicity of the activated datapaths. Such units include the register file (e.g. write 0 to a register and compare it to 0) and the simple-logic units residing in the EX stage such as and, or, xor. Unfortunately, this is not true for all components, particularly for components which have an interplay between bits (such as the adder or shifter units present in the ALU). For these components, a more general, functional test sequence has to be used. In other words: as there is no way of knowing what the actual connections inside the units are, there is no way of telling if a fault will occur and / or if it will be masked. As a result of this, real guarantees of the correctness of the component can only be given by testing all the inputs and observing all its corresponding outputs.

¹The controllability of a signal is the ease of setting the signal to the desired value. The observability is the ease of which a signal can be observed, e.g. by writing it to a register.

This is of course an involved task, and considering the fact the ALU in the baseline design has two 32-bit operands (plus a carry-in signal), this would require $2^{65} = 36,893,488,147,419,103,232$ tests, which is not feasible for at-runtime testing due to the massive overhead this would induce. Nevertheless, *general* test sequences can still be designed in case at some point gate-level details become available. These general test sequences use immediates to specify the inputs of these units. When no details are available the test sequences use a random variable, which can be filled in with a more concrete number when gate-level details are available. The resulting sequences can be viewed along with the other sequences in Appendix C.

Changes have to be made to the instruction memory in order to replace the nops by test sequences. Recall that in SISC, the instruction memory size is reduced by injecting nops in the hardware. However, as scheduling test sequences will consume these nops, this compression is less viable. As such, the instruction memory has to be increased. This results in the need for a new jump-and-link instruction, as the 12-bit immediate is not large enough to target all addresses. The new jump-and-link instruction is discussed in the next section.

Finally, a `bneqz` instruction is added to the processor, to reduce the overhead of comparisons for determining if an error has occurred or not. This instruction is discussed in section 3.4.2.3.

It should be said that it is also possible to add hardware to the processor in order to improve testability. This can be useful in particular when long test sequences are required to check for a potential fault, e.g. the tested component is hard to control. This topic will be revisited in in section 3.4.3.

3.5.2.2 Jump-And-Link Extension (jalo)

As mentioned in the previous section, the scheduling of test sequences requires the instruction memory size to be increased. This is due to the fact that these sequences are scheduled instead of nops, and as such compressing the binary by means of injecting the nops in the hardware is not viable. Moreover, this causes the jump-and-link address (*format: jal #target (12 bit immediate)*) to not be able to cover all addresses. There are two ways to deal with this problem:

1. Load the address into a register in the register file, and have the jump-and-link instruction work based on a register access rather than directly the target address,
2. Add an additional register to hold the upper part of the target address.

Since 15 bits are enough to hold the target address for the new instruction memory, the choice is made to add a small register (3 bits) to which the upper part is first written, after which the jump and link can take place. At the cost of the additional register overhead (presumed small due to the small size), this allows less pressure on the register file, and no additional scheduling needs to be done to avoid nops caused by data-dependencies between the load and use operations.

An additional instruction `jalo` (*format: jalo #target (3-bit immediate)*) is created to occupy this initial load. As such, jump-and-linking is done as follows:

```
jalo #targetHigh
jal #targetLow
```

3.5.2.3 The branch-when-not-equal-to-zero (bneqz) instruction

As depicted in figure 3.7, the results of a test sequence need to be compared to the expected results. When using the baseline ISA however, this can be a tedious task. Assume the test sequence generates the constant c as a result. Then, the following steps have to be taken to compare the functions' result to the expected result:

1. Load c into a register;
2. Do a set-equal operation on c and r , storing the result to c ;
3. Branch to the error-handling address in the assembly file using the beqz function on c .

It should be noted that loading a 32-bit constant to a register already consumes 7 instructions, due to the fact that we can only load or add an immediate 8 bits at a time. This means that the total process would take an additional nine instructions, which causes even the longest test sequence to more than double its size. In other words, a high amount of time is spent on observing the fault.

Instead, an attractive alternative is to make sure the expected constant is zero. In this case, two instructions are still needed (set if equal to zero, and branch-if-equal-to-zero (beqz)). As most test sequences require 3 instructions to generate the result (equal to zero, see appendix C), the overhead of using the two instructions is 66.67%.

However, it is clear that these two instructions (invert and branch if equal) can very simply be represented by one instruction: branch if *not* equal to zero. As this only requires an inverter and a mux to change the beqz instruction to the bneqz instruction, the hardware overhead is minimal. Furthermore, the critical path is not affected, since the branching unit is not on the critical path.

In essence, adding the bneqz instruction allows one instruction less to be used per test sequence which allows more test sequences to be scheduled.

The instruction format of the bneqz instruction is:

```
bneqz rX, #addr
```

in which rX is the register we compare to, and $\#addr$ an 8-bit immediate. The address is absolute, and as such, it needs to be one of the first 255 instructions. Due to the high amount of test sequences which are scheduled, relative branches are not suitable using the 8 bit target address. An alternative would be to branch to an address stored in a register - but this would require additional hardware, or having an existing register be occupied throughout the program. As such, the instruction is not implemented by storing a target address in a register.

The resulting processor, with increased instruction memory size, the `bneqz` instruction and the change to jump-and-link will be known as FT/SF/HF throughout the rest of this thesis.

3.5.2.4 Example

As an example, the methodology described in section 3.4.2.1 is applied. In particular, we will try to create a stuck-at-1 fault for a register in the register file. Generating such a stuck-at-fault requires us to make sure the register is set to 0, and if it is not equal to zero, an error is detected. This example is chosen as it is the simplest possible sequence which can be created due to the fact that it operates straight on the operand of the `bneqz` instruction. However, it should hopefully demonstrate the methodology.

First, we are required to determine the activated datapaths for all instructions. While not shown here, we can determine the controllability and observability of each signal from the datapath activated for the instruction. The list resulting from this effort would be excessively large and not suitable for reporting here. For now, we state that the `li` instruction is an ideal candidate for writing 0 to a register (i.e. the controllability of the register is ideal). It should be noted that using other operations, such as a `mov rX, r0` / `xor rX, rX` are also viable alternatives, as they also require only a single instruction to set the register to 0 (see Appendix B).

Now that we have everything setup for generating the fault, a `bneqz` instruction is all which is required to determine if a fault has occurred. The total sequence is therefore:

```
li rX, 0
bneqz rX, error
```

In which `rX` is the register we are targeting. Note that the sequence is generic: It is up to the scheduler to fill in the register number. In particular, this sequence also covers the following errors (fault type - signal name - signal stage):

- SA1 IR(7 downto 0) (DE)
- SA1 (signextended) (DE)
- SA1 (mux → OPR2) (DE)
- SA1 (op1 → out1) (DE)
- SA1 in the register file (output) (DE)
- SA1 (IR 11 downto 8, register selection, depends on `rX nr`) (DE)
- SA0 (IR 11 downto 8, register selection, depends on `rX nr`) (DE)
- SA1 (OPR2) (EX)
- ALU: SA1 for result for `li` / `mov` type (EX, MEM, WB)
- SA1 Result data line (EX)

- SA1 mux $\rightarrow$ res (EX)
- SA1 res $\rightarrow$ memres (MEM)
- SA1 MEMRES $\rightarrow$ wb_data (WB)
- SA1 WBDATA (shown in DE)
- SA1 (all wb register lines, and associated pipeline registers, depends on rX nr) (ALL)
- SA0 (all wb register lines, and associated pipeline registers, depends on rX nr) (ALL)

3.5.2.5 Scheduling

In order to make use of the test sequences generated following the methodology mentioned in the previous sections, they have to be properly scheduled in the program code. There are two ways of doing this:

1. Changing the compiler, or
2. Post-processing the assembly file.

Making changes to the compiler has several advantages. First and foremost, the required information (free slots to schedule in, and which register can be used) is readily available, as the scheduler already requires this information. In this sense, an optimal schedule is simple to achieve. Furthermore, it provides an elegant solution: In the CoSy Compiler (a superset of the used Synopsis Compiler Designer), each of the so-called engines (functions performing tasks on the intermediate representation of the compiler) can be run or ignored by using simple command-line options.

However, this also means that a whole new tool (CoSy) has to be mastered. Especially considering the fact that the back-end generator is potentially the hardest part to master (as the documentation is less structured, and a lot of information is to be processed and taken into account), this is a rigorous task. Given the time allocated for this project, the entire scheduler is made in the form of a C program and operates by means of post-processing the assembly file. As a result, the scheduling might not be optimal, as it is not always known from inspecting the assembly file what register is free to use throughout the program.

The scheduling takes place in the following steps:

1. Lexical analysis (tokenizer). The resulting information consists of a set of instruction types, used registers, used immediates and determining the basic blocks with associated control flow graph. The information is stored in an array containing the intermediate representation. Each nop-instruction also has a nop_info structure attached to it, which tracks if the nops is used by a test sequence in a later stage of the program.

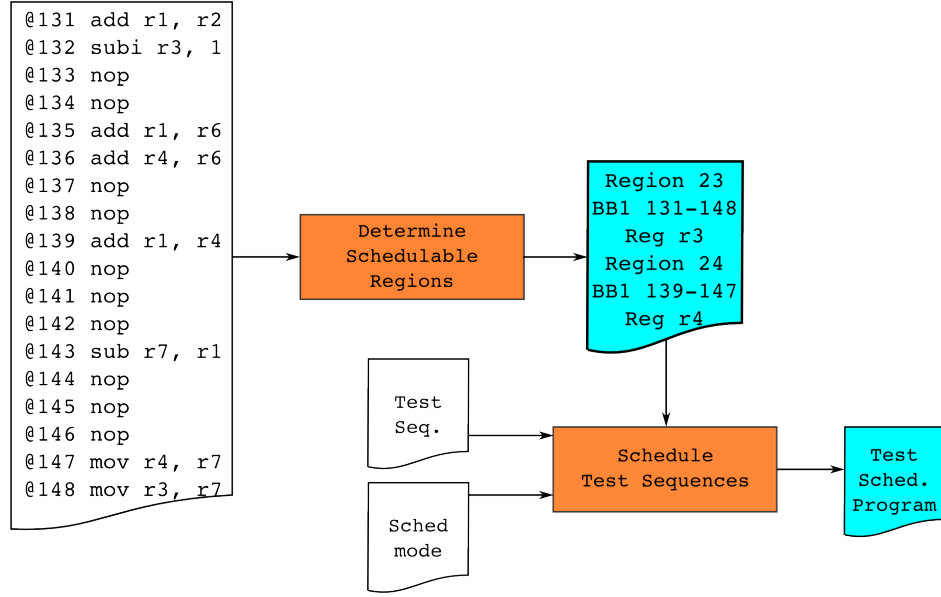


Figure 3.9: The tool flow for scheduling test sequences.

2. Determination of schedulable regions. These regions consist of the start and stop address of the region (i.e. the zone in which the register is free) and the associated register. Registers may be considered free when:
 - (a) From the last time the register is used, to the address of a mov, li or not instruction. As these operations do not care what the original contents of the destination were, we are free to manipulate the contents of this register to our own needs.
 - (b) From the last time the register is used, to a function return (jump to return address) in the case the value is callee-saved,
 - (c) From the start of a function call, to the first time the register is used by a mov, li or not instruction.
 - (d) Throughout the entire function, if the register is not used at all.
3. Schedule the instructions, based on the given schedulable regions and test sequences.

The scheduling process is depicted in figure 3.8. In an initial implementation, a nop coverage of 50-70% has been found. It is expected that if the tool was integrated in the compiler (and as such has true access to free and used register regions), this percentage would be increased.

Scheduling is done by first specifying what test sequences are desired. For example, the programmer may specify whether the scheduler must put more effort on making sure the register file is secure when for example the amount of stalls is expected or is measured to be scarce, or not schedule certain sequences as certain functional units are

never or very rarely used. Alternatively, the programmer may go for another approach, for example doing round-robin on the instruction sequences. This process can be done for the entire program (using all schedulable regions) down to per-block or even per-region level.

In particular, four scheduling schemes are created for evaluation. These schemes are:

1. Scheduling tests for the register file;
2. Scheduling tests for the register file and simple units (and, xor);
3. Round-robin scheduling using all test sequences;
4. Round-robin scheduling using all test sequences and self-test instructions (section 3.4.3).

It is expected that in the case of round-robin scheduling, the fault coverage will be low due to the high amount of functional tests. This is due to the fact that there may simply not be enough stalls available to schedule enough tests for each functional unit. Scheduling register file-only (or in particular: testing a subset of functional blocks) guarantees a high coverage for a small part of the program.

After these specifications, the scheduler will mark the nops present in the used schedulable regions. Care is taken to ensure that the data dependencies inherent to the test sequences will not result in performance loss, i.e. enough cycles are placed between each instruction. Marking modifies the `nop_info` structure of the corresponding nop instructions, which is part of the intermediate representation. Stored in each `nop_info` structure are the test sequence number, the test sequence size, the instruction number this nop will be replaced by (e.g. the third instruction in a four-instruction test sequence) and the register(s) and value(s) the sequence is supposed to use. In essence, this means that each modified nop will know how to be translated into a part of a test sequence.

When no more nops can be marked, or enough sequences are scheduled to suit the programmer's needs (i.e. each functional block are guaranteed to be correct using the current scheduled tests), the intermediate representation is handed off to an emit function. This function copies the original assembly file, with the key difference that when a nop is used for scheduling, the nop is replaced by the appropriate instruction from the target test sequence.

3.5.3 Self-Test Instructions

Certain structures are untestable using test sequences. These untestable paths are, for example, all circuits unique to control-flow operations, covering the branching unit, the BPC, the PC and the logic in the fetch stage. It should be noted that while the control unit is not directly testable, it is tested indirectly by scheduling numerous test sequences, since the results will at some point be different than expected in the case a control line has a stuck-at fault.

By designing for testability (DFT), hardware structures and associated instructions are added to the processor and compiler, to provide a direct way of controlling and / or observing specific signals or components. For example, a custom instruction may load a

value directly to the BPC register, the data of which can then be compared for stuck-at faults. It should be noted that the paths which are not testable with test sequences are small both in number and in size. Typically, adding fault tolerance specifically designed for such paths will come at a high overhead, since only a small part of the processor will make use of the added hardware. However, due to the safety constraints set by the field the processor is designed for, these paths have to be guaranteed to function correctly.

While adding test instructions for untestable paths is mandatory for detecting errors, it is also possible to apply DFT to components which *are* testable by test sequences. Such an addition may be introduced to, for example, increase the controllability or observability of the signals under test. For example, testing the adder is a rigorous task, and the test sequences reported for this unit ranging from 2 to 8 instructions long. On the other hand, this unit needs to be tested using the most sequences, due to the complexity of the circuit. In order not to waste all our resources (i.e. available nop slots) on testing the adder, adding custom hardware and instructions for testing the adder can be added. In such a way, the sequence can be simplified to a single instruction, which allows for 8 tests to be scheduled instead of one. While being a nice suggestion, the choice is made not to implement this in our design. As there is still no method of structurally testing the more complex units, massive amounts of tests are still required, which are typically not schedulable in the assembly file to begin with without messing with performance. As such, the added fault tolerance will not outweigh the added area and power reduction overheads.

The technique is, however, applied to the untestable control-flow parts of the processor. In section 3.4.3.1, our implementation for the control-flow operation testing, consisting of additional hardware and a test instruction is presented.

3.5.3.1 Control-Flow Testing

By creating an instruction, which mimics the control-flow operations yet does not commit a result (i.e. the PC and fetch address are not actually changed), we can test if the control path and associated logic is functioning correctly. In particular, the additions to test the control-flow operations, consist of a three-fold of additions:

1. **Additions to the datapath**, in order to put a value in the datapath and propagate it to the fetch stage, in the same way this happens in a jump / branch;
2. **A comparator** at the fetch logic, to determine if the value has arrived correctly or not;
3. **A custom instruction “br_tst”** to set the inputs (i.e. target address) necessary for testing above mentioned additions.

Furthermore, the PC is never available for testing for stuck-at faults, as it is in use 100% of the time. To make sure we can detect errors in the PC, we add a shadow PC as in [3], which functions exactly like the PC. A comparator checks if both the PC and shadow PC have the same value, i.e. under the assumption of single stuck-at faults, the error is detectable. The overhead for testing this register is considerable with respect

to the size of the PC, considering more than twice the hardware is required to test it. However, since the PC is the only register which needs such a detection mechanism and is a relatively small part in the micro-architecture, this is estimated to be an acceptable overhead. Furthermore, due to the constant switching in the PC, the power consumption is expected to be increased substantially. However, this method does allow detection of both soft and hard faults in the PC. The additions are depicted in red in figure 3.9.

It should be noted that adding hardware to the processor inevitably results in an increased chance for faults. For example, if the shadow PC corrupts but the PC does not, an error is detected, even though the original PC never corrupted to begin with. However, as long as the additions are small and are able to detect errors both in the circuits they test and themselves, this is an acceptable overhead. These overheads are discussed in section 4.2.

3.5.4 Summary

In this section, the implementation specifics of the technique to tolerate hard faults by instruction-based testing has been presented. First, a method is shown on how to create the test sequences, which are functional tests to detect hard faults. These sequences are scheduled by post-processing the assembly file. In particular, three sets have been proposed (*register file only*, *register file and simple functions*, *round-robin on all sequences*), which can be used depending on the availability of stalls in the program and the coverage required. A new instruction (`bneqz`) is added to simplify these tests.

Furthermore, an addition to the untestable paths which is related to control-flow-operations has been proposed in this section. To prevent a fault in the program counter to result in erroneous behavior a shadow PC is added. This shadow PC acts like a copy of the PC and an error signal is raised when the two are not equal. Additions are made to the processor to control and observe the required control signals, and a custom instruction “`br.tst`” is added to test these.

