Masterarbeit

Design and Implementation of a Fault Injection System for Verifying Generic CPU Safety-Tests

Christopher Preschern

Institut für Technische Informatik
Technische Universität Graz
Vorstand: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

Begutachter: Ass.Prof. Dipl.-Ing. Dr.techn Christian Steger
Betreuer: Ass.Prof. Dipl.-Ing. Dr.techn Christian Steger
Dipl.-Ing. Dr.techn. Christian Josef Kreiner

Graz, im Jänner 2014

Kurzfassung

Die vorliegende Arbeit beschäftigt sich mit Selbsttests von CPU-basierten Systemen. Für kritische Anwendungen ist es oft nötig sicherzustellen, dass defekte Hardware nicht zu einer Fehlfunktion des Gesamtsystems führen kann. Bei einer CPU muss zum Beispiel ständig überprüft werden, ob die internen Register funktionstüchtig sind und ob die CPU Instruktionen sich wie spezifiziert verhalten. Um dies zu überprüfen werden zur Laufzeit des kritischen Programms zusätzliche Berechnungen durchgeführt welche das korrekte Verhalten der CPU testen. Das sind die sogenannten CPU Selbsttests.

In dieser Arbeit wird ein Überblick über Selbsttests von CPU-basierten Systemen gegeben. Speziell wird darauf eingegangen wie solche Tests in Safety-kritischen Systemen auszusehen haben bzw. welche Komponenten getestet werden. Safety-kritische Systeme sind Systeme in welchen eine Fehlfunktion zu Personenschäden führen kann. Insbesondere werden die Tests auf den IEC 61508 Safety Standard ausgelegt. Der Hauptteil der Arbeit spezialisiert sich auf die Tests der CPU-internen Komponenten (Registerbänke, Program Counter, ...) und es werden CPU-Architektur unabhängige Tests vorgestellt, welche diese CPU-internen Komponenten testen. Zur Verifikation der Tests wird ein System zur Fehlerinjektion entwickelt, welches gezielt Fehler in der CPU simulieren kann um die korrekte Funktionalität der Tests zu überprüfen. Mit diesem Fehlerinjektionssystem wird für zwei CPU Architekturen gezeigt, dass die vorgestellten Safety-Tests eine genügend hohe Fehlererkennung haben um sie in einem Safety-kritischen System mit Sicherheitsintegritätslevel 2 (SIL 2) nach dem IEC 61508 Standard einzusetzen.

Abstract

This thesis covers self-tests for critical CPU-based systems. For critical applications it is necessary to ensure that faults in the hardware do not lead to overall system failures. For CPUs this means that the correct functionality of the CPU components (e.g. internal registers, program counter, ...) has to be checked during runtime. It has to be ensured that the CPU works as specified. To check the CPU functionality, additional self-test programs are executed.

This thesis gives an overview of self-tests for CPU based systems. It is described which parts of CPU-based systems have to be tested and which faults have to be detected in critical systems. More specifically, safety-critical systems are handled. A system is safety-critical, if a malfunction of the system can cause human injuries or even poses a threat to human lives. In particular, the focus is on the IEC 61508 safety standard and the tests are designed to fit the requirements of this safety standard. The main part of this thesis focuses on CPU architecture independent safety self-tests for the core components of the CPU (internal registers, program counter, ...). For the verification of these safety-tests, a fault injection system is developed and used to inject specific faults into CPU registers. These faults should be detected by the safety-tests. This fault injection approach is applied to two different CPU architectures to show that the proposed generic safety-tests detect a sufficiently high number of faults to be used in systems with IEC 61508 safety integrity level 2 (SIL 2).

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Credits

This master thesis was carried out at the Institute for Technical Informatics, Graz University of Technology.

At this point I want to thank my girlfriend Silke, my friends, and my family for enduring me during the time of my study.

Special thanks to Christian Steger and Christian Kreiner who helped me when working on this thesis with much good advice. Without them it would have been a lot more difficult to achieve the aims of this thesis. I also want to thank Armin Krieg and Johannes Grinschgl for helping me to set up the fault injection system.

Graz, Jänner 2014 Christopher Preschern

Contents

1 Introduction **10**
1.1 Motivation and Goal . . . 10
1.2 Outline . . . 11

2 Technical Background and Related Work **12**
2.1 Safety-critical Systems . . . 12
2.1.1 Introduction to Safety-Critical Systems and to Safety Standards . . 12
2.1.2 Fault, Error, Failure . . . 13
2.1.3 IEC 61508 . . . 14
2.2 Software-based Self-Tests for CPU-based Systems . . . 17
2.2.1 Fault Types . . . 18
2.2.2 Safe Failure Fraction and Diagnostic Coverage . . . 18
2.2.3 Memory Tests and IEC 61508 . . . 19
2.2.4 RAM Test Methods . . . 21
2.2.5 Cache Test Methods . . . 25
2.2.6 CPU Core Tests and IEC 61508 . . . 28
2.2.7 CPU Core Tests . . . 30
2.3 Fault Injection . . . 32
2.3.1 Purpose of Fault Injection . . . 32
2.3.2 Generic Fault Injection Architecture . . . 32
2.3.3 Hardware-Based Fault Injection . . . 33
2.3.4 Software-Based Fault Injection . . . 34
2.3.5 Simulation-Based Fault Injection . . . 34
2.4 Related Work . . . 35
2.4.1 Deterministic Tests for CPU Components . . . 36
2.4.2 Indirect CPU Testing . . . 37
2.4.3 Safety-related CPU Fault Injection Frameworks . . . 37

3 Design of a Fault Injection System for the Verification of Generic CPU Safety-Tests **39**
3.1 System Requirements . . . 39
3.2 Fault Injection System Design . . . 40
3.3 Design of Generic CPU Safety-Tests . . . 41
3.3.1 CPU Core Tests . . . 41
3.3.2 Quicksort Test . . . 42

3.3.3 SHA-1 Test . . . 42
3.3.4 CRC Test . . . 43
3.3.5 March Test . . . 43

4 Implementation of the Fault Injection System 47
4.1 Simulation Testbench . . . 47
4.2 TCL Script . . . 48
4.2.1 Forcing Modelsim Values . . . 49
4.2.2 Watching Modelsim Values . . . 49
4.2.3 Reading Modelsim Values . . . 50
4.2.4 Execution Failures . . . 50
4.3 Software Components . . . 51
4.4 Setting Up Processor Testbenches . . . 52
4.4.1 LEON3 Testbench . . . 52
4.4.2 Plasma/MIPS Testbench . . . 53

5 Verification and Evaluation of the Safety-Tests by Fault Injection 55
5.1 Processor Architecture . . . 55
5.2 Fault Injection Target Registers . . . 57
5.3 Test Results . . . 58
5.3.1 Diagnostic Coverage . . . 58
5.3.2 Running the Tests on a Safety-Critical Device . . . 60
5.3.3 Simulation Time . . . 61
5.4 Discussion of the System Requirements . . . 62

6 Conclusion 64
6.1 Discussion of the Results . . . 64
6.2 Future Work . . . 65

A TCL Source Code 67

B Source Code of the Generic Test Programs 71
B.1 Main Test Routine . . . 71
B.2 CPU Core Tests . . . 71
B.3 Quicksort Test . . . 75
B.4 SHA-1 Test . . . 76
B.5 CRC Test . . . 76
B.6 March Test . . . 78

C List of Registers for Fault Injection 80
C.1 Plasma/MIPS . . . 80
C.2 Leon3 . . . 80

D Publication 82

Biblography 86

List of Figures

2.1 Relation between Faults, Errors, and Failures 14
2.2 IEC 61508 Safety Lifecycle 15
2.3 2-set Associative Cache Memory [DPS11] 26
2.4 Mapping of March Test Commends to Cache Tests [DPS11] 27
2.5 MATS March Test for Cache Memories 27
2.6 Typical Fault Injection Components and Interactions (based on [BP03]) . . 33

3.1 Components of the Fault Injection System 41
3.2 Basic Structure of the CPU Muliplier and Divider Tests 43
3.3 Basic Structure of the ALU and Shifter Tests 44
3.4 Basic Structure of the Quicksort and SHA-1 Tests 45
3.5 Basic Structure of the CRC and March Tests 46

4.1 Modelsim Testbench 48
4.2 Fault Injection System TCL Script 49
4.3 Simplified Version of the TCL Code 50
4.4 Involved Software Components 51
4.5 Test Run Sequence 51
4.6 Xconfig Tool 52
4.7 Xgrlib Tool 53

5.1 Workflow for Simulating the Self-Tests 55
5.2 Plasma/MIPS Architecture (opencores.org/project,plasma) 56
5.3 LEON3 Architecture [Aer13] 56
5.4 Overview of the Registers where the Faults are Injected 58
5.5 Sequence of the Fault Injections 61

List of Tables

2.1 Software Design and Development Techniques and Measures [Int10] 16
2.2 Failure Probability for Different SILs [Int10] 16
2.3 Maximum Allowed SIL for a Given Safe Failure Fraction Depending on the Level of Hardware Fault Tolerance [Int10] 19
2.4 IEC 61508 Memory Test Requirements [Int10] 20
2.5 IEC 61508 Memory Test Techniques/Measures for Invariable Memories [Int10] 21
2.6 IEC 61508 Memory Test Techniques/Measures for Variable Memories [Int10] 22
2.7 March Memory Test Notation 24
2.8 Commonly Used March Test Sequences [TB06] 24
2.9 IEC 61508 CPU Test Requirements [Int10] 29
2.10 IEC 61508 CPU Core Test Techniques/Measures [Int10] 30
2.11 Hardware-Based Fault Injection Approaches Described in Literature 34
2.12 Software-Based Fault Injection Approaches Described in Literature 35
2.13 Simulation-Based Fault Injection Approaches Described in Literature 36

5.1 Plasma/MIPS Stuck-At and Transient Fault Diagnostic Coverage of the Generic Test Programs 59
5.2 LEON3 Stuck-At and Transient Fault Diagnostic Coverage of the Generic Test Programs 59
5.3 Number of Test Runs for each Register 62

Chapter 1

Introduction

Developing safety-critical systems according to industry standards is mandatory in most safety-critical applications. Part of the requirements for safety-critical CPU-based systems is to ensure that the CPU functions as intended. One way to ensure that is to use self-tests for the CPU. This thesis presents generic CPU self-tests and validates them against the requirements of the IEC 61508 safety standard.

1.1 Motivation and Goal

Safety-critical systems are system, which pose a threat to human health of even to human lives if they malfunction. To ensure that such malfunctions are very unlikely, safety-critical systems are usually carefully developed and rigorously tested. Safety standards provide requirements and guidelines to construct safety-critical systems and in industry most safety-critical systems have to be certified according to a standard. A very prominent safety standard is the IEC 61508 standard which is a general safety standard describing upper limits for failures in time, depending on the system's level of criticality. To decrease the failures in time for a specific system component, the standard suggests to execute self-tests for that component during runtime. Such a safety self-test can detect component faults before they lead to an overall system failure.

For CPU-based systems for example, the IEC 61508 standard requires the execution of self-test programs which detect a certain percentage of hardware faults. Such hardware faults in a CPU can be random bit-flips or permanent register defects. Usually it is rather difficult to develop such self-test programs from scratch, because this requires very detailed knowledge about the hardware and about appropriate operations to test relevant parts of the CPU. Also, even if such a self-test is developed, it is still very difficult to verify that the self-test actually detects a sufficiently high amount of CPU faults, because such faults occur very rarely and can therefore not sufficiently be observed during the regular operation of the CPU. To simulate faults in the CPU, fault injection is usually applied to to achieve a statistically reasonable amount of faults in order to evaluate whether the self-tests detect enough of them.

Related to CPU self-tests and fault injection, this thesis provides following three main contributions:

- The thesis gives an introduction to safety-critical systems with focus on self-tests of CPU-based systems. Furthermore, an introduction to the IEC 61508 safety standard is given and the self-test requirements from the standard are described. A thorough overview of related work on these tests is given. Furthermore, the thesis describes the basic concept of fault injection and provides a literature overview of fault injection frameworks.

- A fault injection system is presented which injects faults in a Modelsim simulation. The system is specifically tailored to the safety domain and can easily be used to verify self-tests for CPUs used in safety-critical environments.

- Generic self-test for CPUs are presented. The generic tests are CPU architecture independent and aim to test CPU components which explicitly have to be tested according to the IEC 61508 standard. The self-tests are simulated on two different CPU architectures (Plasma/MIPS and LEON3) and faults are injected in this simulation to evaluate whether the fault detection ratio of the self-tests is sufficiently high to fulfill the requirements of IEC 61508 standard to achieve Safety Integrity Level 2 (SIL2) certification. The generic self-tests can provide safety-engineers who have to develop CPU tests for a specific CPU with a good starting point for the development of specific self-tests.

1.2 Outline

This thesis is structured as follows: Section 2 gives an introduction to safety-critical systems and focuses on the IEC 61508 safety standard. In particular, self-tests for safety-critical CPU-based systems are covered in detail. Furthermore, this section introduces the concept of fault injection and presents related work. Section 3 provides the design for a fault injection system which can be used to inject faults into CPU simulations. Furthermore, this section shows the design of generic self-tests for CPUs. Section 4 presents the implementation of the fault injection system specifically tailored to test safety-critical CPU-based systems. Section 5 describes the usage of the fault injection system to validate the generic self-tests against the IEC 61508 requirements for CPU self-tests. This validation is shown for two different CPU architectures. Section 6 concludes this work and gives an outlook of possible future work. The Appendix contains the source code for the fault injection system, as well as the source code for the generic self-test programs. Furthermore, the Appendix contains poster and a paper which were published based on the work in this thesis.

Chapter 2

Technical Background and Related Work

This section explains the topics of safety-critical systems with focus on hardware self-tests and fault injection and covers related work on these topics.

2.1 Safety-critical Systems

This section gives an introduction to safety-critical systems, explains some of the terminology, and provides an introduction to the IEC 61508 safety standard with special focus on software-based self-tests for CPU-based systems.

2.1.1 Introduction to Safety-Critical Systems and to Safety Standards

Safety-critical systems are systems which can cause harm to their environment if they malfunction. In particular, systems are considered as safety-critical if they can cause injuries to people or if they pose a threat to human lives. A computer-based system is not safety-critical by itself; however, it can become safety-critical in context with its environment. Such a computer-based system can then be described with the term *functional safety* which is defined as follows:

Functional Safety is the part of the overall system safety which depends on the correct functioning of safety-related systems for risk reduction. The intended functions of these systems, i.e. the safety functions, must be executed under defined fault conditions with a defined high probability [BÖ7].

An example for a critical computer-based system which has to keep up functional safety can be found in a modern car. The brakes of the car are not mechanically connected to the breaking pedal. Instead the breaking pedal sends a message to a microcontroller which then activates the breaks. If the microcontroller does not activate the breaks due to an error, human lives could be in danger. Therefore, the breaking system of a car is a safety-critical system.

To ensure that systems which are developed in a safety-critical domain are of high quality and just rarely malfunction, such systems are usually developed and certified according to a safety-standard. Sometimes this is required by law and each system developed for a

specific safety domain has to be certified. For example, the European Machinery Directive 2006/42/EC lists different safety domains and states which standards are preferred for these domains to show that a product complies to health and safety requirements which are required by law [ABB10]. In some other cases, it is not explicitly required by law to certify a safety-critical product; however, in industry this is often seen as 'best practice'.

Such safety standards are developed and maintained by standardization organizations like the ISO (International Organization for Standardization) or IEC (International Electrotechnical Commission) for example. There are different safety standards for different safety domains. For example, the IEC 26262 standard specifically covers automotive safety-critical systems. This standard emerged from the IEC 61508 standard which covers safety-critical electrical, electronical, and computer-based systems in general and serves as a basis for many other domain-specific standards. For example the EN 50128 standard for the railway domain or the IEC 61513 standard for nuclear power plants are based on IEC 61508. There are numerous other standards for other domains and there are also similar standards for the same domains in which cases it depends in the local country laws or on the local industry directives which standard has to be applied.

2.1.2 Fault, Error, Failure

This section briefly explains the terms **fault**, **error**, and **failure**. These terms are used for functional safety in general. However, the following explanations are biased towards the IEC 61508 standard.

According to the IEC 61508 safety standard [Int10], a **fault** is defined as follows: *"An abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function."* A **fault** is an incorrect state or process which can cause the system to perform in an unintended manner. The **fault** itself does not cause harm to anyone if it is properly handled and detected or if it does not become active [ALRL04].

If a **fault** becomes active, it is called an **error**. According to the IEC 61508 safety standard, an **error** is: *"A discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition."* A similar definition is given in the IEC 65A standard [Int92] (on which the IEC 61508 standard is partly based), where an **error** is defined as a detected deviation from the specification. Thus, compared to a **fault**, an **error** does already influence the specified system functionality [vdM95].

An **error** becomes a **failure**, if the system is not capable to deliver the specified external services anymore due to the **error**. This means that a **failure** is externally observable and can have a negative impact on the system's environment. IEC 61508 defines a **failure** as: *"A failure is the termination of the ability of a functional unit to provide a required function or operation of a functional unit in any way other than as required."*

Figure2.1 shows the relation between **faults**, **errors**, and **failures**. An **error** is an activated **fault** which might propagate and become a **failure**. A **failure** of a system component can be seen as a **fault** in the context of its surrounding system [ALRL04].

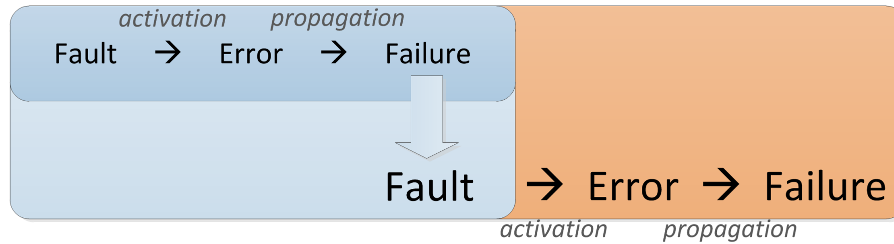


Figure 2.1: Relation between Faults, Errors, and Failures

2.1.3 IEC 61508

General Information

The IEC 61508 standard is published by the International Electrotechnical Commission (IEC). The first version was published in 1998 and in 2010, the second edition was published. The standard covers safety-related electrical, electronic, and programmable electronic systems in general and serves as a basis for several other domain-specific safety standards [LPP10]. The IEC 61508 standard consists of the following parts:

- *IEC 61508-1* - General requirements
- *IEC 61508-2* - Requirements for electrical/electronic/programmable electronic safety-related systems
- *IEC 61508-3* - Software requirements
- *IEC 61508-4* - Definitions and abbreviations
- *IEC 61508-5* - Examples of methods for the determination of safety integrity levels
- *IEC 61508-6* - Guidelines on the application of IEC 61508-2 and IEC 61508-3
- *IEC 61508-7* - Overview of techniques and measures

For IEC 61508 safety certification it is mandatory to follow the first three parts of the standard. Parts 4-7 just provide additional information about how to apply the first three parts.

The Safety-Lifecycle

The safety-lifecycle describes a process framework which has to be used to achieve IEC 61508 certification. The framework describes different process phases in a product lifecycle (e.g. requirements specification, development, maintenance, ...) and gives specific requirements and methods how to conform with this process phases. Each phase describes a set of outputs (e.g. test result documents) which have to be delivered for a system developed according to the standard.

Figure 2.2 shows the IEC 61508 safety lifecycle. The initial planning phases include the system concept definition and an overall hazard analysis. These steps result in the safety requirements for the system. In the requirements, each safety-critical component has to be

allocated to a Safety Integrity Level (SIL). The different planning phases require that plans for system verification, validation, installation, and maintenance have to be constructed before the system implementation. The realization phase can be split up in several other phases. There are different more specific realization phases described in IEC 61508-2 for hardware components, and in IEC 61508-3 for software components. In general, the realization phases cover a development lifecycle for safety-related hardware and software. The steps of the realization phase include requirements specification, design and development, system integration, operation and maintenance, and safety validation including validation planning. After the realization phase, the standard covers process phases regarding the overall system integration, safety validation, and operation and maintenance. These steps were already previously planned in the corresponding planning phases. Another phase in the safety-lifecycle handles system modifications. It is interesting that software maintenance is seen as system modifications. Thus, if a software bug is corrected, the system has to be re-certified to still be safe according to the standard. This can sometimes be counter-productive, because if a system is safety certified, usually the system developer does not want to change it afterwards because of the high re-certification costs. Thus, if a bug which does not have severe impact on the system, most likely this bug will not be corrected, because if it is corrected the system is not safety certified anymore or i would have to be re-certified. The last phase of the safety-lifecycle covers decommissioning and disposal of the system [PKK13].

