Gerhard Schönfelder, BSc

FIES: A Fault Injection Framework for the Evaluation of Self-Tests

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

BSc. Dipl.-Ing., Andrea Höller

Institute for Technical Informatics

Dipl.-Ing. Dr. techn., Christian Kreiner

Graz, January 2015

Kurzfassung

Für die Entwicklung und Prüfung von Software-basierenden Selbsttests ist ein geeignetes Fault Injection Framework unerlässlich. Dies ist besonders von Bedeutung, wenn die Tests Anforderungen von Sicherheitsstandards einhalten müssen und kommerzielle Prozessoren ohne öffentlich zugänglichen Quellcode verwendet werden. Aufgrund von kostengünstiger Hardware werden heutzutage immer öfters gewöhnliche Allzweck-(Mikro-)Prozessoren in Sicherheitsbereichen, zum Beispiel für Steuerungs- und Regelungsaufgaben, verwendet. Das Problem dieser Prozessoren ist, dass sie nicht für den Sicherheitsbereich entwickelt worden sind und somit die korrekte Funktionalität der einzelnen Hardware-Komponenten durch die Verwendung von geeigneten Maßnahmen oder Methoden gewährleistet werden muss (Anforderung des Sicherheitsstandards). SBSTs bieten eine ansprechende Lösung um diese funktionale Sicherheit zu erreichen. Jedoch weisen die meisten Techniken, die vom Sicherheitsstandard vorgeschlagen werden, eine hohe Laufzeit bzw. Komplexität auf und können somit nicht auf moderne Systeme bzw. Prozessoren angewendet werden.

Diese Arbeit beschäftigt sich damit, bessere und schnellere SBST-Methoden für die Hauptkomponenten der CPU (ALU, Register, Schieberegister, Addierer, Multiplizierer, etc.) sowie für den Arbeitsspeicher (RAM) zu finden und miteinander, anhand ihrer Fehlerabdeckungsraten, zu vergleichen. Um zu garantieren, dass eine ausreichend-hohe Fehlerabdeckungsraten erreicht wird, um zumindest eine Sicherheitsanforderungsstufe von 3 (SIL 3) gemäß IEC 61508 zu erreichen, wird ein spezielles Fault Injection Framework in dieser Arbeit entwickelt. Dieses Framework ermöglicht es, jegliche Art von Fehlern, die von dem IEC 61508 Sicherheitsstandard gefordert werden, zu jeder Zeit für einen kommerziellen ARM9-Prozessor, bei dem der Quellcode nicht öffentlich zugänglich ist, zu simulieren.

Abstract

An appropriate fault injection framework for a closed-source processor is indispensable for the development and the verification of Software-Based Self-Tests (SBSTs). This is especially important to be compliant with the requirements of the IEC 61508 safety standard. Nowadays, the usage of general-purpose (micro-)processors for safety-critical systems, especially in the control and automation sector, have become a common practice due to their low hardware costs. The problem is that these general-purpose processors are not developed for safety issues and hence the correct functioning of the hardware has to be ensured by appropriate techniques. SBSTs are an attractive solution to achieve functional safety, but the proposed techniques by the IEC 61508 safety standard are outdated and not applicable to modern processor-systems, because of their long execution times.

In this master thesis, faster and better SBST techniques for the CPU-core elements (ALU, register, shifter, adder, multiplier etc.) and the main memory (RAM) are compared and validated against their fault coverages. In order to guarantee a sufficiently high fault detection rate to be compliant with at least safety integrity level (SIL) 3 according to the IEC 61508, a special fault injection framework is developed. This framework is able to simulate every by the IEC 61508 safety standard required fault at any time for a closed-source ARM9-processor.

AFFIDAVIT

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. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

....
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 parents and my whole family for enabling my education in the last years and my girlfriend Sandra for her patience and understanding during my master thesis.

Special thanks to my supervisors Andrea Höller and Christian Kreiner, who helped me with technical and organizational issues as well as with many good advices during my master thesis. Furthermore, I want to thank all my friends who helped me in any manner during my whole study.

Graz, January 2015 Gerhard Schönfelder

Contents

1 Introduction **9**
- 1.1 Motivation . . . 9
- 1.2 Goals . . . 10
- 1.3 Outline . . . 11

2 Technical Background and Related Work **12**
- 2.1 Safety-critical Systems . . . 12
  - 2.1.1 Introduction to Safety-Critical Systems and Safety Standards . . . 12
  - 2.1.2 Fault, Error, Failure . . . 14
  - 2.1.3 IEC 61508 . . . 15
  - 2.1.4 Safety Integrity Level . . . 15
  - 2.1.5 Techniques and Measurements . . . 16
- 2.2 ARM926EJ-S Core Module . . . 17
  - 2.2.1 Generic Architecture Overview . . . 17
  - 2.2.2 Memory Management Unit . . . 17
  - 2.2.3 Register . . . 18
  - 2.2.4 ARM9EJ-S CPU . . . 18
- 2.3 Fault Injection . . . 18
  - 2.3.1 Objectives of Fault Injection . . . 19
  - 2.3.2 Overview of the Fault Injection Environment . . . 19
  - 2.3.3 Hardware-Based Fault Injection . . . 20
  - 2.3.4 Software-Based Fault Injection . . . 21
  - 2.3.5 Simulation-Based Fault Injection . . . 23
  - 2.3.6 Emulation-Based Fault Injection . . . 23
  - 2.3.7 Hybrid Fault Injection . . . 24
- 2.4 Software-Based Self-Test . . . 25
  - 2.4.1 Fault Classification . . . 26
  - 2.4.2 Self-Tests and IEC 61508 . . . 26
  - 2.4.3 CPU-core Tests . . . 30
  - 2.4.4 RAM Tests . . . 32
- 2.5 Related Work . . . 37
  - 2.5.1 QEMU-based Fault Injection Framework . . . 37
  - 2.5.2 Software-Based Self-Tests (SBSTs) for CPU . . . 38
  - 2.5.3 Software-Based Self-Tests (SBSTs) for RAM . . . 39