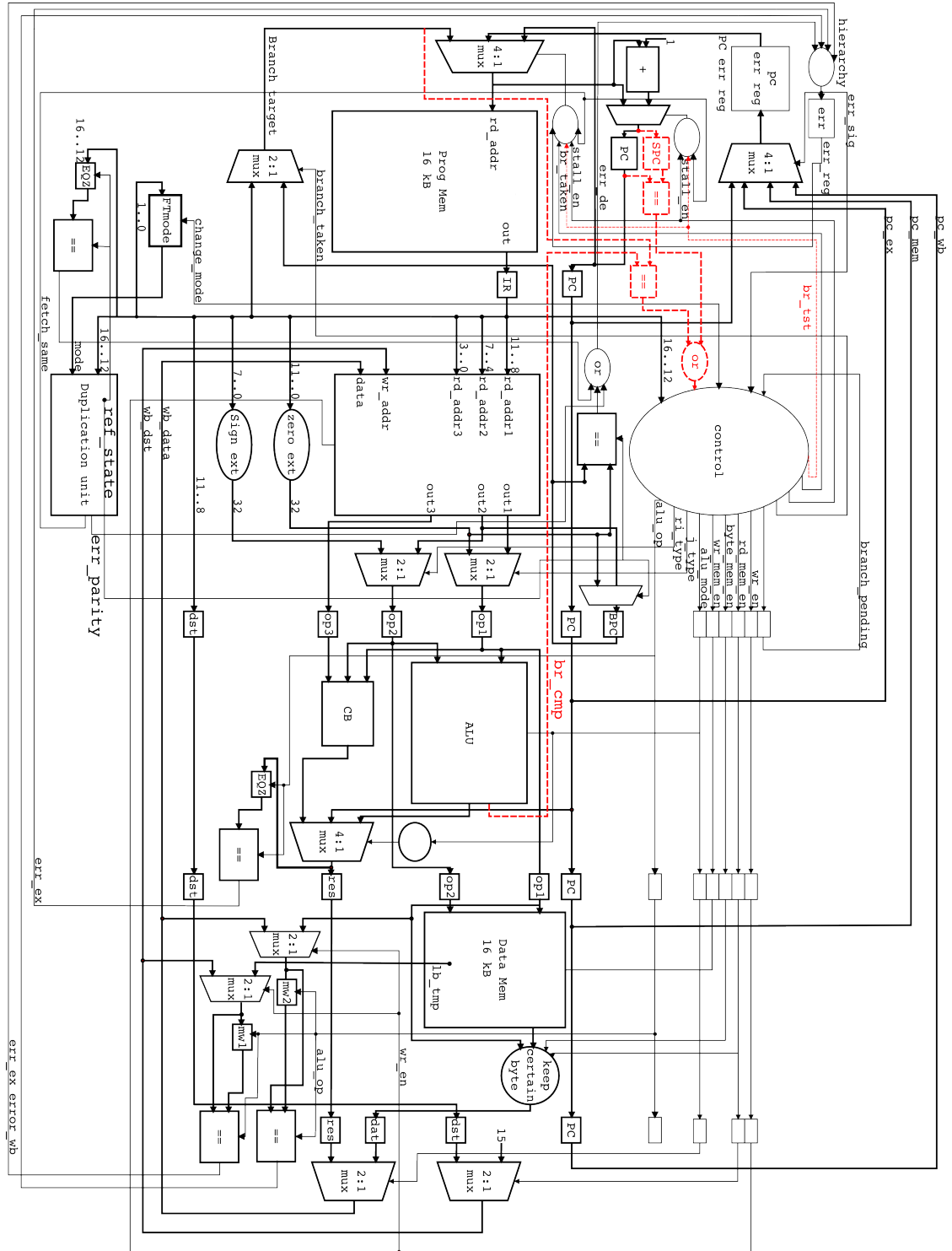


Figure 3.10: Self-Test additions for detecting errors in the control-flow logic.

In this chapter, the experimental results from the processor enhanced with fault tolerance are given. First, our method of evaluating the processor is given in section 4.1, after which the complete tool flow is presented in section 4.1.4. Finally, the results from the evaluation are given and discussed in section 4.2.

4.1 Experimental Setup

The experimental setup consists of three main attributes, which are:

1. The testing method used for soft faults,
2. The testing method used for hard faults,
3. Determining the overheads associated to each processor design and benchmark (area, power, energy and performance).

These methods will be presented in the following sections. For convenience, we summarize here the different designs, duplication policies, test-sequence schedules and the used benchmarks.

The experiments are done on the processor designs in table 4.1. The designs are encoded by means of the following acronyms:

- FT: Fault-Tolerant;
- SF: Soft-Fault tolerance (instruction duplication);
- HF: Hard-Fault coverage possible (increased memory size and instructions to facilitate test sequences);
- FP: optimized Fetching Protocol (refetch next cycle);
- ST: Self-Test additions (shadow PC, br_tst instruction).

For each of the processor functions the results are taken based on the duplication policies, which are (as discussed in section 3.3.4.2):

1. No duplication;
2. CF instr. duplication (duplicate control-flow instructions only);
3. full CF instr. duplication (duplicate control-flow instructions and the updating of loop variables);

Table 4.1: The various processor designs and features

Name	Soft Fault	Impr. re-fetching	Incr. mem. size	bneqz	jalo	shadow PC	br_tst
Baseline	-	-	-	-	-	-	-
FT/SF	x	-	-	-	-	-	-
FT/SF/FP	x	x	-	-	-	-	-
FT/SF/HF	x	-	x	x	x	-	-
FT/SF/HF/ST	x	-	x	x	x	x	x

4. full instr. duplication (duplicate all instructions).

Furthermore, the following schedule sets are used for the test sequences (as discussed in section 3.4.2.5):

1. Register file only;
2. Register file and simple operations (and, xor etc.);
3. All tests (round-robin);
4. All tests (round-robin) with the self-test instruction. Note this schedule is only run on the FT/SF/HF/ST processor.

Finally, the benchmark is the same as in SISC. This is an encryption program, which is split over several functions. In particular, the program is a combination of a number of functions, all run as separate benchmarks. As the encryption is symmetric, the decryption and encryption function are the same[1]. The code belonging to each benchmark is given in Appendix E. The encryption program consists of:

1. Running the CRC benchmark 132 times,
2. Running the decryption benchmark 1 time,
3. Running the encryption function 131 times,
4. Running the emptyMain function 1 time,
5. Running the keys benchmark 2 times.

4.1.1 Testing Method (Soft Faults)

As can be read in section 2.4, soft faults are modeled by transient faults, which exist only for a single cycle. However, multiple faults on one signal line are allowed in our model, e.g. a whole dataline may corrupt, rather than just a single bit. As all processor designs use the same method to detect soft faults (excluding the shadow PC added in FT/SF/HF/ST), the error measuring is done on FT/SF. The technique described here is only used for simulation. The hardware mentioned here is not synthesized and has no impact on the synthesized results which will be discussed in section 4.2.2.

To determine the amount of faults manifesting into errors in the FT/SF(/xx) design(s), we replace the contents of a signal with a random value. The random value is generated by an LFSR (linear feedback shift-register).

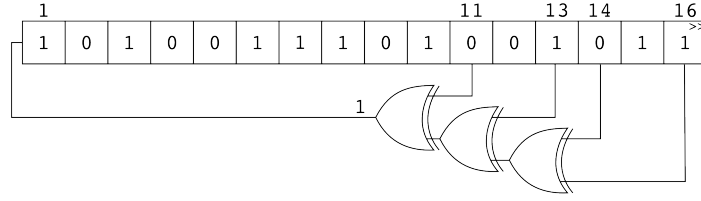


Figure 4.1: A 16-bit maximal-length LFSR.

An LFSR operates by shifting the register one bit to the right in each cycle. To create a stream of values, the most significant bit is connected in a specific way to the other bits in the register. This specific way is called the feedback polynomial. For example, the most-significant bit can be set by applying an or-function over the lowest two bits.

In certain cases, the LFSR will go through a long list of states ($2^n - 1$) called the maximal length, and the corresponding polynomial is called the maximal-length feedback polynomial. Given in figure 4.1 is the maximum-length LFSR for an LFSR of size 16. As the maximum size of our signals is 32-bits, a 32-bit LFSR is used. Other maximal-length feedback polynomials can be found at [31].

The LFSR is connected to all signals under investigation. In order to simplify testing, debugging and to show which faults are most likely to corrupt the output, all possible faults have been grouped in five separate sets. These sets are:

1. The fetched (used) instruction,
2. The registers in the register file,
3. Control signals and control path,
4. Other signals in the datapath (e.g. the result of an ALU operation),
5. The fault-tolerant processor additions (to guarantee the additions themselves do not cause additional errors).

Note that the faults in the PC are not modeled in any of the fault sets. This is due to the fact that such behavior will be similar to the behavior in the fetched instruction fault set, and thus can be derived from the first fault set. By comparing the overheads of the FT/SF/HF processor to the overheads of the FT/SF/HF/ST processor an estimation will be made on how much fault tolerance would be added to the processor using a shadow PC and how much overhead a shadow PC adds.

When a fault is generated, one signal will be replaced with random bits of the LFSR. The signal is chosen based on a modulo operation on the LFSR value, and takes into account the different sizes of the test signals and the fan-in / fan-out, as a higher fan-in / -out corresponds to a higher amount of possible faulty nodes. For example, if a test set would consist of the load-word control signal(1 bit, fan-out = 1, fan-in = 2) and the store-word control signal (1 bit, fan-out = 1, fan-in = 1) the ratio of generating errors will be 2:1. The test-sets are grouped in such a way that signals of roughly the same size are grouped together in order to simplify testing, yet get accurate results.

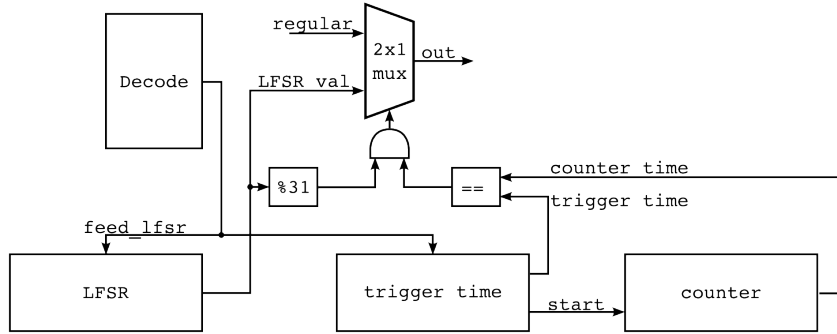


Figure 4.2: Replacing a data signal with the LFSR value.

For testing purposes, a small test program has been created, which takes roughly 5.000 cycles to complete on the Baseline. As many tests will be required to have some sensible results, the test program is small, yet contains all instructions and all general cases we are interested in (for example, a branch followed by a jump, or an arithmetic instruction followed by a branch). As the technique is architecture-level and treats instructions of the same class (e.g. arithmetic) in the same way, there is no need to test each possible sequence of instructions.

A custom instruction has been added to be able to feed the LFSR from the test program (format: `feed_LFSR #feed_value (12 bits)`). Feeding the LFSR is done by executing 2 `feed_LFSR` instructions, starting a counter (which is also added to the processor) which counts up to the trigger time. For each test set, the following steps are executed:

1. A random value is generated in the bash shell (terminal), which is the trigger time. The trigger time is guaranteed to be within the total time it takes the program to finish execution, and takes into account the duplication mode,
2. This value is fed to the LFSR using the `feed_LFSR` instruction,
3. At the trigger time, the LFSR will overwrite a random signal in the fault set,
4. When execution completes, or a time-out occurs (i.e. execution does not complete, and as such we probably had a control flow error along the way), the results are compared to the known correct result of the test program. This is done by analyzing the `.vcd` file generated during the test and comparing this to the known correct `.vcd` file.

This process is run for each test set 4.000 times for each duplication mode, meaning a total of 16.000 tests are run per fault set (or 80.000 in total). In order to compare our results to the baseline in a fair way, the same tests are also executed on the baseline. Figure 4.2 provides a visualization of the hardware additions required for generating the faults. When a `feed_LFSR` instruction is decoded in the decode stage, the value is fed to the LFSR and the trigger time register, which starts the counter. When this counter reaches the trigger time the value of the target signal, derived

by a modulo operation on the LFSR value, is replaced by the LFSR value. Note that it is assumed in this figure the width of the target signal is the same size of the LFSR. If this is not the case, only a number of bits of the LFSR replace the target signal.

Other than the fault coverage for all soft faults, we are interested in how many control-flow faults are covered. After all, some of the proposed duplication modes specifically target control-flow errors. To check if a fault has been caused by a fault in control-flow, each basic block starts with an instruction to add a unique value to r0. As r0 is not used in the test program, this is allowed. When the program executes in the correct order, the value in r0 will match the expected value. As such, the result in r0 is compared to the expected value. Note that r0 may corrupt due to, for example, a corruption in the adder when the value is added. As such, this scenario is explicitly prevented by not allowing the LFSR to overwrite a value if r0 is the destination target. Note that the number of control-flow errors is a subset of the total number of errors.

It is, of course, possible to run a larger test program and run more tests per program. However, as the technique for soft faults only looks at the archetype of instructions (e.g. arithmetic or control-flow), we do not need to guarantee that every possible instruction combination is guaranteed to happen. For example, an and-instruction followed by an xor-operation is treated the same way as an and-instruction followed by a load-word operation.

It should be noted that by replacing random values with bits from the LFSR does not always guarantee a fault will manifest as an error, even in the case of the Baseline processor. After all, if the signal on which the fault is generated is not used at that particular time, the fault is masked. However, as the fault sets taken are the same in both the Baseline and the fault-tolerant processor, the chance of fault masking is the same for both processors. In other words, comparing the results of the fault-tolerant processor to the Baseline processor effectively cancels out the effects of masked faults.

Using the number of faults manifesting into errors as described in the previous paragraphs it is possible to determine the added fault tolerance. To do so, we compare the number of faults manifesting into errors in the SF/FT(xx) design(s) to the number of faults manifesting into errors in the Baseline design.

4.1.2 Testing Method (Hard Faults)

Hard faults are modeled as behavioral faults, as stated in section 2.4. Getting accurate hard-fault test results is time consuming operation. In classic fault injectors, a faulty processor with a single stuck-at fault is generated. This can be done in modelsim by, for example, forcing a signal to a value by a do-script. Due to the complexity and the number of signals in the Baseline processor, this would result in several millions of faulty processors, all which have to be tested for completeness. In turn, an accurate method for testing hard faults is not possible. Instead, it is shown that each of the test sequences work for the expected cases. For each test sequence, the following is done:

1. For each of the instructions in a test sequence, a random signal / register it activates is put on a stuck-at 1 or stuck-at 0;

2. The same test program as in section 4.1.1 is used, but now the test sequence is scheduled in the test program;
3. The test program is run, and it is checked whether the fault is detected or not. This is done by checking if the PC is stuck at the error trap address from the `bneqz` instruction.

This is a very coarse-grain technique and, as such, the results will simply indicate if the test sequences “pass” or “fail” the test. Future work would include a more sophisticated evaluation to hard faults.

4.1.3 Determination of Overheads

Except for the fault coverage, the overheads need to be measured. After all, while fault tolerance is important, it needs to be done with limited resources due to the constraints imposed by the biomedical environment. Furthermore, comparing the added fault tolerance to the added overheads is an important aspect in determining if the design is viable for a biomedical platform and if this design is more viable than other solutions. The same tools and benchmarks are used as with SISC (Baseline) and the design is evaluated at the same target frequency (20 MHz). All the steps and tools involved are briefly discussed in the following section.

4.1.4 Tool Flow

This section will provide a step-by-step guide to what tools, scripts and files are required and used for verifying and evaluating the processor designs. The toolflow used throughout the project is depicted in figure 4.3.

Starting at the top-left, the two main inputs used while designing the architecture are shown: `.lisa` files, which together form the processor description written in lisa, and the compiler description (`.cfg`) files. The `.lisa` design files can be compiled by Synopsys Processor Designer into synthesizable VHDL code. Similarly, Synopsys Compiler Designer can use the `.cfg` files to generate the lcc compiler (the c-compiler generated for the target architecture). Note that programming the `.lisa` and `.cfg` files is facilitated by Synopsys Processor Designer and Synopsys Compiler Designer respectively.

While rare, it may happen that the generated VHDL files are not correct (see Appendix D). Errors in the VHDL files will be picked up either during VHDL compilation, or during design verification. Design verification is performed by compiling small test programs for the processor, and feeding the test programs through the modelsim simulator. Note that it is also possible to manually write the memory files for design verification - although this requires a lot of manual labour. In our setup the lcc compiler is set to create an assembly file rather than a program binary, as to allow some post-processing on the generated assembly file. In the current design, post-processing options are:

- Default: Replacing the `ft_set` function calls by the `ft_set` instruction (as can be read in section 3.3.4.1);
- Removing nops;

- Scheduling test sequences;
- Creating the LFSR feed for the soft-fault testing method.

Any of these post-processing options can be chosen to be executed per-program compilation. After post-processing, the `lcc` compiles the assembly file into a program binary. The `exe2txt` tool transforms this program binary into memory maps, which are the files required by the `modelsim` simulator to emulate the instruction and data memories. As can be seen in the toolflow, errors may occur during simulation (i.e. the expected and actual behavior are not the same). These errors may be due to an incorrect `lisa` model, which means the processor (or compiler) description has to be corrected. These errors may also be due to the faults injected for testing fault tolerance (see section 4.1.1). In this case the errors are used for fault-coverage measurements, which is done by a script.

When the design is running correctly in the simulator, it can be synthesized by Synopsys DC as Processor Designer creates synthesizable VHDL. Synopsys DC requires the VHDL files as input, along with library files which are specific to a certain technology. In our particular design, we have used the UMC 90nm library. This technology is used as 1) it was available and 2) SISC has been synthesized in this technology and thus this allows for a fair comparison. A few options can be set, including the generation of reports. As a result, the synthesis report (for debugging), the area report and the netlist VHDL files are generated. As Processor Designer produces synthesizable VHDL code, no errors are expected during synthesis.

Using the netlist, it is possible to get accurate power measurements. While power simulation is possible in Synopsys DC, the power simulator produces inaccurate output, as it simply assumes half of the bits in a signal are flipped in a clock cycle. By extracting the activity switching file from `Modelsim`, it is possible to get more accurate power consumption reports. To extract the activity (`.vcd`) file, the netlist, the library and the memory simulation files need to be compiled and simulated in `modelsim`, using the same testbench which is used during design verification. We use a clock frequency of 20 MHz, which is our target frequency.

After this process, all that remains is converting the `.vcd` file to a `.saif` file, which is done by using the `vcd2saif` tool provided with Synopsys DC. This file is read by Synopsys PT, which produces the timing and power reports.