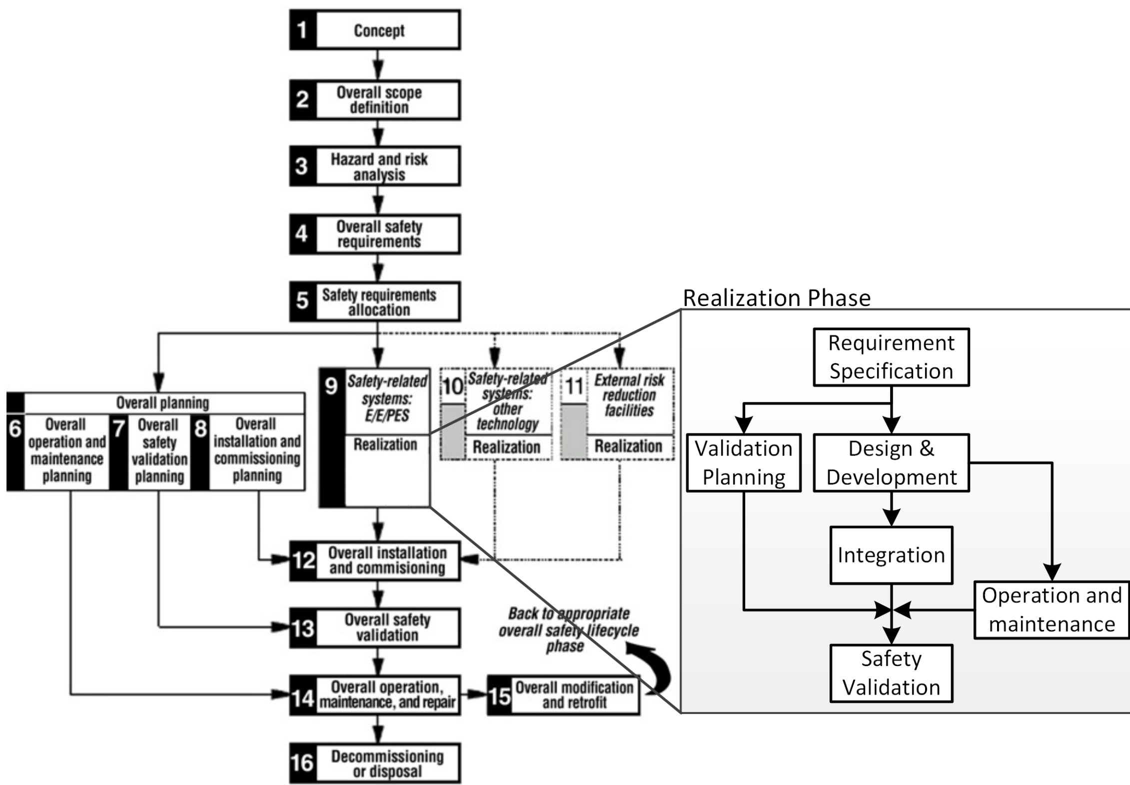


Figure 2.2: IEC 61508 Safety Lifecycle

Technique/Measure	**SIL 1**	**SIL 2**	**SIL 3**	**SIL 4**
Suitable programming language	Highly Recommended	Highly Recommended	Highly Recommended	Highly Recommended
Strongly typed programming language	Highly Recommended	Highly Recommended	Highly Recommended	Highly Recommended
Language subset	-	-	Highly Recommended	Highly Recommended
Certified tools and certified translators	Recommended	Highly Recommended	Highly Recommended	Highly Recommended
Tools and translators: increased confidence from use	Highly Recommended	Highly Recommended	Highly Recommended	Highly Recommended

Table 2.1: Software Design and Development Techniques and Measures [Int10]

Safety Integrity Levels

The IEC 61508 safety standard defines different Safety Integrity Levels (SILs). These safety integrity levels represent the overall level of risk reduction for a safety-critical system. The levels range from SIL 1 (lowest level) to SIL4 (highest level). The standard requires different levels of safety for a system or its components, depending on the SIL. For this, some sections of the standard contain tables which describe safety methods which can be used to achieve a higher level of safety. Depending on the SIL, the standard then says whether it is mandatory to apply this method or not. Figure 2.1 shows an example for such a table for software design and development measures. If a technique or measure is "recommended" for a SIL, then it has to be applied if there is not a good reason why it cannot be applied. If a technique or measure is "highly recommended" then it is mandatory to apply it and just in rare cases an exception can be made. The specific techniques and measures in Table 2.1 are further explained in part 7 of the IEC 61508 standard.

The SILs also define average levels of probability for a dangerous failure. Table 2.2 shows these probabilities as defined by the IEC 61508 standard. This table clearly shows which of the SIL levels are more rigorous. SIL4 is the highest safety level and most difficult to achieve, while SIL1 is the lowest safety level.

Safety Integrity Level	**Average frequency of a dangerous failure of the safety function per hour**
4	$\geq 10^{-9} to < 10^{-8}$
3	$\geq 10^{-8} to < 10^{-7}$
2	$\geq 10^{-7} to < 10^{-6}$
1	$\geq 10^{-6} to < 10^{-5}$

Table 2.2: Failure Probability for Different SILs [Int10]

Techniques and Measures

Part 7 of the IEC 61508 safety standard presents a collection of techniques and measures which are recommended to apply for meeting the safety requirements stated in parts 1-3 of the standard. This part of the standard also explains the aim and describes how to apply the techniques or measures. Furthermore, the standard provides references to more detailed descriptions. The techniques and measures are divided into the following categories:

- *Techniques and Measures to protect against random hardware failures* - The application of some of these techniques and measures is required by part 2 of the standard, which describes hardware development. Some examples are: RAM self-tests, majority voting, reference sensors.
- *Techniques and Measures to protect against systematic failures* - The techniques and measures are related to project management and to the overall system design and verification. They are not specially tailored for hardware or software development. Some examples for the techniques and measures are: Black-box testing, checklists, Failure modes and effects analysis (FMEA).
- *Techniques and Measures to achieve software safety integrity* - This part covers techniques and measures related to software development and testing. Some examples are: Coding standard, software diversity, UML.
- *Techniques and Measures for ASIC design* - This part covers techniques and measures for the design of ASICs as well as for testing ASICs. Some examples are: Design for testability, validation of the soft core, burn-in tests.

The usual way to achieve safety certification for a system is to have a look at the lists of requirements given in part 1-3 of the IEC 61508 standard. Each requirement recommends one or more technique or measure to fulfill the requirement. Table 2.1 is an example for techniques which are required by the standard to achieve a high quality software design. The safety engineer has to choose appropriate techniques and measures. If other techniques are used, it has to be well argued why this is done and it has to be shown to the safety certification authority that the chosen technique can also fulfill the requirement. Usually this is a lot more difficult than just choosing one of the suggested techniques or measures.

2.2 Software-based Self-Tests for CPU-based Systems

This section introduces software-based self-tests and explains the requirements for software-based self-tests for CPU-based systems according to IEC 61508. Furthermore, this section presents some software-based self-tests and gives related work with focus on RAM, cache memory, and CPU core tests.

A self-test is a process on a component where the component checks itself with built-in measures and reports the test results [GPZ04]. Software-based self-tests require no additional hardware and are easier to implement for existing hardware. However, of course such software-based self-tests require additional processing time resources. In some cases,

software-based self-tests are not as accurate as test functionality which is build in hardware might be, because in complex systems, the software often does not have access to all parts of the hardware and thus cannot test everything. In such cases, built-in hardware tests or hybrid test approaches can increase the quality of the testing procedure.

2.2.1 Fault Types

In safety-critical systems it has to be ensured that low-level hardware faults are detected and do not become systems failures. The IEC 61508 safety standard considers two different types of hardware faults: *Permanent Faults* and *Transient Faults*. The safety standard just handles single faults and poses no requirements regarding the detection of multiple faults which occur simultaneously.

Permanent faults which are considered by the safety standard are permanent faults in the hardware where, for example, a bit is stuck at a specific value. Therefore, they are called *Stuck-At Faults*. An example for a stuck-at fault is an output of a flip-flop register who's output is always a logical 0, independent form the input combination. In this case, a stuck-at 0 fault is present at the flip-flop. Stuck-at faults can for example origin in gate-to-source bridging failures in CMOS logic. Further information about the origin of stuck-at faults can be found in [Vis91].

Transient faults are triggered by environmental conditions and result in no long-term damage in the hardware. For example, environmental radiation can cause a single bit-flip in the hardware. If the logical value 1 is stored in a flip-flop, for example, and if a transient fault occurs, then the logical value 0 is read when accessing the flip-flop. However, compared to permanent faults, the electrical circuit takes no further damage and still functions properly. Transient faults can occur if alpha particles hit a part of the electrical circuit. Due to the decreasing size of electrical circuits, alpha particles pose an increasing threat and more care has to be taken to properly handle transient faults [CRP+96].

2.2.2 Safe Failure Fraction and Diagnostic Coverage

According to the safety standard [Int10], the safe failure fraction is defined as follows: *"A property of a safety related element that is defined by the ratio of the average failure rates of safe plus dangerous detected failures and safe plus dangerous failures"*. This can be better illustrated as an equation:

$$\begin{aligned} SafeFailureFraction(SFF) &= \frac{\sum \lambda_S + \sum \lambda_{Dd}}{\sum \lambda_S + \sum \lambda_{Dd} + \sum \lambda Du} = \\ &= \frac{noncritical\ failures + detected\ critical\ failures}{noncritical\ failures + detected\ critical\ failures + undetected\ critical\ failures} \end{aligned} \tag{2.1}$$

The SFF describes the overall ratio of dangerous failures which have to be detected in a safety-critical system. The SIL of a safety-critical system defines the maximum upper bound for the allowed SFF (see Table reftab:sff. The upper bound for the SFF depends on the *hardware fault tolerance (HFT)* of the system. The HFT is the minimum number of faults that can cause the system to not meet its safety functionality. If, for example, a CPU calculates a safety-critical functionality and a HFT of 1 has to be reached, then the

CPU can be duplicated and the safety critical functionality is calculated by each of the two CPUs separately. One hardware fault in a CPU does not influence the other CPU and therefore a single fault cannot cause an overall system failure [SS01].

	Hardware Fault Tolerance		
SFF	**0**	**1**	**2**
$< 60\%$	SIL -	SIL 1	SIL 2
$60\% - 90\%$	SIL 1	SIL 2	SIL 3
$90\% - 99\%$	SIL 2	SIL 3	SIL 4
$> 99\%$	SIL 3	SIL 4	SIL 4

Table 2.3: Maximum Allowed SIL for a Given Safe Failure Fraction Depending on the Level of Hardware Fault Tolerance [Int10]

In order to increase the number of detected failures of the SFF, self-tests can check if a system fault is present and avoid that it becomes a failure. In this way the $\sum \lambda_{Dd}$ value can be increased and the $\sum \lambda_{Du}$ value can be decreased to increase the overall SFF value. The ratio of detected faults is called diagnostic coverage (DC) and is defined as follows:

$$DiagnosticCoverage(DC) = \frac{DetectedFaults}{PossibleFaults} = \frac{\sum \lambda_D}{\sum \lambda_{All}} \tag{2.2}$$

The diagnostic coverage value is a measure for the self-tests to be evaluated and compared. Before calculating the DC value, one has to know the possible faults that can occur for a system. This can be found out by applying a failure mode and error analysis (FMEA) which is described in detail in part 7 of the safety standard. For some components of typical CPU-based systems, the safety standard already provides a list of fault sources which have to be considered. These lists will be given in the following sections in which the self-tests for specific components of CPU-based systems are covered.

2.2.3 Memory Tests and IEC 61508

This section covers IEC 61508 RAM self-test requirements as well as testing methods for RAM memory proposed by the standard.

A long-term study on memory faults conducted by Google concludes that RAM faults are an issue and have to be handled properly [SPW09]. In the study, several servers were monitored regarding RAM faults which were detected by using an ECC controller for the RAM. The paper reports that on average, there is a 8% chance for a fault to occur per year for each RAM DIMM. This study shows that RAM faults are an actual problem and have to be covered in the safety domain.

IEC 61508 Memory Tests Requirements

Part 2 of the IEC 61508 standard provides the requirements for memory self-tests which are shown in Table 2.4. This table shows which fault types have to be detected if a specific level of faults coverage is required.

For example, if a system with a hardware fault tolerance value of 1 has to achieve SIL 3, then this system requires a SFF of at least 90% according to Table 2.3. Assuming all failures to be critical, the system requires a DC value of at least 90%. Thus, one has to assume the fault types in the second from right column in Table 2.4 for that system and the tests have to detect at least 90% of these faults in order for the whole system to be SIL 3 capable.

	Requirements for diagnostic coverage claimed		
	Low (60%)	**Medium (90%)**	**High (99%)**
Invariable memory	Stuck-at for data and addresses	DC fault model for data and addresses	All faults that affect data in the memory
Variable memory	Stuck-at for data and addresses	DC fault model for data and addresses Change of information caused by soft errors	DC fault model for data and addresses Dynamic cross-over for memory cells Change of information caused by soft-errors No, wrong or multiple addressing

Table 2.4: IEC 61508 Memory Test Requirements [Int10]

Memory Test Approaches suggested by the IEC 61508 Standard

The IEC 61508 safety standard presents different techniques to test invariable and variable memories. The standard gives DC values that can usually be achieved with these tests. These values provide safety engineers with a rough estimation to develop a testing concept which has to be presented to the safety certification authority during the early safety certification steps. However, the DC values have to be confirmed during later phases of the safety certification through testing approaches which are further discussed in Section 2.3.

Table 2.5 presents the memory test approaches suggested by the IEC 61508 standard for invariable memories and Table 2.6 presents IEC 61508 tests for variable memories. The tests for invariable memories are limited to RAM tests. However, in a modern CPU usually also a cache test is required which is covered in the following sections.

Technique/Measure	**Description**	**Achievable DC value**
Word-protection multi-bit redundancy	Every memory word is protected by additional redundant bits. The resulting memory words should have a hamming distance of at least 4. Each time the work is read, the redundant bits have to be checked.	Medium (90%)
Modified checksum	A memory block checksum is stored. Each time the memory is read, the checksum is checked.	Low (60%)
Signature of one word (8-bit)	A CRC8 checksum is calculated for a memory block and checked when reading from the memory	Medium (90%)
Signature of one word (16-bit)	A CRC16 checksum is calculated for a memory block and checked when reading from the memory	High (99%)
Block replication	The memory is duplicated and the two memory values are compared when reading from the memory	High (99%)

Table 2.5: IEC 61508 Memory Test Techniques/Measures for Invariable Memories [Int10]

2.2.4 RAM Test Methods

This section basically explains the memory test methods suggested by the IEC 61508 standard. Furthermore, it focuses on how these tests can be applied to modern DDR RAM memories. The test descriptions are based on the descriptions given in the IEC 61508 standard if not referenced otherwise.

Checkerboard Test

The Checkerboard tests aims to detect static bit faults. A pattern of alternating 0s and 1s is written to the memory. The memory cells are inspected in pairs. The first address for the pair is chosen and the second address is formed by bit-wise inverting the first address. The content of the two cells should be the same. The cell-pairs are checked in two test runs. In the first test run, the variable address is defined by running from lower to higher addresses and in the second test run, the variable address runs from higher to lower addresses. The whole test is repeated with an inverted initial memory contents.

The test requires $10n$ memory accesses (read or write), where n is the size of the memory to be checked.

Walkpath Test

The Walkpath test detects static, dynamic and cross-talk faults in the memory. It requires $n + n^2$ memory access steps. The following steps describe the Walkpath test:

Technique/Measure	Description	Achievable DC value
RAM test checkerboard or march	A specific pattern is written to the RAM and later on read to detect stuck-at bit faults.	Low (60%)
RAM test walk-path	The test initializes the RAM with a pattern. The first RAM cell value is inverted and the contents of the remaining cells is checked. Then the first cell is inverted again. This is done for every memory cell.	Medium (90%)
RAM test "galpat" or "transparent galpat"	Tests the RAM in the same way as the walk-path test, but additionally also tests the currently inverted cell after each read operation on other cells	High (99%)
RAM test Abraham	The Abraham test is also known as MATS test and consists of a specific sequence of read and write operations applied to the memory.	High (99%)
Parity bit for RAM	Each word is extended by a parity bit which is checked when reading the data.	Low (60%)
RAM monitoring with a modified Hamming code, or detection of data failures with errordetection-correction codes (EDC)	Values saved to the memory are extended by several bits to form a code with Hamming distance of at least 4.	Medium (90%)
Double RAM with hardware or software comparison and read/write test	The data is duplicated on two different memories. The values of the duplicated data are compared when reading the data.	High (99%)

Table 2.6: IEC 61508 Memory Test Techniques/Measures for Variable Memories [Int10]

- The memory range to be tested has to be uniformly initialized (all 1s or all 0s)
- One memory cell is inverted and the contents of all other cells is inspected and checked if it changed. Then the inverted memory cell is inverted again to restore its value. This step is repeated for all memory cells which have to be checked.

Galpat Test

The Galpat test detects some static and a large number of coupling bit faults. It works similar to the Walkpath test, but additionally in each step (for each inverted cell) checks the content of the inverted cell after each read access for checking the other cells. Thus, the Galpat RAM test requires $2n^2 + n$ computation steps.

An alternative version of the regular Galpat test is the transparent Galpat test. This test is similar, but does not initialize the content of the memory in the first step. Instead, it calculates a signature (this is any value which represents the memory content like a hash value or a CRC value for example) of the remaining RAM cells, before inverting a cell. After inverting the cell, the signature is computed again and checked against the first signature. Compared to the regular Galpat test this has the advantage that the memory content is preserved. However, instead of simply looking at the pre-known content of the RAM, a signature has to be computed which increases the overall computation steps of the test to computing $2n$ signatures (each of which requires reading n memory values) and comparing n signatures. Additionally n^2 memory read steps for checking the inverted cells are required.

Abraham Test

The Abraham test is presented in [AT78] and aims to detect stuck-at faults as well as coupling faults. The test walks through the whole memory up and down and sets or reads memory cells. This test is also known as the MATS march test and will be covered in more detail in one of the following section which covers march tests. The Abraham test takes about $30n$ memory accesses.

March Tests

March RAM tests are briefly mentioned in the IEC 61508 standard as an alternative to the Checkerboard RAM test. March tests are one of the most commonly used types of memory tests. March tests are a type of memory tests which go through the memory in a specific order, write memory cells, and check their content. There are many different march tests which focus to test for different types of memory faults. The march tests all have a common structure and use a common notation.

A march test consists of a set of *March Elements* which describe memory operations that have to be applied to a memory cell and the order in which the cells are traversed. After all operations of a march element are applied to a cell, they are applied to the next cell. A march tests consists of a set of march elements which are sequentially applied. Table 2.7 shows the march test notation.

The following sequence of march elements describes a sample march test, the MATS (modified algorithmic test sequence) test which is also known as part of the Abraham test:

- $\Updownarrow (w0)$
- $\Updownarrow (r0, w1)$
- $\Updownarrow (r1)$

Symbol	Description
$\Uparrow$	address from 0 to $n-1$
$\Downarrow$	address from $n-1$ to 0
$\Updownarrow$	either way
$w0$	write 0 to the memory cell
$w1$	write 1 to the memory cell
$r0$	read a cell who's value should be 0 (test fails if different)
$r1$	read a cell who's value should be 1 (test fails if different)

Table 2.7: March Memory Test Notation

The test first writes the value 0 to all memory cells, then walks through all cells (the order does not matter) and for each cell checks if the value is 0 and writes the value 1 into the cell. In the last step the test goes through all cells and checks if the value of the cell is 1. The whole test requires $4n$ memory access steps. Table 2.8 shows some commonly used march tests.

Algorithm	March elements
MATS	$\Updownarrow(w0); \Updownarrow(r0, w1); \Updownarrow(r1)$
MATS+	$\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0)$
MATS++	$\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0, r0)$
MARCH X	$\Updownarrow(w0); \Uparrow(r0, w1); \Uparrow(r1, w0); \Updownarrow(r0)$
MARCH C-	$\Updownarrow(w0); \Uparrow(r1, w0); \Downarrow(r0, w1); \Downarrow(r1, w0); \Updownarrow(r0)$

Table 2.8: Commonly Used March Test Sequences [TB06]

For all march tests, there also exists a transparent march test which achieves the same fault coverage. By replacing the write-operations of a march test by memory-flip operations and by skipping the memory initialization, any march test can be converted to its transparent version which has got the advantage that the initial memory content is not lost after the test. The exact algorithm how such a test can be converted is described in [LTW05].

March Tests for DDR-RAM memories

Conventional march tests have the problem that they do not discover coupling faults and address faults in DDR RAM memories, because DDR memories send write commands to a memory controller which gathers write operations and writes several cells at once. This can mask coupling faults in a way that a regular march test cannot detect them. The same goes for address faults, because a DDR read operation reads several cells which can also mask faults and makes them difficult to discover by march tests.

One possible solution to this problem is to explicitly mask the write operations so that the memory controller just writes one cell per write operation. Another solution is to shift the start address of the memory addresses for the march test. Both solutions are described

in detail in [SBS08] where also for DDR memories adapted test patterns for march tests are presented.

2.2.5 Cache Test Methods

In this section basics about cache memories and an approach to use march tests for cache memories will be discussed.

Basics about Cache Memories

The aim of the cache memory controller is to copy main memory data which is often accessed to the cache memory to provide a quicker memory access for this data to the CPU. If the cache memory is full, then the replacement policy tells which data in the cache has to be overwritten. If the replacement policy can choose any location in the cache to replace, then the cache is called *fully associative*. However, this approach has the drawback, that if the CPU wants to have data, the whole cache has to be checked for that data. Another approach is that each data address in the main memory has exactly one dedicated memory address in the cache where it can be present. Such a cache is called *direct mapped*. In this case, the CPU just has to check one cache address to see if the data is present in the cache. However, this approach has the disadvantage that probably several data which has to be put to the same memory location has to be stored in the cache at a time. This leads the cache replacement algorithm to replace the data in the cache even if other parts of the cache may still be empty.

The way a cache is normally organized is a tradeoff between the two approaches discussed above. In *associative caches*, a main memory address has several dedicated places in the cache where it can be stored. For example, for a 4-way associative cache, each data in the main memory has 4 address in the cache where it can be stored. Such an approach is tradeoff between having to look at all cache addresses for the required data and between not optimally using the available cache memory.

For the CPU to identify which data is present in the memory, cache data is stored in cache data lines. Such a data line consists of the actual data and tags which are parts of the main memory address. Figure 2.3 shows the organization of an example cache and the mapping to main memory addresses. The main memory index-part of the address determines in which set of the cache memory the data will be stored. The main memory tag-part of the address determines in which set of the cache memory the data will be stored, and the offset-part of the address determines the exact data chunk in a cache line.

Adapting March Tests for Data Cache Memories

A common technique to test cache memories with software-based self-tests is to adapt march tests. If it is possible to have well-directed write and read commands for the cache memory lines, then one can simply apply a march test. However, due to the indirect access to the cache memory with a cache controller, it is difficult to come up with a method for such well-directed accesses and the access method strongly depends on the cache architecture (e.g. N-set associative cache).