3 Concept and Design **41**
3.1 System Requirements . . . 41
3.2 Fault Injection Framework based on QEMU . . . 41
3.2.1 QEMU Overview . . . 42
3.2.2 Details about QEMU's Translation Process . . . 43
3.2.3 Design of Fault Injection Components . . . 45
3.3 Software-Based Self-Tests (SBSTs) . . . 52
3.3.1 SBSTs for CPU-core Elements . . . 53
3.3.2 SBSTs for RAMs . . . 57

4 Implementation **63**
4.1 Fault Injection Framework . . . 63
4.1.1 Implementation Overview of QEMU . . . 63
4.1.2 Implementation of the Fault Injection Components in QEMU . . . 65
4.1.3 Compiling and Executing FIES . . . 76
4.2 Software-Based Self-Tests (SBSTs) . . . 78
4.2.1 Overview of SafeRTOS . . . 78
4.2.2 Implementation of SBSTs in SafeRTOS . . . 81

5 Results **84**
5.1 Test Results . . . 84
5.1.1 Memory Test Results . . . 84
5.1.2 CPU-core and Register Test Results . . . 87
5.2 Simulation Time . . . 89
5.3 Validation of the System Requirements . . . 90

6 Conclusion **92**
6.1 Further Work . . . 93

A FIES Source Code **94**

B SBST Source Code **98**
B.1 Pseudo Transparent March C- Test . . . 99
B.2 Transparent March SS Test with Multiple Input Signature Register . . . 101
B.3 TWM-TA-modified Abraham Test . . . 102
B.4 CPU-core Tests . . . 105
B.5 Register Tests . . . 107

Literaturverzeichnis **109**

List of Figures

2.1 Overview of safety standards 13
2.2 Fault-error-failure cascade 14
2.3 Basic components of a fault injection environment 20

3.1 Simulation model of QEMU 44
3.2 Structure of the fault injection framework 46
3.3 Monitor output of the command *info registers* in QEMU 48
3.4 Visualization of the fault injection process 52
3.5 Flow-chart of multiplier, adder, divider and shifter test 55
3.6 Flow-chart of ALU and condition code flags test 56
3.7 Flow-chart of Walking Bit test for testing the registers 57

4.1 Execution tracing of the QEMU-framework 66
4.2 Dependencies of FIES 67
4.3 Sequence diagram of an access-triggered memory fault injection 68
4.4 The two consoles of FIES 78
4.5 Context diagram of SafeRTOS 80

List of Tables

2.1 Safety Integrity Levels - continuous mode . . . 16
2.2 Functional safety assessment . . . 16
2.3 Examples of hardware-based fault injection approaches . . . 21
2.4 Examples of software-based fault injection approaches . . . 22
2.5 Examples of simulation-based fault injection approaches . . . 23
2.6 Benefits and drawbacks of fault injection techniques . . . 24
2.7 Maximal SIL for a given SFF depending on HFT . . . 26
2.8 CPU fault sources defined by IEC 61508 . . . 28
2.9 IEC 61508 techniques/measures for testing CPU-core . . . 29
2.10 Variable memory fault sources defined by IEC 61508 . . . 29
2.11 IEC 61508 techniques/measures for testing variable memories . . . 30
2.12 List of single-cell static FFMs with according FPs . . . 33
2.13 List of single-cell dynamic FFMs with according FPs . . . 34
2.14 List of two-cell static FFMs with according FPs . . . 34
2.15 March memory test notation . . . 36

3.1 QEMU binary translation with TCG-micro-ops . . . 45
3.2 Realization of RAM fault models . . . 50
3.3 Realization of CPU fault models . . . 51
3.4 Individual and overall Safe Failure Fraction for the CPU-core . . . 57
3.5 Description of memory testing algorithms . . . 58
3.6 Comparison of fault coverages for different memory tests and FFMs . . . 58
3.7 Data pattern for detecting intra-word CFid and CFdst in a 16-bit memory 59
3.8 Data pattern for detecting intra-word CFsts in a 16-bit memory . . . 60
3.9 Operation sequences for detecting intra-word CFdst in a 16-bit memory . . 60
3.10 Description of transparent memory testing algorithms . . . 61
3.11 Description of transparent symmetric memory testing algorithms . . . 62

5.1 Comparison of test results for memory self-tests . . . 86
5.2 Comparison of test results for register self-tests . . . 87
5.3 Achieved diagnostic coverages for CPU-core elements . . . 88
5.4 Results of SBSTs for CPU-core elements applied to the memory . . . 89
5.5 Simulation times for a fault injection experiment . . . 90

Chapter 1

Introduction

The compliance with safety standards is a mandatory step for the development of safety-critical systems. In the case of (micro-) processor-based safety critical systems it is important to guarantee the correct functionality of the hardware parts. For this purpose many self-tests have been developed. These self-tests have to been evaluated against the requirements of the safety standard. Therefore so-called fault injection methods have been introduced. The aim of this thesis is to develop a suitable fault-injection framework for an ARM9 core on the one hand and to present self-tests and compare them against the requirements of the IEC-61508 safety standard on the other hand.

1.1 Motivation

Nowadays, the usage of microprocessors for safety-critical systems in the control and automation sector has become indispensable. A malfunction of a safety-critical system can endanger human life and can cause the death of humans in the worst case. The importance to comply with safety standards can be shown using the example of the Therac-25 radiation therapy system, which caused the death of humans in six known cases by an accidental massive radiation overdoses caused by a software error [Pul01]. Hence, different standardization institutes like the International Electrotechnical Commission (IEC) defined requirements, which have to be fulfilled by safety-critical systems. These requirements are specified in safety standards. One common safety standard is for example the IEC 61508, which provides four Safety Integrity Levels (SIL). The highest safety level (SIL-4) defines that the probability of a failure per hour (PFH) has to be smaller than 10^{-8}. This means that for a period of 100 years, the PFH is still under 1% (exact value: 0.8%). In order to achieve such a SIL, it is important to guarantee that every part of the hardware works correctly. For this purpose so-called self-tests have been developed, which test the hardware according to the requirements specified in IEC 61508. To fulfill these requirements for a specified SIL, a certain percentage of hardware faults (test coverage) has to be detected.