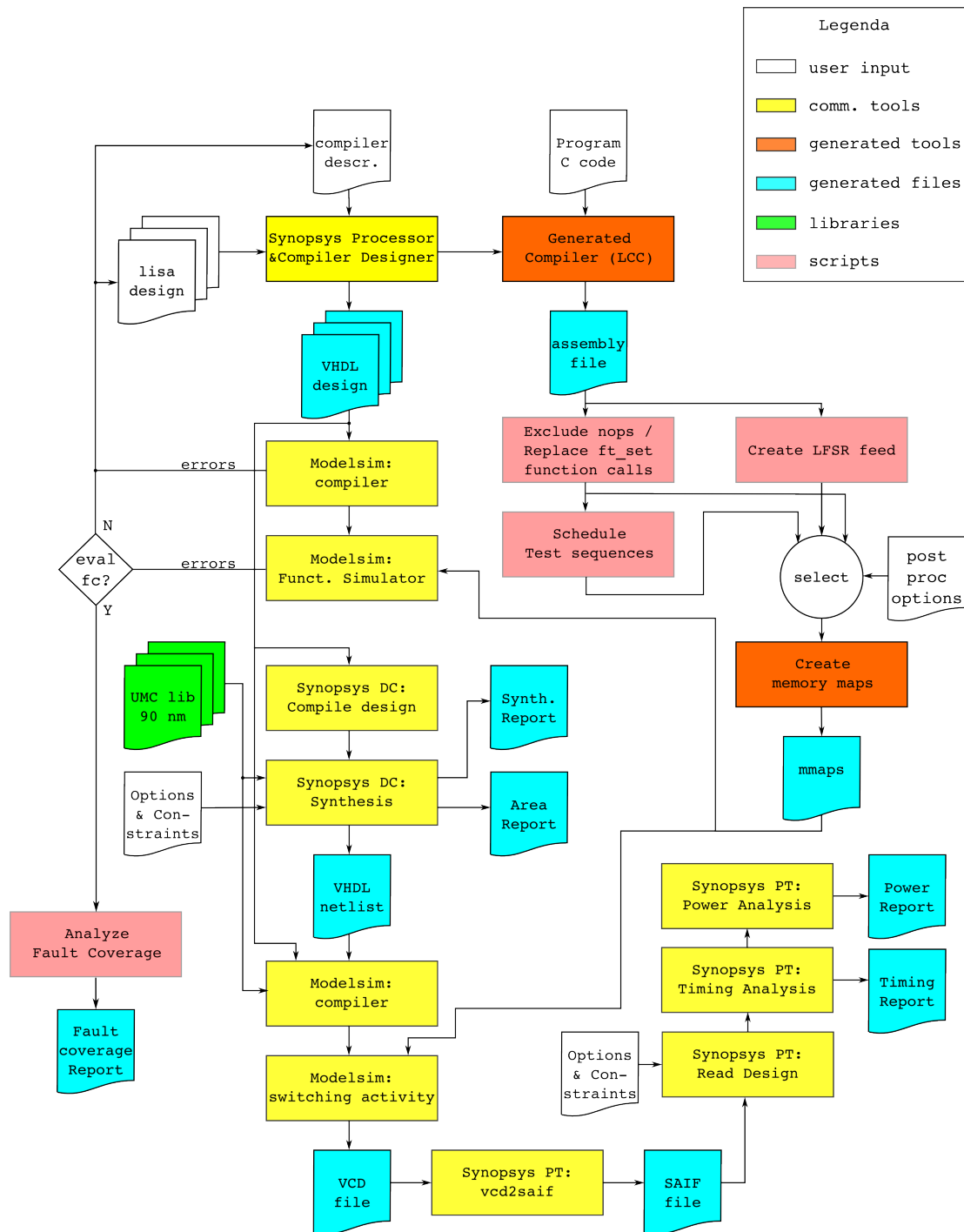


Figure 4.3: The toolflow used for evaluating the design.

4.2 Experimental Results

In this section, the experimental results are given. First, the results of the soft- and hard-fault coverage are presented and analyzed in section 4.2.1. The performance, area, power consumption and energy consumption are discussed in section 4.2.2. Finally, some compound metrics are introduced in section 4.2.3 in order to compare the added fault coverage to the added overhead, allowing us to compare our technique to the solutions mentioned in chapter 2.

4.2.1 Fault Coverage

In this section, the fault coverage added by the two separate techniques is discussed. First, we discuss the added fault-tolerance to soft-faults by instruction duplication in section 4.2.1.1, after which we will discuss the fault detection added by the instruction-based tests in section 4.2.1.2.

4.2.1.1 Fault-Tolerance to Soft-Faults

The fault-tolerance added by the instruction duplication technique, measured by the method presented in section 4.1.1, is depicted in figure 4.4 for the various test sets and duplication policies. Each column corresponds to the number of faults manifesting into errors, out of which the control-flow errors are depicted as the solid fill. The various columns correspond to the different fault sets as introduced in section 4.1.1 and are shown for the various duplication policies. First, we will discuss the total amount of soft errors, after which we will take a closer look at the control-flow errors. Throughout the section, the results of all fault sets are explained relative to the results of the same fault set when no duplication is employed.

When no duplication is employed, it is observed that only 1524 out of 4.000 faults result in an error when the fetched instruction is corrupted. However, as the instruction is read and used 100% of the execution time, it would be expected that all faults will result in errors. The reason for this observed behavior is two-fold:

1. The corrupted instruction depends on data currently used in the pipeline. Due to the stall-detection unit, the instruction will in this case be re-fetched. However, in the following cycle the fault no longer exists, i.e. an error does not manifest, and regular execution resumes;
2. The corrupted instruction is part of putting the contents of a to-be-used register on the stack / returning them from the stack, for example during a function call. This may for example occur during the end of the main function, when arguments are re-loaded from the stack. Since only the result of the function is considered, it is possible that faulty register contents may have slipped past our fault measurement method.

Note that in the last case an error *has* occurred, but this is not detected by our testing method. However, testing for all possible types of errors would heavily impact the testing time, which is already considerable in case only the result is checked. To

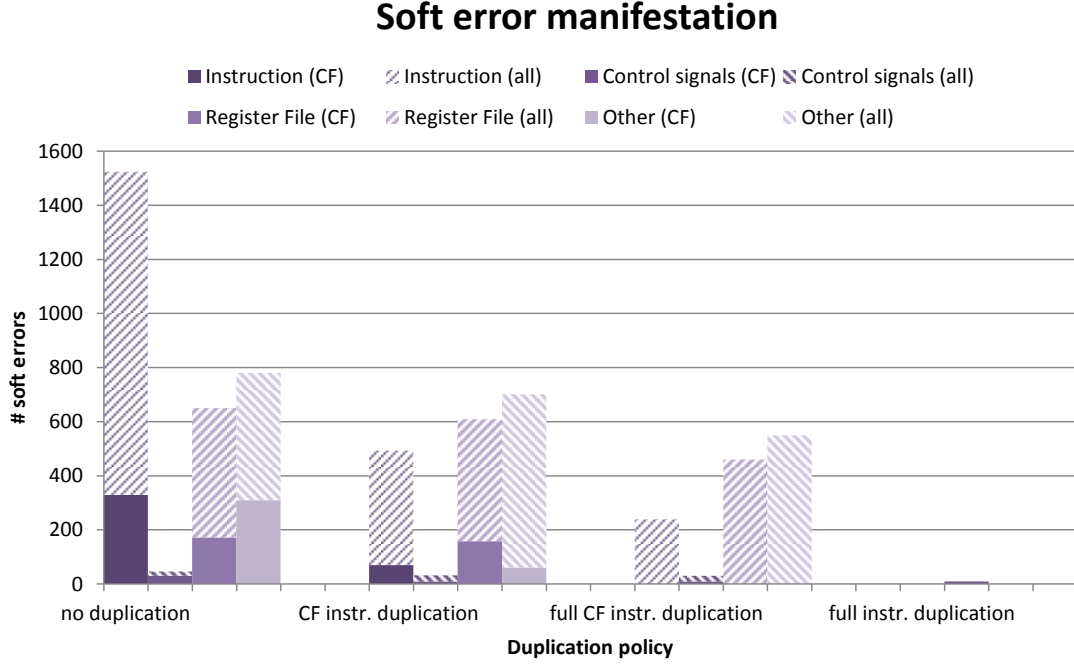


Figure 4.4: Manifested soft errors per test set.

summarize, it is expected (but not proven) more errors *will* have occurred, but as we compare our results to the Baseline (which is tested by the same mechanism) the added fault tolerance will be considered a good approximation of the actual fault tolerance.

Furthermore, it can be seen that in case of control-flow instructions only, the faults in the instruction fault set are reduced to one-third of the baseline results. The reason for this is two-fold:

1. The opcode's parity bit detects half of all faults in the fetched instruction. Consider the parity bit and the opcode of an instruction as two separate fields. The parity bit detects a corruption when the parities of these fields are not the same (e.g. the parity bit is true while the parity of the opcode is false and vice versa). The chance an error is prevented is $P_{prev} = P_p(odd) \cdot P_o(even) + P_p(even) \cdot P_o(odd)$, in which P_p is the chance the parity bit will be odd / even and P_o the chance an opcode will be odd / even. As the chances in this equation can be approximated to be 50% when taking (many) instructions into the equation, the chance a mismatch will be detected by the parity bit is 50%.
2. Looking at the ISA, Control-flow instructions are 4/22 instructions. The duplication unit will detect if a regular instruction corrupts into a control-flow instruction and vice versa. As such, the number of prevented errors N_{prev} is related to the number of executed control-flow and regular instructions by $N_{prev} = P(c \rightarrow r) \cdot N_c + P(r \rightarrow c) \cdot N_r$. In this formula, P is the chance a control-flow instruction will corrupt into a regular instruction and vice versa and N is the number of control-flow

and regular instructions executed during program execution. While the chances are fixed numbers (based on the ISA, $P(c \rightarrow r) = 18/22$ and $P(r \rightarrow c) = 4/22$), the number of executed control-flow or regular instructions vary based on the program. Due to the instruction mix in our test program, the number of detected errors will be reduced from the aforementioned 50% to roughly 33%.

On the other sets, it is shown that duplicating control flow statements only has a minor impact (under 10%). These faults are only detected, when they are involved in the control-flow instructions. As such, we can relate this number to the number of control flow instructions executed in the program.

The number of errors is reduced even further when full CF instr. duplication is employed. As in this mode the instructions within a fault-tolerant section (e.g. updating a loop variable) are duplicated, the fault tolerance is increased as expected. The claim that the number of errors is low in these fault-tolerant sections is confirmed by the findings for the full instr. duplication policy. It is shown that for the full instr. duplication policy, almost no errors occur. In fact, the only errors which occur are when one of the control signal corrupts. This is not something we have thought of before, and can be prevented by increasing the Hamming distance of the control signals. Note that control signals are typically a small part of the processor area. As such, the total error contribution (taking size into account) is minimal.

The design has also been evaluated in terms of control-flow errors. The control-flow errors are represented in figure 4.4 by the bottom solid of each column. In this figure it is shown that when the CF instr. duplication policy is used, there is still a reasonable chance of control-flow errors occurring. These errors exist due to the fact that, for example, loop variables may be corrupted. The CF instr. duplication only prevents control-flow errors with the target address at the end of a basic block, and as such a miscalculated target will not be detected.

In the other cases (full CF instr. duplication and full instr. duplication) it is shown that no control-flow errors have occurred, i.e. all errors have been from instructions not affecting the control flow.

Out of the four fault tests, some contribute more to the total amount of errors due to the size, fan-in and fan-out of the signals and circuits under test. For example, the control signals are much smaller than the ALU. This contribution is modeled as a number of weights: the instruction has a weight of one, control signals 2, and the register file / others a weight of 15. By comparing the number of errors occurring with fault tolerance to the number of errors without fault tolerance, the fault coverage can be determined.

The scaled number of errors is depicted in figure 4.5. The figure depicts the results for both “all soft faults” and “control-flow-only faults”, both including and excluding the effects of the PC.

The PC is directly related to the fetched instruction, and is thus somewhat covered in the fetched-instruction-fault model. However, a corruption in the PC will result in an error which cannot be recovered from. As such, even when full duplication is employed, 2000 out of 4000 faults will result in errors. We have modeled this effect by such an

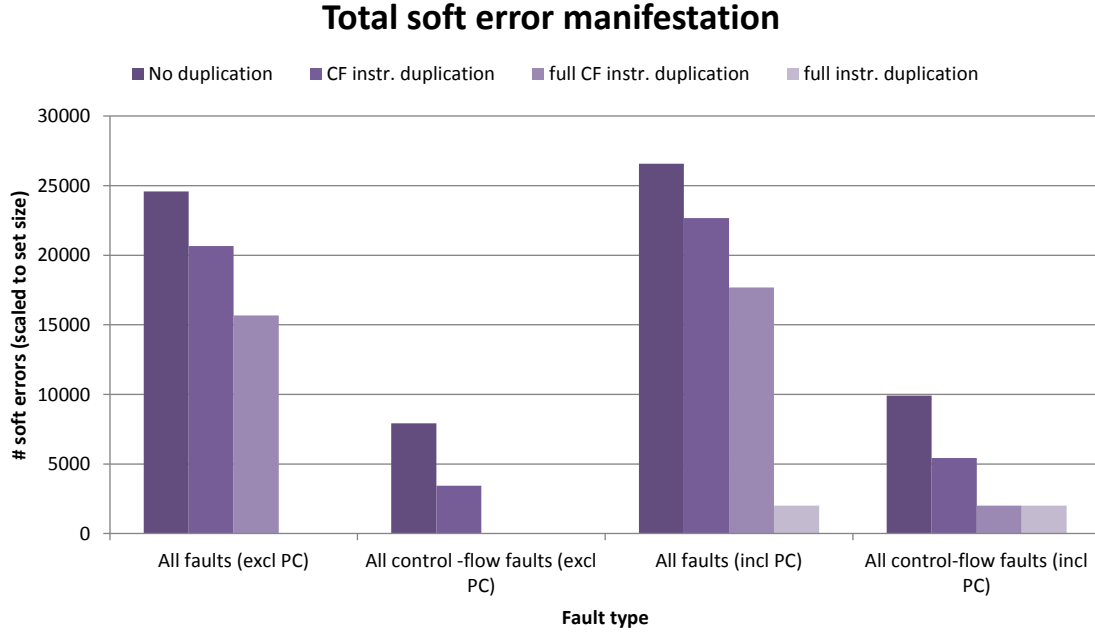


Figure 4.5: The total amount of manifested soft errors.

analysis and in this way we provide an estimate of the effect of PC corruption. As shown in figure 4.5, the number of errors is thus somewhat higher when the PC is taken into account for all policies.

Table 4.2 shows the fault tolerance added to the processor, and is directly related to the amount of errors occurring for the various duplication policies shown in figure 4.5. With respect to the total fault coverage, it is shown that the coverage is low for all policies except for full instr. duplication. This is to be expected, as these policies are not intended to provide maximum fault-tolerance. In case the full instr. duplication policy is used, 99.97% of all modeled faults are prevented from manifesting into errors.

Some of the policies (full CF and CF instr. duplication) intend to prevent control-flow errors. It is depicted that these functions substantially reduce the control-flow errors. Full CF instr. duplication has roughly the same coverage as full instr. duplication when only control-flow errors are considered (see the second and fourth group of bars). The minor variations are due to the randomness of our fault injector. Recall that CF instr. duplication is intended mainly as a policy to support other (compiler-level) techniques when a high degree of fault-tolerance is required. As such, the fault coverage of using this technique on its own is low.

When including the modeled effects of PC corruption, the fault tolerance of all policies decreases by roughly 7.5%. For a safety-critical application this is not reliable. The shadow PC in the FT/SF/HF/ST processor eliminates such a scenario (not shown in table 4.2, as a single error will not be able to corrupt both the PC and the shadow PC. As will be shown later, adding a shadow PC is a cost-effective method of preventing both hard and soft errors due to faults in the PC.

Table 4.2: Amount of soft-fault tolerance added over the baseline processor

Technique	All (incl PC)	CF (incl PC)	All (excl PC)	CF (excl PC)
CF instr. duplication	15.89 %	56.63 %	14.70 %	45.21 %
full CF instr. duplication	36.21 %	99.85 %	33.49 %	79.73 %
full instr. duplication	99.97 %	99.87 %	92.44 %	79.74 %

4.2.1.2 Fault Tolerance to Hard-Faults

As can be read in section 4.1.2, an accurate result for hard faults would take too much time to achieve. Instead, we have tried a number of tests to demonstrate the concept works. In these scenarios, a program with scheduled test instructions is executed on a core with a randomly chosen stuck-at fault. In all of the scenarios, it is shown the fault is detected, however, the number of tests have been far too selective and minor to draw any conclusions. Certainly, it is expected that some faults (e.g. in a complex block such as the ALU) will not always be detected. More work has to be done in order get more meaningful results, in particular for 1) more complex blocks (ALU, shifter unit) and 2) situations where test sequences which test a particular fault are not reachable, e.g. an error manifests which forces the program to never reach the test sequence which checks for the originating fault.

Out of all nops available in the benchmarks 64% has been used to schedule test sequences. The number could be higher when the method is integrated with the compiler as more details regarding register availability and scheduling are available. However, it is not expected that 100% of all nops can be used as, for example, all registers may be used at some point during program execution.

4.2.2 Overheads

The overheads in the system are area, performance, power consumption and energy consumption. They depend on the processor version, the set degree of fault tolerance and the benchmark which is executed.

Throughout this section, it should not be forgotten that the FT/SF/HF and FT/SF/HF/ST processors are based on the FT/SF processor. It will be shown that the FT/SF/FP processor has lower overheads across the board. As such, a margin of the overheads introduced in the FT/SF/HF and FT/SF/HF/ST processors can be neglected if they were to be changed to the same re-fetching policy as in FT/SF/FP.

It will be shown in section 4.2.2.3 that energy consumption is a major overhead in our proposed solution mainly due to the increased execution time. Not increasing the battery capacity, and thus the size, results in a decrease of operation time. To make a fair comparison to other fault-tolerant techniques, the required increase of battery size has also not been taken into account in the techniques discussed in section 2.5, i.e. we compare the energy overheads.

We will try to compare our technique to two other techniques: TSC and TMR. As reported in section 2.5, we assume that TMR adds 100% fault tolerance at the cost of 200% area and 200% power overheads. For TSC a ratio of 100% area and 100% power is assumed, also for 100% fault tolerance.

Other techniques are not as easy to compare, as they either 1) only cover a small set of the architecture or 2) not enough details are available to make a good estimation. Comparing our technique to software duplication (calculate twice, compare once, software only) would show if it is beneficial to add additional hardware to the processor to run the comparison or not. The overheads of this technique depend on a number of factors, including the data-dependencies in the program (instruction scheduling), the size of the register file (register spilling) and micro-architectural features (e.g. a forwarding unit). An accurate estimation of these overheads can currently not be obtained and as such we will not compare our technique to software duplication.