A computer program just has direct access to the RAM memory. Thus, to test a cache, a program has to allocate variables in RAM memory at addresses which will be stored at

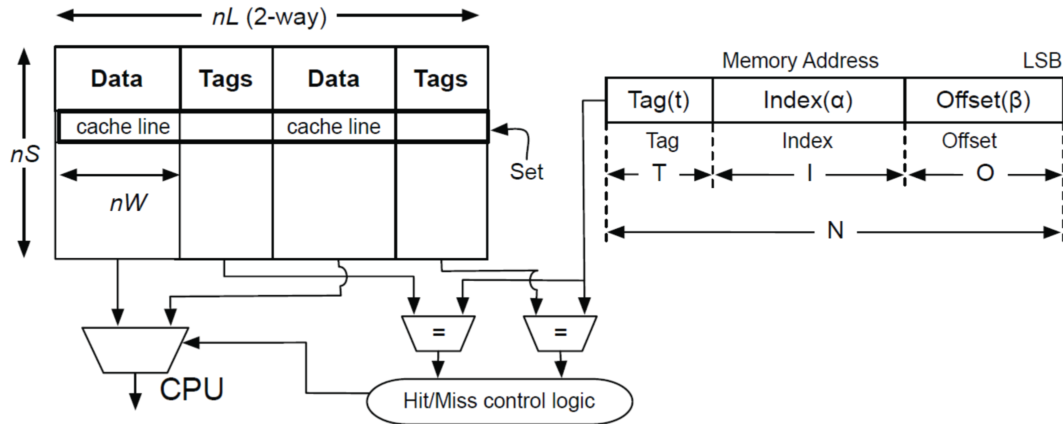


Figure 2.3: 2-set Associative Cache Memory [DPS11]

a specifically desired cache location. For example, by defining the RAM address, one can exactly define in which cache set the data will be stored. By allocating and accessing the variables in a specific sequence, one can also define in which of the N ways in the cache memory, the data will be stored.

With this approach, a program can structurally access cache lines. For a march test it is also important, that there is an order in which the cache lines are accessed for the test. In RAM memories this order is to go from lower addresses to higher addresses and vice versa. It does not matter whether the addresses are numerically ordered, it does just matter that the march test accesses the data in a specific order (each cache line once) and that the test can also access the data in the reverse order. It is usually no problem to access the specific set of cache lines in an order, because the RAM address of the program variable determines the set. However, to access the specific cache line in a set is more complicated, because where exactly the cache line in a set stands, depends on the order in which the data was first accessed. A program has to invalidate the data in a cache set and then read N variables (for an N-way associative cache) to fill the cache set. The order of these read operations determines in which way of the set the data is stored. Now, the variables are mapped to exactly one cache line and march tests can be applied to the cache memory [DPS11].

Mapping March Test Elements to Cache Memory Test Elements

To use the march test algorithms for cache memories, the simple march test elements ($\Uparrow, \Downarrow, w1, w0, r1, r0$) have to be mapped to the actual commands which are applied to the cache memory as to some extent already described in the section above. Figure 2.4 shows such a formal mapping which will be furthermore used for the cache test notation. With this mapping common march test notations can now be adapted to be used for describing cache tests. The data backgrounds are a specific data pattern which is written or read from the cache memory. Tests can be run with one single or with multiple data patterns. More details on the effectiveness of running multiple tests with different patterns can be found in [HL06].

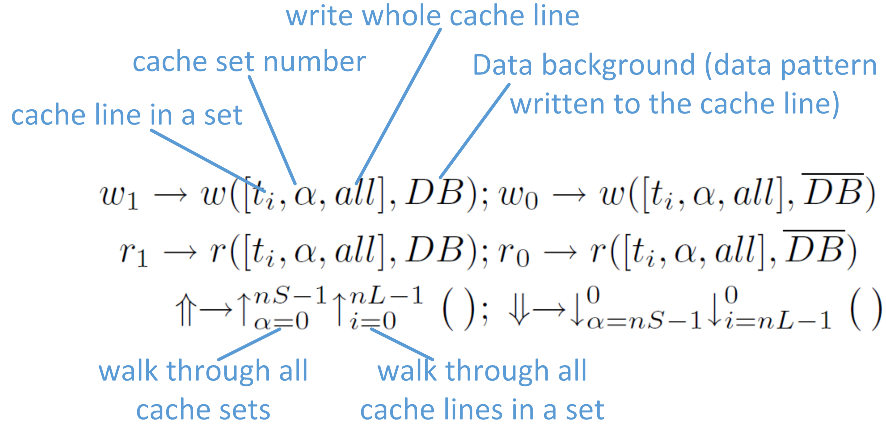


Figure 2.4: Mapping of March Test Commends to Cache Tests [DPS11]

MATS March Cache Test

The presented mapping from march test notation to cache tests can be used to apply an march test algorithm for testing the cache memory. Figure 2.5 shows as an example the MATS test adapted for cache memories.

$$\uparrow_{\alpha=0}^{nS-1} \uparrow_{i=0}^{nL-1} \; w([t_i, \alpha, all], \overline{DB})$$
$$\uparrow_{\alpha=0}^{nS-1} \uparrow_{i=0}^{nL-1} \; r([t_i, \alpha, all], \overline{DB}) \; w([t_i, \alpha, all], DB)$$
$$\uparrow_{\alpha=0}^{nS-1} \uparrow_{i=0}^{nL-1} \; r([t_i, \alpha, all], DB)$$

Figure 2.5: MATS March Test for Cache Memories

Tests for the Cache Tag-Values

With the above described approach it is possible to adapt march tests for the part of the cache which contains the data. However, as already shown in Figure 2.3, each cache line also consists of a tag part which contains a part of the RAM memory address.

To test this part, the variables for the cache test have to be allocated in a way that the tag-part of their addresses contains the data pattern (background) which is used for the test. A problem is that the tag-values in the same cache set cannot be the same (in that case an address would be loaded twice in the cache). Thus, for a march test, different backgrounds (tags) are used within a cache line.

Another problem is that for a march test, the cache has to be walked through in a specific order and in its reverse order. However, the cache sets are usually filled sequentially. This means that one cannot determine the order in which the tag-values in cache sets are accessed. To tackle this problem, dummy values can be written to the cache memory until the required part in a cache set is accessed.

One more problem is that the cache tag cannot be read. However, usually it can be identified if a memory access resulted in a cache hit or a cache miss. Thus, one can check if a tag value is correct or not by checking if the correct data is delivered when accessing the cache address or whether the access results in a cache miss. Such a cache miss could be detected by modifying the data in the RAM memory, but not modifying it in the cache. This could be achieved by disabling the cache, writing the value of a variable to the RAM, and re-enabling the cache.

The above mentioned solutions now provide an approach to walk through the tag-cache in a specific order (and its reverse order), to write data patterns to the tags, and to detect if the patterns are incorrect. These are the required operations for a march test and with them it is possible to apply any existing march test also to the tag-values in a cache memory [DPS11].

Tests for the Instruction Cache

CPUs with a Harvard Architecture have a separated data and instruction cache, while Von Neumann CPUs have one single cache for instructions and for data. The cache of Von Neumann CPUs can be tested with the approaches described above while Harvard CPUs have to tested in a different way.

To have specified write operations for the march test of the instruction cache, the test program has to access a new program part which is then loaded to the cache. Such write operations could also be realized as a memory pre-fetch if that is supported by the CPU. Read operations are the actual execution of the code instruction. If the read operation does not obtain the expected value, then a different instruction would be executed. Thus, to test if the read was successful, the executed instruction output can be compared with a pre-calculated, expected instruction output.

The instruction and tag values of the instruction cache can now be tested similar to the data cache. The instruction value pattern can be defined by the actual instruction which is used and the tag value can be defined by the address where the instruction is stored in the memory. More detailed information about how to test instruction cache memories can be found in [DPS11].

2.2.6 CPU Core Tests and IEC 61508

This section covers IEC 61508 CPU test requirements as well as testing methods for the CPU which are proposed by the standard. The section also covers related work on CPU self-tests.

IEC 61508 CPU Core Test Requirements

Part 2 of the IEC 61508 standard provides the requirements for CPU self-tests which are shown in Table 2.9. This table shows which fault types have to be detected if a specific level of fault coverage is required. The requirements focus on core parts of the CPU like the execution of instructions, or the file registers. These CPU parts are in literature often referred to as CPU-core components. Therefore, here related tests are furthermore called CPU core self-tests. The CPU core test requirements in Table 2.9 just cover some parts of the overall CPU. For example, specific CPU components like a shifter unit or an

arithmetical unit are not explicitly addressed by the standard. Usually such components also have to be tested if they are used in safety applications.

	Requirements for diagnostic coverage claimed		
	Low (60%)	**Medium (90%)**	**High (99%)**
Register, internal RAM	Stuck-at for data and addresses	DC fault model for data and addresses Change of information caused by soft-errors	DC fault model for data and addresses Dynamic cross-over for memory cells Change of information caused by soft-errors No, wrong or multiple addressing
Coding and execution including flag register	Wrong coding or no execution	Wrong coding or wrong execution	No definite fault assumption
Address calculation	Stuck-at	DC fault model Change of addresses caused by soft-errors	No definite fault assumption
Program counter	Stuck-at	DC fault model Change of addresses caused by soft-errors	DC fault model Change of addresses caused by soft-errors
Stack pointer	Stuck-at	DC fault model Change of addresses caused by soft-errors	DC fault model Change of addresses caused by soft-errors

Table 2.9: IEC 61508 CPU Test Requirements [Int10]

CPU Core Test Approaches suggested by the IEC 61508 Standard

The IEC 61508 safety standard presents different techniques to test the CPU core. Table 2.10 presents the CPU core testing approaches suggested by the IEC 61508. One of the techniques and measures for CPU core components is to use software-based self-tests. According to the standard this method just leads to a low or medium diagnostic coverage value. However, the tests can still be used to achieve a high DC value, if that higher DC

value can be shown during the verification of the tests. The other techniques (such as comparator, voter, reciprocal comparison, ...) require some kind of hardware redundancy to be applied. This theses will focus on CPU tests which do not require any modifications to the hardware. Therefore, from the methods shown in Table 2.10, just software-based self-tests will be covered.

Technique/Measure	Description	Achievable DC value
Comparator	The output of the CPU component is compared to a redundant unit.	High (99%)
Majority voter	Multiple CPU unit results are taken as input for a voter which decides for the majority of the provided results	High (99%)
Self-test by software: limited number of patterns (one channel)	Data patterns serve as an input for the CPU unit and calculations on these patterns are checked for their correctness	Low (60%)
Self-test by software: walking bit (one channel)	CPU registers are tested by setting them to zero, and then setting one bit after another. After each step the register contents is read and compared to the expected value.	Medium (90%)
Self-test supported by hardware (one channel)	Additional hardware to support self-tests is integrated.	Medium (90%)
Coded processing (one channel)	Processing units are designed with special failure detection units.	High (99%)
Reciprocal comparison by software	Two processing units exchange their final and intermediate results reciprocally. A comparison is carried out by each of the two units.	High (99%)

Table 2.10: IEC 61508 CPU Core Test Techniques/Measures [Int10]

2.2.7 CPU Core Tests

Literature distinguishes between two main approaches to test CPU cores [PG05], [PGSR10].

- **Functional Tests** - This kind of CPU testing tests at a rather high level of abstraction. The interface for functional tests is the instruction set of a CPU. The tests apply a specific sequence of CPU instructions and judge by the outcome of these instruction (e.g. by a register value at the end of the computations) whether the CPU works properly, or whether a fault has occurred. The advantage of functional CPU testing is that such tests are more portable and do not require in depth knowledge about the CPU hardware. It is rather easy to set up testbeds for functional testing, because the CPU instructions can directly be accessed by software. A main

disadvantage of this approach is that it achieves limited fault coverage for the CPU, because just limited information about the CPU can be used as information for what should be and what can be tested. This means that it might not be possible to discover some faults with this type of testing.

- **Structural Tests** - Structural CPU tests are on a low level of abstraction and require detailed information about the CPU hardware. Structural tests are tests which operate on the register-transfer-level (RTL) or on the gate-level of a CPU. The tests also apply CPU instructions for testing the CPU, but the testing instructions are chosen based on additional information from the RTL or gate-level implementation of the CPU. For example, if two registers are close together on the physical chip, then it is more likely that they cause interference for one another. Thus, the CPU test should consider to more thoroughly test the interaction between these two registers compared to registers which are physically not close to each other. Usually, with this additional information, higher values for the fault coverage of the CPU can be achieved. Main disadvantages of structural tests are that they require in-depth knowledge, they require high development effort, and they are not portable to other CPUs.

A further orthogonal classification between CPU tests is whether they are deterministic or randomized.

- **Deterministic Tests** - Such tests provide a well-known fault coverage for a specific CPU. The test always uses the same test data or test sequence and always tests the same components. For some CPU tests, formal proves are known which confirm a minimum fault coverage for CPU components (e.g. the multiplier) independent of the actual architecture of the component [PGK+01]. For such a test it is not even necessary to verify the fault coverage, which can be quite convenient.

- **Randomized Tests** - Such CPU tests are not deterministic and their coverage has to be verified. Randomized tests can be tests which apply some kind of learning mechanism to optimize the test inputs during a learning phase in order to end up with a fixed test sequence which is then applied during operation. Randomized tests can also be tests which simply apply random inputs for test cases (e.g. check the multiplier by using random inputs and checking the output by diverse computation) in order to test a lot of inputs over time.

When constructing test routines for a CPU, a standard way is to divide the CPU into its main components (e.g. ALU) and to test these main components. For each component the instruction set which can influence the component is analyzed. From these instructions, the ones which are well-observable (which cause the component to produce a reaction which is observable by using other CPU instructions) are chosen and self-tests are build with these instructions [KGPZ02], [KPGZ02].

The CPU core tests applied in this thesis are not based on such a CPU architecture analysis, but are simply a collection of well-known tests for CPU core components and of commonly used test routines. The tests are non-randomized and test the CPU on a functional level. Some of the tests have a proven minimum fault coverage and for some other tests the fault coverage has to be verified.

2.3 Fault Injection

In this section the reason for applying fault injection and different types of fault injection as well as an overview of a generic architecture of a fault injection system will be explained. Most of the content in this section about fault injection is based on [BP03].

2.3.1 Purpose of Fault Injection

When testing the reliability or the safety of a system, it is necessary to observe the system behavior in case of faults, because the system requirements for safety or reliability require the system to operate or to perform a specific action in case of faults. However, in some cases these faults occur very rarely which makes it difficult to test the system's behavior in case of faults.

For example, consider a system which has to detect the occurrence of RAM memory faults. According to [SPW09], such a fault occurs with 8% probability within a year in an average RAM DIMM. This means that on average one has to wait more than 6 years for a single fault to occur. To provide a qualitative good statement about the quality of a RAM fault detection mechanism, several faults (in the order of thousands) have to be observed. Due to the low occurrence rate of the faults, it is impossible to do that. Therefore, fault injection is applied to simulate or induce such faults.

Fault Injection can also be applied to simulate software faults. This could test the robustness of a program against design or programming faults during the development of the program. One example for software fault injection would be to delay the delivery of a software routine or to simulate a failed software routine and to check if the program acts as intended (e.g. shut down gracefully). However, the rest of this section will rather focus on fault injection of hardware faults which is later on used to test software-based CPU self-tests.

2.3.2 Generic Fault Injection Architecture

Figure 2.6 presents the main components which are necessary to perform fault injection. Not all of these components are necessary for all types of fault injections, but most of them can very well be mapped to the fault injection frameworks described in the next sections.

- The **Controller** coordinates the fault injection test. It starts and stops the test runs and retrieves the results.
- The **Target System** is the system which executes the functionality to be tested while the faults are injected.
- The **Fault Injector** injects the fault into the target system. The type, amount, and duration of faults is taken from the **Fault Library**.
- The **Workload Generator** provides commands (execution instructions, programs) which the target system executes during the test run. The Workload Generator takes these commands from its **Workload Library**.
- The **Data Collector** reads raw test data from the target system outputs.

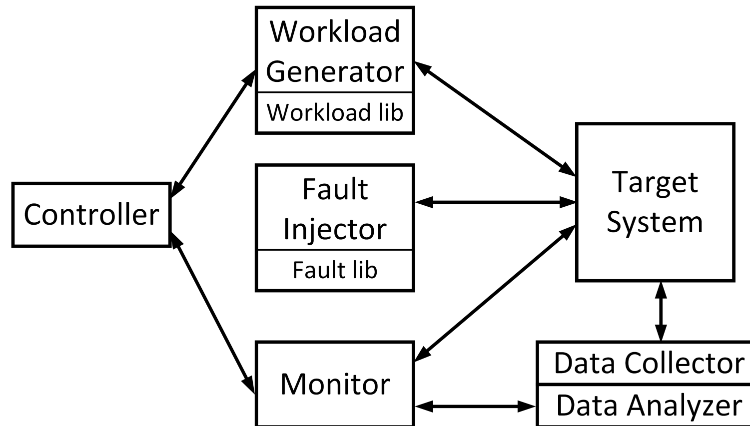


Figure 2.6: Typical Fault Injection Components and Interactions (based on [BP03])

- The **Data Analyzer** performs pre-processing of the test data.
- The **Monitor** watches the execution of commands on the target system and collects the test data.

2.3.3 Hardware-Based Fault Injection

Hardware-based fault injection manipulates actual hardware during execution. The two main approaches are the *forcing technique* and the *insertion technique*.

- The forcing technique simply overwrites a specific value in the system. An example would be to externally set a specific voltage level to the pin of a chip.
- The insertion technique modifies a hardware component or replaces it with a faulty component. This can invoke unintended behavior in other components. For example, a RAM memory can be replaced with a partially malfunctioning RAM memory to observe the program behavior when no reliable memory is present.

One man advantage of hardware-based fault injection is that the injections can be performed very fast and they can be applied to the actual software running on the hardware. Thus, the software does not have to be modified.

A main disadvantage of hardware-based fault injection is that it is tailored to a specific hardware which makes it very difficult to port the fault injection framework to a different platform. Often, the construction of a hardware-based injection framework is rather complicated and requires special purpose hardware. The observability of the test results might also be limited, because often it is impossible to monitor specific hardware states or registers.

Table 2.11 presents some hardware-based fault injection approaches and gives a short description of them.

Name	Description
AFIT	Pin-level fault injection. The external pins of chips are short-circuited to introduce faults [GBS04].
RIFLE	A system to deterministically inject pin-level faults [MMS94].
FOCUS	User specified faults are injected at run-time and the propagation to the chip pins is measured [Cho96].
FIST	A RISC processor with error detection capabilities in hardware evaluated with a heavy-ion fault injection framework [GOR89]
MESSALINE	System which forces pin-levels [ACC+93].
MARS	MARS is a time-triggered computer system designed with high fault detection capabilities. [Fuc96].

Table 2.11: Hardware-Based Fault Injection Approaches Described in Literature

2.3.4 Software-Based Fault Injection

Software-based fault injection does not modify the hardware, but the software running on it. For example, a network stack can be modified to duplicate packets from time to time. This might influence other system components (e.g. router or switch hardware) in an unintended way which can then be detected.

An advantage of software-based fault injection is that it can be performed on the actual hardware which does not have to be modified. This makes the approach more portable than hardware-based fault injection and is also often more easy to achieve, because no special purpose hardware is necessary. The injections can still be performed in acceptable speed in the order of magnitude of the software execution processing speed which is in most cases sufficiently fast.

A disadvantage of software-based fault injection is that the software running on the system has to be modified and that not every kind of fault can be injected, because the faults that can be addressed directly by software might be limited. For example, in a CPU it is not possible to set or reset every CPU register by software. Also not every register value can be read which makes the observability of software-based fault injection limited.

Table 2.12 gives an overview of software-based fault injection approaches from literature and provides a short description of them.

2.3.5 Simulation-Based Fault Injection

Simulation-based fault injection simulates hardware and injects faults into this simulation. For example, the hardware description language code of a CPU can be simulated with a simulation tool while registers are modified by a fault injection component. In the

Name	Description
BOND	BOND is a Windows NT tool for fault injection. It performs deterministic and statistical injections into program code, registers, heap data, and system calls [BBCP00].
XCEPTION	A software-based fault injection tool with debugging and performance monitoring capability [CMCS04].
MAFALDA	A tool to test microkernels by fault injection. Injections are done as library calls or as software traps [RSFA99].
DOCTOR	The fault injection environment generates workloads, injects software faults, and collects performance data. Memory, CPU, and communication faults can be injected [Han95].
EXFI	Fault injection framework based on the Trace Exception Mode of microprocessors [BPRR98].
FIAT	A distributed real-time fault injection framework [SVS+88].

Table 2.12: Software-Based Fault Injection Approaches Described in Literature

simulation, the CPU runs a test program which monitors and detects the injected hardware faults. This approach will be used in Chapter 3 to verify CPU self-tests.

The main advantage of simulation-based fault injection is that any type of fault can be injected (limited to the detail-level of the simulation model) and that the results of the fault injection are very well observable.

The main disadvantage is that a simulation model of the hardware is required. It is also necessary to verify that the model complies with the actual hardware which will be used. This means that the model sometimes has to be verified which can be a lot of effort. Another disadvantage is that the simulations are a lot slower than fault injections in the actual hardware or software.

Table 2.13 gives an overview of simulation-based fault injection approaches from literature and provides a short description of them.

2.4 Related Work

This section covers related work on deterministic and proven CPU fault tests which will be later on used. Also indirect CPU tests will be handled. Furthermore, this section covers related work on fault injection frameworks used in the safety domain.

Name	Description
VFIT	A VHDL-based fault injection tool [GBGG04].
MEFISTO-C	A tool which can apply scan-chain implemented fault injection and fault injection based on built-in test logic [FSK98].
ALIEN	Mutation testing tool for the VHDL language [RS04].
VERIFY	Extension of the VHDL language with fault injection signals [STB97]
POWER-MODES	A flexible VHDL fault injection and power estimation framework specifically tailored for testing security attributes during early phases of chip design [KGS+12]

Table 2.13: Simulation-Based Fault Injection Approaches Described in Literature

2.4.1 Deterministic Tests for CPU Components

For the test algorithms for some specific CPU components, a test coverage for the algorithm can be proven. An example for that is shown in [PGK+01], where a test algorithm for the CPU multiplier unit is presented. Additionally for the multiplier test, a proof is provided which shows that the fault coverage for the multiplier unit is higher than 99% - independent of the actual multiplier architecture which is used by the CPU. [PGK+01] also provides other tests for CPU components like tests for a barrel shifter and for the ALU. A high fault coverage is shown for these tests for a specific CPU architecture, but for these components no proof for a minimum coverage is given.

[GPH+08] focuses on testing the pipeline of the CPU. A generic approach is presented which distributes a test program over the RAM memory in order to test the address logic of the CPU. Specific test routines which test pipeline hazards are also described. Additionally, an approach is described how to enhance an existing test program with the presented pipeline tests (e.g. by distributing the program all over the RAM memory).