These self-tests could be realized in hardware or software. One big advantages of Software-Based Self-Tests (SBSTs) is that they are cheaper and easier to implement than Hardware-Based Self-Tests.

Furthermore, SBSTs have to detect permanent faults like stuck-at-errors caused by long-term damage of the hardware and transient faults like bit-flips that can be caused

for example by environmental radiation. But the verification of these SBSTs against the requirements of the IEC 61508 (test coverage) with a sufficiently high fault detection rate is challenging, because transient faults occur very rarely and are hard to observe during the regular operation of the system. Therefore, a fault-injection (FI) is needed, which is able to simulate these faults at every time.

A possible approach to inject faults, is to use a Hardware Description Language (HDL) code to simulate the target system (processor). But in our case, it is not possible to use these approaches, because the used CPU core (ARM9) is not an open-source project and hence the needed HDL code is not public available.

In order to summarize the previous section following main questions have to be answered in this thesis:

- Is a suitable FI method available, which is able to simulate an appropriate fault model at every time?
- Is this FI method able to emulate an ARM9 processor?
- Which SBST method can hold the required test coverage for a specific hardware part (at least a test coverage of 90%)?
- Does the SBST implementation fulfill the safety standards (SIL-3)?

1.2 Goals

The goal of this work is to satisfy the following requirements:

SBSTs for ARM9 processors: A software has to be provided, which tests if the main parts of the CPU (ALU, registers, shifter, adder and multiplier) and RAM work correctly.

Verification of the SBSTs: In order to verify that the SBSTs satisfy the required test coverages, a special injection framework is needed, which is able to simulate an appropriate fault model (permanent and transient) at every time for the closed-source ARM9 processor.

IEC 61508 compliant SBSTs: The developed test-methods should satisfy the required IEC 61508 safety standard at least at SIL-3.

Fully-automated simulation: The self-tests and the FI should run without human interaction in order to reduce the run-time of simulation and testing.

Easy-to-use: The choice of faults as well as the time of injection and other relevant parameters should be easy to set up.

1.3 Outline

This thesis is structured as follows:

Chapter 2 gives the technical background about safety-critical systems and safety standards with focus on the IEC 61508 standard. It describes the relevant parts of an ARM9-processor for the IEC 61508 and gives a short architecture overview of the used processor. This chapter covers also the different FI methods and discusses their advantages and disadvantages. It presents the SBST methods for the different hardware parts in Section 2.4. Furthermore, this chapter contains the current available approaches and theories about FI-methods and SBSTs in the related work section.

Chapter 3 describes the details about the concept and the design for the FI framework and the different SBSTs. It delivers insights into the chosen FI approach including the used technologies with their advantages and disadvantages. It gives an overview about the used QEMU-emulator and explains how a fault injection can be realized by using QEMU. Furthermore, this chapter describes the chosen SBST for every hardware part in detail and shows if the SBST can deliver the required test coverage.

In **Chapter 4** the implementation details are presented. This includes a listing of the QEMU source files, where changes are made to achieve the FI. It also describes the different SBST implementations as well as the used programming languages and toolchains. Furthermore, a short introduction to the provided system framework is given in this chapter.

Chapter 5 presents the result of this work. It shows the usage of the FI framework and verifies the detection rates of the different SBSTs. Furthermore an evaluation of the performance of the FI framework is given in this chapter and the compliance with the requirements is verified and discussed.

Chapter 6 summarizes the thesis and gives possible suggestions for the improvement of this work.

Chapter 2

Technical Background and Related Work

This chapter covers the basic background knowledge about safety-critical systems and safety standards, ARM9-core modules, fault-injection techniques, software-based self-tests and presents the current existing approaches in the related work section.

2.1 Safety-critical Systems

This section gives an overview about the safety-critical systems, defines some basic keywords, explains the structure of the IEC-61508 safety standard and shows some techniques and measurements for software-based self-tests.

2.1.1 Introduction to Safety-Critical Systems and Safety Standards

Safety-Critical Systems (SCS) are systems which have influence on the environment indirectly or directly if they malfunction. A malfunction of a SCS can result in significant damage of properties or the environment and can also endanger human lifes and result in loss of life in the worst case. Commonly known cases of some malfunctions of SCS are listed in [Pul01].

There are many different definitions available of the term SCS in the literature. All in all they have the same statement in common. As said in [Kni02], a system becomes safety critical if the consequences of a failure lead to an unacceptable risk. Hence a computer-based system is not safety-critical by itself without considering the environment. The traditional application areas of such SCS are for example computer-based control and automation systems in railways, airplanes, nuclear plants etc. Nowadays, the humankind moves to a situation in which computers are pervasive and so a further area of SCS arises, the so-called non-traditional systems. Some examples for that application areas are systems in transportation control, banking and finance, electricity generation and distribution, telecommunication and the management of water. The malfunction of such a system would not endanger human life directly, but can lead to economic loss or loss of services, which consequences can be serious. For example, a malfunction of the telecommunication network at the right time, makes it impossible for a person to dial the emergency call, which can result in serious injuries or death.