4.2.2.1 Processor Area

The total area of the various design is depicted in figure 4.6. It is shown that the FT/SF processor adds roughly 20% area overhead compared to Baseline. Using the different refetching policy, FT/SF/FP is roughly 18% larger than Baseline. The additions in FT/SF/HF (the bneqz instruction and required size increase of the memory controller) add an additional 2.3% in area. Furthermore, it is shown that the FT/SF/HF/ST processor, which implements the br_tst instruction and shadow PC, is 28% larger than Baseline.

The area presented here is only the processor area. As a parity bit is appended to each 16-bit word, an additional 6% area overhead is added to the instruction memory. Furthermore, in the case of the FT/SF/HF and FT/SF/HF/ST processors, the instruction memory size is four times the size in the Baseline, in order to allow test sequences to be scheduled. Compared to both TSC and TMR, the increase in processor area is much smaller.

Two remarks are given for FT/SF/HF and FT/SF/HF/ST:

1. Recall that the designs are based on the FT/SF design. As such, the overhead of the added fetch logic can safely be removed, and will shrink when the design is changed to the optimized version in FT/SF/FP (which is a reduction of roughly 2%);
2. It may be beneficial to remove the stall-detection unit. In the current processor versions this unit is still in the processor, since it is required for the instruction duplication to prevent errors due to ignoring data dependencies. It may be opted at some point to move the instruction duplication to compiler level (i.e. duplicate the instructions statically). This will, however, remove the benefit of setting the degree of fault tolerance at runtime.

4.2.2.2 Performance

The performance consists of two components: The number of cycles required to finish execution, and the maximum obtainable frequency (the critical path). First, the number of cycles will be discussed at length, after which the critical path will be discussed.

The number of cycles required to finish execution relies on several factors:

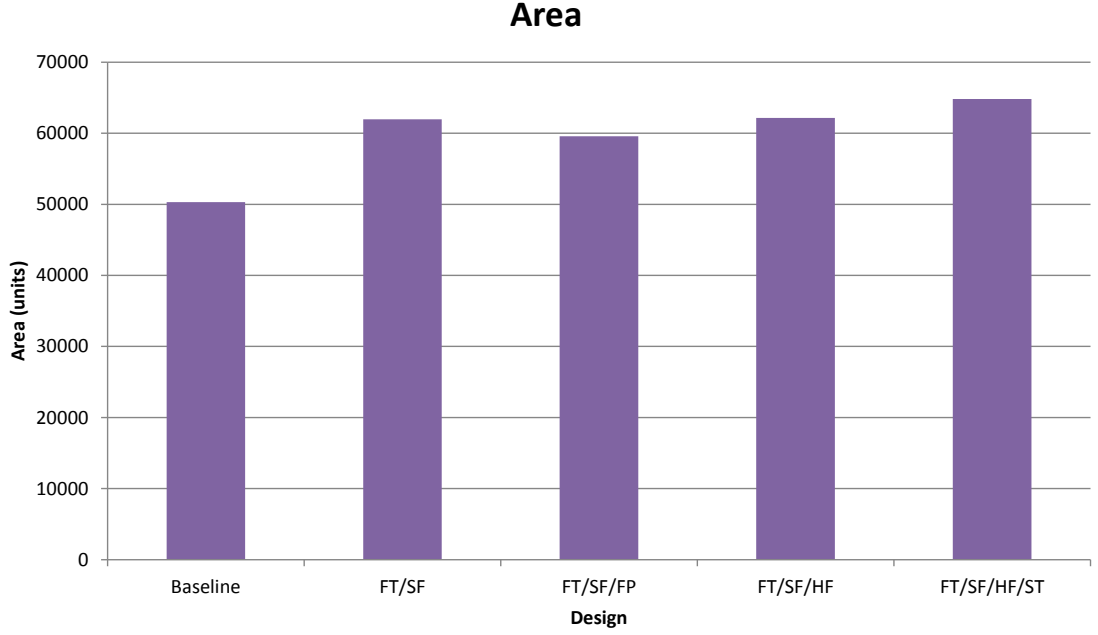


Figure 4.6: The total processor area.

1. The benchmark run on the processor;
2. The degree of soft-fault coverage;
3. The degree of hard-fault coverage.

Figure 4.7 depicts the number of cycles required for each of the benchmarks to complete on the FT/SF/HF processor, based on the duplication policy. In this figure, the results for emptyMain are multiplied by 10 in order to show the results more clearly, while the results for the total encryption program are reported in thousands. The bottom layer (solid fill) of the stacked bars represent the number of cycles required to execute the benchmarks *without* any test sequences scheduled. Each bar represent the number of cycles required to execute the benchmarks *with* test sequences scheduled. While we have implemented 4 different scheduling modes, the results are very similar and as such we report the average case of test sequence schedules. We will first discuss the results without test sequence scheduling, after which the results obtained from test sequence scheduling will be discussed. The overheads mentioned in this section are relative to the results obtained without soft- or hard-fault coverage unless stated otherwise.

As expected, the number of cycles required to finish execution depends on the duplication mode. Duplicating control-flow instructions only adds an overhead between approximately 1 and 12%, depending on the benchmark. Benchmarks which are heavy on control-flow operations will have a higher performance overhead using this duplication mode (e.g. keys). The overhead of using this mode is 9% in the total encryption program.

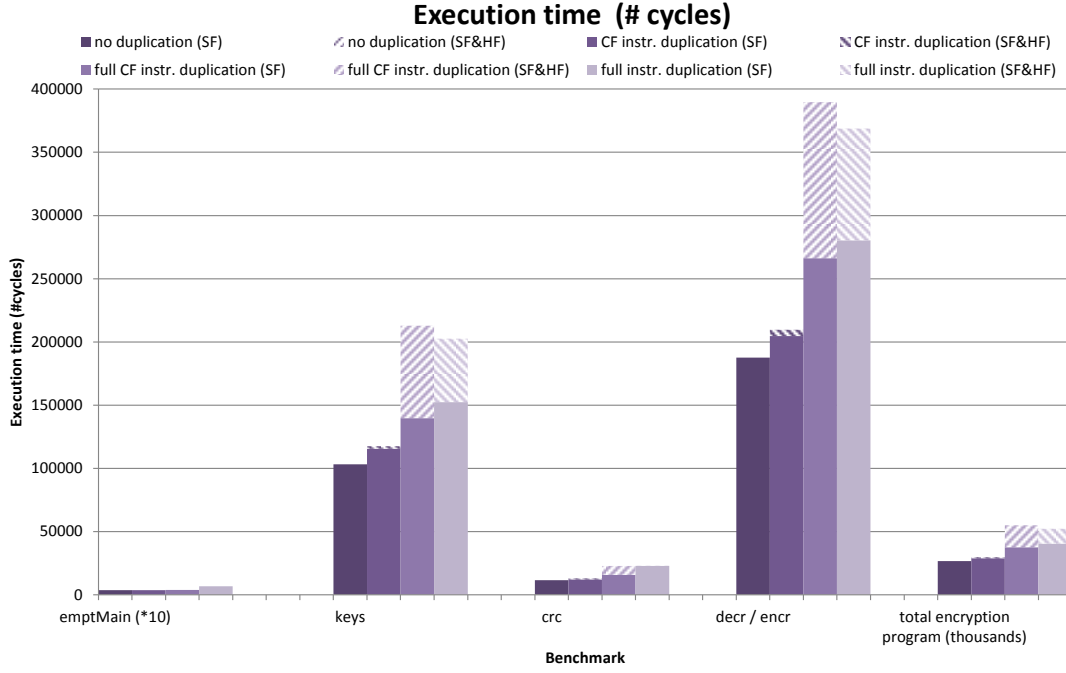


Figure 4.7: The number of cycles required to execute the benchmarks.

When full CF instr. duplication is used, an additional overhead is required due to the fact that more `ft_set` instructions need to be executed to reconfigure the duplication module. Depending on the benchmark, the overhead of this scheme varies between 5% and 42%, adding a 41% overhead to the total encryption program. This is a considerable overhead, in particular when compared to the full instr. duplication mode which will be discussed in the next paragraph. It is expected that the overhead of this policy will be lower when more code is executed between loop-variable updates.

The performance overhead going from no duplication to full duplication is between 48 and 97% depending on the benchmark, adding a 52% overhead to the total encryption program. This is an odd result at first: After all, by executing every instruction twice, a performance overhead of 100% can be expected. The answer can be found in the data-dependencies of the program, in particular the amount of stalls found in the program. By executing the duplicated instruction in the timeslot where we would normally expect a stall, the performance overhead is reduced. As such, benchmarks which have a minor amount of stalls (`emptyMain` and `crc`) will have an overhead close to the expected 100%, while benchmarks with a greater amount of stalls (`keys` and `decr / encr`) have a reduced performance penalty.

In addition to the number of cycles required to complete program execution *without* any hard fault detection, figure 4.7 shows the number of cycles required to complete program execution *with* hard fault detection (test sequence scheduling). Looking at the first bar of each column it is clearly visible that no performance overhead is added when no instructions are duplicated (when compared to the results without test sequence

scheduling). When CF instr. duplication is used, the overhead varies between 1% and 14%, increasing the performance overhead of the total encryption program to 12%, which is 3% more than when no test sequences were scheduled. Each test sequence contains a control-flow instruction (`bneqz`) which has to be duplicated, which explains the increase in performance overhead.

A major increase in overhead is found when full CF instr. duplication is used. The performance overhead varies from 6% for the `emptyMain` upto 108%(!) for the `encr / decr` algorithms. The performance overhead for the total encryption program is 107%, which was 41% when no test sequences were scheduled. In other words, the performance drops by more than a factor 2 when full CF instr. duplication is employed. Our explanation for this as follows: most of the test sequences are scheduled in the time slots where the fault-tolerance mode is set to full (i.e. when a loop variable is updated). As such, the performance overhead in these code segments will approach 100%. As an additional overhead is required for executing the `ft_set` operations, combined with the fact that aforementioned code segments are executed most frequently, the overhead will be greater than 100%.

Finally, it is shown that in case of full instruction duplication, the overhead comes close to the expected 100% (97% performance overhead is added to the total encryption program, up from 52% in case no hard-fault tolerance was added). This is due to the fact that the stalls which could be used to mask some of the performance overhead (see the discussion on full instruction duplication earlier in this section) are now used for test sequences. Other than a number of stalls which could not be scheduled, the execution time is doubled.

It should not be forgotten that the trade-off between the different duplication policies, also, depends on the application. We have evaluated our design using an encryption algorithm, which is heavily dataflow oriented. It would be interesting to see what the differences would be on a more control-flow oriented program.

As TSC and TMR add no performance overhead, these techniques naturally perform better in terms of performance than our technique.

For the SiMS framework a frequency higher than 10-20 MHz is not expected to be required, due to the expected performance requirements of the target programs. After all, increasing the frequency can be associated with a higher average power consumption, which is already constrained by the environment. At the same time, a frequency of 20 MHz must be guaranteed for certain applications due to real-time constraints. In figure 4.8 the critical-path delay for each of the designs is shown.

It is shown that for the FT/SF design, there is a significant overhead in the critical path (roughly 20%). However, the system can still reach an operating frequency of 69 MHz, well within range of the target application. This increase in overhead is removed when the re-fetching policy is changed to that in FT/SF/FP. It should be noted that the critical-path delay of the FT/SF/FP, FT/SF/HF and FT/SF/HF/ST is lower than that of the baseline processor. While not 100% sure of the underlying causes, the following explanation is given:

1. The synthesis tool uses heuristic algorithms, and as such a statistical error is introduced.

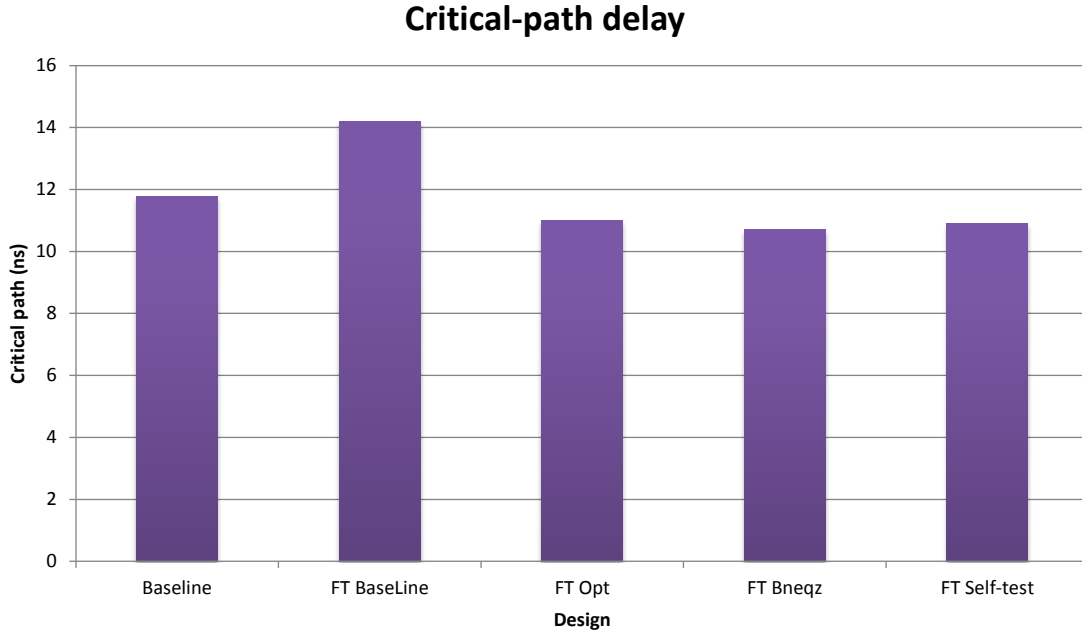


Figure 4.8: The critical path for the various designs.

2. For a given technology, the delay of connecting small logic blocks may be greater than that of a large logic block (e.g. a tree of or-gates versus one large or-gate). To save area, the synthesis tool may prefer to use small logic blocks. When more inputs are given to a circuit (when for example a certain function is made more complex) it may be beneficial use the large block area-wise. As such, not only the area but also the speed (and thus: the critical path) are optimized.

4.2.2.3 Power Consumption

The average power consumption is an important figure, as it can be related to the heat dissipation of the system. A high power consumption can thus cause a variety of harmful conditions, such as tissue burns. Energy consumption, the product of power consumption and execution time, can be related to battery lifetime and is thus an important figure for biomedical implants. We will first discuss the average power consumption, after which we will discuss the energy consumption. The average power consumption is depicted in figure 4.9. In this figure, the FT/SF/HF (sched) represents the FT/SF/HF processor with the test sequences scheduled.

While there are marginal differences, the power consumption is roughly 20% higher than the baseline processor for all designs, even when no duplication is done. This is due to the fact that no clock-gating is involved, so even when no duplication is desired all components are still active although their results are discarded. The FT/SF/HF design has a slightly higher power consumption due to the more complex of the memory controller (as the memory size has increased) and the idle power used for the `br_tst`

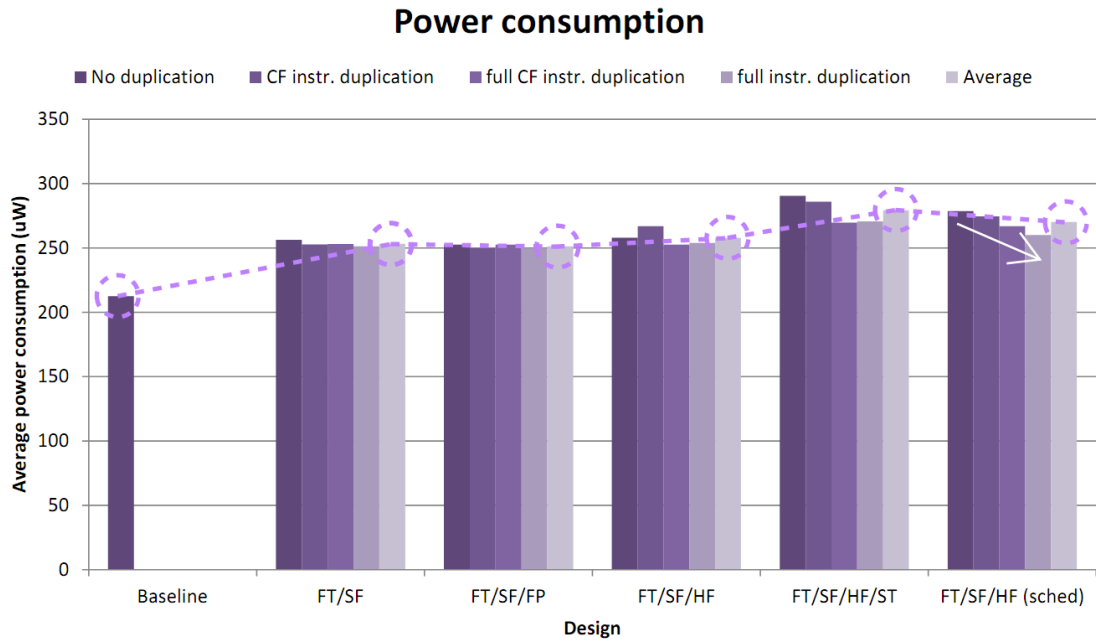


Figure 4.9: The average power consumption for the various designs and duplication policies.

instruction. Both designs have a slightly larger power consumption than FT/SF/FP of around 5%.

It is shown that as more functionality is added to the design, more power is consumed on average. As expected, the self-test additions in FT/SF/HF/ST (assumed to be mainly the shadow PC and comparator) increase the power consumption by quite a margin. In fact, the self-test version adds 13% additional average power consumption over the FT/SF/HF design. As expected, the power consumption overhead of the shadow PC is significantly higher than the area overhead, due to the amount of transitions occurring in the PC (and as such, in the shadow PC and comparator).

An interesting observation lies in the fact that the average power consumption goes down when more instructions are duplicated, as depicted most clearly by the white arrow (slope) inside the last set of columns in figure 4.9. In particular, a 10% decrease in average power consumption can be identified for the FT/SF/HF (sched) design when going from no duplication to full instruction duplication. This observation can be explained as follows: assume that we have the following instruction sequence somewhere in the program:

```
Li r1, 42
nop
mov r2, r4
```

In the case of no duplication, a nop enters the pipeline stage after the Li. This will cause numerous transitions in for example the EX stage, as the results from the

previous input (li r1, 42) will now be changed to the results of the nop (which will be 0). In the next cycle, numerous transitions will again occur, as the mov r2, r4 instruction will cause transitions in all stages after the nop. When we duplicate the li instruction, our executed code will look like this:

```
Li r1, 42
Li r1, 42
mov r2, r4
```

In this case, the second li r1, 42 instruction will not cause any transitions, as the pipeline is already stable at the correct input. As such, a high amount of switching is avoided. As such, duplication can be seen as a technique to reduce average power consumption. In fact, a method for reducing power consumption by “duplicating” stalls is described in [32], with similar results.