[Bao03] focuses on tests for peripheral CPU components. In particular, the paper focuses on PLL tests. One approach to test the PLL is to measure the mean voltage level of the PLL output and to compare it to a calculated mean voltage value which results from the applied clock frequency.

In [TJD10], a CPU test approach is explained, which builds on tests for CPU components. Theses tests are collected in a library with the aim to reuse them for other CPU architectures. However, the paper does not provide details about the tests themselves.

To minimize the effort for developing CPU self-tests and to minimize the runtime for CPU self-tests, [KGPZ02] presents an approach how to build low cost self-tests by just targeting the main CPU components. The ALU and the shifter unit are identified as the main components of the used Parwan CPU. For these two components, deterministic tests are presented and the test coverage on the specific CPU is verified.

[Dey02] also suggests to write tests for single components of the CPU. Additionally,

it is suggested to structurally reason about the overall fault coverage of the CPU tests by combining the tests of the components in a fault tree. If some of the tests are redundant, then that can be identified in the fault tree.

[CDS+00] analyzes the CPU and its components. Tests are just developed for the main components (e.g. the ALU, the shifter). For the tests, the components are analyzed regarding the instructions which affect them. This information is based on the hardware description language model of the processor.

Safety-related self-tests for an ARM7 processor are presented in [TP07]. The walking pattern method (described in the IEC 61508 standard) is used for flash and RAM memory tests. A GALPAT test (also described in the IEC 61508 standard) is applied to the CPU registers and custom tests are developed to test the ALU and the CPU flags. The paper does not describe how the fault coverage of the tests is verified, but claims that the tests achieve the required fault coverage (which is 90% in the paper).

2.4.2 Indirect CPU Testing

The so far proposed test approaches just test the core components of the CPU and do not cover all CPU registers or elements. However, when testing the multiplier, for example, also other components are tested. The CPU pipeline for example, will also be tested, because each instruction has to go through the pipeline. Therefore, also other CPU components are tests, even if the test is not specifically aimed for this component. This approach to test parts of the CPU is called *indirect testing*.

[CD01] present tests for the shifter and the ALU components of a CPU. The fault coverage of the tests for these components as well as for the whole CPU is verified. The tests achieve 99% fault coverage for the shifter and the ALU and achieve 90% fault coverage for the whole CPU. This shows that also other parts than just the shifter and ALU are tested by the tests for these two components.

A similar approach is presented in [KXP+03]. A CPU is analyzed and its main components are identified. For these main components (adder, multiplier, ...), functional tests are written and these tests are verified. For the verification, faults are injected into the main components as well as into other components (e.g. multiplexers, pipeline). The results show that these components are also tested by the tests for the main components and that for the overall processor, a fault coverage of 94.5% is achieved.

2.4.3 Safety-related CPU Fault Injection Frameworks

In [TPN07], an approach to verify self-tests for processors in safety-critical environments is presented. The tests are developed for the memory and the CPU core of an ARM7 processor and are verified with a fault injection approach. The JTAG interface of the processor is used to read and write CPU registers. Additionally the scan chain logic and an internal feature to halt the processor is used to inject faults. For the injection, the processor is halted, a register value is externally written, and the the processor continues processing. This fault injection is applied while self-tests run on the processor in order to see whether the tests detect the injected faults. The injection approach and the necessary tool setup is described on more detail in the paper [TPN07].

A simulation-based verification approach for self-tests applied to safety-critical systems is presented in [KPG+13]. The paper describes deterministic self-tests for CPU core

components as well as a fault injection framework which is used to verify the tests. For the fault injection approach, the hardware description language code of a Plasma/MIPS CPU is modified. Saboteurs are included in the code to have some elements in the CPU which can be used to modify CPU register values by the help of external triggers. The modified code is emulated on an FPGA in order to speed up the fault injections. For the CPU and the provided self-tests, a fault coverage of more than 90% can be verified.

Chapter 3

Design of a Fault Injection System for the Verification of Generic CPU Safety-Tests

In this section, an approach to verify software-based safety self-tests is presented. First, the requirements for a fault injection system to verify safety-tests for CPUs will be presented. Next, the general design for such a fault-injection system and the general design of the generic CPU safety-tests is given.

3.1 System Requirements

R1 - Generic Self-Tests: A set of CPU-independent safety self-tests has to be provided. There should not be major modifications necessary for applying the provided self-tests to other CPU architectures.

R2 - Self-Test Verification: Self-tests for CPU have to be verified. In order to judge whether a test works successfully, it has to be tested under faulty CPU behavior. Due to the low error rate in regular CPUs, fault injection has to be applied to obtain statistically relevant and observable test result data.

R3: IEC 61508 compliant self-tests: The self-tests have to comply to the requirements of the IEC 61508 safety standard for CPU self-tests for at least SIL2.

R4 - Simulation time: The simulation is applied once per processor to verifiy the self-tests and perhaps once more during safety-certification if a certification authority demands to see whether a specific test works. In the first case, a high number of fault injections are necessary and in the second case just few fault injections are necessary. This means that the overall simulation time will be a lot higher for the first case. However, in the first case it does not matter very much if the simulation takes longer (up to some weeks). In the second case the simulation should not be longer than about one hour.

R5 - Fully automated simulation: The simulation for testing all safety-relevant CPU parts with all necessary fault injections according to the IEC 61508 standard have to

be fully automated and no human interaction must be necessary, because otherwise injecting the faults would be a lot more effort for the operator - especially if the simulations take quite a long time.

R6 - Easy to use: It should be easy to set up the self-test verification through fault injection for a different CPU architecture.

3.2 Fault Injection System Design

The fault injection system injects faults into a *Modelsim* simulation of a CPU. The faults are forced with Modelsim commands and the results of the fault injection are read from GPIO pins of the CPU. Figure 3.1 shows the design of the fault injection system. Most of the elements shown in the generic fault injection system architecture (Figure 2.6, page 33) can also be found in this figure:

- **Target System** - The system under test is a CPU which is simulated with Modelsim. Just open source CPUs can be used for fault injection, because the HDL source code has to be available for this simulation.

- **Workload Generator** - The software-based self-test programs are compiled for the specific CPU and run in the simulation while the faults are injected. In this chapter, the self-test programs specifically aimed for CPU-core tests are covered in more detail.

- **Controller** - The main part of the fault injection system is a TCL script which coordinates the fault injections and which interprets the CPU output. The TCL script actually also includes the **Fault Injector**, **Fault lib**, and **Monitor** components. For one simulation run, the TCL script starts the CPU simulation and after a specific time injects a fault. The type of fault is either transient or permanent. The register where the fault has to be injected is taken from a pre-defined list of registers which are relevant (registers which are explicitly required to test in the IEC 61508 standard). After the fault is injected, the CPU simulation finishes its test run and signals the TCL script that it finished by setting a GPIO pin. On a different GPIO pin, the CPU simulation signals whether a fault was detected or not. The TCL script reads the GPIO values to check if the test succeeded or failed. The TCL script executes several test runs for each fault type and for each register and reports the number of injected and the number of detected faults for each pre-defined register and fault type. Thus, the script coordinates all fault injections and does not need any human interference for producing a complete report about the fault detection ratio of the predefined set of faults for the predefined set of registers.

Compared to Figure 2.6 on page 33, the fault injection system does not explicitly include a **Data Collector** or **Data Analyzer** component, because the task of the data collect is done by a single line of code of the TCL script (read GPIO pins) and the task of the data analyzer is done by a single line of code of the TCL script as well (check if the read GPIO is 0 or 1). Therefore, those two components are not explicitly mentioned in Figure 3.1.

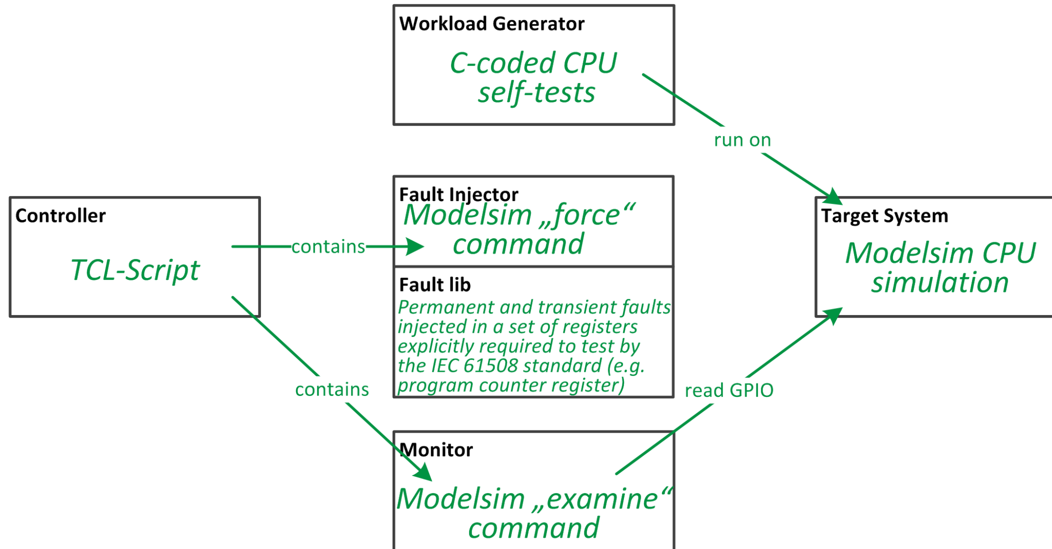


Figure 3.1: Components of the Fault Injection System

3.3 Design of Generic CPU Safety-Tests

This chapter describes a set of safety self-test programs which are suitable to test for CPU faults. The reason for just focusing on CPU-core tests and not also covering memory tests in more detail is that for memory tests, many well known and from safety standardization organizations accepted approaches are known (as described in section 2.2). However, for CPU core tests no such approaches are described by the safety standard. Therefore, the practical part of this thesis especially focuses on CPU-core tests to show and verify which tests can achieve the safety requirements by the IEC 61508 standard.

Some of the self-test programs are taken from literature. In particular, deterministic test programs with proven or at least experimentally known high fault coverage ratio for some components of the CPU are taken from literature. Other test programs are adapted from the *MiBench* benchmark suite. *MiBench* is an open source benchmark suite which provides test programs for several application domains like automotive, security, or networking [GRE+01]. The full source code of the test programs is given in Appendix B.

3.3.1 CPU Core Tests

The first set of test programs is taken from [PGK+01]. In that paper, several test programs for components of the CPU core are presented. The covered components are the multiplier, the divider, the ALU, and the shifter. According to the paper, all of the tests provide a fault coverage of more than 99% verified for an Intel 8051 processor. For the multiplier unit, a fault coverage of more than 99% is guaranteed, independent of the multiplier architecture [PGK+01]. This guarantee can be given, because the paper presents a mathematical proof that the multiplier test can achieve this fault coverage, because more than 99% of the multiplication functionality is tested by the proposed test.

The used CPU core test program consists of overall four tests (for multiplier, divider, ALU, and shifter). The four tests are based on [PGK+01], but are re-written in C-code

(and not in Assembler as in [PGK+01]). The CPU core test program executes the four tests one after another and reports a failure if any of the tests fails. In the following, the four test programs are described:

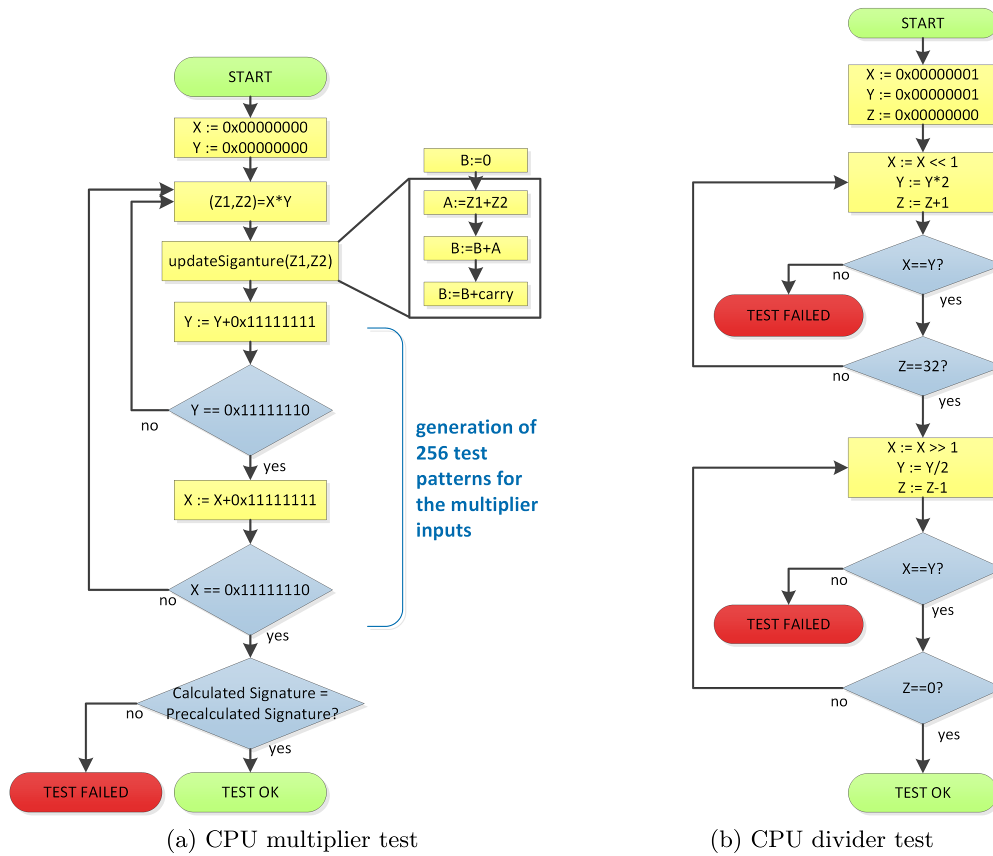
- The basic structure of the multiplier test is shown in Figure 3.2a. 256 multiplier inputs are calculated deterministically. The result of the 256 multiplications is accumulated in a signature register which is compared to a pre-calculated value at the end of the test. If the signature register is equal to the pre-calculated value, the test was successful. If it differs, a fault has occurred.
- The divider test structure is presented in Figure 3.2b. The test uses a diverse calculation to the binary division in order to test the divider unit. A division by two is the same as a bit-shift-right operation. The value 1 is multiplied 32 times and then divided 32 times. After each calculation, the result is compared to the result of the same mathematical calculation which is conducted by shifting left or right. If any of the overall 64 results differ, the test fails which means that a fault is detected.
- The ALU test is shown in Figure 3.3a. It uses diverse calculations of logical expressions with AND, OR, XOR, and NOT. These diverse calculations are applied for different data patterns.
- The shifter test (Figure 3.3b) applies shifting operations to a deterministic input and applies shift left and right operations to this input. The test compares the output value of these operations to a pre-calculated signature register value. If the two values match, the test is successful. If the two values differ, the test fails.

3.3.2 Quicksort Test

The quicksort test is a generic test which (just like the following tests) does not aim to test a specific component of the CPU. The reason for adding such a generic test is to evaluate whether the IEC 61508 test requirements for CPUs can be fulfilled by tests which are not specifically developed to test CPUs, but which indirectly test the CPU functionality. The quicksort test is based on a quicksort algorithm from the *MiBench* test suite. The test initializes an unsorted array, runs the quicksort algorithm, and checks for each element in the array whether they are in correct order. If the elements are not sorted, the test fails. Figure 3.4a gives an overview of the test.

3.3.3 SHA-1 Test

The SHA-1 test is a generic test which just like the quicksort test does not aim to test specific CPU components. The algorithm is based on the *MiBench* test suite. The test calculates a SHA-1 hash value and compares it to to a pre-calculated SHA-1 hash value. Figure 3.4b shows the basic structure of the SHA-1 test.



(a) CPU multiplier test (b) CPU divider test

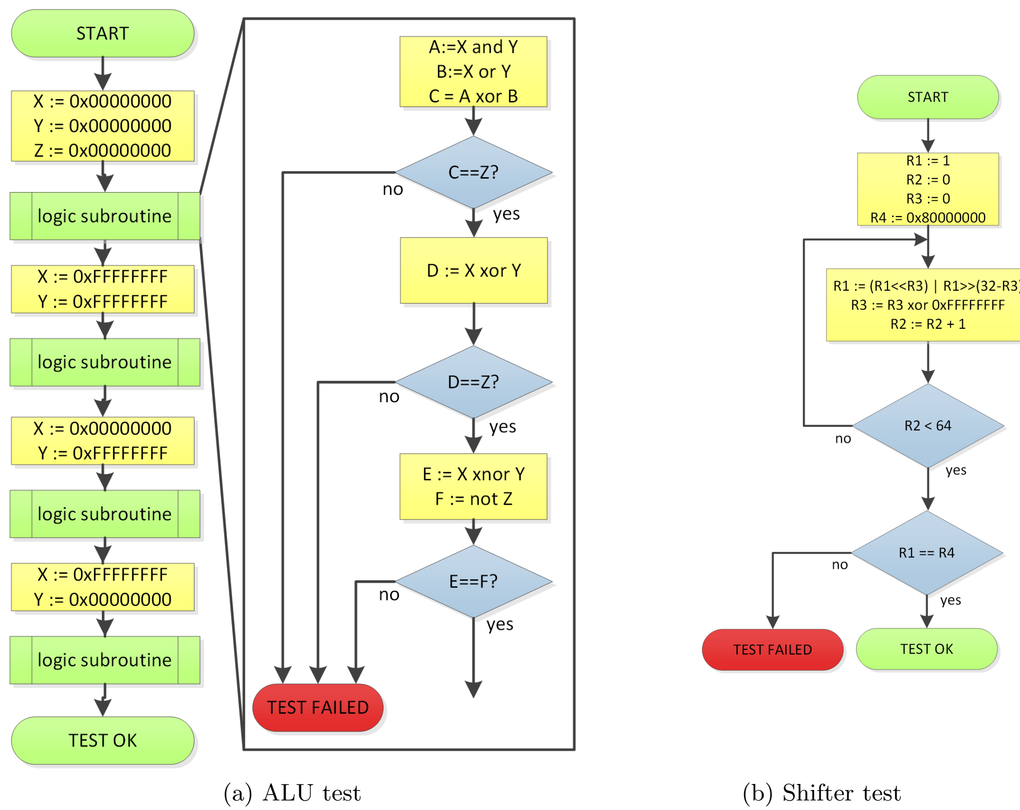
Figure 3.2: Basic Structure of the CPU Muliplier and Divider Tests

3.3.4 CRC Test

The CRC test is a generic test which does not aim to test specific components of the CPU. It is based on the *MiBench* test suite. Two diverse CRC algorithms are used and their results are compared. If their results differ, the test fails. The first algorithm simply calculates a CRC by using simple logical operations without any optimization and without any pre-calculation. The second algorithm computes a CRC value by using a pre-calculated table which already contains CRC values for all 8-bit data. The basic structure of the CRC-test is shown in Figure 3.5a.

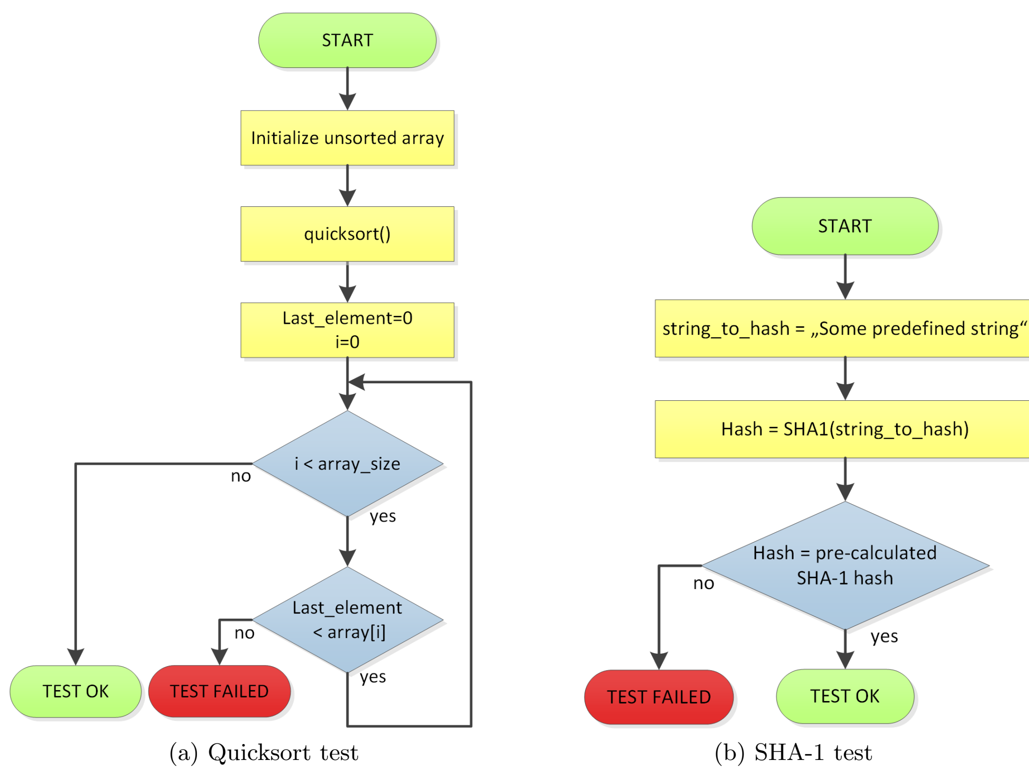
3.3.5 March Test

The march test is a test for memories and is based on a program presented in [Bar99]. It aims to test permanent single faults and coupling faults of memory cells and does not target any specific CPU components. The test initializes a memory region with some pattern and then checks if that pattern was actually written to the memory region by reading the values. Then, the data in the memory is inverted and the inverted values are read and checked. If the test reads an unexpected value, it fails. Figure 3.5b shows an overview of the march test.