These above-named reasons indicate that a computer-based system has to work dependable. The issue is that the current chip sizes of integrated circuits gets smaller and smaller and hence transient faults caused by environmental radiation occur more and more frequently, which can lead to unpredictable behavior of the system. In this context the term *functional safety* has an important meaning, which is defined as follows:

> *„Functional Safety is the part of the overall safety of a system or piece of equipment that depends on the system or equipment operating correctly in response to its inputs, including the safe management of likely operator errors, hardware and software failures and environmental changes. It is an additional step beyond the traditional product safety assessment and tackles our ever increasingly complex world of interoperating technologies and the hazards they cause [TUE14b]. “*

There are devices in several domains where functional safety is indispensable and hence there is an obligation by law to fulfill requirements for ensuring functional safety. These requirements are specified in safety standards, which are defined by standardization institutes. There are different national and European standardization institutes, but the most important ones are the international standardization institutes like the ISO (International Organization for Standardization) or the IEC (International Electrotechnical Commission). Figure 2.1 shows that every industrial domain has its own specialized safety standard like the IEC 62061, which gives requirements related to the machinery domain. These more specialized safety standards are derived and adapted from the IEC 61508 safety standard. The IEC 61508 is one of the most important generic safety standard and due to this fact, the IEC 61508 is often called „The mother standard for functional safety.“

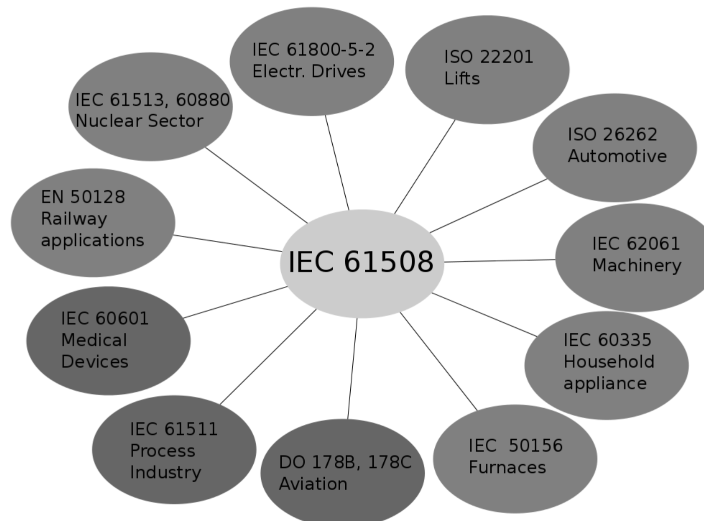


Figure 2.1: Overview of safety standards. Every industrial domain has its own safety standard, which is derived and adapted from the IEC 61508 safety standard. The IEC-61508 is a generic safety standard which is called „The mother standard for functional safety.“ This image is adapted from [TUE14a].

2.1.2 Fault, Error, Failure

This section defines the keywords fault, error and failure and explains the difference between these words, which is important for understanding the following sections. In the German language all three words have the same translation, but in the IEC 61508 every word has a different meaning.

A fault has the following definition in the IEC 61508: *„An abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function [IEC10].“* A fault is a cause, which can rise a system breakdown, but not every fault leads to a fail of a system. It depends on the systems tolerance, the safety management and if the part, where the fault occurs, is currently active or processed. An example of a fault can be a software bug or a permanent or temporary damaged hardware part.

The definition of an error is: *„A discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition [IEC10].“* This means that an error is a situation where a fault becomes apparent. An example is, if the CPU accesses a memory cell where a bit is permanently stuck at a different logic level as expected from the system.

If a SCS is not able to handle the error and to get the system in a safe state, an error becomes a failure. A failure is defined in the IEC 61508 as: *„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 [IEC10].“* A failure already has a negative influence on the environment and is externally observable.

Figure 2.2 visualizes the above mentioned fact that a fault can lead to an error, if the fault is activated. This error can become a failure if the SCS fails to handle the error and this can trigger a fault in another hardware part or functional unit and so on. The last failure in a cascade can lead to a hazard for the environment or a person, which can result in an accident and in injuries and loss of human life.

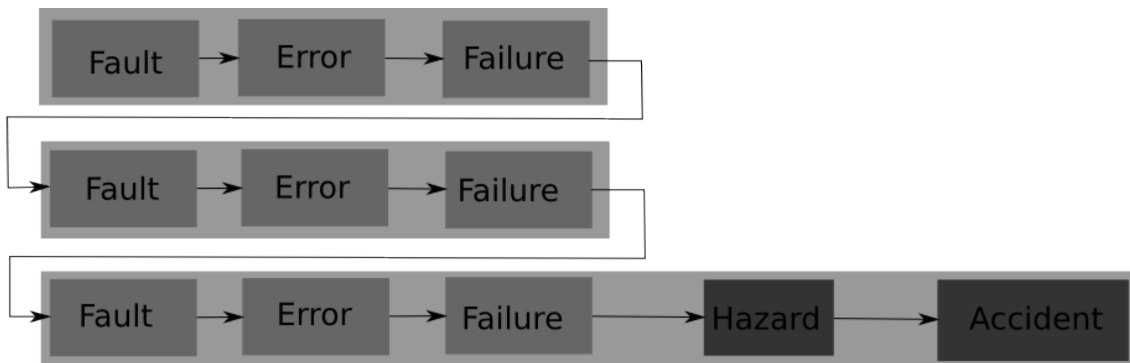


Figure 2.2: The image shows the fault-error-failure cascade. An active fault can lead to error and an error can be propagated to a failure under appropriate circumstances. This can again activate a fault in another hardware part or functional unit. If the last failure in the cascade leaves the system level, a hazard can be triggered, which can result in an accident and furthermore in injuries and loss of human life. This image is adapted from [Kal05].

2.1.3 IEC 61508

The IEC 61508 is a safety standard for electrical, electronic, and programmable electronic (E/E/PE) equipment. The first version of the IEC 61508 was published in the mid-1998 and the actual version is published 2010 by the International Electrotechnical Committee (IEC). This safety standard consists of the following seven 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 parts 2 and 3

IEC 61508-7: Overview of techniques and measures

The parts 1-4 are about technical requirements and their compliance is necessary to achieve an IEC 61508 certification. The last four parts (4-7) contain additional information for supporting the understanding of the standard.

The IEC 61508 is a generic safety standard, which focuses attention on the risk-based safety-related system design [Exi06]. The objective of this standard is to provide a cost-effective implementation for the development of E/E/PE safety-related system on the one hand and to help other industrial domains to develop domain-specific safety standard, which are based on the IEC 61508 on the other hand.