This also explains why adding test sequences incurs a higher power consumption overhead when no instructions are duplicated, compared to when all instructions are duplicated: Consider the following code segment, where 3 nops are scheduled (due to data-dependencies):

```
Li r1, (BB1>>16)
nop
nop
nop
stfl r1, 8
```

In case no tests are scheduled in this code segment, only two transitions occur: From the li to the nops, and from the last nop to the stfl instruction. In case these nops are used for test sequences, up to 5 transitions will occur: from the li to the 1st test instruction, from the 1st test instruction to the 2nd test instruction, etc. As such, the power consumption rapidly increases. These transitions will occur both when no instructions are duplicated and when all instructions are duplicated. However, as the program executes faster when no duplication is performed, the average power consumption will be higher. Note, however, that while the average power consumption will be lower for full instr. duplication, the energy consumption will increase as the total execution time increases as shown in section 4.2.2.2. As TSC and TMR add 100% and 200% resp. power consumption overheads, we can conclude that our technique performs better with respect power consumption.

4.2.2.4 Energy Consumption

Figure 4.10 shows the total energy consumption for the various processor designs and duplication policies. The energy consumption is equal to the product of execution time and the average power consumption. As discussed in the previous section, the average power consumption depends on both the complexity of the design and the duplication policy (a more complex design consumes more power, while duplicating more instructions

typically reduces the average power consumption). Furthermore, it has been shown in section 4.2.2.2 that the duplication policy has an impact on the execution time. As such, duplicating more instructions affects the energy consumption both positively (by reducing the average power consumption) and negatively (by increasing the execution time). We will first discuss the energy consumption based on the complexity of the processor designs (i.e. independent of the duplication policy), after which we will discuss the impact of the various duplication policies.

Looking at figure 4.10 we can see that as the designs increase in complexity, the energy consumption increases. This is most apparent from comparing the averaged energy consumption for all designs, which is the average of all duplication policies for each individual processor design, depicted as the last column for each processor design. The averaged energy consumption of the FT/SF processor is 46% higher than the Baseline, while that of the FT/SF/HF/ST (the most complex) processor design is 85%, thus the range in energy overhead is [46%, 85%] with respect to the baseline. The FT/SF/FP design performs slightly better than the FT/SF design in terms of energy consumption, which indicates that the energy consumption can be reduced by applying the improved fetching policy present in FT/SF/FP. This is expected, as the reduced average power consumption in FT/SF/FP (compared to FT/SF) causes a reduction in energy consumption as well.

Other than the complexity of a design, it is shown that for each design the energy consumption increases when more instructions are executed (i.e. the execution time is higher). A more fault-tolerant duplication policy (i.e. more instructions are duplicated) reduces the average power consumption while also increasing the execution time. For each of the duplication policies, we can see the following effects on the execution time and power consumption compared to the Baseline¹:

1. No duplication: 0% increase in execution time, 20% increase in power consumption;
2. CF instr. duplication: 9% increase in execution time, 18.9% increase in power consumption;
3. Full CF instr. duplication: 41% increase in execution time, 18.9% increase in power consumption;
4. Full instr. duplication: 52% increase in execution time, 18.2% increase in power consumption.

While the power-consumption increase is lowered when more instructions are duplicated (as shown in the previous section), the increase in execution time by far outweighs the power savings. As such, the energy consumption follows a similar trend as the increase in execution time, as discussed in section 4.2.2.2. However, this does imply that the energy consumption does not increase linearly with an increased execution time. In the most fault-tolerant scenario (scheduled test sequences and full duplication), the energy consumption is up to 157% higher.

¹Coming from the data depicted in figure 4.7 and figure 4.9. Presented are the values for the FT/SF design. However, it should be noted that all presented processor designs follows the same trend.

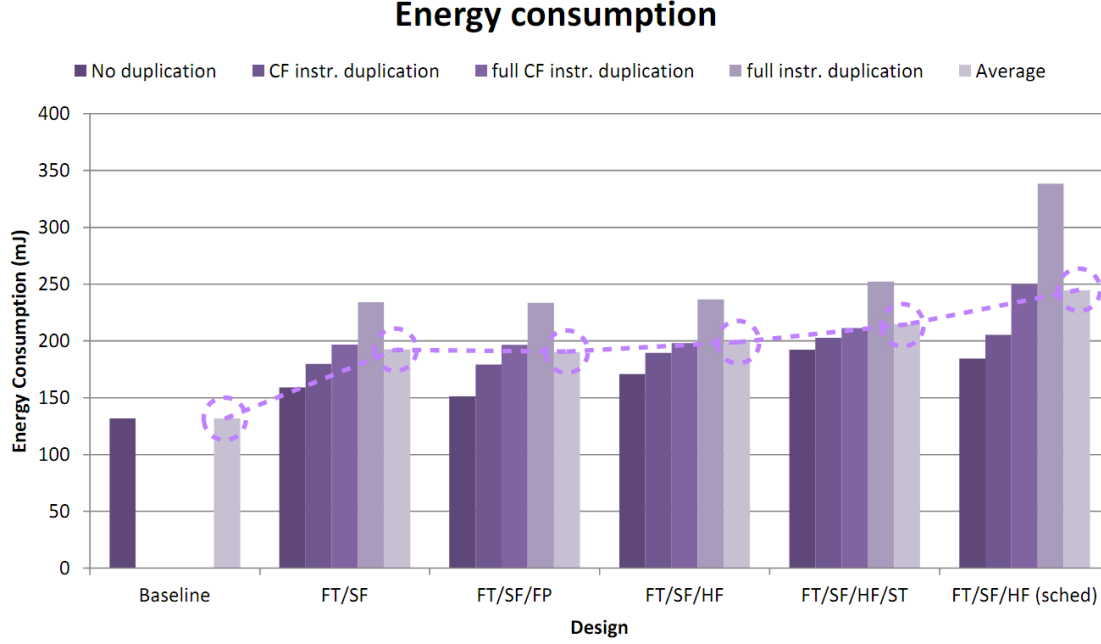


Figure 4.10: The energy consumption for the various designs and duplication policies.

Summarizing this and previous section, the power consumption exhibits a somewhat stable overhead varying between 20% and 35% compared to the Baseline. The consumption depends on the executed program, the duplication policy, and whether or not test sequences are scheduled. When maximal fault tolerance is desired energy consumption exhibits a major overhead and is up to 157% higher when compared to the energy consumption of the Baseline.

4.2.3 Compound Metrics

In the previous sections we have shown the various overheads added by our proposed technique compared to the Baseline design. It has been shown that while the area and power consumption overheads are more or less constant with regards to different fault-tolerance policies, the performance and energy consumption can vary heavily.

In this section, compound metrics are introduced to compare our design to other designs. These metrics are the Power-Area (PA) product, the Energy-Area (EA) overhead product, the fault tolerance (FT) / PA ratio and the fault tolerance (FT) / EA ratio. The PA product is important as both power consumption and area are important figures due to the constraints set by the environment. As such, this metric represents the total overheads related to the hard constraints (power consumption and area) imposed by the biomedical environment. The EA product takes energy consumption (and thus execution time) also into account. As such, this metric represents the total overhead of the system. The added fault tolerance found in 4.2.1.1 is then normalized by the PA and EA. This will give a metric to compare the added fault tolerance to the overheads (in which a larger value will be a more optimal solution), which can be compared Chapter

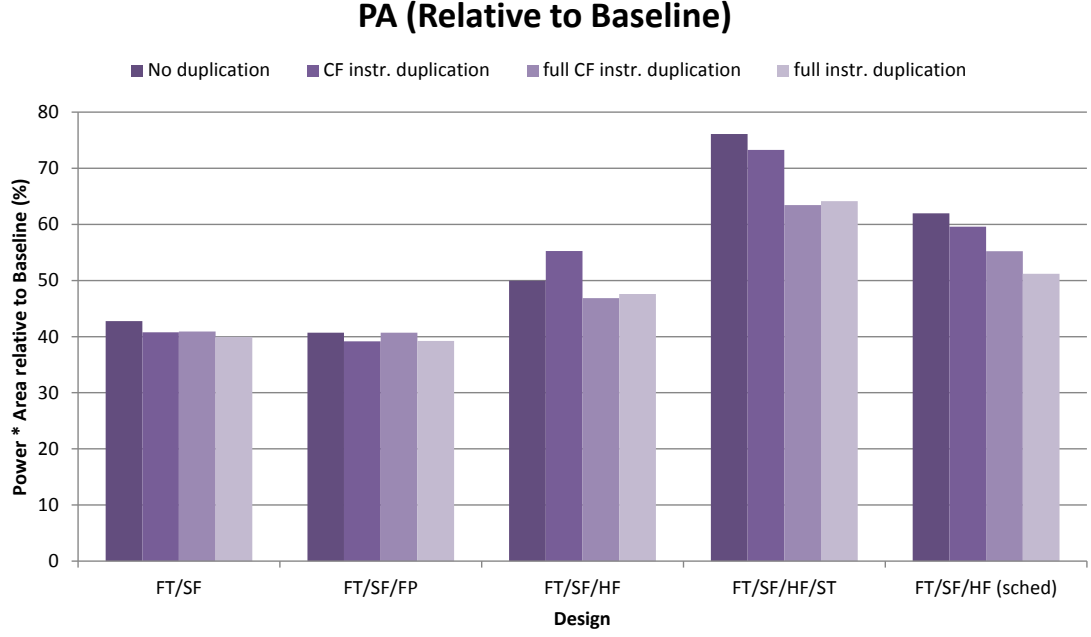


Figure 4.11: The PA metric relative to Baseline.

2. The compound metrics are discussed in the following subsections.

4.2.3.1 The PA metric

The PA compound metric summarizes the power consumption and area overhead. As area and power consumption are the hard constraints set for the system, this number represents the total overhead related to these constraints.

The PA compound metric, relative to Baseline, is depicted in figure 4.11. In FT/SF/FP, only 40% overhead is added when compared to the Baseline. The FT/SF/HF/ST processor has both a relative high increase in area and power consumption (as can be read in section 4.2.2.1 and section 4.2.2.3). As such, the PA ratio varies between 63% and 76% depending on the duplication mode. As the area is constant, the PA product is driven by the same factors as the power consumption (i.e. duplication policy, the test sequence scheduling mode and the executed program).

To compare the overheads for TSC and TMR we calculate the PA ratio based on the values assumed in section 2.6. TSC adds 100% area and 100% power consumption, and the PA metric relative to the Baseline is thus 300% $((2 * 2 - 1) * 100\%)$. For TMR, the PA metric is 800% $((3 * 3 - 1) * 100\%)$. As such, we can safely say that our technique has less overhead with respect to the hard constraints posed by the environment.

4.2.3.2 The EA metric

The EA compound metric summarizes the energy consumption and area overhead. As energy consumption is determined by both the execution time and the power consump-

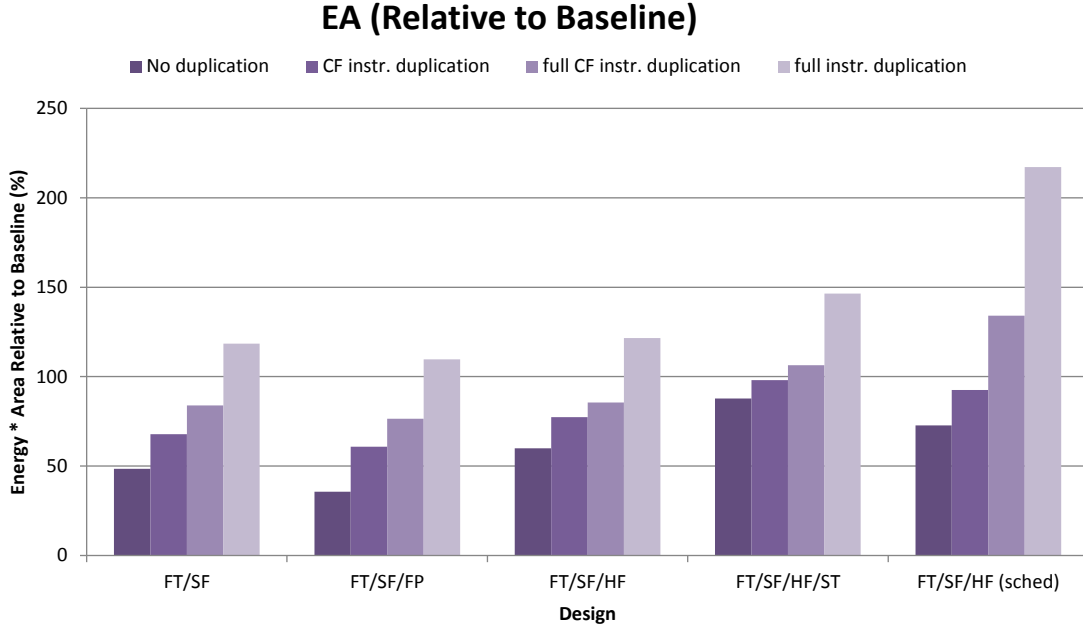


Figure 4.12: The EA metric relative to Baseline.

tion, this metric summarizes the total system overhead. Figure 4.12 depicts the EA metric relative to Baseline.

While the FT/SF/HF/ST has a bigger PA product than the FT/SF/HF (sched) processor it is shown that for EA, FT/SF/HF (sched) has the highest EA product. This is expected as the EA product depends on execution time, which is higher when test sequences are scheduled. As the area is constant for a particular design, the EA metric follows the same trend as reported for energy (section 4.2.2.3). When test sequences are scheduled without the self-test additions (i.e. in design FT/SF/HF (sched)) the EA product is up to 217% higher than the Baseline. As the self-test additions add 13% energy consumption on average, the EA product is 257% when maximum fault-tolerance is desired. For TSC and TMR, no performance overheads are added. As such, the EA overhead is the same as the PA overhead (300% and 800% resp.). Consequently, while the energy consumption of our technique is higher than the energy consumption of TSC, the total system overhead is lower.

4.2.3.3 The FT / PA metric

To justify adding overheads to the baseline design, the added fault tolerance must be taken into account. After all, if solution A adds three times the overhead compared to solution B, while only adding 2% fault coverage, solution A should be considered the worse solution. The FT / PA compares the added soft-fault tolerance to the hard constraints of power consumption and area. A high FT / PA value is desirable, as such a value would indicate a small overhead is paid for a high degree of fault tolerance, by comparison.

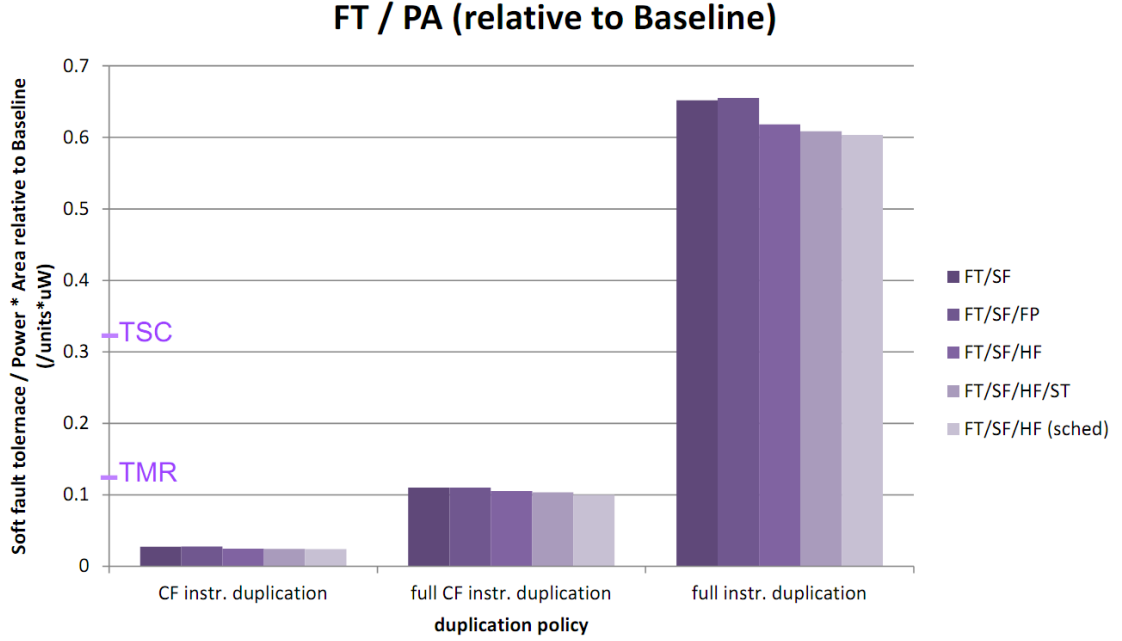


Figure 4.13: The FT / PA metric relative to Baseline.

The resulting FT / PA ratios for the various designs and duplication policies are depicted in figure 4.13. It is shown that when duplication is only done for a subset of instructions (e.g. CF instr. duplication), the associated FT / PA is low indicating that there is a high overhead for a small fault coverage. As the fault tolerance remains the same for all designs for a particular duplication policy, the variation between designs (for a given duplication policy) is a function of the PA compound metric above (section 4.2.3.1).

Furthermore, it is shown that when the duplication policy is changed to a higher degree of fault tolerance, the FT / PA ratio becomes higher. This is in line with our expectations, as it has been demonstrated in section 4.2.1 that the fault tolerance is a function of the duplication policy. Furthermore, it has been shown that as more instructions are duplicated, the power consumption overhead is decreased (section 4.2.2.3). As a result, the FT / PA ratio is boosted by both these factors, showing it is more optimal to do more duplications in terms of both added fault tolerance and power consumption.

When compared to TMR and TSC, duplicating a subset of instructions gives a lower FT / PA ratio than TMR. This is not unexpected: As the fault tolerance is very low while our PA metric is high, a high overhead is paid for minimal fault tolerance. However, both our technique and TMR / TSC add roughly 100% fault tolerance when we apply full instruction duplication. As such, the FT / PA ratio will mainly depend on the PA metric when we are duplicating all instructions. As has been shown in section 4.2.3.1, the PA metric of our technique is better than the PA metric for TSC / TMR when full duplication is employed. We, thus, conclude that our technique outperforms both TSC and TMR when the added fault tolerance is compared to the added area if we apply full

instruction duplication.

Note that only the fault tolerance to soft faults is compared: TSC and TMR have a better detection (and for TMR, also correction) mechanism for hard faults. However, soft faults are by far more common than hard faults. Since the test sequences are assumed to provide a good coverage to hard faults, the results will remain the same if hard fault detection would have also been taken into account (remember from section 4.2.3.1 that the PA is not substantially increased when test sequences are scheduled).