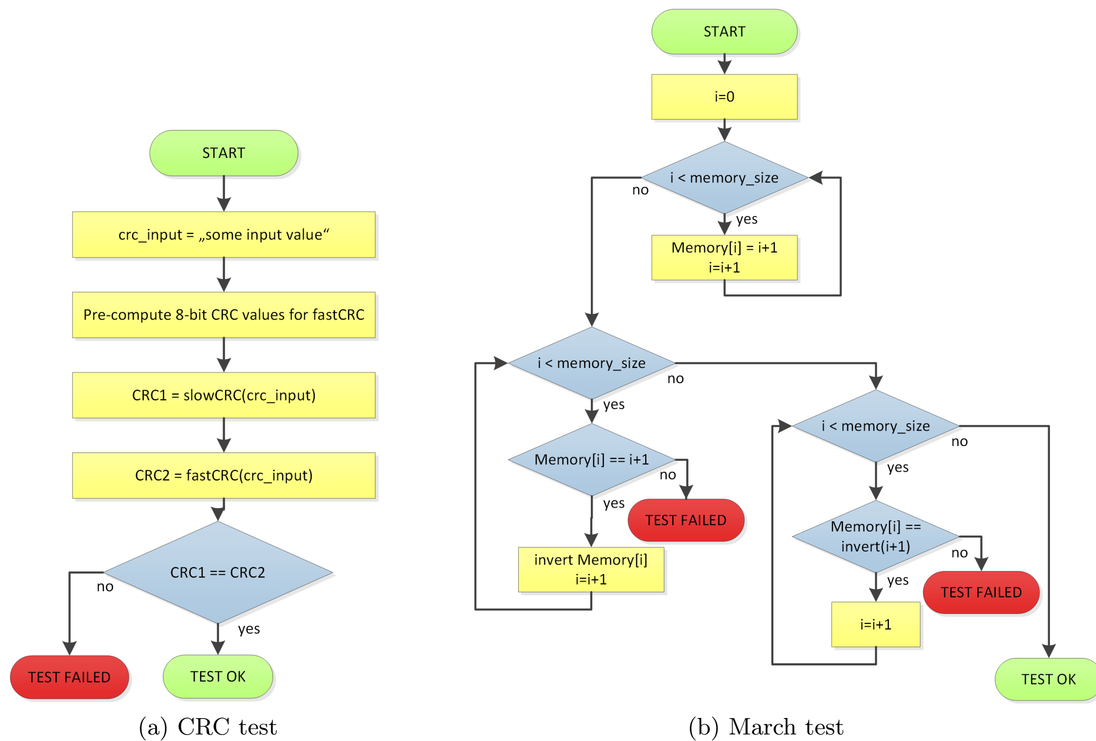
(a) ALU test (b) Shifter test

Figure 3.3: Basic Structure of the ALU and Shifter Tests



(a) Quicksort test (b) SHA-1 test

Figure 3.4: Basic Structure of the Quicksort and SHA-1 Tests



(a) CRC test (b) March test

Figure 3.5: Basic Structure of the CRC and March Tests

Chapter 4

Implementation of the Fault Injection System

This chapter describes the implementation of the fault injection system. First, the used CPU simulation testbench is described. Next, a TCL script which coordinates the fault injections into the simulation testbench is described. The core elements of the TCL script which inject the faults and which observe the results are covered in detail. Then, the different software components of the fault injection test setup and their runtime behavior are described. A the end of this chapter, a description about how to set up the testbenches for the used CPU simulations is given.

4.1 Simulation Testbench

The Modelsim tool is used to simulate the hardware description language code of a CPU. For some open source CPUs, a testbed with the HDL code for the CPU can be obtained from *opencores.org*.

For the fault injection system, the HDL code for the testbed does not have to be modified and can directly be used for the presented fault injection approach. The only CPU-specific modification is that a library to read and write the GPIO pins of the specific CPU has to be provided. However, this library is usually provided with a CPU testbed. Thus, just the function names of the library functions have to be modified in order to match the later on presented self-tests.

Figure 4.1 shows a screenshot of the Modelsim testbench environment. In the Modelsim environment, every signal in the CPU can be watched if it is added to the watch window (right side of the figure) before the simulation. However, the presented self-test verification system does not use this approach to present results, but gives debug information (the box at the bottom of the figure) to provide test summaries. The feature of Modelsim to allow watching the exact values of a register over time was still interesting, because during the development of the fault injection system this allowed easy verification whether the operations (e.g forcing a register value) were correctly applied and actually had an effect.

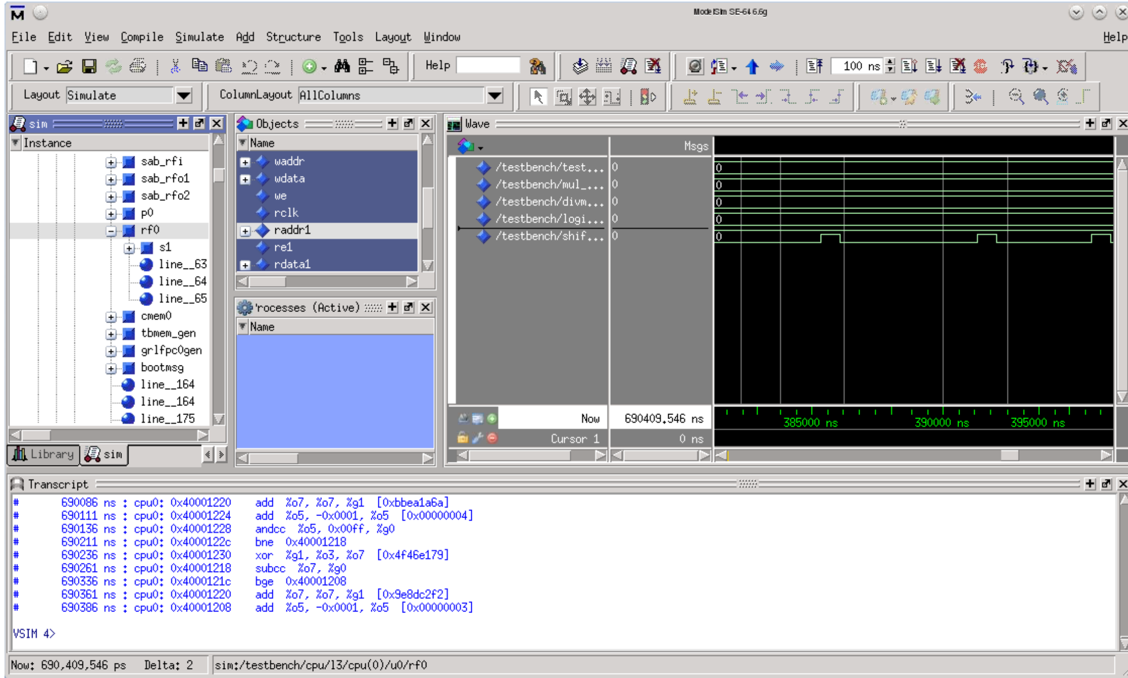


Figure 4.1: Modelsim Testbench

4.2 TCL Script

A TCL script is used to coordinate the fault injections and to send commands to the Modelsim testbench. TCL (Tool Command Language) is a script language which is supported by several computer-aided design tools [MH97]. Also Modelsim supports TCL and provides several TCL commands to interfere with Modelsim simulations.

Figure 4.2 describes the interaction between the TCL script (left), the Modelsim testbench (middle) and the self-test programs simulated by the testbench (right). The TCL starts the testbench simulation which simulates the test programs. When a test program starts, it signals that to the TCL script via a GPIO pin. The TCL script then injects a fault (forces a single register value). After the test program finishes, it signals to the TCL script via the GPIO pins whether a fault was detected or not. After the test run is finished, the TCL script resets the simulation and triggers the next test run. The script starts a test run for each fault type (transient, stuck-at 0, stuck-at 1), for each test program, and for each register which is pre-configured in the script (more information about the selection of the specific register is given in Section 5). Additionally, the TCL script stores the results for each test run (register, type of fault, whether the test was successful or not) and reports these results by the end of the simulations. This has the advantage that the script coordinates all test runs and makes it possible to run the test fully automatically without any human interference required.

Figure 4.3 shows a simplified version of the TCL code which coordinates the fault injections. The following sections give further explanations of this TCL code and of the specific commands it uses. The actually applied TCL code is presented in Appendix A.

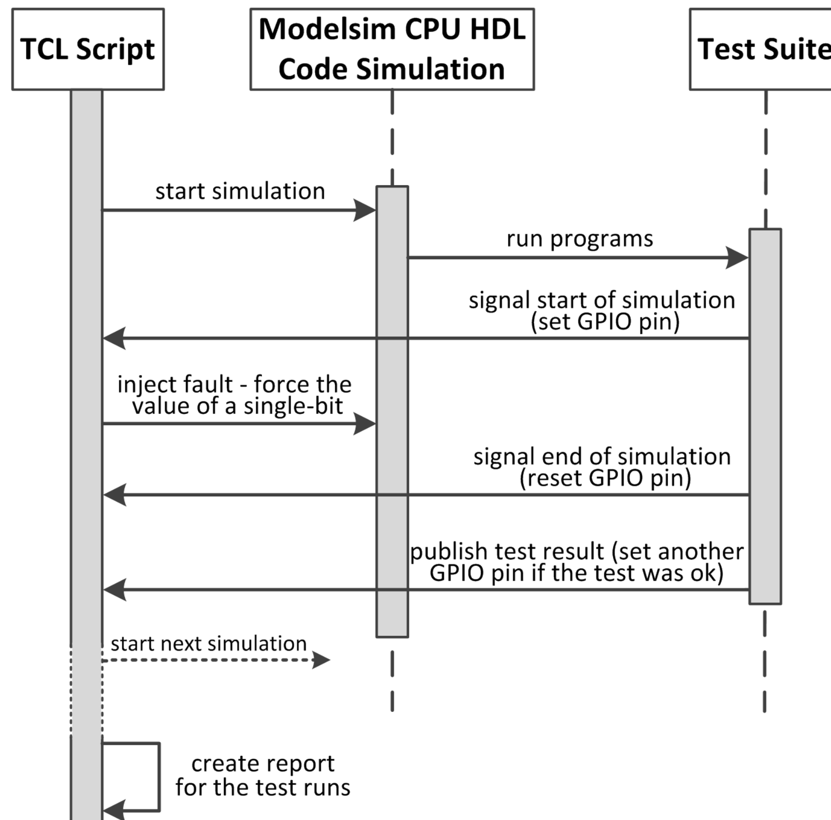


Figure 4.2: Fault Injection System TCL Script

4.2.1 Forcing Modelsim Values

To inject the faults, the TCL script overrides Modelsim register values. This can be done with the Modelsim **FORCE** command. The force command can be parametrized with the duration how long the value should be forced or it can also be parametrized to just flip a register value. For a stuck-at fault, the TCL script forces the value of a single register to either 0 or 1 for the whole duration of the test program. For a transient fault, the script just flips a register value at a random time during the test run. To know during which period of time the fault has to be randomly injected, first, the test program is run without injecting a fault. At the end of this test run, the duration the test takes is measured. This is the maximum time for the random transient fault injection.

4.2.2 Watching Modelsim Values

To react on signal changes in the simulation, Modelsim provides the **WHEN** command. The when-command takes two parameters and executes the second parameter when the first parameter gets fulfilled. This means that the second parameter is just executed exactly when the first arguments goes from value 0 to 1. It is not executed during a period while the first argument just stays one. The when command is used to detect the start and the end of the test program in order to determine the time when the fault has to be injected.

```
onbreak{
  incr faults_detected
  resume
}

foreach current_register $register_list {
  #inject in 32bits of the registers
  for {set i 0} {$i<32} {incr i} {
    set force_element $current_register
    append force_element "($i)"
    RESTART -FORCE
    NOWHEN *
    #gpio(0)=high --> test program running
    WHEN {sim:/gpio(0)==0} {
      NOFORCE $force_element
      NOFORCE sim:/gpio(1)
    }
    if {$fault_to_inject=="STUCKAT"} {
      WHEN{sim:/gpio(0)==1} {FORCE -FREEZE $force_element $zero_or_one}
    }
    if {$fault_to_inject=="TRANSIENT"} {
      #realize a random delay
      WHEN{sim:/gpio(0)==1} {FORCE -FREEZE sim:/gpio(1) 1 $random_time}
      WHEN {sim:/gpio(1)==1} {FORCE -DEPOSIT $force_element [expr![EXAMINE $force_element]]}
    }
    RUN $simulation_time
    if {[EXAMINE sim:/gpio(2)]==1} {
      incr faults_detected
    }
  }
}
echo $faults_detected
```

Figure 4.3: Simplified Version of the TCL Code

4.2.3 Reading Modelsim Values

To read GPIO values from the Modelsim simulation, the **EXAMINE** command is used. This command returns the current value of a parametrized register in the simulation. The main usage of the examine command is at the end of each test run. The examine command is used to read the test result values (which are written to GPIO pins). Additionally, the examine command is used during the injection of transient faults. A transient fault is a register value flip. In order to flip a value, the value is first read with the examine command and the inverse value is then forced to the register.

4.2.4 Execution Failures

In some cases, an execution failure occurs during the execution of the CPU tests. Such a failure could be a CPU halt which could occur if the CPU is being put into an invalid state due to the fault injection. If the fault injection, for example modifies the program counter and makes it point to a memory region which does not contain CPU instructions, such a CPU halt would occur in the simulation. In such a case, the TCL script would not be notified that the test program ended, because it never would. However, in case of a CPU failure, the TCL script is automatically interrupted and an error is reported, because handling the error would require manual interferences, which is undesirable. The TCL script uses the *ONBREAK* command to define a reaction for such CPU failures. If a break occurs, simply the next test run is started.

4.3 Software Components

The main software components are the fault injection TCL script and the CPU self-tests. However, additionally to them, also a component is required which handles the invocation of the self-tests and the communication of the self-tests with the GPIO pins. Figure 4.4 shows the software components.

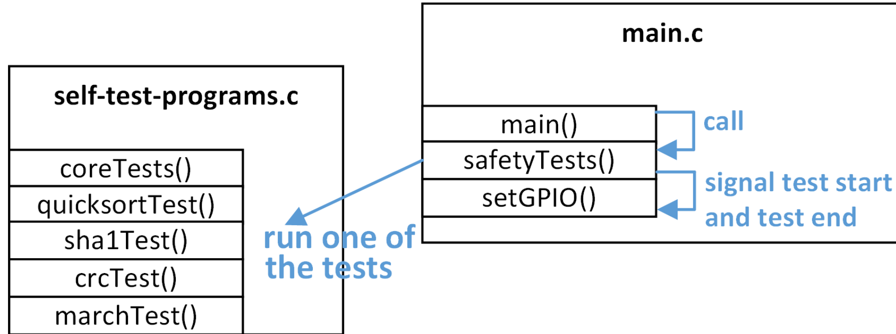


Figure 4.4: Involved Software Components

The main program calls a routine which first sets the GPIO pins to signal the TCL script the start of a test. Next, the routine calls one of the safety-test programs and signals with the GPIO pins when the test is finished. Figure 4.5 shows such a test run. The test runs are controlled by the TCL script. The script starts a test run with a specific test, injects the fault and waits for the test results. After the results are obtained, the next test run with the next test program is started.

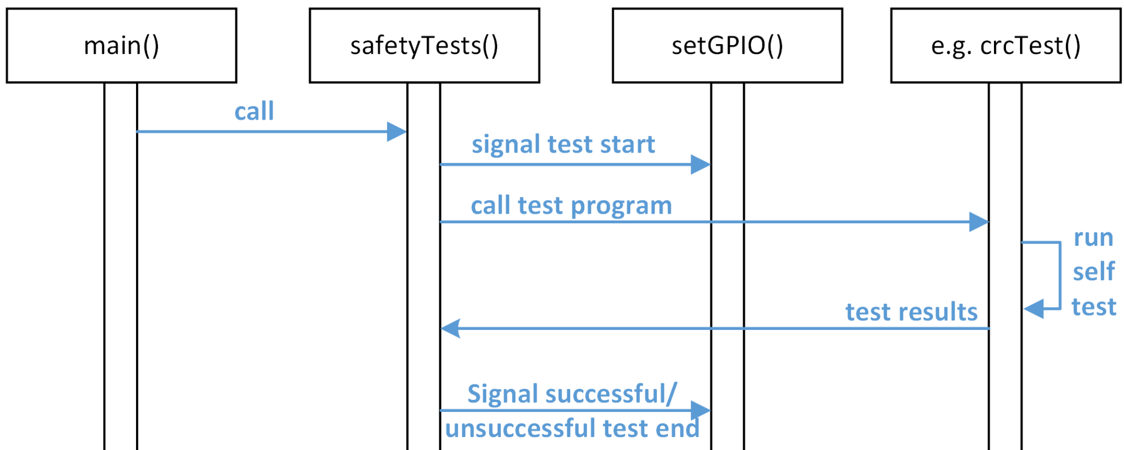


Figure 4.5: Test Run Sequence

4.4 Setting Up Processor Testbenches

This section explains how to set up the testbenches for the LEON3 and for the Plasma/MIPS processor. Those two processors were used to simulate the self-tests while applying fault injections.

4.4.1 LEON3 Testbench

This section describes how the self-tests are cross-compiled to be simulated on a CPU.

Obtaining the Testbench - The LEON3 testbench can be obtained from Gaisler Research in the GRLIB package [Aer13]. The obtained package has to be unpacked.

Configuring the CPU - The CPU functions can be enabled or disabled via a configuration tool. This graphical tool can be launched from the extracted `/designs/` directory with `make xconfig`. The tool allowed one to adjust processor features like the cache, for example. For the cache one can adjust whether it should be active or even its size. Figure 4.6 shows the graphical configuration tool. For the presented simulation, no configuration of `make xconfig` were changed.

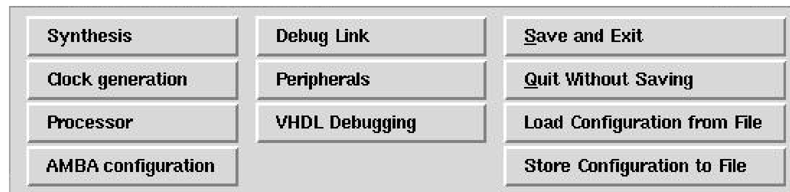


Figure 4.6: Xconfig Tool

Compiling the CPU - For compilation, also a graphical tool can be used. This tool can be launched from the `/designs/` directory with `make xgrlib`. The tool contains a *Build* button which can be clicked to compile the sources. Figure 4.7 shows a screenshot of this tool.

Compiling the Programs which run on the CPU - To compile the self-test programs, a LEON3 cross-compiler has to be installed. The compiler can be acquired from the Gaisler website. Some code (e.g. `/designs/example.c` can be compiled with the following commands:

- `sparc-elf-gcc example.c -o example`
- `sparc-elf-mkprom example -o example.exec`
- `sparc-elf-objcopy -O srec example.exec example.srec`

Installing the Compiled Program - For the LEON3 CPU to actually run the compiled program, either the compiled standard program can be replaced (`cp example.srec prom.srec`) or the testbench can be modified in order to load the new program (references in the `testbench.vhd` file to `prom.srec` have to be replaced with `example.srec`).

Figure 4.7: Xgrlib Tool

Starting Modelsim - To start the simulation environment, the testbench has to be called with the following sequence of commands:

- `make clean`
- `make vsim`
- `make testbench`

Running the TCL script - To start the simulation, the TCL script has to be loaded by selecting `Tools->Tcl->Execute Macro` from the menu. Now the TCL script can be selected and will start.

4.4.2 Plasma/MIPS Testbench

This section describes how the self-tests are cross-compiled to be simulated on the Plasma/MIPS CPU. The instructions are based on [Ope13]. Compared to the LEON3 CPU it is a bit easier, because no configurations have to be made.

Obtaining the Testbench - The Plasma/MIPS testbench can be obtained from opencores.org. The obtained package has to be unpacked.

Obtaining the Crosscompiler - To be able to compile programs for the Plasma/MIPS CPU, a cross-compiler has to be set up. The instructions are based on [Eri09]. The following tools were built:

- `GNU Binutils`
- `GNU MPFR (Muliple-Precision Floating-point with Rounding`

- Newlib
- GNU GCC

Compiling the Program - For the compilation, simply the just compiled cross-compiler is called with `gcc -c -g example.c`

Installing the Program on the Processor - The compiled file (`example.txt`) has to be copied to the simulation directory of the Plasma/MIPS processor (the root directory in the unpacked testbench) in order for the processor to simulate this application.

Opening the Testbench - Modelsim has to be opened and the `test_bench.vhd` file of the Plasma/MIPS VHDL testbench has to be loaded.

Running the TCL script - To start the simulation, the TCL script has to be loaded by selecting `Tools->Tcl->Execute Macro` from the menu. Now the TCL script can be selected and will start.

Chapter 5

Verification and Evaluation of the Safety-Tests by Fault Injection

This chapter shows the evaluation of the presented generic self-test programs for CPUs. This chapter also describes on which processors the tests were simulated and it also describes the test results regarding fault coverage and simulation time in detail. Furthermore, the chapter evaluates the requirements from Section 3.1 qualitatively.

The fault injection system presented in section 3.2 is used to evaluate how many faults can be detected by the tests presented in 3.3. The overall workflow for simulating the self-tests on a processor with the presented fault injection system is shown in Figure 5.1. The main tasks were to set up the simulation testbench and to define the registers where the faults have to be injected. Running the simulation itself and interpreting the results was not very time-consuming.

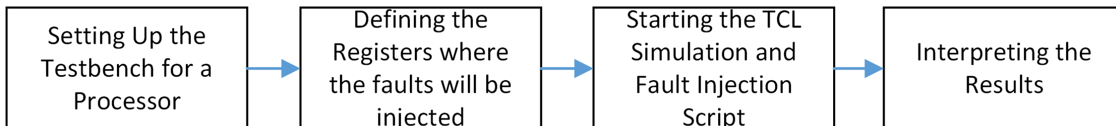


Figure 5.1: Workflow for Simulating the Self-Tests

5.1 Processor Architecture

For the evaluation of the self-tests, two different processors are simulated while injecting faults into the simulation model. A Plasma/MIPS and a LEON3 CPU are used. The reason for choosing these two CPUs is that both CPU cores are open source and publicly available with an easy to use testbench.