2.1.4 Safety Integrity Level

The IEC 61508 certification process is based on two basic concepts. The first is the safety life cycle, which explains the process of developing a safety-related system. It is referred to [IEC10] for further information. The second concept are the Safety Integrity Levels (SILs), which define the level of risk reduction. The IEC 61508 defines four SILs, where SIL1 is the lowest possible level and SIL4 the highest level of risk reduction, which is the most difficult level to achieve. The IEC 61508 distinguishes between *low demand mode*, *high demand mode* and *continuous mode* for SIL classification. The choice of these operation modes are based on the operation frequency and the proof test frequency. If the frequency of demands for operation is smaller than once per year and smaller than twice of the proof test frequency, the system runs in a low demand mode. If the frequency of demands for operation is greater than once per year and greater than twice of the proof test frequency, the system runs in a high demand mode. For a continuous mode, the system has to run continuously and can be called as a very high demand mode. Table 2.1 shows such a SIL classification for a continuous mode, where the risk reduction is presented as a probability of dangerous failure per hour.

Safety Integrity Level	**Average frequency of a dangerous failure of the safety function per hour**
SIL4	$\geq 10^{-9}$ to $< 10^{-8}$
SIL3	$\geq 10^{-8}$ to $< 10^{-7}$
SIL2	$\geq 10^{-7}$ to $< 10^{-6}$
SIL1	$\geq 10^{-6}$ to $< 10^{-5}$

Table 2.1: The different SILs and the corresponding probabilities of a dangerous failure of the safety function for a safety-related system, which runs in continuous mode [IEC10].

Furthermore, the SIL classification gives recommendations about the usage of different techniques and measurements. An example is shown in Table 2.2, where highly recommended means that this technique or measurement has to be applied or used. If a method is recommended it should be used except there is a good reason, why it cannot be used. There are also methods, which are labeled with no recommendation and methods which are not recommended. In this case, the method should not be used, if there is no good reason for the usage of this technique.

Assessment/Technique	**SIL1**	**SIL2**	**SIL3**	**SIL4**
Checklists	R	R	R	R
Decision/truth tables	R	R	R	R
Software complexity metrics	R	R	R	R
Failure analysis	R	R	HR	HR
Common cause failure analysis of diverse software (if diverse software is actually used)	-	R	HR	HR
Reliability block diagram	R	R	R	R

Table 2.2: Functional safety assessment: shows the highly recommended (HR), recommended (R) and assessments or techniques with no recommendation (-), which are defined in [IEC10].

2.1.5 Techniques and Measurements

The IEC 61508-7 contains different techniques and measurements, which are recommended to apply for achieving a special test coverage required by the SIL [IEC10]. Furthermore this part of the safety standard gives further information for the used methods and explains how they should applied. The seventh part contains the following four subcategories:

Techniques and measures to protect against random hardware failures: These methods are useful for the compliance with the safety requirements specified in the second part of the IEC 61508 (hardware development). Some examples for these methods are software-based self-tests for the functional units of the CPU, RAM self-tests or tests with redundant hardware.

Techniques and measures to protect against systematic failures: These techniques try to minimize the failure committed in the project management and the overall

system design and verification process. Some example techniques are worst-case analysis or black-box tests.

Techniques and measures to achieve software safety integrity: These parts recommends different techniques in the development and testing step to achieve a safe and qualitative product. Examples for these methods are UML, diverse or modular programming.

Techniques and measures for ASIC design: This part contains methods for the design and testing of ASICs. Examples are static run-time analysis, burn-in tests or validation of soft-cores.

2.2 ARM926EJ-S Core Module

This section gives an overview of the used processor and explains the necessary processor parts for the further understanding of this thesis.

2.2.1 Generic Architecture Overview

The ARM926EJ-S belongs to the general-purpose ARM9 microprocessor family [ARM08]. It has a 32-bit RISC-CPU with an ARMv5TE instruction set architecture (ISA), which supports 32-bit ARM- and 16-bit Thumb instructions. This ISA allows the user to switch between a high performance and a high code density (small code size). Furthermore, this processor module supports instructions for Jazelle Java extension, which allows an efficient execution of Java byte code. DSP extensions and Floating Point Unit (optional) are also supported by the processor. The core module has a five stage pipeline and supports AMBA-AHB interfaces for multilayer AHB-based systems. The ARM926EJ-S has a Harvard-based caching architecture and a Memory Management Unit (MMU). An overview of all processor blocks and their interaction among each other can be seen in [ARM08, p. 27].

2.2.2 Memory Management Unit

The MMU of the ARM926EJ-S is an ARMv5 architecture, which provides virtual memory features and a two-level page table, where a single set is stored in the main memory [ARM08]. This allows the MMU to perform address translations, permission checks and memory region attributes for instruction and data accesses. The MMU uses a Translation Lookaside Buffer (TLB) to cache frequently used pages for the avoidance of unnecessary and slow path table walks. The whole TLB consists of a main TLB and a lockdown TLB. The main TLB is a two-way, set-associative cache with 32 entries per way. This results in a total number of 64 entries. The lockdown TLB is an eight-entry fully-associative cache, which allows to lockdown special address translations to avoid a slow path table walk. The MMU supports different mapping sizes, like 1 MB as sections, 64 KB as large pages, 4 KB as small pages and 1 KB as tiny pages. Another feature of the ARMv5 MMU is the hardware path table walk, which decreases the translation time of a not-cached address translation.

2.2.3 Register

The ARM9EJ-S CPU has a total number of 37 32-bit registers, where 31 of these registers are for general-purpose and six registers are status registers [ARM02]. The accessibility of these registers depends on the processor state and operating mode.

If the CPU is in ARM state and user mode, 16 registers (r0 to r15) and the Current Program Status Register (CPSR) are directly accessible. The CPSR contains the condition code flags and the current mode bits. The registers r0 to r13 are for general-purpose, where the register r13 contains in most cases the stack pointer (SP). The register r14 is the Link Register (LR), which contains a copy of the Program Counter (PC), if a branch is executed. The r15 contains the PC. If the CPU switches to the privileged mode, an additional register is accessible. The Saved Program Status Register (SPSR), which contains condition code flags and the mode bits saved as a result, if the exception caused entry to the current mode.