Finally, an interesting conclusion is that the shadow PC implemented in FT/SF/HF/ST keeps the FT / PA nearly the same to the same processor version without the shadow PC (FT/SF/HF). By adding a shadow PC, the fault tolerance is increased by 7.5%, while the area and power consumption overheads are increased to respectively 5% and 12%. Therefore, the FT / PA ratio stays roughly the same as depicted in figure 4.13.

In summary, the FT / PA ratio varies between 0.05 and 0.72, with mild variations based on which design is used. When a subset of instructions is duplicated the processor performs worse when compared to the techniques mentioned in chapter 2 in terms of the FT / PA metric. However, when full instr. duplication is used, the FT / PA metric is better than these techniques. Adding a shadow PC does not lead to any significant degradation in the FT / PA metric. Therefore, a shadow PC should be implemented into the design if the power and area budget allows this.

4.2.3.4 The FT / EA metric

By comparing the added fault tolerance to the energy consumption and area, the total system cost is compared to the added fault tolerance. Similar to FT / PA, this allows us to compare our technique to other techniques, and determine its viability. The benefit of doing FT / EA analysis is that it takes performance and, thus, battery lifetime also into account, which is important for implantable devices.

Figure 4.14 shows the FT / EA metric for the various processor designs and duplication policies. Similar to FT / PA, the results are better when a higher degree of fault tolerance is set. However, it should be noted that all results are slightly depressed when compared to figure 4.13 (i.e. the results are slightly worse) . The low amount of fault tolerance added when only a subset of instructions is duplicated results in a low FT / EA ratio. In case full instr. duplication is used, the metric varies between 0.29 and 0.44. A considerable drop is measured when test sequences are scheduled. As the scheduling of test sequences decreases performance when full instr. duplication is used (as shown in section 4.2.2.2), the energy consumption is considerably higher than when no test sequences are scheduled. However, while not modeled here, this does add a degree of fault tolerance to hard faults to the application.

Our added fault tolerance is close to 100% for full duplication. As such, the difference in FT / EA between our technique and TSC / TMR is in the EA metric. It has been shown in section 4.2.3.2 that our technique performs better in terms of EA than TMR, and very similar to TSC. As such, it can be said that our technique requires less overhead for the same amount of fault tolerance compared to TSC and TMR, as depicted in figure on the y-axis of figure 4.14. It should be mentioned that incorporating our techniques

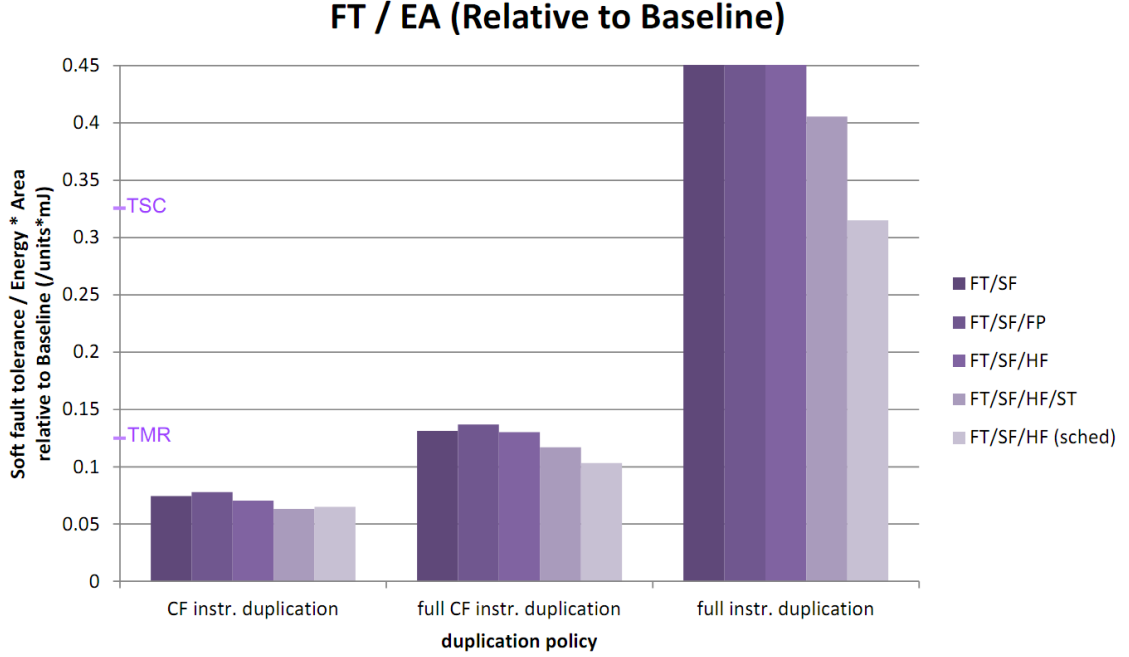


Figure 4.14: The FT / EA metric relative to Baseline.

are as easy as implementing TMR, and far easier than incorporating TSC in any design.

Finally, it is also shown that the shadow PC has a minor impact on the FT / EA metric, the reason of which has been explained in the previous section. As such, a shadow PC should be implemented if the energy and area budgets allow this.

In summary, energy consumption is the main overhead of the system. Comparing the added fault tolerance to the total system overhead, it is shown that the FT / EA ratio is very low when only a subset of instructions is duplicated. When full instruction duplication is used, our techniques require less overhead than TSC and TMR, while providing the same soft-fault tolerance.

4.3 Summary and Discussion

In this chapter, the method used for evaluating the design and associated results have been presented. First, the experimental setup for evaluating the coverage of the various processor designs to soft faults, hard faults and overheads has been described as well as the involved tool flow in section 4.1. This setup is then used to evaluate the processors. Section 4.2.1.1 has shown that as more instructions are being duplicated, almost 100% fault tolerance can be achieved to soft faults, depending on whether a shadow PC is added (99.97% coverage at the additional cost of 5% area and 12% power) or not (in which case the actual fault tolerance peaks at 93%) as shown in table 4.2. The shadow PC does not affect the ratio between overheads and added fault tolerance, and should be incorporated in the design if the power consumption, energy consumption and area budget allows this. Moreover, the area and power overheads of all designs could be

Table 4.3: Summarized overheads (relative to Baseline)

Processor	Dup. mode	Power	Energy	PA	EA	FT/PA	FT/EA
Baseline	none	212.6 (μ W)	132.0 (μ J)	10.7 (W*#l.cells)	6.6 (J*#l.cells)	-	-
FT/SF	none	+20.60%	+20.60%	+42.80%	+48.5%	-	-
FT/SF	CF. instr.	+18.88%	+36.27%	+40.78%	+67.86%	0.39	0.07
FT/SF	full CF. instr.	+18.98%	+49.25%	+40.91%	+83.85%	0.71	0.13
FT/SF	full instr.	+18.17%	+77.38%	+39.95%	+118.50%	0.71	0.46
FT/SF/FP	none	+18.82%	+14.58%	+40.71%	+35.69%	-	-
FT/SF/FP	CF. instr.	+17.50%	+35.85%	+39.16%	+60.88%	0.40	0.08
FT/SF/FP	full CF. instr.	+18.80%	+49.01%	+40.69%	+76.47%	0.72	0.14
FT/SF/FP	full instr.	+17.56%	+77.03%	+39.23%	+109.65%	0.71	0.48
FT/SF/HF	none	+21.33%	+29.41%	+49.96%	+59.95%	-	-
FT/SF/HF	CF. instr.	+25.61%	+43.55%	+55.26%	+77.42%	0.37	0.07
FT/SF/HF	full CF. instr.	+18.83%	+50.10%	+46.87%	+85.52%	0.64	0.13
FT/SF/HF	full instr.	+19.41%	+79.25%	+47.60%	+121.55%	0.68	0.45
FT/SF/HF (sched)	none	+36.64%	+45.74%	+61.98%	+72.77%	-	-
FT/SF/HF (sched)	CF instr.	+34.45%	+53.65%	+59.59%	+92.51%	0.34	0.06
FT/SF/HF (sched)	full CF instr.	+26.81%	+60.19%	+55.21%	+134.10%	0.65	0.10
FT/SF/HF (sched)	full instr. dup.	+27.36%	+91.17%	+51.19%	+217.10%	0.64	0.32
FT/SF/HF/ST	none	+31.05%	+39.78%	+76.11%	+87.84%	-	-
FT/SF/HF/ST	CF instr.	+29.12%	+55.75%	+73.28%	+98.02%	0.32	0.06
FT/SF/HF/ST	full CF instr.	+25.58%	+89.40%	+63.45%	+106.46%	0.58	0.12
FT/SF/HF/ST	full instr. dup.	+22.32%	+156.55%	+64.14%	+146.38%	0.61	0.41

reduced by respectively 5% and 1% when the same error correction strategy is adopted in FT/SF/FP. The difference in using this fetching policy has only been noticed at a later point, which is why this has not been implemented in our processor designs as a mainstream improvement.

Furthermore, control-flow errors can be avoided by either duplicating all instructions, or by duplicating all control-flow instructions and instructions involved in updating loop variables. In section 4.2.1.2 it has been shown the test sequences and self-test additions work using a number of scenarios. However, more work is required conclusively decide on the applicability of the technique. Finally, it has been shown in section 4.2.2 that the area and power overheads vary between 20 and 37% depending on the processor design, as summarized in the first column of table 4.4 and the third column of table 4.3, respectively. Replacing nops by duplicated instructions is a method found to effectively reduce the power overhead, and therefore reduces the energy overhead in case much duplication is required. Performance varies heavily depending on the duplication policy and scheduling mode (between 0 and 107.2%) as summarized in table 4.5, as does the energy consumption (up to 157% overhead) shown in the fourth column of table 4.3.

The compound metrics, as summarized in the last four columns of table 4.3, have shown that our technique performs better in terms of utilizing the power consumption and area overheads for fault tolerance compared to the techniques discussed in section 2.5. Energy consumption has a substantial overhead, in particular when test sequences are scheduled and full duplication is employed. Due to the small amount of added fault tolerance when only a subset of instructions is duplicated, our technique does not perform as well as some of the techniques discussed in section 2.5. However, when all instructions are duplicated (i.e. the maximum degree of fault tolerance is set), our technique generally performs better than these techniques, as it requires less overhead to achieve 100% fault tolerance. Our technique is as easy to incorporate into a processor design as TMR, and

Table 4.4: Area and critical path overheads (relative to Baseline)

Processor	Area (logic cells)	Critical-path delay (ns)
Baseline	50292	11.8
FT/SF	61950 (+23.2%)	14.2 (+20.8%)
FT/SF/FP	59560 (+18.43%)	11.0 (-6.4%)
FT/SF/HF	62161 (+23.60%)	10.7 (-9.03%)
FT/SF/HF/ST	64817 (+28.88%)	10.9 (-7.31%)

Table 4.5: Performance overhead (relative to baseline)

Duplication mode	Execution time w/o sched(#cycles)	Execution time w sched(#cycles)
none	26481	26481 (+0%)
CF instr.	28833 (+8.9%)	29588 (+11.7%)
full CF instr.	37458 (41.5%)	54859 (+107.2%)
full instr.	40306 (+52.2%)	52073 (+96.6%)

much easier than TSC, in which each component has to be designed from scratch.

Conclusions and Future Work

In this thesis we have presented a combination of techniques for fault tolerance, targeting both hard and soft faults. Soft faults are tolerated by an instruction duplication technique, in which all instructions or a subset of instructions is duplicated. The set of instructions which needs to be duplicated can be (re)set at runtime, allowing a dynamic trade-off to be made between performance and fault tolerance. Hard faults are tolerated by running test sequences when the processor stalls and by hardware additions for functionally untestable paths. The techniques have been applied to the SISC Baseline processor and have been analyzed in terms of fault tolerance, area, performance, power consumption and energy consumption as well as a number of compound metrics.

5.1 Summary

In chapter 2, the SiMS project which strives to create a framework for biomedical implants was presented. This chapter furthermore provides background knowledge on faults and the SISC Baseline processor, a five-stage RISC processor designed within the SiMS framework. The Baseline was used to implement the fault-tolerance techniques on and it provides a point of reference. Furthermore, several architecture-level techniques have been surveyed to get an understanding of possible techniques, after which these solutions were compared in terms of area, power consumption, performance, fault coverage, and implementation time.

In chapter 3, the proposed architecture was described in detail, including the logic involved in instruction duplication, error detection and error correction. The method of setting the subset of instructions which needs to be duplicated is introduced, along with the current three instruction sets which can be duplicated (no duplication, control-flow instructions or full duplication). These duplication modes, which represent the degree of (required) soft-fault tolerance, can be configured by instruction level configurability. The required parity bit is also discussed, and it is shown what role it plays in the design. Furthermore, four duplication schemes were proposed, targeting various performance and fault tolerance requirements.

Other than the technique for soft-fault tolerance, the proposed method of designing and scheduling test sequences was discussed in this chapter. It has been shown how test sequences were designed, and how these were scheduled by means of processing an assembly file. Furthermore, it has been shown how we target untestable paths, by adding design for testability and a shadow program counter.

In chapter 4, the method of evaluating the design was presented. It was shown how the added fault tolerance for both soft and hard faults was measured and how the overheads (area, performance, average power consumption and energy consumption) were determined. These metrics were determined for various processor designs and degrees of

fault tolerance. The fault tolerance to soft faults using our method is 99.97%, which can be increased to 100% by increasing the Hamming distance of certain control signals. It has been shown that using test sequences for testing against hard faults works, although more research needs to be conducted into the fault coverage and effective scheduling methods, among others. The area and average power consumption overheads are roughly 20% for the processor strictly targeting soft faults. These overheads were increased up to 25% area and 33% power consumption for a design which has capabilities to test functionally untestable paths. The designs have also been evaluated in terms of a number of compound metrics, which summarize the total overheads imposed by the design. These overheads were compared to the added fault tolerance, in order to compare the cost and gain. It was shown that energy consumption and performance were the main overheads when a high degree of fault tolerance was set (targeting both soft and hard faults). The used techniques perform better in terms of the hard constraints (area, power consumption) and total system overhead when compared to the techniques discussed in section 2.5 if the maximum degree of fault tolerance was set. When performance is preferred over fault tolerance, the performance and thus the energy consumption were much lower, although the overheads are high compared to the added fault tolerance.

5.2 Concluding Remarks

Two targets have been set for the architecture: Having it be viable for biomedical implants, and having the used fault-tolerance techniques be more viable than other techniques. These have been formulated as the research questions “Is the proposed fault-tolerance architecture suitable for a biomedical implant?” and “How do the techniques used in the proposed fault-tolerance architecture compare (in terms of area, power, performance and fault-coverage) to other, well-known techniques?”.

To answer the first question, we have shown that 99.997% of all soft faults can be tolerated (when a shadow PC is added to the design) with area and average power consumption overheads being as small as 25% and 33% respectively (compared to the Baseline processor). In addition, a 6% increase in the instruction memory size is required due to the parity bit. In particular, it has been shown that duplication can even lead to a reduction in power consumption when the duplicated instruction replaces a nop, thus avoiding signal transitions. As indicated, the area and power consumption overheads could be lower by duplicating instructions at compile-time rather than at runtime. However, this comes at the cost of an increased instruction memory and losing the benefits of runtime duplication.

The energy overhead is directly related to the performance and is therefore directly related to the required degree of fault tolerance (i.e. the duplication policy and the amount of test sequences scheduled). In a setting where both soft faults and hard faults are assumed to be well-covered (full instruction duplication with a high degree of test sequences scheduled in the program), the energy overhead can be as high as 257%. On the other hand, the energy overhead can be as low as 40-60% (depending on whether a shadow PC is added or not) when no duplication is required (i.e. the design is set to achieve higher performance at the cost of reduced fault tolerance). These overheads

could be reduced further if clock gating would be employed.

In short, the area and power consumption overheads are minor, in particular when compared to the amount of faults covered. The energy consumption is considerable, especially when the highest degree of fault tolerance is required. However, a trade-off can be made between performance and fault tolerance, both at processor-design time and at-runtime (through instruction level configurability). As the area and power consumption overheads are minimal, **the design can be considered viable for a biomedical implant with respect to the hard constraints of area, power consumption and, to some degree, in energy consumption.**

To answer the second question “How do the techniques used in the proposed fault-tolerance architecture compare (in terms of area, power, performance and fault-coverage) to other, well-known techniques?”, we have compared the overheads and fault coverage of our techniques to two other fault-tolerance techniques. It is shown that both triple modular redundancy and self-checking circuits come with overheads which are similar or exceed those of our technique. The techniques used in our architecture find a balance between techniques which have a high performance overhead (software-based instruction duplication) and techniques which have high area and power consumption overheads (triple modular redundancy and self-checking circuits), while having a similarly high fault coverage.

It has been shown that the performance and energy consumption are the major overheads of the techniques used in our architecture. However, comparing the added fault tolerance to the overheads of the techniques presented in section 2.5, **our used techniques may be considered better for a biomedical implant than the other techniques, given that a non-trivial decrease in performance and an increase in energy consumption may be tolerable.**

5.3 Thesis Contributions

The work presented in this thesis has addressed a number of issues concerning the design of a fault-tolerant processor, suitable for biomedical implants. The main contributions in this thesis are:

- A survey on a number of architecture level techniques, out of which a subset of techniques is combined to create a new fault-tolerant architecture suitable for biomedical implants targeting both soft- and hard faults,
- The design and implementation of the aforementioned techniques, and applying them to the Baseline processor. Our architecture consists of the following details:
 - Instruction duplication is done by the processor at run-time rather than at compile-time. as originally proposed in [29]. This allows us to make a trade-off between fault tolerance and overheads at runtime;
 - A parity bit guards against opcode corruption. This allows the “original” instruction to corrupt as well as the duplicated instruction, which is a limitation in the technique proposed in [3];

- A shadow program counter guards against both soft and hard faults in the program counter. This idea is a straightforward implementation of what has been proposed in [3];
 - Additional hardware is added to each stage for the storing and comparing of the committing results, i.e. error detection. This is similar to what has been done in [3] for the target-address of control-flow operations. However, we are novel in applying the same technique to all stages which commit results;
 - Error correction is done by re-fetching from the last correctly executed instruction onward, and flushing the pipeline. As this is a preemptive technique, this method requires no mechanism to roll the processor back to the last known correct state, which is a key difference when compared to checkpointing (section 2.5.1.1).;
 - A method to allow the design to make a trade-off between performance and fault tolerance by means of instruction-level configurability, depending on the needs of the application. The required degree of fault tolerance is set for a subset of instructions in a block of code, using a custom instruction. Using a custom instruction is one of the two ways of setting the degree of fault tolerance as proposed in [25]. This method allows for less flexibility but a less restricted instruction set as compared to the other possible implementation (instruction flags). We are novel in using this technique in combination with aforementioned techniques;
 - Test sequences are created to test the functionally testable paths in the processor. The methodology used to derive the test sequences is the same as in [12];
 - Scheduling these sequences is done in a round-robin style, taking the place of stall (nop) instructions. This is similar to what has been proposed in [9], where it is stated that replacing the stalls with test sequences hides the performance overhead of executing these test sequences. Testing for hard faults in this way is very suitable to the baseline architecture, due to the high amount of stalls present in the programs compiled for the baseline. These stalls exist due to the inherent data-dependencies in the programs, and the dependencies occurring due to the 16-bit ISA; and
 - Specific hardware structures are added to the processor to test the functionally untestable paths. An example of a functionally untestable path is the logic involved in updating the program counter. This is similar to what has been proposed in [30] and is similar to design for testability.
- Evaluating this design in terms of added fault tolerance, area, power consumption, energy consumption and performance, along with a number of compound metrics,
 - Comparing this design to other designs and determining its viability for biomedical implants.