The Plasma/MIPS CPU is a 32bit RISC processor containing basic core units such as an ALU, a multiplier, and a shifter. The CPU uses a tree stage pipeline. The Plasma/MIPS CPU is often used in education or in scientific projects, because the source code is small due to the limited features of the CPU and it is easily understandable, because it is clearly written and very well documented. Figure 5.2 shows an overview of the CPU architecture.

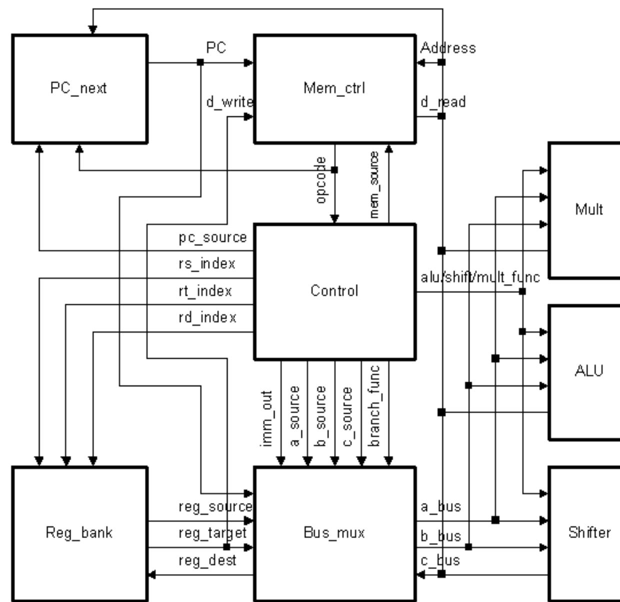


Figure 5.2: Plasma/MIPS Architecture (opencores.org/project,plasma)

The LEON3 CPU a 32bit processor. It is based on the SPARC-V8 architecture and it is an open source CPU core maintained by *Aeroflex Gaisler*. The LEON3 processor code is highly configurable. The processor uses a 7-stage pipeline and has got separate data and instruction cache. Advanced features such as a memory management unit can be configured to be included in the processor. Compared to the Plasma/MIPS processor, the LEON3 is much more complicated due to the many possible configurations and due to the many more features which the processor provides. Figure 5.3 shows and overview of the main components of the LEON3 processor.

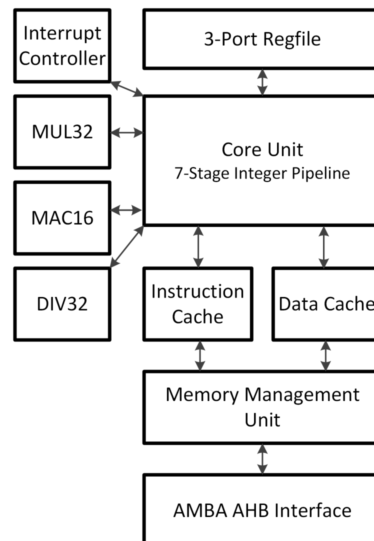


Figure 5.3: LEON3 Architecture [Aer13]

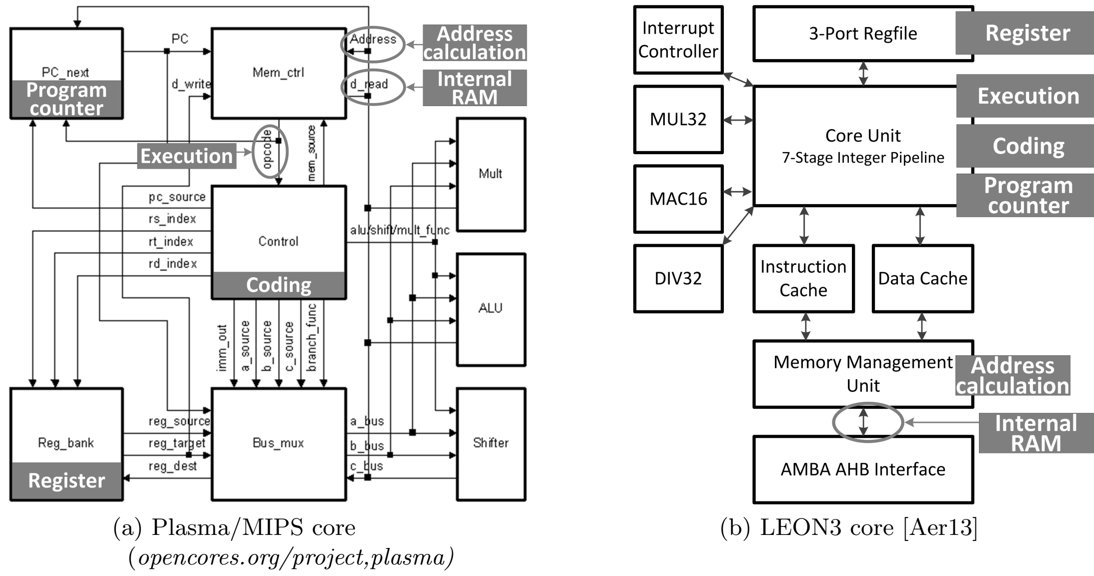
5.2 Fault Injection Target Registers

One of the main tasks when evaluating the fault coverage of the self-tests on a specific processor architecture was to come up with a list or relevant registers for which faults should be injected. To obtain data of the overall test coverage for the whole processor, it would be necessary to inject faults into all registers. However, such a fault simulation would require much to long simulation time. Therefore, just register which are relevant according to the IEC 61508 standard were selected. The functionality which is relevant for safety processor tests as described in detail in Section 2.2.6 are:

- Register, internal RAM
- Coding and execution
- Address calculation
- Program counter
- Stack pointer

To test this functionality, relevant registers which are required for this functionality were selected and the fault injection was just applied to these registers. For example, the IEC 61508 standard requires to test the execution of instructions. To test this, fault were injected into the registers of the execution stage of the pipeline. Such registers were located for all of the above given functionality apart from the stack pointer. The stack pointer is not defined on a hardware level for the examined processors, but it is defined by the compilation toolchain. Therefore, stack pointer tests were applied on a software level. Instead of defining the register in the TCL script, for the stack pointer test, a software routine is written. This software simply uses an assembler command to flip a bit of the stack pointer. The TCL script is used to trigger this test routine and to tell the test routine which bit of the stack pointer should be flipped. The reason for also including the stack pointer tests into the TCL script is that the stack pointer test can then be integrated in the overall test framework which means that then all tests can be automatically run without any human interaction.

Figure 5.4a and Figure 5.4b show which registers were used for fault injection. The list of the exact registers is given in Appendix C.



(a) Plasma/MIPS core (*opencores.org/project,plasma*)

(b) LEON3 core [Aer13]

Figure 5.4: Overview of the Registers where the Faults are Injected

5.3 Test Results

This section presents the results of the fault injections and discusses the diagnostic coverage and the simulation time.

5.3.1 Diagnostic Coverage

Table 5.1 shows the diagnostic coverage values for the Plasma/MIPS CPU and Table 5.2 shows the diagnostic coverage values for the LEON3 CPU. The diagnostic coverage values were calculated with the following formula:

$$DiagnosticCoverage(DC) = \frac{DetectedFaults}{PossibleFaults} = \frac{\sum \lambda_D}{\sum \lambda_{All}}$$

All the diagnostic coverage values in both tables show very high values of more than 90%. These are rather high values for diagnostic coverage, in particular because also transient faults were covered. On the one hand it is not surprising that the test programs have a high diagnostic coverage, because if a stuck-at error in the stack pointer occurs, it is most likely that any test program would detect the error. On the other hand it is very surprising to see that some of the test programs which were not specifically developed to test CPU cores (for example the CRC test) achieve even a higher diagnostic coverage than the generic core tests, which were specifically developed to test CPUs.

With a fault coverage value of more than 90%, according to the IEC 61508 standard, a test is suitable to be used in a SIL2 environment. This means that any of the 5 presented test programs would be qualified to be used in such a safety-critical environment. Of course one has to be careful with this interpretation, because the fault simulations just took place on two specific processors and could probably achieve a lower diagnostic coverage on different processor architectures. However, the fault injections indicate that the tests

		Register, Internal RAM	*Coding and Execution*	*Address Calculation*	*Program Counter*	*Stack Pointer*	***Diagnostic Coverage***
Generic CoreTests	Stuck-At	100.0%	99.5%	100.0%	100.0%	100.0%	**99.9%**
	Transient	94.2%	96.9%	94.6%	95.3%	100.0%	**96.2%**
Qsort Test	Stuck-At	84.8%	99.5%	81.7%	85.6%	100.0%	**90.3%**
	Transient	97.7%	93.8%	93.8%	100.0%	100.0%	**97.6%**
CRC Test	Stuck-At	100.0%	97.9%	100.0%	100.0%	100.0%	**99.6%**
	Transient	99.1%	100.0%	100.0%	100.0%	100.0%	**99.8%**
SHA-1 Test	Stuck-At	100.0%	100.0%	100.0%	100.0%	100.0%	**100.0%**
	Transient	99.2%	100.0%	97.5%	96.3%	100.0%	**98.6%**
March Test	Stuck-At	85.5%	92.2%	100.0%	81.9%	100.0%	**91.9%**
	Transient	97.7%	100.0%	100.0%	100.0%	100.0%	**99.6%**

Table 5.1: Plasma/MIPS Stuck-At and Transient Fault Diagnostic Coverage of the Generic Test Programs

		Register, Internal RAM	*Coding and Execution*	*Address Calculation*	*Program Counter*	*Stack Pointer*	***Diagnostic Coverage***
Generic CoreTests	Stuck-At	100.0%	100.0%	96.8%	100.0%	100.0%	**99.4%**
	Transient	87.7%	90.6%	86.0%	89.3%	100.0%	**90.7%**
Qsort Test	Stuck-At	95.3%	97.7%	100.0%	70.3%	100.0%	**92.7%**
	Transient	76.5%	96.9%	100.0%	96.9%	100.0%	**94.1%**
CRC Test	Stuck-At	93.8%	97.3%	100.0%	85.9%	100.0%	**95.4%**
	Transient	96.5%	93.8%	100.0%	93.8%	100.0%	**96.8%**
SHA-1 Test	Stuck-At	100.0%	100.0%	100.0%	82.8%	100.0%	**96.6%**
	Transient	96.9%	96.1%	96.7%	97.1%	100.0%	**95.1%**
March Test	Stuck-At	99.3%	99.1%	98.4%	70.3%	100.0%	**92.3%**
	Transient	100.0%	98.5%	100.0%	96.9%	100.0%	**99.1%**

Table 5.2: LEON3 Stuck-At and Transient Fault Diagnostic Coverage of the Generic Test Programs

achieve a high coverage and it can be assumed that also on other processors this coverage might be high. If the tests are simulated on many other open source processors and also achieve a high coverage for them, one might even be able to argue that the test should achieve a high coverage independently from the processor architecture. With this argumentation one could use the tests on another processor without verifying them with fault injection. This could be beneficial if a processor is used where the source code is not openly available and for which therefore no tests can be simulated.

Due to the high diagnostic coverage values of the tests, they are qualified to be used in

SIL2 environments. However, this does not mean that they are sufficient to be used for a processor which has to be SIL2 certified for a specific environment. The IEC 61508 safety standard gives specific parts of the CPU (e.g. stack pointer) which have to be tested, but the standard does not say that these parts are sufficient for the whole processor. These parts just have to be tested in any case. This means that the presented self-tests can be used to test the basic functionality of a processor, but additional tests might still be necessary. For example, if a processor has got an Ethernet interface on the chip, this interface is most likely not sufficiently tested by the proposed self-test programs. However, if this interface is used in a safety-critical environment, it also has to be tested. In such a case, the proposed generic self-test programs can be taken to test the basic processor functionality, and additionally specific tests for the Ethernet interface would have to be developed.

One interesting test program is the last one: The March Test. According to the IEC 61508 standard, such a test has to be applied to test the RAM memory of a safety-critical system in any case. It is interesting to see that this test does not just check the RAM memory, but also tests the CPU. It could even be argued that it would be sufficient to just apply the memory test and have the CPU 'indirectly' tested through these tests. However, one has to be careful with this argumentation, because the IEC 61508 standard says that the ALARP (As Low As Reasonably Practicable) principle always has to be applied. This means that a measure which can reduce the probability of failure of a safety-critical system always has to be applied if the cost to implement this measure are reasonable when related to the hazard which would be induced when not implementing this measure. In the case of the self-tests this would mean that even though the March Test has a diagnostic coverage of more than 90% and therefore would be capable of being used in a SIL2 environment, the March Test would still not be sufficient, because the Generic Core Tests have a higher diagnostic coverage and applying them does not induce much cost (low execution time, no development time as the program is generic). Therefore, in any case the generic core tests (or better tests) have to be part of a safety-critical CPU if one has the information form the presented diagnostic coverage data in Table 5.1 and 5.2.

5.3.2 Running the Tests on a Safety-Critical Device

A possible approach to build safety tests for a processor is to take all of the 5 presented test programs and run them one after the other. According to the safety standard, a self-test either has to be applied:

- every time before the device is involved in a safety-critical functionality (this applies if the system has a hardware fault tolerance of 0 which means that a failure of the device can lead to a safety-critical state)
- once every day (this applies if the system has a hardware fault tolerance of more than 0 which means that more than one device [e.g. more than one CPU] has to fail in order to bring the system into a safety-critical state)

For a system with hardware fault tolerance of more than one, the presented tests could be scheduled in parallel to the regular CPU functionality and they would have to run once a day which does not require a lot of the overall processing power.

5.3.3 Simulation Time

The simulation ran on a computer with a 6-core 3.2GHz AMD Phenom-II CPU and 16GB RAM. The simulation time was rather long, because the CPU is not emulated on hardware, but the VHDL code is simulated with Modelsim in software. However, just few registers had to be tested, because just registers explicitly relevant according to the IEC 61508 standard were considered. A test for a single fault injection which includes initializing the CPU, starting the test, running the test, injecting the fault, finishing the test, and reporting the test result took about 10 seconds in average. The exact test duration strongly depends on the runtime length of the test program (e.g. the generic core tests are a lot quicker than the SHA-1 test). For the Plasma/MIPS CPU, a single test run was about 10 times faster. This is the case, because the LEON3 CPU is much more complicated than the Plasma/MIPS CPU. This means that many more registers have to be simulated and therefore the overall simulation of the LEON3 CPU is slower.

These slow test times were still sufficient, because the testbench is fully automated. This means that after the tests were parametrized (configuration of the registers where the faults have to be injected), no more human interaction was necessary and the faults were injected for every test program and for every bit of the register. Figure 5.5 visualizes this. For stuck-at faults, two different test runs took place. One for stuck-at 1 and one for stuck-at 0. A stuck-at fault was always injected for the whole duration of the test run. For transient faults, several test runs took place, because the behavior of the test program depends on the time when the fault is injected. Therefore, several test runs (9-10 test runs) took place for transient faults, each at a random time during the test program. Table 5.3 calculates in detail the amount of faults which were injected for each register which is overall 1920. When considering that a single test run on the Plasma/MIPS processor takes about 1 second, this means that testing a single register for all the fault types and all the test programs takes about half an hour. Such test runs for a single register of the LEON3 CPU take even longer. For the LEON3 CPU the tests for a single register require about 5 hours.

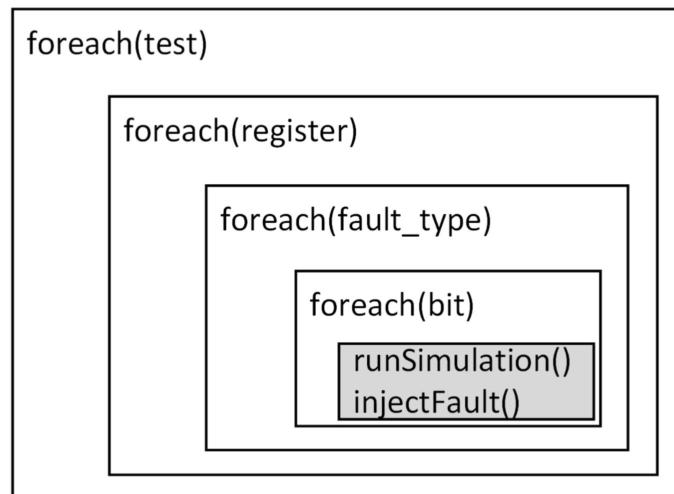


Figure 5.5: Sequence of the Fault Injections

Stuck-At	**32** (one injection for each register bit) **x 5** (number of test programs) **x 2** (stuck-at 1, stuck-at 0)	**320**
Transient	**32** (one injection for each register bit) **x 5** (number of test programs) **x 10** (several test runs with fault injected at a randomly different time)	**1600**
SUM		**1920**

Table 5.3: Number of Test Runs for each Register

Again, these slow test times are still sufficient, because the tests are fully automatic and just few registers had to be tested. For both CPUs about 30 registers had to be tested and the overall simulation took about one week. Altogether about 120,000 faults were injected during this simulation. However, for more complex simulations (for example, for more complex test programs) or for more exhaustive simulations (more registers, more faults to inject), the proposed fault injection system would not be sufficient. In such a case, the CPU would have to be simulated in hardware in order to be faster. The speedup with such an approach would be significant (according to literature about factor 1000 [BP03]). The disadvantage of a hardware-based simulation would be that one has to modify the HDL code in order to inject faults. With the presented fault injection system that is not necessary.

5.4 Discussion of the System Requirements

R1 - Generic Self-Tests: The self-test programs are written in C. Thus they are not processor specific and can also be applied to other CPU architectures. However, the downside of having C-code for the self tests is that the compilation step brings uncertainties regarding the actual assembler code which is run at the target processor. However, for the specific CPU architectures covered in this thesis, this did not matter.

R2 - Self-Test Verification: The self-tests were verified with fault injection. For every self-test and every relevant target register, several transient and stuck-at faults were injected for every bit.

R3: IEC 61508 compliant self-tests: The faults were injected into CPU registers relevant according to the IEC 61508 standard. As required by the standard, single bit transient and stuck-at faults were injected. On the LEON3 and Plasma/MIPS processor, the presented self-tests achieve a fault coverage of more than 90% which is required by the safety standard for the tests to be used in a SIL2 environment.

R4 - Simulation time: The simulation time for injecting a single fault on the used simulation hardware is between one and ten seconds depending on the test program and on the actually simulated CPU. For verifying all relevant registers of a CPU for all

tests this leads to an overall simulation time of about a week which is acceptable. For just injecting specific faults to check whether a specific test works (which might be required during safety certification), just a few seconds of simulation time are necessary, which is acceptable as well.

R5 - Fully automated simulation: The presented TCL script automates the whole simulation and does not require any human interaction after the registers to be tested have been defined. This means that an operator can simply call the script and collect all test results when the whole simulation is done.

R6 - Easy to use: The fault injection system does not require any modifications in the HDL code of the processor which makes its application very easy. This means that somebody using the fault injection system does not require a deep understanding of the processor source code. However, one has to define the target registers of the processor. This means that still one has to look at the HDL code to chose the appropriate registers. When using the fault injection system, one also does not have to cope with the actual fault types which have to be injected according to the safety standard, because the fault injection system already injects all necessary fault types.

Chapter 6

Conclusion

6.1 Discussion of the Results

This thesis gave an overview of the IEC 61508 standard and on software-based self-tests to test memory components and CPU components. Furthermore, the basics of fault injection testing were explained and different fault-injection approaches were discussed. A fault injection system to test a set of presented generic CPU self-tests was described and fault injections were conducted. The fault injection results were presented and the suitability of the proposed self-test programsfor safety-critical environments was discussed.

The following points highlight the main outcomes of this thesis of which a reader could benefit:

- The design and implementation of a fault injection system specifically tailored for safety-tests which is very easy to handle and can easily be configured to be used for other HDL code. This fault injection system could be used by others to verify self-test in safety-critical domains. The main advantage of the system is that it does not require any modification of the HDL code, but just configuration of the TCL script which controls the fault injection and the test simulation. The software-based simulation makes the tests very slow, however, for the safety domain this is not a big problem, because often during safety certification just single faults have to be injected to prove to a safety certification authority that a specific test is able to detect specific types of faults. For such uses, the presented fault injection system simulation speed is sufficient.

- Insight into the field of 'indirect testing' of hardware components with tests which are actually aimed to test other components. Most of the presented generic tests did not aim at testing CPU components. However, it was shown that still they achieve a high diagnostic coverage for the CPU. Sometimes such 'indirect testing' is used in the safety domain if direct testing is not possible. This would be the case if the source code of a processor is not available and self-tests have to be verified. In such a case one could take tests with proven coverage (like the presented multiplier test) and argue for the overall CPU test coverage with indirect testing. This thesis provided some insight into indirect testing and also presented diagnostic coverage values for such indirect tests (like for example the March Test).

- A set of generic test programs which can be used as a basis to construct self-tests for processors in safety-critical environments. One of the main benefits of the thesis for someone who wants to develop self-tests for a CPU is that the thesis provides a set of well-proven self-tests which are even architecture independent. Of course this architecture independence brings the uncertainty of what the compiler will make when translating the code to a target architecture. However, for the two presented different CPU architecture, it did not make much of a difference. In any case, the presented tests can be taken as a starting point when developing self-tests for a specific CPU which has to be safety-certified.

6.2 Future Work

The current state of the work provides verification results for self-tests on two different processor architectures. However, there is plenty of ideas which could enhance this work:

- Additional CPU architectures. It would be of great interest to evaluate the self-tests of more different CPU architectures. This would help to argue for the processor architecture independence of the generic self-tests. If the self-tests achieve a high coverage also for other processor architectures, then this would hint that they also have a high coverage for CPU architectures where such simulations cannot be applied (for example because their source code is not available).

- More self-tests. The current self-tests are based on literature or on the *MiBench* test suite. It would be of interest to evaluate further self-tests to see whether other tests are better and to see whether any of the test-programs manages to detect faults in the by IEC 61508 required registers where faults apparently are rather simple to detect (for example, a fault in a stack pointer is very easy to detect for any program).

- Get a safety certification authority involved. Currently, the fault injection system and the self-tests are developed without much feedback from a safety certification authority. At the beginning of this work, the main concept of the self-tests and the fault injection system were discussed with TÜV Rheinland. They approved of the concept, however, they so far did not assess it in detail and it was so far not yet used during a safety certification. Future work on this point will be to actually use the argumentation of the generic self-tests which indirectly test CPU components during a safety certification.