If the CPU is in the thumb state and user mode, only the registers r0 to r7 as well as PC, SP (r13), LR (r14) and CPSR are directly accessible. There are banked SPs, LRs, and SPSRs for each privileged mode. The higher registers (r8-r15) are not part of the standard register set, but they can be used for fast temporary storage. For example a special MOV-instruction can transfer a value from lower to higher register. After this a CMP-instruction can be applied to compare the value of the lower register with the higher register.

2.2.4 ARM9EJ-S CPU

As said before, the ARM9EJ-S core implements the ARMv5TE architecture, which supports a 32-bit ARM and 16-bit Thumb instructions. The CPU is able to switch between these two states, which enables an optimization between code density and performance. An ARM study shows that the code size in Thumb state is typically 35% smaller than equivalent ARM code, while providing 160% of the effective performance in constrained memory bandwidth operations [ARM02].

The ARM9EJ-S core implements a 32-bit RISC CPU with a 32-bit address space, 32-bit registers, 32-bit barrel shifter and Arithmetic Logic Unit (ALU), enhanced 32-bit MAC block and a 32-bit memory transfer. Furthermore, the memory system is based on a Harvard architecture, which allows accessing data and instructions concurrently. This results in a significant decrease in cycles per instruction. The CPU also implements a five-stage pipeline in ARM state and a six-stage pipeline in Jazelle state. The processor core supports DSP-extensions with a single-cycle 16x16 and 32x16 MAC implementations. This improves the performance over ARM7-based CPUs by a factor of two to three.

2.3 Fault Injection

This section describes the purposes of Fault Injection (FI) platforms and gives an overview to the generic architecture of a FI. Furthermore, it presents and explains the different groups of FI approaches. The most information in this section is based on [ZAV04], if it is not otherwise stated.

2.3.1 Objectives of Fault Injection

The main objective of a FI is the assessment of the dependability for a fault-tolerant system or component. As said in [BDL96], FI approaches can be grouped in static and dynamic FI. The static FI methods are used to modify the program's source code text and the dynamic FI methods are used to modify the state of an executing program. The most commonly known static FI is the mutation testing, which tests the robustness of a program. Through small syntactic changes, like an operator exchange, the recovery code fraction for example could be tested, which is normally not executed during the run-time of a program. Dynamic FI methods are used to simulate the behavior of a system while a hardware fault modifies special memory bits or register contents. Dynamic FI is important to ensure the functional safety of a system or component by testing the implemented fault detection methods. In order to guarantee that a certain test coverage or fault detection rate is fulfilled, many tests of the system or component under faulty conditions have to be executed. The issue is that most of the hardware faults rarely occur. Due to this low occurrence rate of faults, a fault injection is used to simulate such faults in an adequate number.

According to [ZAV04], the usage of FI techniques can yield the following seven benefits:

- The functional behavior under faulty condition of the fault-tolerant system.
- The assessment of the efficacy of fault tolerance techniques.
- A forecasting of the system's behavior in the case of a fault occurrence.
- An estimation of the failure coverage and latency of fault tolerance techniques.
- An assessment of the effectiveness of fault tolerance techniques for different workloads.
- Identification of bottlenecks and weak points in the design (where one fault can crash the system).
- The analysis of the system's behavior when and while a fault occurs (fault propagation for example).

In summary, FI methods can be used to simulate the behavior of a fault-tolerant system or component while a hardware or software fault occurs. The next sections of this thesis will focus only on dynamic FI methods, which inject hardware faults.

2.3.2 Overview of the Fault Injection Environment

Figure 2.3 shows the typical architecture of a FI system. It consists of the following nine components:

Target system: Executes the program or functionality, which should be tested while faults are injected.

Fault injector: Injects faults in a target system while it executes commands from workload generator.

Fault library: Stores the types, durations, locations of faults and the time when the faults should be triggered.

Workload generator: Generates the workload (instructions or programs) as input for the target system.

Workload library: Stores sample workloads for the target system.

Controller: Controls the fault injection.

Monitor: Tracks the commands, which are executed and initiates data collector if it is necessary.

Data collector: Collects and stores the raw test data.

Data analyzer: Analyzes the data and performs a data pre-processing.

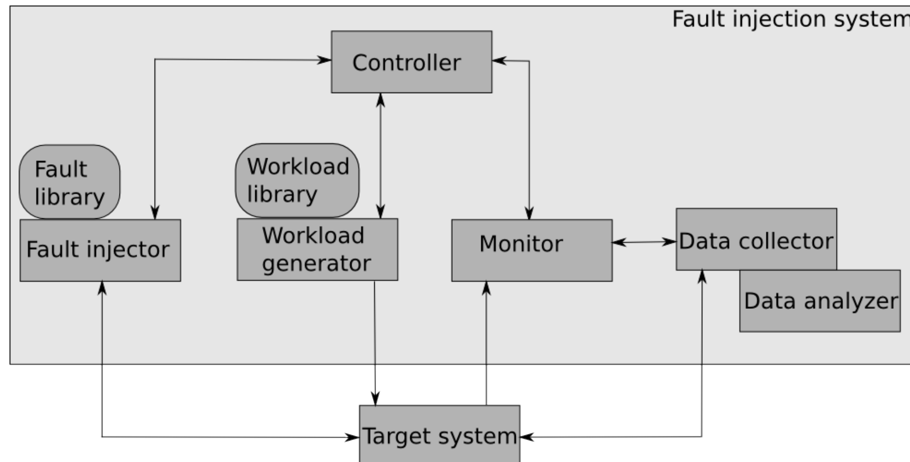


Figure 2.3: This image shows the components of a typical FI environment and their interaction among themselves. This image is adapted from [HTI97].