5.4 Future Work

The work presented in this thesis has answered a number of concerns related to a fault-tolerant architecture suitable for a biomedical implant. However, there are a number of topics which could use further investigation. These are:

- Determining the fault coverage of the test sequences, by means of fault-injection. It is expected that such an effort could not only classify the degree of fault detection added by the test sequences, but will also help to improve the existing test sequences;
- Adding fault recovery for hard faults. While detection is possible with the current techniques, the best result possible is to stop the processor in case a fault is detected. Fault recovery could come in the form of spare modules, or reconfigurable hardware. This will require an error to be isolated (rather than just detected), after which the spare module / reconfigurable hardware can replace the erroneous module or signal;
- In order to facilitate the previous point, fault isolation is required. It has to be investigated whether the current test sequences are enough to achieve this goal, or if additional support is required.
- Dynamic scheduling of test sequences. It is observed that in case the test sequences are scheduled statically in code which is executed multiple times (e.g. a for loop), the same sequence is executed over and over again. By dynamically scheduling the test sequence, a (much) higher coverage could be obtainable. As an alternative to this, loop unrolling could be possible. It is to be determined if this is possible within the limited area and energy budget biomedical implants have;
- The current compiler is not optimal, and improving the compiler could vastly improve the attainable performance. Moving the test-sequence generator to the compiler could further improve the obtainable results (amount of reachable nops, schedulable registers, etc.) when compared to the post-processing of the assembly file. Finally, moving more work to the compiler could reduce the processor area (e.g. by removing the stall-detection unit);
- Combining the hard and soft fault mechanisms may be beneficial. For example, when a soft fault is detected by the instruction duplication technique, a check is first performed to see if the hardware is not defect;
- Automated duplication policies. Rather than having the designer manually set and reset the required degree of fault tolerance, it can be interesting to specify the deadlines required by a function and let the compiler inject the duplication modes automatically. Furthermore, the processor could change the duplication policy at run-time depending on a number of reasons, for example when the battery is low;
- Clock gating the components which are added for fault tolerance when no duplication is required. Currently, the power consumption is practically the same

regardless of the duplication policy, as the duplication hardware is not shut down when it is not required to be active; and

- Apply instruction duplication when a nop is detected, regardless of the duplication policy, as in [32]. This allows for less transitions to occur in the pipeline, and can thus be used to reduce power.



Table A.1: SISC specifications (fixed)

Design Name	Area (units)	Crit. path	Exec. time (#cycles)	Power (uW)	Energy (uJ)
BaseLine	50292	11.76	24,825,020	212.617	131.956
Optimized1	50471	11.73	18,038,302	219.826	99.132
Optimized2	54185	11.77	14,479,006	219.679	79.518

The following errors have been corrected in the original SISC architecture (the resulting specifications are given in table A.1):

1. Removed an additional register (IR), which only buffered the signal between the fetch and decode stage. In turn, this IR actually made the system a 6-stage pipeline, rather than 5-stage which was intended.
2. Fixed some of the descriptions in the matcher (in the emulated modulo, divide and multiply functions, not enough basic blocks were added).
3. Fixed the scheduler. In Lisa, the compiler designer is only capable of accepting references to properly identify data dependencies. As SISC is written using pipeline registers, no scheduling was possible, as a dependency was seen between all the operands (even when they were not the same, e.g. r1 and r10). To allow the tool to automatically generate the data dependencies, references have been added to the required operands (i.e. the operands are now both pipeline registers and references). Note that in this case an additional area overhead can be added if the synthesis tool is not smart enough. To overcome this overhead, two processors exist: One using only pipeline references for synthesis, and one using both pipeline registers and the required references to create the compiler.

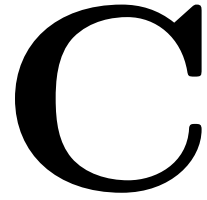
Colored fields are the instructions added to the ISA for the fault-tolerant processor designs.

Table B.1: Instruction Formats (baseline and optimizations)

Formats:																
rrr_type	opcode				rd				rs1				rs2			
rr_type	opcode				rd				rs1				funct			
ri_type	opcode				rd				imm							
jump_type	opcode				imm											
nop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
rr_type:																
beqz	0	0	0	1	rd				rs				0	0	1	1
sgtu	0	0	0	1	rd				rs				1	1	1	1
sgt	0	0	0	1	rd				rs				1	1	1	0
se	0	0	0	1	rd				rs				1	1	0	1
lb	0	0	0	1	rd				rs				1	0	1	1
jr	0	0	0	1	rd				rs				1	0	1	0
and	0	0	0	1	rd				rs				0	0	1	0
lw	0	0	0	1	rd				rs				1	0	0	0
mov	0	0	0	1	rd				rs				0	1	1	1
not	0	0	0	1	rd				rs				0	1	0	0
or	0	0	0	1	rd				rs				0	1	0	1
sw	0	0	0	1	rd				rs				1	0	0	1
xor	0	0	0	1	rd				rs				0	1	1	0
sub	0	0	0	1	rd				rs				0	0	0	1
add	0	0	0	1	rd				rs				0	0	0	0
ri_type:																
subi	0	1	1	1	rd				imm							
addi	0	1	1	0	rd				imm							
li	0	1	0	0	rd				imm							
sftru	0	1	0	1	rd				imm							
sftl	0	0	1	1	rd				imm							
sfr	0	0	1	0	rd				imm							
bneqz	1	0	1	1	rd				imm							
br_tst	1	1	1	0	rd				imm							
rrr_type:																
cb	1	1	0	1	rd				rs1				rs2			
jump_type:																
jal	1	0	0	1	imm											
j	1	0	0	0	imm											
jalo	1	1	1	1	imm											

Table B.2: SISC ISA (baseline and optimizations)

name	format	assembly	action
jr	rr	jr rd	branch to addr in rd
and	rr	and rd,rs	$rd \leftarrow rd \text{ and } rs$
lw	rr	lw rd,rs	$rd \leftarrow \text{mem}[rs]$
sw	rr	sw rd,rs	$\text{mem}[rs] \leftarrow rd$
mov	rr	mov rd,rs	$rd \leftarrow rs$
not	rr	not rd,rs	$rd \leftarrow \text{not } rs$
or	rr	or rd,rs	$rd \leftarrow rd \text{ or } rs$
xor	rr	xor rd,rs	$rd \leftarrow rd \text{ xor } rs$
sub	rr	sub rd,rs	$rd \leftarrow rd - rs$
add	rr	add rd,rs	$rd \leftarrow rd + rs$
lb	rr	lb rd,rs	$rd \leftarrow \text{mem}[rs]$ (byte)
se	rr	se rd,rs	if($rd == rs$) then $rd \leftarrow 1$ else $rd \leftarrow 0$
sgt	rr	sgt rd,rs	if($rd > rs$) then $rd \leftarrow 1$ else $rd \leftarrow 0(\text{sgn})$
sgtu	rr	sgtu rd,rs	if($rd > rs$) then $rd \leftarrow 1$ else $rd \leftarrow 0(\text{uns})$
beqz	rr	beqz rd,rs	if($rd == 0$) then branch to $\text{Imem}[rs2]$
subi	ri	subi rd,imm	$rd \leftarrow rd - \text{imm}$
addi	ri	addi rd,imm	$rd \leftarrow rd + \text{imm}$
li	ri	li rd,rs	$rd \leftarrow \text{imm}$
sftl	ri	sftl rd,rs	$rd \leftarrow rd \ll \text{imm}$
sftr	ri	sftr rd,rs	$rd \leftarrow rd \gg \text{imm}(\text{sign extension})$
sftru	ri	sftru rd,rs	$rd \leftarrow rd \gg \text{imm}(\text{zero extension})$
bneqz	ri	bneqz rd,imm	if($rd \neq 0$) then branch to $\text{Imem}[\text{imm}]$
br_tst	ri	br_tst rd,imm	execute bneqz with arguments generated based on imm. Do not branch, but check target address.
cb	rrr	cb rd,rs1,rs2	exchanges the $rs2^{th}$ -byte of rd by the LSB of rs1
j	jump	j imm	branch to $\text{Imem}[\text{imm}]$
jal	jump	jal imm	branch to $\text{Imem}[\text{imm}]$ and $r15 \leftarrow PC + 1$
jalo	jump	jalo imm	$\text{JalReg} \leftarrow \text{imm}$
jal	jump	jal imm	branch to $\text{Imem}[\text{JalReg} \& \text{imm}]$ and $r15 \leftarrow PC + 1$



List of Test Sequences

In this appendix, the designed test sequences are given. For each test sequence, only the signals not reported before or unique to the test sequence are reported, to conserve space. As such, it should be noted many test sequences actually cover the same signals, but each of them also covers a unique datapath.

For each sequence, it is reported what the sequence is, and what signals it covers.

Sequence 1:

Li rX, 0

Bneqz rX, #trap

Covers:

- SA1 IR(7 downto 0) (DE)
- SA1 (signextended) (DE)
- SA1 (mux → OPR2) (DE)
- SA1 (op1→out1) (DE)
- SA1 in the register file (output) (DE)
- SA1 (IR 11 downto 8, register selection, depends on rX nr) (DE)
- SA0 (IR 11 downto 8, register selection, depends on rX nr) (DE)
- SA1 (OPR2) (EX)
- ALU: SA1 for result for li / mov type (EX, MEM, WB)
- SA1 Result data line (EX)
- SA1 mux→res (EX)
- SA1 res → memres (MEM)
- SA1 MEMRES → wb_data (WB)
- SA1 WBDATA (shown in DE)
- SA1 (all wb register lines, and associated pipeline registers, depends on rX nr) (ALL)

- SA0 (all wb register lines, and associated pipeline registers, depends on rX nr) (ALL)
- (IMPL: SA0 ri_type)

Note: Assuming we will use all registers, we can safely assume this instruction can always cover SA1/0 for wb register lines and register selection.

Sequence 2:

Li rX, 0x0F
Sftl rX, 4
Subi rX, 0xF0
Bneqz rX, #trap

Covers:

- SA0 li IR(7 downto 0) (DE)
- SA0 (signextended) (DE)
- SA0 (mux → OPR2) (DE)
- SA0 (opr2) (EX)

Can also be used to detect some of the combinational stuff due to the subi.

Sequence 3:

Mov rX, r0
Bneqz rX, #trap

Covers:

- SA1 for IR (7 downto 4, reg select) (DE)
- SA1 for out2 (DE)
- ALU: SA1 mov part result (EX)

Sequence 4:

Not rX, r0
Mov rX, rX
Not rX, rX
Bneqz rX, #trap

Covers:

- SA1 and SA0 for IR (7 downto 4, reg select, depending on register nr) (DE)
- SA1 and SA0 for out2

- ALU: SA0 mov/li part result (EX)
- ALU: SA1 for not part result (EX)

Sequence 5:

Mov rX, r0
Or rX, rX
Bneqz rX, #trap

Covers:

- ALU: SA1 at one of the inputs or the output of the OR (EX)

Sequence 6:

Not rX, r0
OR rX, r0
Not rX, rX
Bneqz rX, #trap

Covers:

- SA0 at the output of the OR (EX)
- SA0 at the first input of the OR (EX)

Sequence 7:

mov rX, r0
not rY, r0
OR rX, rY
Not rX, rX
Bneqz rX, #trap

Covers:

- SA0 at the output of the or (EX)
- SA0 at the 2nd input of the OR (EX)

Sequence 8:

Not rX, r0
And rX, rX
Not rX, rX
Bneqz rX, #trap

Covers:

- ALU: All SA0 in the AND (EX)

Sequence 9:

Not rX, r0
 Xor rX, r0
 Not rX, rX
 Bneqz rX, #trap

Covers:

- ALU: Detects SA1 on 2nd input, SA0 in 1st input, and a SA0 on the output of the xor (EX)

Sequence 10:

Mov rX, r0
 Not rY, r0
 Xor rX, rY
 Not rX, rX
 Bneqz rX, #trap

Covers:

- Alu: SA1 on 1st input, SA0 in 2nd input, and SA0 on output of the xor (EX)

Sequence 11:

Xor rX, rX
 Bneqz rX, #trap

Covers:

- Detects SA1 at the output. Note that we do not care what rX was

Sequence 12:

not rX, r0
 subi rX, 1
 bneqz rX, #trap

Covers:

- ALU: SA0 on not (EX)
- Note that SA1 is already covered by many others Eg seq 3
- Can detect some of the combinational errors inside the ALU.

Sequence 13:

Mov rX, r0
 Add rX, rX
 Bneqz rX, #trap

Covers:

- ALU: SA1 on practically any one of the lines in the adder (as any 1 will correspond to a sum / carry, which should always be seen at the output)

Sequence 14:

Not rX, r0
 Sub rX, rX
 Bneqz rX, #trap

Covers:

- ALU: May detect some SA0 on the input lines.

Sequence 15:

Li rX, \$val
 Subi rX, \$val
 Bneqz rX, #trap

Covers:

- ALU: SA1 / SA0 in lower 8 bits of the ALU

Sequence 16:

Li rX, \$valH1
 Sftl rX, 8
 Addi rX, \$valH2
 Sftl rX, 8
 Addi rX, \$valL1
 Sftl rX, 8
 Addi rX, \$valL2
 Sub rX, rX
 Bneqz rX, #trap

Covers:

- ALU: May be used to detect errors in the sub (full size), of which the inputs are always the same (as op1 and op2 are the same in the sub). Could also be used for add (will skip the inversion step of the 2nd operand and will not have a carry in for the first bit). Also detects error with addi, and this sequence could also be changed to incorporate subi.

Sequence 16:

Li rX, \$val
 Sftl rX, \$times
 Li rY, \$val2
 Sftl rY, \$times
 Sub rX, rY
 Sftl rX, \$times
 Subi rX, \$val2-\$val

Bneqz rX, #trap

Covers:

- Can be used to test subtraction for 2 different values (8 bit) with different patterns. One may for example contain 01010101, the other 00011101. First, we shift these 8 bit values to the location we want to test by shifting \$times. When in place, we can subtract, which will test the adder specific to these locations. Then we shift back and remove the remainder we expect, and see if we still have 0
- This can also be done to incorporate add / addi into the mix.
- ALU: SA1/SA0 for the sftl / sftl instructions.
- This is, again, highly dependent on the type of adder we synthesize. For example, a ripple carry adder would be fairly easy to test like this, as we only need inputs of 2-3 bits to simulate all signals. However, this is most likely not the case for other, more complex adders.

Sequence 17:

Sw r0, StackPointer

Lw rX, StackPointer

Bneqz rX, #trap

Covers:

- SA1 for opr1 → Exopr1 (EX)
- SA0 for opr1 → exopr1 depending on register nr (EX)
- SA1/SA0 for opr2 → Exopr2 depending on StackPointer(EX)
- Implicitly checks SA1 / SA0 (both) ex_wrmem_en
- SA1/SA0 for Exopr1 → data depending on contents (MEM)
- SA1/SA0 for Exopr2 → addr depending on contents (MEM)
- SA1 for out → keep (MEM)
- Sa1 keep → OUT (MEM)
- SA1 data → mux (WB)
- Implicitly checks SA1 for rd_mem_en
- Implicit checks SA1 for Byte_mem_en

Sequence 18:

Not rX, r0

Sw rX, StackPointer

Lw rY, StackPointer
 Not rY, rY
 Bneqz rY, #trap

Covers:

- SA0 for out → keep (MEM)
- SA0 for keep → data (MEM)
- Sa0 for data → mux (WB)

Sequence 19:

Sw r0, StackPointer
 Lb rX, StackPointer
 Bneqz rX, #trap

Covers:

- SA1 for expr1 → keep (MEM)
- Implicit checks SA0 for Byte_mem_en

Sequence 20:

Not rX, r0
 Sw rX, StackPointer
 Lb rY, StackPointer
 Subi rY, 0xFF
 Bneqz rY, #trap

Covers:

- SA0 for expr1 → keep (MEM)

Sequence 21:

Li rX, r0 ;;rX = 0
 Not rY, r1 ;;rY = FF
 Li rZ, \$val ;;rZ = \$val
 Cb rX, rY, rZ ;;rX = 0000FF00 (example, if rZ was 2)
 Sftl rX, 8*\$val ;;rX = 000000FF
 Subi rX, 0xFF
 Bneqz rX, #trap

Covers:

- SA1 / SA0 for IR (3 downto 0) (somewhat implicit) (DE)

- SA1 / SA0 for out3 $\rightarrow$ opr3 (DE)
- SA1 exopr1 $\rightarrow$ CB (EX)
- SA0 exopr2 $\rightarrow$ CB (lower 8 bits, but this is all we need to check for CB) (EX)
- SA0/SA1 for opr3 $\rightarrow$ CB (EX)
- SA1 for all bytes not covered by rZ (cb $\rightarrow$ resmux) (EX)
- SA0 for all bytes covered by rZ (cb $\rightarrow$ resmux) (EX)

Sequence 22:

Li rX, \$val
 Se rX, rX
 Beqz rX, #trap

Covers:

- ALU: SA0 for the zero flag used for SE (EX)

Sequence 23:

Li rX, \$val
 Se rX, r0
 Bneqz rX, #trap

Covers:

- ALU: SA1 for the zero flag used for se (\$val;0) (EX)

Sequence 24:

Li rX, \$val
 Sgt rX, rX
 Beqz rX, #trap

Covers:

- ALU: SA1 for the sign flag used for sgt / sgtu (EX)

Sequence 25:

Li rX, \$val
 Sgt rX, r0
 Beqz rX, #trap

Covers:

- ALU: SA0 for the zero flag used for se (EX)

Sequence 26:

Not rX, r0

Sftru rX, 32

Bneqz rX, #trap

Covers:

- ALU: SA0 in the sftru operand (EX)
- ALU: SA1 in the sftru result (EX)

Sequence 27:

Not rX, r0

Sftru rX, 0

Not rX, rX

Bneqz rX, #trap

Covers:

- ALU: SA1 in the sftru operand (EX)
- ALU: SA0 in the sftru result (EX)

Processor / Compiler Designer Review

D

In this appendix, a short user review of the Processor / Compiler Designer tools from Synopsis is given. The tool provides an excellent for rapidly designing a processor and associated compiler. However, there are a few sticky points in both the documentation, (missing) functionality and user experience. Furthermore, some additions to the tool would be highly beneficial to both the functionality of the tool and user experience. The following points deserve some attention:

1. A number of topics are either missing or severely hard to find in the documentation. While most information is present in the documentation, some issues faced during this project are hard to find and underestimated by the writer of the documentation. For example, the fact pipeline registers cannot be used for the compiler designer is only mentioned in two locations as a “note”, stating that a pipeline register-based architecture *may* generate sub-optimal scheduler (as far as we have seen, this means it *no* code scheduling is done whatsoever) This is a key parameter for the design, and it would have been much better to not only highlight this difference at the start of several topics (e.g. the difference between pipeline registers and references). , but possibly even make a note on this when the compiler designer detects no references are in the code.
2. Taking up on the previous point, it should be possible that pipeline registers can be used for creating a compiler. The knowledge of data dependencies can be known statically, and as such there is no reason why this limitation exists.
3. The undo and redo function is a mess. It appears the buffer is not flushed after an action, and as such redoing is still possible after an undo. Older code which has been undone will often reappear at the original location when a redo is executed, messing up the code. Furthermore, when the user tries to “undo” this faulty redo, the faulty redo will not be undone: rather, the last taken action will be, messing up the code even further.
4. In some cases, some of the functions working on the bit data type will generate incorrect VHDL code (e.g. referring to a sub-vector of bits 9 downto -3), without giving any form of warning. Care has to be taken using these functions.
5. Related to the previous point, some definite improvements can be made in the error and warning reports. While most warnings are fine and can be at least traced back to the function body the error has occurred, a number of warnings are only given after for example processor generation. The errors given in this tool usually refer to locations in the intermediate representation rather than locations in the actual

lisa code. If the processor is not generated frequently, finding this error can be very tricky, and has led to a number of design roll-backs to prevent the error.