- Use the fault injection system in different domains. It would be of interest to use the presented system not just to test safety-critical self-tests, but also to use of for other domains. For example, systems which have to undergo Common Criteria security certification also require fault injection tests. The fault injection system could be used there and it would be interesting to see whether the slow fault injections are sufficient also for this domain.

- Emulate the tests on hardware. One of the main drawbacks of the presented fault injection system is that the simulation time is rather long. Simulating a very complex CPU or simulating very long test programs might be infeasible. To speed up the

fault injections, the VHDL code could be emulated in hardware. It would be very interesting to see how much faster the fault injections could be applied. However, it would require major modifications of the fault injection system, because the TCL script could not be used anymore to control and watch the simulation. Instead one would have to communicate to the CPU hardware and react on the GPIO pin signals of the CPU.

Appendix A

TCL Source Code

```
#REGISTER WHERE A FAULT IS INJECTED
set signal_to_force "/register/to/force"

#TRANSIENT OR STUCKAT OR STACK(=transient error on stack-pointer)
set fault_type "STUCKAT"

#FORCE VALUE 0 OR 1 FOR STUCK-AT
set force_value "1"

#COUNT PROCESSOR HALTS
onbreak {
  set broken 1
  incr broken_count
  resume
}

#NEEDED FOR BINARY OUTPUT
proc dec2bin int {
  binary scan [binary format I $int] B* res
  set res
  set res [string range $res 0 31]
}

set start_time [clock seconds]
set result ""
set result_summary ""

for {set i_modes 0} {$i_modes < 3} {incr i_modes} {
    if {$fault_type != "STACK"} {
      if {$i_modes == 0} {
        set fault_type "STUCKAT"
        set force_value "1"
      }
```

```
      if {$i_modes == 1} {
        set fault_type "STUCKAT"
        set force_value "0"
      }
      if {$i_modes == 2} {
        set fault_type "TRANSIENT"
      }
    } else {
        incr i_modes
        incr i_modes
    }

set broken_count 0
set infinite_count 0
set found_count 0
set ok_count 0

#INJECT IN 32 REGISTERS
for {set i 0} {$i < 32} {incr i} {

  set broken 0
  set force_signal_element $signal_to_force
  append force_signal_element "($i)"
  set cycle_count 0;

  #RESET OLD COMMANDS
  if {$i>0} {
    nowhen *
  }

  restart -force
  run 74000

  #JUST INJECT THE FAULT IF gpio(0)=high
  when -label gpiowhen {sim:/tbench/gpio0_out(1) == 0} {
    force -freeze sim:/tbench/gpioa_in(1) 0
    noforce sim:/tbench/gpioa_in(1)
    if {$fault_type != "STACK"} {
      noforce sim:$force_signal_element
    }
    set cycle_count 0;
  }

  if {$fault_type == "TRANSIENT"} {
    append result "$now: INJECT TRANSIENT FAULTS - $force_signal_element \n"
    when -label transienttrigger {sim:/tbench/gpio0_out(1) == 1} {
```

```
    force -freeze sim:/tbench/gpioa_in(1) 1 20000 ns -cancel 30000 ns
  }

  when -label transientwhen {sim:/tbench/gpioa_in(1) == 1} {
    set transient_input [examine $force_signal_element]
    if {$transient_input == 1} {
      force -freeze sim:$force_signal_element 0 -cancel 10000 ns
      force -deposit sim:$force_signal_element 0 10100 ns
    }
    if {$transient_input == 0} {
      force -freeze sim:$force_signal_element 1 -cancel 10000 ns
      force -deposit sim:$force_signal_element 1 10100 ns
    }
  }
}

if {$fault_type == "STUCKAT"} {
  append result "$now: INJECT STUCKAT FAULTS\n"
  when -label stuckatwhen {sim:/tbench/gpio0_out(1) == 1} {
    force -freeze sim:/tbench/gpioa_in(1) 1
    force -freeze sim:$force_signal_element $force_value
  }
}

if {$fault_type == "STACK"} {
  append result "$now: INJECT TRANSIENT FAULTS ON STACK\n"
  when -label stackwhen {sim:/tbench/gpio0_out(5) == 1} {
    force -freeze sim:/tbench/gpioa_in [dec2bin $i] -cancel 200 ns
  }
}

#RUN AND INJECT FAULTS
run 7500000

#STORE INJECTION RESULTS
set pin_out [examine /tbench/tests_found]
set test_ok [examine /tbench/tests_ok]
set faults_injected [examine /tbench/test_count]
append result "$now: $i. run: injected: $faults_injected, found
               $pin_out, ok: $test_ok - broken:$broken\n"

if {$broken == 0} {
  if {$pin_out > 0} {
    incr found_count
  } else {
    if {$test_ok < 2} {
```

```
      incr infinite_count
    } else {
      if {$test_ok > 7} {
        incr infinite_count
      } else {
        incr ok_count
        }
      }
    }
  }
}

#PRINT FINAL RESULTS
nowhen *
echo "\n\nRESULTS: \n"
echo $result
append result_summary "\n $fault_type faults in $signal_to_force\n"
if {$fault_type == "STUCKAT"} {
 append result_summary "  Forced value $force_value\n"
}
append result_summary "$broken_count BREAKS\n"
append result_summary "$found_count FOUND\n"
append result_summary "$infinite_count INFINITE\n"
append result_summary "$ok_count OK \n\n"
}

set delta_time [expr [clock seconds]-$start_time]
set delta_minutes [expr $delta_time / 60]
echo $result_summary
echo "SIMULATION TIME: $delta_minutes Minuten\n"
```

Appendix B

Source Code of the Generic Test Programs

B.1 Main Test Routine

```
int main(void)
{
  UINT16 main_i;
  TBSendStartOfTest(); //signal the start of the benchmark to the TB
  for(main_i = 0; main_i < 10; main_i++)
  {
    initPIO();
    stack_fault_pattern = 0;
    setPIO(0x20);  //to be recognized by TCL for Stack-Pointer Test
    int temp = safety_test(); //CALL THE SPECIFIC TEST ROUTINE HERE
    if(temp == 42) //tests ok
    {
      setPIO(0x4);
    }
    else  //found fault
    {
      setPIO(0x8);
    }
  }
  TBSendEndOfTest(); //signal the end of the benchmark to the TB
  return 0;
}
```

B.2 CPU Core Tests

```
int test_shifter()
{
  unsigned int R1 = 1;
```

```
  unsigned int R2 = 0;
  unsigned int R3 = 0;
  unsigned int R4 = 0x80000000;
  setPIO(0x2);
  while(R2 < 0x43)
  {
    R1 = ((R1<<R3) | (R1>>(32 - R3)));
    R3 = R3^0xFFFFFFFF;
    R2 = R2 + 1;
    if(R2>0x43)
    {
      initPIO();
      return -1;
    }
  }
  if(R1 != R4)
  {
    initPIO();
    return -1;
  }
  initPIO();
  return 42;
}

int test_mult()
{
  unsigned int x = 0;
  unsigned int y = 0;
  unsigned int z1,z2;
  unsigned long long m_res = 0;
  unsigned long long temp;
  unsigned int a_res = 0;
  unsigned int carry = 0;
  unsigned int golden_value = 0;
  int count = 0;
  setPIO(0x2);
  while(x != 0x11111110)
  {
    y = 0;
    while(y != 0x11111110)
    {
      m_res = (unsigned long long)x * (unsigned long long)y;
      temp = m_res;
      z1 = (unsigned int)temp;
      z2 = (unsigned int)(temp>>32);
      temp = temp>>32;
```

```
      a_res = z1 + z2;
      temp = (unsigned long long)golden_value + (unsigned long long)a_res;
      golden_value = (unsigned int)temp;
      carry = (temp>>32);
      golden_value = golden_value + carry;
      y = y + 0x11111111;
    }
    x = x + 0x11111111;
  }
  if(golden_value==0xffffffdc)
  {
    initPIO();
    return 42;
  }
  else
  {
   initPIO();
   return -1;
  }
}

int logic_test_sub(unsigned int x, unsigned int y, unsigned int z)
{
   unsigned int a,b,c,d,e,f;
   a = x&y;
   b = x|y;
   c = a^b;
   if (c != z)
     return -1;
   d = x^y;
   if (d != z)
     return -1;
   e = !(x^y);
   f = !z;
   if (e!=f)
     return -1;
   return 42;
}

int test_logic()
{
   setPIO(0x2);
   unsigned int x,y,z,return_value=0;
   x = 0x00000000;
   y = 0x00000000;
   z = 0x00000000;
```

```
   return_value = return_value + logic_test_sub(x,y,z);
   x = 0xffffffff;
   y = 0xffffffff;
   z = 0x00000000;
   return_value = return_value + logic_test_sub(x,y,z);
   x = 0x00000000;
   y = 0xffffffff;
   z = 0xffffffff;
   return_value = return_value + logic_test_sub(x,y,z);
   x = 0xffffffff;
   y = 0x00000000;
   z = 0xffffffff;
   return_value = return_value + logic_test_sub(x,y,z);
   if (return_value != 4*42)
   {
     initPIO();
     return -1;
   }
   else
   {
     initPIO();
     return 42;
   }
}

int test_muldiv()
{
  unsigned int x = 0;
  unsigned int y = 0;
  unsigned int z1,z2;
  int golden_value = 42;
  int count = 1;
  setPIO(0x2);
  while(count < 100)
  {
    y = count;
    x = x + count;
    z1 = x * y;
    z2 = z1 / x;
    if(z2 != y)
    golden_value = -1;
    count = count + 1;
  }
 initPIO();
 return golden_value;
}
```

B.3 Quicksort Test

```
//http://www.codeuu.com/
void q_sort(int numbers[], int left, int right)
{
  int pivot, l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  while (left < right)
  {
    while ((numbers[right] >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbers[left] = numbers[right];
      left++;
    }
    while ((numbers[left] <= pivot) && (left < right))
      left++;
    if (left != right)
    {
      numbers[right] = numbers[left];
      right--;
    }
  }
  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left < pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot)
    q_sort(numbers, pivot+1, right);
}

int quick_sort_test()
{
  int arr[20] = {108,102,5,1,100,107,101,6,11,109,106,
                8,9,103,10,23,105,44,0,104};
  int elements = 20;
  q_sort(arr,0,19);
```

```
  if (arr[0]==0   && arr[1]==1   && arr[2]==5   && arr[3]==6
 && arr[4]==8   && arr[5]==9   && arr[6]==10  && arr[7]==11
 && arr[8]==23  && arr[9]==44  && arr[10]==100&& arr[11]==101
 && arr[12]==102&& arr[13]==103&& arr[14]==104&& arr[15]==105
 && arr[16]==106&& arr[17]==107&& arr[18]==108&& arr[19]==109)
  {
    return 42;
  }
  else
  {
    return -1;
  }
}
```

B.4 SHA-1 Test

```
int shaTest()
{
  SHA1Context sha;
  SHA1Reset(&sha);
  SHA1Input(&sha, (const unsigned char *) "abc", 3);
  SHA1Result(&sha);
  if(    sha.Message_Digest[0]==0xA9993E36
    && sha.Message_Digest[1]==0x4706816A
    && sha.Message_Digest[2]==0xBA3E2571
    && sha.Message_Digest[3]==0x7850C26C
    && sha.Message_Digest[4]==0x9CD0D89D )
  {
    return 42;
  }
  return -1;
}
```

B.5 CRC Test

```
// based on code from Michael Barr

unsigned int crcTable[256];

unsigned int crcSlow(unsigned char const message[], int nBytes)
{
    unsigned int  remainder = 0;
    int byte;
    unsigned char bit;
    for (byte = 0; byte < nBytes; ++byte)
```

```
    {
        remainder ^= (message[byte] << (32 - 8));
        for (bit = 8; bit > 0; --bit)
        {
            if (remainder & (1 << (32 - 1)))
            {
                remainder = (remainder << 1) ^ 0xF4ACFB13;
            }
            else
            {
                remainder = (remainder << 1);
            }
        }
    }
    return remainder;
}

void crcInit()
{
    unsigned int remainder;
    int dividend;
    unsigned char bit;
    for (dividend = 0; dividend < 256; ++dividend)
    {
        remainder = dividend << (32 - 8);
        for (bit = 8; bit > 0; --bit)
        {
            if (remainder & (1 << (32 - 1)))
            {
                remainder = (remainder << 1) ^ 0xF4ACFB13;
            }
            else
            {
                remainder = (remainder << 1);
            }
        }
        crcTable[dividend] = remainder;
    }
}

unsigned int crcFast(unsigned char const message[], int nBytes)
{
    unsigned char data;
    unsigned int remainder = 0;
    int byte;
    for (byte = 0; byte < nBytes; ++byte)
```

```
    {
        data = message[byte] ^ (remainder >> (32 - 8));
        remainder = crcTable[data] ^ (remainder << 8);
    }
    return (remainder);
}

int crcTest()
{
    crcInit();
    unsigned char crc_input[10];
    crc_input[0] = 'T';
    crc_input[1] = 'E';
    crc_input[2] = 'S';
    crc_input[3] = 'T';
    crc_input[4] = 'C';
    crc_input[5] = 'R';
    crc_input[6] = 'C';
    crc_input[7] = '3';
    crc_input[8] = '2';
    crc_input[9] = '!';

    if (crcSlow(crc_input,10) != crcFast(crc_input,10))
    {
      initPIO();
      return -4;
    }
    else
    {
      initPIO();
      return 42;
    }
}
```

B.6 March Test

```
//Based on code from Michael Barr
int memTestDevice()
{
    int baseAddress[100];
    int nBytes = 100;
    unsigned long offset;
    unsigned long nWords = nBytes / sizeof(int);
    int pattern;
    int antipattern;
```

```
    /* Fill memory with a known pattern */
    for (pattern = 1, offset = 0; offset < nWords; pattern++, offset++)
    {
        baseAddress[offset] = pattern;
    }
    /* Check each location and invert it for the second pass */
    for (pattern = 1, offset = 0; offset < nWords; pattern++, offset++)
    {
        if (baseAddress[offset] != pattern)
        {
            return -1;
        }
        antipattern = ~pattern;
        baseAddress[offset] = antipattern;
    }
    /* Check each location for the inverted pattern and zero it */
    for (pattern = 1, offset = 0; offset < nWords; pattern++, offset++)
    {
        antipattern = ~pattern;
        if (baseAddress[offset] != antipattern)
        {
            return -1;
        }
    }
    return 42;
}
```

Appendix C

List of Registers for Fault Injection

C.1 Plasma/MIPS

```
# FILE-REGISTER
#/tbench/u1_plasma/u1_cpu/u4_reg_bank/data_out1
#/tbench/u1_plasma/u1_cpu/u4_reg_bank/data_out2
#/tbench/u1_plasma/u1_cpu/u4_reg_bank/reg_target_out
# DECODING
#/tbench/u1_plasma/u1_cpu/b_bus
#/tbench/u1_plasma/u1_cpu/b_busd
# EXECUTION
#/tbench/u1_plasma/u1_cpu/opcode
# PROGRAM COUNTER
#/tbench/u1_plasma/u1_cpu/pc_current
#/tbench/u1_plasma/u1_cpu/u1_pc_next/pc_new
# ADRESSING
#/tbench/u1_plasma/ram_address
#/tbench/u1_plasma/u1_cpu/u2_mem_ctrl/address_in
# INTERNAL RAM
#/tbench/u1_plasma/ram_data_r
#/tbench/u1_plasma/ram_data_w
```

C.2 Leon3

```
# FILE-REGISTER
#/testbench/cpu/l3/cpu(0)/u0/rfo_fi.data1
# DECODING
#/testbench/cpu/l3/cpu(0)/u0/p0/iu0/r.d.inst(0)
# EXECUTION
#/testbench/cpu/l3/cpu(0)/u0/p0/iu0/r.e.op1
#/testbench/cpu/l3/cpu(0)/u0/p0/iu0/r.e.op2
```

```
# PROGRAM COUNTER
#/testbench/cpu/l3/cpu(0)/u0/p0/iu0/r.f.pc
# ADRESSING
#/testbench/cpu/l3/cpu(0)/u0/cmem0/crami.icramin.address
#/testbench/cpu/l3/cpu(0)/u0/cmem0/crami.dcramin.address
# INTERNAL RAM
#/testbench/cpu/l3/cpu(0)/u0/cmem0/cramo.icramo.data(0)
#/testbench/cpu/l3/cpu(0)/u0/cmem0/crami.dcramin.data(0)
```

Appendix D

Publication

The work presented in this thesis was published at the 8th IEEE International Design & Test Symposium 2013 in Marrakesh, Morocco. On the following pages, this chapter shows the poster presented at the conference and the published paper.

Verifying Generic IEC 61508 CPU Self-Tests with Fault Injection

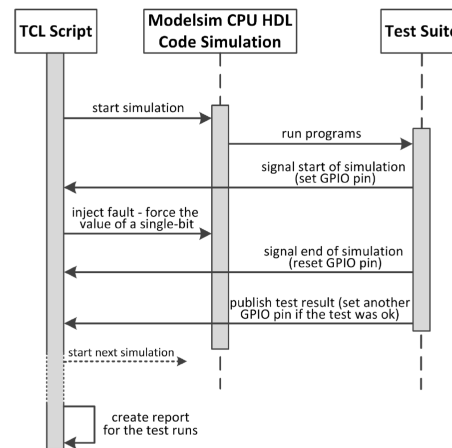
Christopher Preschern, Nermin Kajtazovic, Andrea Höller, Christian Steger, Christian Kreiner
Institute for Technical Informatics
Graz University of Technology

Email: christopher.preschern@tugraz.at, Nermin.kajtazovic@tugraz.at, andrea.höller@tugraz.at, steger@tugraz.at, christian.kreiner@tugraz.at

IEC 61508 CPU Test Requirements

Register, internal RAM
Coding and execution
Address calculation
Program counter
Stack pointer

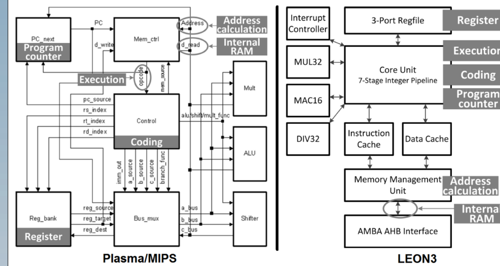
Fault Injection Framework



Generic Self-Tests

- ALU/Multiplier/Divider/Logic Test
- Quicksort Test
- CRC Test
- SHA-1 Test
- March Test

Target Registers



Detected Faults

		Reg., Int.RAM	Coding, Exec.	Address Calc.	Program Counter	Stack Pointer	DC
Generic CoreTests	Stuck-At	100.0%	99.5%	100.0%	100.0%	100.0%	**99.9%**
	Transient	94.2%	96.9%	94.6%	95.3%	100.0%	**96.2%**
Qsort Test	Stuck-At	84.8%	99.5%	81.7%	85.6%	100.0%	**90.3%**
	Transient	97.7%	93.8%	93.8%	100.0%	100.0%	**97.6%**
CRC Test	Stuck-At	100.0%	97.9%	100.0%	100.0%	100.0%	**99.6%**
	Transient	99.1%	100.0%	100.0%	100.0%	100.0%	**99.8%**
SHA-1 Test	Stuck-At	100.0%	100.0%	100.0%	100.0%	100.0%	**100.0%**
	Transient	99.2%	100.0%	97.5%	96.3%	100.0%	**98.6%**
March Test	Stuck-At	85.5%	92.2%	100.0%	81.9%	100.0%	**91.9%**
	Transient	97.7%	100.0%	100.0%	100.0%	100.0%	**99.6%**

Plasma/MIPS

		Reg., Int.RAM	Coding, Exec.	Address Calc.	Program Counter	Stack Pointer	DC
Generic CoreTests	Stuck-At	100.0%	100.0%	96.8%	100.0%	100.0%	**99.4%**
	Transient	87.7%	90.6%	86.0%	89.3%	100.0%	**90.7%**
Qsort Test	Stuck-At	95.3%	97.7%	100.0%	70.3%	100.0%	**92.7%**
	Transient	76.5%	96.9%	100.0%	96.9%	100.0%	**94.1%**
CRC Test	Stuck-At	93.8%	97.3%	100.0%	85.9%	100.0%	**95.4%**
	Transient	96.5%	93.8%	100.0%	93.8%	100.0%	**96.8%**
SHA-1 Test	Stuck-At	100.0%	100.0%	100.0%	82.8%	100.0%	**96.6%**
	Transient	96.9%	96.1%	96.7%	97.1%	100.0%	**95.1%**
March Test	Stuck-At	99.3%	99.1%	98.4%	70.3%	100.0%	**92.3%**
	Transient	100.0%	98.5%	100.0%	96.9%	100.0%	**99.1%**

LEON3

Selected References

[1] T. Tamandl, P. Preininger, and T. Novak, "Testing approach for online hardware self tests in embedded safety related systems," in IEEE Conference on Emerging Technologies and Factory Automation, 2007, pp. 1270–1277.
[2] M. Portela-Garcia, C. Lopez-Ongil, M. Garcia-Valderas, and L. Entrena, "Fault Injection in Modern Microprocessors Using On-Chip Debugging Infrastructures," IEEE Transactions on Dependable and Secure Computing, vol. 8, no. 2, pp. 308–314, 2011.
[3] M. Azkarate-askasua, X. Iturbe, I. Martinez, and R. Obermaisser, "FI4SoC : A Fault Injection Framework for Transient Fault Effects in Embedded MPSoCs," in Proceedings of the Ninth Workshop on Intelligent Solutions in Embedded Systems (WISES)2. IEEE, 2011. [4] G. Grießnig, R. Mader, C. Steger, and R. Weiß, "Fault Insertion Testing of a Novel CPLD-based Fail-Safe System," in Design, Automation & Test in Europe Conference & Exhibition (DATE 09). IEEE, 2009.
[4] J. Perez, M. Azkarate-askasua, and A. Perez, "Codesign and Simulated Fault Injection of Safety-Critical Embedded Systems Using SystemC," in European Dependable Computing Conference. IEEE, 2010, pp. 221–229.
[5] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, "Deterministic software-based self-testing of embedded processor cores," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition*. IEEE Comput. Soc, Aug. 2001, pp. 92–96.
[6] A. Krieg, C. Preschern, J. Grinschgl, C. Steger, C. Kreiner, R. Weiss, H. Bock, and J. Haid, "Power And Fault Emulation for Software Verification and System Stability Testing in Safety Critical Environments," IEEE Transactions on Industrial Informatics, vol. 9, no. 1, pp. 1199–1206, 2013
[7] N. Kranitis, G. Xenoulis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Application and analysis of rt-level software-based self-testing for embedded processor cores," in *International Test Conference, 2003. Proceedings. ITC 2003.* IEEE, 2003, pp. 431–440.
[8] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda, "Microprocessor Software-Based Self-Testing," IEEE Design & Test of Computers, vol. 27, no. 3, pp. 4–19, May 2010.