2.3.3 Hardware-Based Fault Injection

Hardware-Based Fault Injection (HWFI) uses additional hardware to inject faults into the hardware of a target system. HWFI approaches can be grouped in two main categories, depending on the faults and their locations:

HWFI with contact: The fault injector has direct physical contact to the target system. The fault is injected by producing voltage or current, which is injected into a pin of the target. Pin-level active probes and socket insertion are common examples for HWFI with contact.

HWFI without contact: The fault injector has no direct physical contact to the target system. In the most cases, an external source produces a physical phenomenon, like heavy ion radiation or electromagnetic interference. This phenomenon induces a spurious current in the target chip.

Table 2.6 summarizes the advantages and disadvantages of the HWFI methods and Table 2.3 shows some examples of HWFI with a short description of them.

Name of tool	Description
RIFLE	Pin-level fault injection system for dependability validation [MRMS94].
FOCUS	Design automation environment used for analyzing a microprocessor-based jet-engine controller. FOCUS uses a hierarchical simulation environment for tracing the impact of transient faults and is able to measure the error propagation within the chip [Cho92].
MESSALINE	Pin-level forcing system, which uses active probe and sockets [AAA+90].
FIST	Inject transient (heavy-ion radiation) and power disturbing faults by using contact and contactless methods [KLD+94].
MARS	Time-triggered, fault-tolerant, distributed system which consists of several computer nodes with synchronous TDMA and three level Error Detection Mechanism (hardware-software, system-software and application-software level) [Fuc96].

Table 2.3: This table presents some examples with a short corresponding description of common HWFI techniques in the literature.

2.3.4 Software-Based Fault Injection

Software-Based Fault Injection (SWFI) is helpful to assess the consequences of hidden software bugs. These methods are based on the execution of additional software on the target system, which modifies the system state. SWFI exploits cooperative and communicative functions to inject faults in the target system and are more oriented towards implementation details. SWFI approaches can inject almost all sorts of hardware and software faults, which are accessible by software. Some example for these faults are, register faults, memory faults, dropped or replicated network packets, erroneous error conditions and flags, miss-timings, missing messages, replays or faulty disk reads.

Furthermore, SWFI can be classified in two groups according to the time when faults are injected:

FI during compile-time: Where and when a fault should be injected, has to be specified before the program image is loaded and executed. A modified piece of code alters the program instruction and generates an erroneous software image, which activates the fault, if the image is executed on the target system.

FI during run-time: This group needs a mechanism to trigger the fault injection during run-time. Possible mechanism are time-out, exception/trap and code insertion. Time-out mechanism uses a timer, which generates an interrupt when the timer expires after a predefined time. The called interrupt handler injects the faults. Exception/trap mechanisms use a hardware exception or software trap to transfer

the control to the fault injector. The advantage of this mechanism is that faults can be injected whenever certain events or conditions occurs. Code insertion techniques add instructions rather than changing original instructions. The advantage of this method is that the fault injector can run in user-mode rather than system mode (higher privileged mode) like the trap method.

Table 2.6 summarizes the advantages and disadvantages of the SWFI methods and Table 2.4 shows some examples of SWFI with a short description of them.

Name of tool	**Description**
FERRARI	FERRARI uses software traps to inject CPU, memory and bus faults, which are triggered by program counter (location) or timer. It changes the content of registers or memory and model transient and permanent faults [KKA92].
FTAPE	Injects faults in user-accessible registers of CPU, memory and disk-subsystems. CPU and memory faults are injected by bit-flips and faults in the disk-subsystem are realized by routines in the driver code [TI95].
FIAT	FIAT is a real-time distributed accelerated FI, which injects faults in messages (corrupted, lost, delayed), tasks (delayed, abnormal termination) and timers [SV+88].
XCEPTION	Exploits advanced debugging and performance monitoring features of modern processors to inject faults. XCEPTION triggers faults by processors own exception and uses fault masks in combination with bit-level operations to model stuck-at, bit-flips and bridging faults [CMS98].
DOCTOR	This approach allows injections into CPU, memory and network-communication. It uses a sophisticated method to modify memory contents and supports three triggering mechanisms. Time-out for memory faults, traps for non-permanent CPU faults and the modification of instructions during compilation for permanent CPU faults [HSR95].
EXFI	EXFI is a FI for embedded microprocessor-based systems based on trace exception mode of microprocessors. This tool is able to model single transient bit-flip faults in the memory and in user registers [BPRR98].
GOOFI	GOOFI can perform different FI techniques on different target systems. It is a user-friendly approach with graphical user interface and is able to inject single or multiple transient bit-flip faults [AVFK03].

Table 2.4: This table presents some examples with a short corresponding description of common SWFI techniques in the literature.

2.3.5 Simulation-Based Fault Injection

Simulation-Based Fault Injection (SIFI) are based on a simulation model, which is used to simulate the hardware where fault should be injected (system under analysis). This simulation model is developed by Hardware Description Languages like VHDL (Very high speed integrated circuit Hardware Description Language) and faults are injected into this VHDL model. It exist two different opportunities to realize SIFI. One is to modify the original VHDL code and the second is to use built-in commands of simulators. Some examples for VHDL code modifications are approaches based on saboteurs (additional FI component) or mutants (dormant code, which is activated by fault injection).

Table 2.6 summarizes the advantages and disadvantages of the SIFI methods and Table 2.5 shows some examples of SIFI with a short description of them.

Name of tool	**Description**
VERIFY	Uses an extension of VHDL for describing faults in a component. VERIFY is a multi-threaded FI, which uses checkpoints and comparison with golden run to improve simulation time of the faulty run [STB97].
MEFISTO-C	Uses a VHDL simulator and injects faults with simulator commands in variables and signals of a VHDL model [FSK98].
HEARTLESS	Is a hierarchical register-transfer-level fault-simulator, which supports permanent stuck-at faults, transient bit-flips and delay faults for complex sequential designs like processor cores [RPB$^+$01].
GSTF	Is a VHDL-based, automatic and model-independent FI tool for injection permanent stuck-at and transient bit-flip faults at gate, register and chip level. GSTF is based on a commercial VHDL simulator [BGGG00].
FTI	FTI aims to generate a fault-tolerant integrated circuit by an original VDHL description and some guidelines for the fault-tolerant techniques and their location in the design provided by a designer. This approach automatically generates a fault-tolerant version by inserting hardware and information redundancy [ELO01].