6. Extracting the compiler to generate CoSy is a mess. Other than numerous missing files when attempting to extract the compiler, the extraction seems to be targeting windows-only, as it is looking for files in very specific folders (e.g. `remwin32.exe`). Any modification to the compiler is therefore only possible by re-creating the entire compiler by means of the micromenu and `edl` files found in the `CoWare/.../compilers/cecc` folder, followed by copying the generated engine files (by the compiler generator) such as the `.cgd` files in the destination folders.
7. More options should be given in the compiler designer for generating the compiler. While the tool itself is very good at creating a basic compiler, high value could be obtained by creating a GUI overlay for other parts in CoSy (e.g. engine additions). It is understood that there is probably no license for this with ACE(CoSy), although this could definitely increase the value of the product (and thus be beneficial to both parties).
8. A tutorial (rather than the design guides) would speed-up learning the tool, due to the amount of information required to create a design.

All in all, the tool is very decent for rapid prototyping architectures using a form of an instruction memory.



Benchmarks

emptyMain

```
\breaklines=true
```

```
\frame=none
```

```
int main(void){
```

```
    unsigned int res;
    int i=0;
    unsigned int MN = 5;
    unsigned int rec_RN;
    unsigned int rec_len;
    unsigned int cmd;
    unsigned int rec_crc;
```

```
    int pos;
    if (one==0)
        one++;
    return 1;
```

```
}
```

```
keys (main)
```

```
int main(void){
```

```
    unsigned int res;
    int i=0;
    unsigned int MN = 5;
    unsigned int rec_RN;
    unsigned int rec_len;
    unsigned int cmd;
    unsigned int rec_crc;
```

```
    int pos;
```

```
    if (one==0){
        //generating the the expanded key
        misty1_keyinit(ek_e);
        misty1_key_destroy(ek_e);
        one++;
    }
```

```

    }
    return 1;
}

decrypt (main)
int main(void){

    unsigned int res;
    int i=0;
    unsigned int MN = 5;
    unsigned int rec_RN;
    unsigned int rec_len;
    unsigned int cmd;
    unsigned int rec_crc;

    int pos;

    if (one==0){
        blockin[0]=msg[0];
        blockin[1]=msg[1];
        misty1_decrypt_block(ek_e);
        rec_len= blockout[0];
        rec_RN= blockout[1];
        one++;
    }
    return 1;
}

encrypt (main)
int main(void){

    unsigned int res;
    int i=0;
    unsigned int MN = 5;
    unsigned int rec_RN;
    unsigned int rec_len;
    unsigned int cmd;
    unsigned int rec_crc;

    int pos;

    if (one==0){

        pos = 3;
        blockin[0]=inputAr[pos];

```

```

        blockin[1]=inputAr[pos+1];

        misty1_encrypt_block(ek_e);

        outputAr[pos]=blockout[0];
        outputAr[pos+1]=blockout[1];

        one++;
    }
    return 1;
}

crc (main)
int main(void) {

    unsigned int res;
    int i=0;
    unsigned int MN = 5;
    unsigned int rec_RN;
    unsigned int rec_len;
    unsigned int cmd;
    unsigned int rec_crc;

    int pos;

    if (one==0){
        pos=4;
        bcopy_u4_byte_LittleEnd(inputAr[pos], 0);
        bcopy_u4_byte_LittleEnd(inputAr[pos+1], 4);
        crc_acc += update_crc (crc_acc, packet, 8);
        one++;
    }
    return 1;
}

gen_crc_table
void gen_crc_table() {
    int i, j;
    unsigned int crc_accum;

    for (i = 0; i < 256; i++) {
        crc_accum = ((unsigned int) i << 24);
        for (j = 0; j < 8; j++) {
            if (crc_accum & 0x80000000L)

```

```

        crc_accum = (crc_accum << 1) ^
            0x04c11db7L;
    else
        crc_accum = (crc_accum << 1);
    }
    crc_table[i] = crc_accum;
}
return;
}

```

update_crc

unsigned int

update_crc(unsigned int crc_accum, unsigned char
***data_blk_ptr, int data_blk_size) {**

```

    int i, j;

    for (j = 0; j < data_blk_size; j++) {
        i = ((int)(crc_accum >> 24) ^ *data_blk_ptr++)
            & 0xff;
        crc_accum = (crc_accum << 8) ^ crc_table[i];
    }
    return crc_accum;
}

```

fi

unsigned int fi(unsigned int fi_in, unsigned int fi_key)

```

{
    unsigned int d9, d7;

    d9 = (fi_in >> 7) & 0x1ff;
    d7 = fi_in & 0x7f;
    d9 = s9[d9] ^ d7;
    d7 = (s7[d7] ^ d9) & 0x7f;

    d7 = d7 ^ ((fi_key >> 9) & 0x7f);
    d9 = d9 ^ (fi_key & 0x1ff);
    d9 = s9[d9] ^ d7;
    return ((d7<<9) | d9);
}

```

bcopy_u4_byte_LittleEnd

void bcopy_u4_byte_LittleEnd(unsigned int kl, int pos) {
 packet[3+pos]=(kl>>24)&0xFF;
 packet[2+pos]=(kl>>16)&0xFF;

```

    packet[1+pos]=(kl>>8)&0xFF;
    packet[0+pos]= kl &0xFF;
}

```

bcopy_u4_byte

```

void bcopy_u4_byte(unsigned int kl, int pos) {
    k[0+pos]=(kl>>24)&0xFF;
    k[1+pos]=(kl>>16)&0xFF;
    k[2+pos]=(kl>>8)&0xFF;
    k[3+pos]= kl &0xFF;
}

```

misty1_keyinit

```

void misty1_keyinit(unsigned int *ek)
{
    int i;

    bcopy_u4_byte(key[0],0);
    bcopy_u4_byte(key[1],4);
    bcopy_u4_byte(key[2],8);
    bcopy_u4_byte(key[3],12);

    for(i=0; i < 8 ; i++) {
        ek[i] = (k[i+i]*256) + (k[(i+i) +1]);
    }

    for (i=0 ; i < 8 ; i++) {
        ek[i+8] = fi(ek[i],ek[(i+1)%8]);
        ek[i+16] = ek[i+8] & 0x1ff;
        ek[i+24] = ek[i+8] >> 9;
    }
}

```

misty1_key_destroy

```

void misty1_key_destroy(unsigned int *ek)
{
    int i=0;

    for(i=0;i<32;i++)
        ek[i]=0;
}

```

fo

```

unsigned int fo(unsigned int *ek, unsigned int fo_in,
    unsigned char k)

```

```

{
    unsigned int    t0, t1;

    t0 = (fo_in >> 16);
    t1 = fo_in & 0xFFFF;
    t0 ^= ek[k];
    t0 = fi(t0, ek[((k+5)%8) + 8]);
    t0 ^= t1;
    t1 ^= ek[(k+2)%8];
    t1 = fi(t1, ek[((k+1)%8) + 8]);
    t1 ^= t0;
    t0 ^= ek[(k+7)%8];
    t0 = fi(t0, ek[((k+3)%8)+8]);
    t0 ^= t1;
    t1 ^= ek[(k+4)%8];
    return ( (t1<<16) | t0 );
}

fl
unsigned int fl(unsigned int *ek, unsigned int fl_in,
    unsigned char k)
{
    unsigned int    d0, d1;
    unsigned char t;

    d0 = (fl_in >> 16);
    d1 = fl_in & 0xffff;
    if ( k % 2 ) {
        t=(k-1)/2;
        d1 = d1 ^ (d0 & ek[((t+2)%8)+8]);
        d0 = d0 ^ (d1 | ek[(t+4)%8]);
    }
    else {
        t=k/2;
        d1 = d1 ^ (d0 & ek[t]);
        d0 = d0 ^ (d1 | ek[((t+6)%8)+8]);
    }
    return ((d0<<16) | d1);
}

flinv
unsigned int flinv(unsigned int *ek, unsigned int fl_in,
    unsigned char k)
{
    unsigned int d0, d1;

```

```

unsigned char t;

d0 = (fl_in >> 16);
d1 = fl_in & 0xffff;

if ( k % 2 ) {
    t = (k-1)/2;
    d0 = d0 ^ (d1 | ek[(t+4)%8]);
    d1 = d1 ^ (d0 & ek[((t+2)%8)+8]);
}
else {
    t = k/2;
    d0 = d0 ^ (d1 | ek[((t+6)%8)+8]);
    d1 = d1 ^ (d0 & ek[t]);
}
return ((d0<<16) | d1);
}

misty1_encrypt_block

unsigned int misty1_encrypt_block(unsigned int *ek)
{
    unsigned int d0, d1;

    d0 = blockin[0];
    d1 = blockin[1];
    /* 0 round */
    d0 = fl(ek,d0,0);
    d1=f1(ek,d1,1);
    d1 = d1 ^ fo(ek,d0,0);

    /* 1 */
    d0 = d0^fo(ek,d1,1);

    /* 2 */
    d0 = fl(ek,d0,2);
    d1=f1(ek,d1,3);
    d1 = d1 ^ fo(ek,d0,2);

    /* 3 */
    d0 = d0^fo(ek,d1,3);

    /* 4 */
    d0 = fl(ek,d0,4);
    d1=f1(ek,d1,5);
    d1 = d1 ^ fo(ek,d0,4);

```

```

/* 5 */
    d0 = d0 ^ fo(ek, d1, 5);

/* 6 */
    d0 = fl(ek, d0, 6);
    d1 = fl(ek, d1, 7);
    d1 = d1 ^ fo(ek, d0, 6);

/* 7 */
    d0 = d0 ^ fo(ek, d1, 7);

/* 8 */
    d0 = fl(ek, d0, 8);
    d1 = fl(ek, d1, 9);

/* round finished */
    blackout[0] = d1;
    blackout[1] = d0;

    return d1;
}

misty1_encrypt_block

void misty1_decrypt_block(unsigned int *ek)
{
    unsigned int d0, d1;

    d0 = blockin[1];
    d1 = blockin[0];

    d0 = flinv(ek, d0, 8);
    d1 = flinv(ek, d1, 9);
    d0 = d0 ^ fo(ek, d1, 7);
    d1 = d1 ^ fo(ek, d0, 6);
    d0 = flinv(ek, d0, 6);
    d1 = flinv(ek, d1, 7);
    d0 = d0 ^ fo(ek, d1, 5);
    d1 = d1 ^ fo(ek, d0, 4);
    d0 = flinv(ek, d0, 4);
    d1 = flinv(ek, d1, 5);

```

```
d0 = d0 ^ fo(ek,d1, 3);
d1 = d1 ^ fo(ek,d0, 2);
d0 = flinv(ek,d0, 2);
d1 = flinv(ek,d1, 3);
d0 = d0 ^ fo(ek,d1, 1);
d1 = d1 ^ fo(ek,d0, 0);
d0 = flinv(ek,d0, 0);
d1 = flinv(ek,d1, 1);
    blackout[0]=d0;
    blackout[1]=d1;
}
```


Bibliography

- [1] D. Siskos, “A co-processor for a secure implantable medical device,” *MSc Thesis*, 2011.
- [2] K. P. e.a., “Cosafe low power hardware-software co-design for safety-critical applications,” *IEEE International Workshop on Biomedical Circuits & Systems*, 2004.
- [3] R. G. Ragel and S. Parameswaran, “Hardware assisted pre-emptive control flow checking for embedded processors to improve reliability,” *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pp. 100–105, 2006.
- [4] I. S. I.D. Elliott, “Implementation of 32-bit risc processor incorporating hardware concurrent error detection and correction,” *Computers and Digital Techniques, IEE Proceedings E*, 1990.
- [5] J. H. Patel and L. Y. Fung, “Concurrent error detection in alu’s by recomputing with shifted operands,” *IEEE Transactions on computers*, vol. C-31, pp. 589–595, 1982.
- [6] M. E. B. Albert Meixner and D. Sorin, “Argus: Low-cost, comprehensive error detection in simple cores,” *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 210–222, 2007.
- [7] T. M. Austin, “Diva: a reliable substrate for deep submicron microarchitecture design,” *32nd Annual International Symposium on Microarchitecture*, pp. 196–207, 1999.
- [8] B. W. Johnson, *The Electrical Engineering Handbook*. 2000.
- [9] H. E. Saeed shamshiri and Z. Navabi, “Instruction-level test methodology for cpu core self-testing,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, p. 673689, 2005.
- [10] “<http://sims.et.tudelft.nl/>,” Consulted on 25/09/2011.
- [11] M. R. Lyu, *Software Fault Tolerance*. 1995.
- [12] D. G. Nektarios Kranitis, Antonis Paschalis and G. Xenoulis, “Software-based self-testing of embedded processors,” *IEEE Transactions on computers*, vol. 54, pp. 461–475, 2005.
- [13] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. 2000.
- [14] D. G. N. Kranitis, A. Paschalis and Y. Zorian, “Effective software self-test methodology for processor cores,” *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002.

- [15] S. Kamal and C. V. Page, "Intermittent faults: A model and a detection procedure," *IEEE Transactions on computers*, vol. c-23, pp. 713–719, 1974.
- [16] N. J. Warter and W.-M. W. Hwu, "A software based approach to achieving optimal performance for signature control flow checking," *Proc. 20th Intl Symp. on Fault-Tolerant Computing Systems*, p. 442449, 1990.
- [17] X. Delord and G. Saucier, "Formalizing signature analysis for control flow checking of pipelined risc microprocessors," *In Proc. 20th Intl Symp. on Fault-Tolerant Computing Systems*, p. 936945, 1991.
- [18] S. Kim and A. K. Somani, "On-line integrity monitoring of microprocessor control logic," *Proc. of the Intl Conf. on Computer Design*, p. 314319, 2001.
- [19] A. Meixner and D. J. Sorin, "Error detection using dynamic dataflow verification," *Conf. on Parallel Architectures and Compilation Techniques*, pp. 104–115, 2007.
- [20] S. G. M. Alireza Vahdatpour, Mahdi Fazeli, "Transient error detection in embedded systems using reconfigurable components," *International Symposium on Industrial Embedded Systems, 2006. IES '06.*, pp. 1 – 6, 2006.
- [21] E. Rotenberg, "Ar-smt: a microarchitectural approach to fault tolerance in microprocessors," *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pp. 84–91, 1999.
- [22] J. Gaisler, "Evaluation of a 32-bit microprocessor with built-in concurrent error-detection," *Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing*, pp. 42 – 46, 1997.
- [23] M. Tremblar and Y. Tamir, "Support for fault tolerance in vlsi processors," *IEEE International Symposium on Circuits and Systems*, vol. 1, pp. 388–392, 1989.
- [24] M. T. K. Gokhan Memik and O. Ozturk, "Increasing register file immunity to transient errors," *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2005.
- [25] B. B. J. Demid Borodin and S. Vassiliadis, "Instruction-level fault tolerance configurability," *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 110–117, 2007.
- [26] J. Shen and J. Abraham, "Native mode functional test generation for processors with applications to self-test and design validation," *Proceedings of the International Test Conference*, pp. 990–999, 1998.
- [27] G. F. Corno, M. Sonza Reorda and M. Violante, "On the test of microprocessor ip cores," *Design Automation & Test in Europe*, 2001.
- [28] A. K. Wei-Cheng Lai and K.-T. T. Cheng, "Test program synthesis for path delay faults in microprocessor cores," *ITC international test conference*, pp. 1080–1089, 2000.

- [29] M. Franklin, "A study of time redundant fault tolerance techniques for superscalar processors," *IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, p. 207215, 1995.
- [30] W.-C. Lai and K.-T. T. Cheng, "Instruction-level dft for testing processor and ip cores in system-on-a-chip," *Proceedings of Design Automation Conference*, 2001.
- [31] P. Alfka, "Efficient shift registers, lfsr counters, and long pseudo-random sequence generators," *Application note (XAPP 052)*, 1996.
- [32] P. e. Lotfi-Kamran, "Stall power reduction in pipelined architecture processors," *21st International Conference on VLSI Design*, pp. 541 – 546, 2008.