TU Graz
Graz University of Technology

Institute for Technical Informatics
www.ITI.TUGraz.at
Head: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Römer

Inffeldgasse 16/1
8010 Graz, Austria
Tel. +43(316) 873 6401
Fax +43(316) 873 6903

Verifying Generic IEC 61508 CPU Self-Tests with Fault Injection

Christopher Preschern, Nermin Kajtazovic, Andrea Höller, Christian Steger, Christian Kreiner
Institute for Technical Informatics, Graz University of Technology, Austria
Email: christopher.preschern@tugraz.at, nermin.kajtazovic@tugraz.at, andrea.hoeller@tugraz.at, steger@tugraz.at, christian.kreiner@tugraz.at

***Abstract*—In this paper we present generic CPU self-test programs and we check if the test programs conform to the IEC 61508 safety standard. We use processor architecture independent test programs to indirectly test the CPU components. We present a fault injection framework which we use to verify the fault detection ratio of the self-tests through simulation on a Plasma/MIPS and on a LEON3 processor.**

I. Introduction

Functional safety certification requires detailed analysis of a system about which faults can occur and which of them are hazardous. For complex hardware such as CPUs, self-tests can be executed to detect hazardous faults before they lead to a critical system failure. Most commercial CPUs are not delivered with such self-tests, which means that when using the CPU in the safety domain, such tests have to be developed individually. Also, these tests have to be verified which is usually done by injecting faults on the CPU to show that the tests can detect these faults. This can be a tedious task, especially if the test programs and the test verification framework have to be built from scratch.

To decrease the effort for developing and verifying CPU tests used in the safety domain, we present and verify a generic test suite which indirectly tests the CPU components and fault types described in the IEC 61508 safety standard. We argue for the universality of the test suite by showing for two different CPU architectures that the tests achieve a high fault detection ratio (diagnostic coverage). We set up a testbench to simulate the processor hardware description language (HDL) code and run the test programs on this simulation while injecting faults into the HDL code in order to obtain the test programs' diagnostic coverage.

II. IEC 61508 Safety Requirements for CPU Tests

Table I shows which components of the CPU according to IEC 61508 explicitly require testing and which types of faults have to be considered. To test the 5 required CPU components, we inject stuck-at and transient faults (soft-errors) into relevant registers. For the **Coding and execution** component, the standard requires to detect wrong or no execution. We simulate this by injecting faults into the opcode registers. With the amount of detected faults we calculate the diagnostic coverage according to the following equation:

$$DiagnosticCoverage(DC) = \frac{\lambda_D}{\lambda_{All}} = \frac{DetectedFaults}{InjectedFaults}$$

If we reach a diagnostic coverage of more than 90%, the test programs are capable to fulfill IEC 61508 SIL2 safety requirements.

CPU Component	Faults which have to be considered for the CPU if 90% diagnostic coverage has to be reached
Register, internal RAM	- Stuck-at for data and addresses - Diagnostic coverage fault model for data and addresses - Change of information caused by soft-errors
Coding and execution	- Wrong coding or no execution
Address calculation	- Stuck-at - Diagnostic coverage fault model - Change of information caused by soft-errors
Program counter	- Stuck-at - Diagnostic coverage fault model - Change of information caused by soft-errors
Stack pointer	- Stuck-at - Diagnostic coverage fault model - Change of information caused by soft-errors

TABLE I. Required CPU tests (IEC 61508-2 Table A.1 [1])

III. Test Setup

Most existing fault injection frameworks modify HDL code, for example to insert saboteurs, and emulate the processor source code on hardware. This requires some knowledge about the processor source code and about how these saboteurs work. Also, many of these fault injection frameworks have the disadvantage that they are not publicly available.

To address these problems, we designed a fault injection framework which does not require any modifications to the HDL code. We simulate the HDL code with Modelsim from Mentor Graphics and execute test programs on this simulation. For fault injection, Modelsim provides the *force* command to inject stuck-at (*force -freeze*) and transient (*force -deposit*) faults and Modelsim provides the *when* and *examine* commands to read register values. Our fault injection is coordinated by a TCL script. For stuck-at faults, the script forces a register value to 0 or 1 during a whole test-run. For transient faults, the script flips a register value at a random time during the test-run. A test-run is simulated for each combination of the test programs and the relevant registers of the 5 CPU components. More details and the full source code of the fault injection framework at available at [2] and [3].

Fault simulation in software is much slower than emulating the processor HDL code on a hardware. However, our software-based self-tests are not very complicated, thus a simulation-based approach is still feasible.

IV. Generic Self-Test

The proposed CPU test programs are written in C code. The reason for not writing the tests in assembler code (which is usually the case for the tests presented in literature) is that the tests should be executable on different CPU architectures. The drawback of having tests in C code is that the compilation process brings uncertainty of how the actual executed code looks like. However, with the evaluation we will show that for different CPUs this has little effect on the test results.

For our first set of self-test programs (*Generic Core-Tests*), we choose the tests presented by [4]. These CPU tests include generic tests for the ALU, the shifter unit, the division unit, and the multiplication unit. The tests compare results of diverse calculations or compare results to a pre-known reference value to detect faults.

For the next sets of test programs we use programs based on the MiBench test suite [5], which provides tests for different application domains such as automotive or security. We use quicksort, CRC32, and SHA-1 test programs. The quicksort test takes an integer array as input and sorts the array. A fault is detected if the resulting array is not in the correct order. The CRC32 test program uses two diverse CRC implementations and checks if the results are the same. The SHA-1 test program computes the SHA-1 hash value for a fixed input and detects a fault if the SHA-1 output differs from the expected result which is hard-coded in the program.

The last test program is a March RAM test. RAM tests have to be applied according to the IEC 61508 safety standard to detect faults in the RAM memory. The March RAM test writes specific data patterns to the memory and checks if they are the same when read. Of course this RAM test also indirectly tests some of the CPU components, because the CPU registers and operations are used as well. The test detects a fault if the data read from the memory differs from the data which was previously written.

V. Applying the Tests

A. Fault Injection Target Registers

We applied the fault injection tests to two different processors. We use the open source testbeds for the Plasma/MIPS processor from *opencores.org* and for the LEON3 processor from *www.gaisler.com*. The most time-consuming task was to find appropriate registers where we want to inject the faults. For the tests of the *Register, internal RAM* CPU component, we injected faults on the bus signals and on the buffer registers in the unit accessing the RAM. We also injected faults into the internal register bank of the CPU. For the *Coding and execution*, the *Address calculation*, and the *Program counter* components, we injected the faults into the corresponding part of the CPU pipeline. For the *Stack pointer* component, we did not locate a specific register in the hardware, but injected the faults with a software-level assembler-routine. The full list of registers and the whole test setup can be found at [2].

B. Diagnostic Coverage

Table II shows the fault injection test results for the Plasma/MIPS processor and Table III shows the results for the LEON3 processor. We can see that all of the test programs reach a diagnostic coverage of more than 90% which is required for SIL2 safety certification. However, one has to take care when intepreting the results that we did not inject faults into all the CPU registers, but just the registers which are explicitly mentioned by the IEC 61508 standard. The standard takes these 5 CPU core components as a basis for testing, but does not say that it is sufficient to just test these registers. This means that the presented safety tests can be taken as a basis for safety testing, but still, depending on the specific processor, additional tests might be required. Still, it is very interesting to see that the required diagnostic coverage of the CPU components can be achieved by test programs which are not specifically tailored to test the CPU.

		Reg., Int.RAM	*Coding, Exec.*	*Address Calc.*	*Program Counter*	*Stack Pointer*	***DC***
Generic CoreTests	Stuck-At	100.0%	99.5%	100.0%	100.0%	100.0%	**99.9%**
	Transient	94.2%	96.9%	94.6%	95.3%	100.0%	**96.2%**
Qsort Test	Stuck-At	84.8%	99.5%	81.7%	85.6%	100.0%	**90.3%**
	Transient	97.7%	93.8%	93.8%	100.0%	100.0%	**97.6%**
CRC Test	Stuck-At	100.0%	97.9%	100.0%	100.0%	100.0%	**99.6%**
	Transient	99.1%	100.0%	100.0%	100.0%	100.0%	**99.8%**
SHA-1 Test	Stuck-At	100.0%	100.0%	100.0%	100.0%	100.0%	**100.0%**
	Transient	99.2%	100.0%	97.5%	96.3%	100.0%	**98.6%**
March Test	Stuck-At	85.5%	92.2%	100.0%	81.9%	100.0%	**91.9%**
	Transient	97.7%	100.0%	100.0%	100.0%	100.0%	**99.6%**

TABLE II. Plasma/MIPS diagnostic coverage

		Reg., Int.RAM	*Coding, Exec.*	*Address Calc.*	*Program Counter*	*Stack Pointer*	***DC***
Generic CoreTests	Stuck-At	100.0%	100.0%	96.8%	100.0%	100.0%	**99.4%**
	Transient	87.7%	90.6%	86.0%	89.3%	100.0%	**90.7%**
Qsort Test	Stuck-At	95.3%	97.7%	100.0%	70.3%	100.0%	**92.7%**
	Transient	76.5%	96.9%	100.0%	96.9%	100.0%	**94.1%**
CRC Test	Stuck-At	93.8%	97.3%	100.0%	85.9%	100.0%	**95.4%**
	Transient	96.5%	93.8%	100.0%	93.8%	100.0%	**96.8%**
SHA-1 Test	Stuck-At	100.0%	100.0%	100.0%	82.8%	100.0%	**96.6%**
	Transient	96.9%	96.1%	96.7%	97.1%	100.0%	**95.1%**
March Test	Stuck-At	99.3%	99.1%	98.4%	70.3%	100.0%	**92.3%**
	Transient	100.0%	98.5%	100.0%	96.9%	100.0%	**99.1%**

TABLE III. LEON3 diagnostic coverage

VI. Conclusion

With this paper we investigated indirect testing of CPU components and we provide a set of generic CPU self-tests and a testing framework at [2]. Our test results show that the test programs achieve a diagnostic coverage of more than 90% which is required for SIL2 safety certification. Some tests were not designed for CPU testing, but were rather simple programs which can commonly be found in embedded applications. The results show that the CPU components mentioned in the IEC 61508 standard can indirectly be tested by these commonly used programs like a CRC calculation or a RAM test.

We think that the presented self-tests provide a good start to develop safety-tests tailored for a specific CPU architecture. With the publicly available source code and the publicly available fault injection framework, it is easy to verify these customized tests without too much effort. We hope that the provided tests will successfully be used on other CPUs to further show that these tests are generic and also achieve a high diagnostic coverage for other CPU architectures.

References

[1] International Electrotechnical Commission, “IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems,” 2010.

[2] Christopher Preschern, “Online Repository of the presented safety test framework: http://sourceforge.net/p/safetytests/ .”

[3] Christopher Preschern , “Verifying Generic CPU Safety-Tests with Fault Injection,” Master’s thesis, Graz University of Technology, Institute for Technical Informatics, 2013.

[4] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, “Deterministic software-based self-testing of embedded processor cores,” in *Design, Automation and Test in Europe*. IEEE, 2001.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” in *4th Annual Workshop on Workload Characterization*. IEEE, 2001.

Bibliography

[ABB10] ABB. Safety and Functional Safety - A General Guide, Technical Report, 2010.

[ACC+93] Jean Arlat, Alain Costes, Yves Crouzet, Jean claude Laprie, and David Powell. Fault Injection and Dependability Evaluation of Fault-tolerant Systems. *IEEE Transactions on Computers*, 42:913–923, 1993.

[Aer13] Aeroflex Gaisler. GRLIB IP Core User's Manual, 2013.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[AT78] J.A. Abraham and S.M. Thatte. Efficient Algorithms for Testing Semiconductor Random-Access Memories. *IEEE Transactions on Computers*, C-27, 6, 1978.

[B0̈7] J. Börcsök. *Functional Safety - Basic Principles of Safety-related Systems*. Hüthig Verlag, 2007.

[Bao03] George Bao. Challenges in Low Cost Test Approach for ARM9 Core Based Mixed-Signal SoC DragonBall -MX1. In *International Test Conference 2003*, 2003.

[Bar99] Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly Media, 1999.

[BBCP00] Andrea Baldini, Alfredo Benso, Silvia Chiusano, and Paolo Prinetto. BOND: An Interposition Agents Based Fault Injector for Windows NT. In *15th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*. IEEE, 2000.

[BP03] Alfredo Benso and Paolo Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Springer, 2003.

[BPRR98] A. Benso, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. EXFI: A low-cost Fault Injection System for Embedded Microprocessor-based Boards. *ACM Trans. Des. Autom. Electron. Syst.*, 3(4):626–634, 1998.

[CD01] Li Chen and Sujit Dey. Software-Based Self-Testing Methodology for Processor Cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20:369–380, 2001.

[CDS+00] Li Chen, Sujit Dey, Pablo Sanchez, Krishna Sekar, and Ying Cheng. Embedded Hardware and Software Self-Testing Methodologies for Processor Cores. In *Proceedings of the 37th conference on Design automation - DAC '00*, pages 625–630. ACM Press, 2000.

[Cho96] G.S Choi. FOCUS: An Experimental Environment for Fault Sensitivity Analysis. *IEEE Transactions on Computers*, 41(12):1515–1526, 1996.

[CMCS04] Diamantino Costa, Henrique Madeira, Joao Carreira, and JoaoGabriel Silva. Xception: A Software Implemented Fault Injection Tool. In *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, pages 125–139. Springer US, 2004.

[CRP+96] Hungse Cha, Elizabeth M. Rudnick, Janak H. Patel, Ravishankar K. Iyer, and Gwan S. Choi. A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults. *IEEE Transactions on Computers*, 4(11):1248–1256, 1996.

[Dey02] S. Dey. Software-Based Diagnosis for Processors. In *Proceedings 2002 Design Automation Conference*, pages 259–262. ACM, 2002.

[DPS11] S. Di Carlo, Paolo Prinetto, and Alessandro Savino. Software-Based Self-Test of Set-Associative Cache Memories. *IEEE Transactions on Computers*, 60(7):1030–1044, 2011.

[Eri09] Stein Owe Erikson. *Low Power Microcontroller Core*. PhD thesis, Norwegian University of Science and Technology, 2009.

[FSK98] P. Folkesson, S. Svensson, and J Karlsson. A Comparison of Simulation Based and Scan Chain Implemented Fault Injection. In *28th International Symposium on Fault-Tolerant Computing*. IEEE, 1998.

[Fuc96] Emmerich Fuchs. An Evaluation of the Error Detection Mechanisms in MARS using Software-Implemented Fault Injection. In *2nd European Dependable Computing Conference*. Springer, 1996.

[GBGG04] Daniel Gil, JuanCarlos Baraza, Joaqun Gracia, and PedroJoaqun Gil. VHDL Simulation-Based Fault Injection Techniques. In *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, pages 159–176. Springer US, 2004.

[GBS04] Pedro Gil, Sara Blanc, and JuanJos Serrano. Pin-level hardware fault injection techniques. In *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, volume 23 of *Frontiers in Electronic Testing*, pages 63–79. Springer US, 2004.

[GOR89] Ulf Gunneflo, Joakim Ohlsson, and Marcus Rimen. A study of the effects of transient fault injection into a 32-bit risc with built-in watchdog. In *Proceedings of the Annual International Symposium on Fault-Tolerant Computing*. IEEE, 1989.

[GPH+08] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi. Systematic Software-Based Self-Test for Pipelined Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16:1441–1453, 2008.

[GPZ04] Dimitris Gizopoulos, A. Paschalis, and Yervant Zorian. *Embedded Processor-Based Self-Test*. Springer, 2004.

[GRE+01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *4th Annual Workshop on Workload Characterization*. IEEE, 2001.

[Han95] S. Han. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems. In *Computer Performance and Dependability Symposium*. IEEE, 1995.

[HL06] Hung-Min Hsu and Kuen-Jong Lee. *Software-Based Test Methodology for Fully Associative Data Cache*. PhD thesis, National Cheng Kung University, 2006.

[Int92] International Electrotechnical Commission. IEC 65A (Secretariat) 122, Software for Computers in the Application of Industrial Safety-Related Systems, 1992.

[Int10] International Electrotechnical Commission. IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems, 2010.

[KGPZ02] N. Kranitis, D. Gizopoulos, A. Paschalis, and Y. Zorian. Instruction-Based Self-Testing of Processor Cores. In *Proceedings 20th IEEE VLSI Test Symposium (VTS 2002)*, pages 223–228. IEEE Comput. Soc, 2002.

[KGS+12] Armin Krieg, Johannes Grinschgl, Christian Steger, Reinhold Weiss, Holger Bock, and Josef Haid. POWER-MODES: POWer-EmulatoR- and MOdel-Based DEpendability and Security Evaluations. *TRETS*, 5(4):19, 2012.

[KPG+13] A. Krieg, C. Preschern, J. Grinschgl, C. Steger, C. Kreiner, R. Weiss, H. Bock, and J. Haid. Power And Fault Emulation for Software Verification and System Stability Testing in Safety Critical Environments. *IEEE Transactions on Industrial Informatics*, 9(2):1199–1206, 2013.

[KPGZ02] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian. Effective Software Self-Test Methodology for Processor Cores. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 592–597. IEEE Comput. Soc, 2002.

[KXP+03] N. Kranitis, G. Xenoulis, A. Paschalis, D. Gizopoulos, and Y. Zorian. Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores. In *International Test Conference (ITC2003)*, volume 1, pages 431–440. IEEE, 2003.

[LPP10] P. Low, R. Pabst, and E. Petry. *Funktionale Sicherheit in der Praxis - Anwendung von DIN EN 61508 und ISO/DIS 26262 bei der Entwicklung von Serienprodukten*. dpunkt.verlag, 2010.

[LTW05] Jin-Fu Li, Tsu-Wei Tseng, and Chin-Long Wey. An Efficient Transparent Test Scheme for Embedded Word-Oriented Memories. In *Design, Automation and Test in Europe Conference and Exhibition (DATE05)*, 2005.

[MH97] Michael McLennan and Mark Harrison. *Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk*. Addison-Wesley, 1997.

[MMS94] Henrique Madeira, Francisco Moreira, and Joao Gabriel Silva. RIFLE: A General Purpose Pin-level Fault Injector. In *First European Dependable Computing Conference*. Springer, 1994.

[Ope13] Opencores, Plasma/MIPS. http://opencores.org/project,plasma, accessed, September 2013.

[PG05] A. Paschalis and D. Gizopoulos. Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 578–583. IEEE Comput. Soc, 2005.

[PGK+01] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian. Deterministic Software-Based Self-Testing of Embedded Processor Cores. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition*, pages 92–96. IEEE Comput. Soc, 2001.

[PGSR10] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. Microprocessor Software-Based Self-Testing. *IEEE Design & Test of Computers*, 27(3):4–19, 2010.

[PKK13] Christopher Preschern, Nermin Kajtazovic, and Christian Kreiner. Catalog of Safety Tactics in the light of the IEC 61508 Safety Lifecycle. In *VikingPLoP 2013*, 2013.

[RS04] Chantal Robach and Mathieu Scholive. Simulation-Based Fault Injection and Testing Unsing the Mutation Technique. In *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, pages 195–215. Springer US, 2004.

[RSFA99] M. Rodrguez, F. Salles, J.-C. Fabre, and J. Arlat. MAFALDA: Microkernel Assessment by Fault Injection and Design Aid. In *3nd European Dependable Computing Conference*. Springer, 1999.

[SBS08] André Borin Soares, Alexsandro Cristovão Bonatto, and Altamiro Amadeu Susin. A New March Sequence to fit DDR SDRAM Test in Burst Mode. In *Proceedings of the twenty-first annual symposium on Integrated circuits and system design*. ACM Press, 2008.

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS/Performance09*, 2009.

[SS01] David J. Smith and Kenneth G. L. Simpson. *Functional Safety - A Straightforward Guide to Applying IEC 61508 and Related Standards*. Elsevier, 2001.

[STB97] V. Sieh, O. Tschache, and F. Balbach. VERIFY: Evaluation of Reliability using VHDL-Models with Embedded Fault Descriptions. In *27th International Symposium on Fault-Tolerant Computing*. IEEE, 1997.

[SVS+88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. FIAT - Fault Injection Based Automated Testing Environment. In *18th International Symposium on Fault-Tolerant Computing*. IEEE, 1988.

[TB06] Matthieu Tuna and Mounir Benabdenbi. Software-Based Self-Test of Register Files in RISC Processor Cores using March Algorithms . In *LATW IEEE Latin-American Test Workshop digest of papers*, 2006.

[TJD10] A Tsertov, A. Jutman, and S. Devadze. Testing Beyond the SoCs in a Lego Style. In *East-West Design & Test Symposium (EWDTS)*, pages 334–338. IEEE, 2010.

[TP07] T. Tamandl and P. Preininger. Online Self Tests for Microcontrollers in Safety Related Systems. In *2007 5th IEEE International Conference on Industrial Informatics*, pages 137–142. IEEE, 2007.

[TPN07] Thomas Tamandl, Peter Preininger, and Thomas Novak. Testing Approach for Online Hardware Self Tests in Embedded Safety Related Systems. In *IEEE Conference on Emerging Technologies and Factory Automation*, pages 1270–1277, 2007.

[vdM95] M.J.P. van der Meulen. *Definitions for Hardware/Software Reliability Engineers*. Simtech b.v, 1995.

[Vis91] Gangaikond S. Visweswaran. The Effects of Transistor Source-to-Gate Bridging Faults in Complex CMOS Gates. *IEEE Journal of Solid-State Circuits*, 26(6):893–896, 1991.