Table 2.5: This table presents some examples with a short corresponding description of common SIFI techniques in the literature.

2.3.6 Emulation-Based Fault Injection

The aim of Emulation-Based Fault Injection (EMFI) is to decrease the simulation time of the SIFI approaches and to take into account the effects caused by the circuit environment. The circuit under analyze is implemented on an FPGA and the development board is connected to a host computer, which defines and controls the fault injection experiment and displays the results.

Table 2.6 summarizes the advantages and disadvantages of the EMFI methods.

Techniques	Advantages	Disadvantages
HWFI	• Can access locations, which are not directly accessible. • High time-resolution for hardware triggering and monitoring. • Fast experiments (near real-time). • Not intrusive (no modification of the target). • Well suited for low-level fault models. • No model development or validation are required. • Modeling of permanent faults at pin level.	• High risk of the damage of the target system. • Low portability, low observability and limited controllability. • Requires special-purpose hardware for fault injection. • Limited set of injection points and injectable faults. • High level of device integration, multiple chip hybrid circuit and dense package technologies limit accessibility.
SWFI	• Is able to test applications and operating systems. • FI can run near real-time. • No special-purpose hardware, low complexity, low development and low implementation costs. • No model development or validation are required. • Can model new classes of faults.	• Limited set of injection instants (instruction level only). • Cannot inject faults in location, which are not accessible for the software. • Modification of the source code is required. • Limited observability and controllability. • Very difficult to model permanent faults.
SIFI	• Can support all system abstraction levels (electrical, logical, functional and architectural). • Not intrusive (no footprint of the testing mechanism). • Full control of fault models and injection mechanism. • No special-purpose hardware (low costs). • High observability and controllability. • Modeling of transient and permanent faults. • Allows reliability assessment at different design stages.	• Large development effort. • Time consuming (experiment length). • No real time fault injection. • Model may not include design faults. • Accuracy of the results depends on the accuracy of the model. • Models are hard to define.
EMFI	• Shorter injection time than SIFI. • Reduction of run-time by partially or totally implementing the input pattern generation in an FPGA.	• Initial VHDL description must be synthesizable and optimized to reduce run-time. • Costs for emulation hardware or/and implementation complexity. • Analyzes only the functional consequences and not the temporal impact of a fault. • Restricted number of faults by the limited number of I/O-ports of the FPGA. • High speed communication between host computer and emulation board is necessary.

Table 2.6: This table summarizes the advantages and disadvantages of different fault injection methods [ZAV04].

2.3.7 Hybrid Fault Injection

A hybrid FI method is a combination of two or more of the above-mentioned techniques which aims to combine the benefits of both or all used techniques. A combination of a

HWFI and a SWFI can combine the advantage of versatility of the SWFI and the accuracy of a HWFI. Such a hybrid approach is well suited for measuring extremely short latencies. A further example is the combination of SIFI and HWFI or SWFI to combine the advantage of controllability and observability of the SIFI. For hardware parts which are not directly accessible, like a (Arithmetic and Logic Unit) ALU which is embedded in a CPU, a SIFI approach can perform an accurate fault injection.

An example for a hybrid FI was developed at the Chalmers University of Sweden [GS95], which combines a SWFI and SIFI. The execution of code runs at full-speed on the target system, while no fault injection is triggered. In the case of a fault injection, the simulator gets active and provides detailed access to the target. This method uses an operational-profile-based fault injection, which only injects fault in those parts (for example memory cells) which contains live data.

2.4 Software-Based Self-Test

This section describes the principle of software-based self-tests and explains, which role they play in the context of safety-critical systems and the IEC 61508. Furthermore, this section introduces the used fault model and describes the self-testing techniques proposed by the IEC 61508. Finally, a short overview of the state-of-the-art methods is given in the related work section.

According to the definition in [GPZ04], a self-test is the ability of an electronic component to test itself against occurring faults and to deliver observable results. The availability of an embedded processor in core-based System on Chip (SoC) architectures, makes it possible to run test generation, test application and test response capturing of self-tests as software-routine instead of a specially synthesized hardware modules. These solutions are grouped as Software-Based Self-Tests (SBSTs), which have some significant advantages against hardware-based self-tests. The first advantage of SBSTs is that they are non-intrusive. SBSTs do not add additional hardware or performance overhead to the circuit and do not change the circuit structure or need a modification of the instruction set architecture (ISA). SBSTs are low-cost and low-power self tests, because SBSTs do not need external testers, add no additional chip area, delay or power consumption overheads to the circuit. Furthermore, SBSTs allow at-speed testing of the circuit or components, because all test patterns are applied with the actual system frequencies. SBSTs are also very flexible, because the software code is easy to modify or to adapt to other components.

Self-Tests are important techniques to guarantee the right functioning of components and hence to achieve safety certificates for safety-critical systems. But the development of SBSTs, especially for embedded processors, includes challenging tasks. Embedded processors are no simple combinational unit or finite state machine, they are components with well-optimized designs in term of performance and power consumptions. They consist of several components like arithmetic units, storage elements, interconnection modules, which are optimized to achieve the best performance at instruction execution level. Some components like pipelines or multiplexers are not easy-to-test, because they are not directly accessible by the instruction set. In the case of memory testing, the challenging task is to reach high fault coverages with the smallest possible runtime. Nowadays, a small runtime is even more important, because the memory sizes are drastically increasing.

2.4.1 Fault Classification

The functional hardware faults of components, which should be detected by SBSTs, can be classified according to their occurring or active time.

The IEC 61508 safety standard defines two groups of fault types: *Permanent Faults* and *Transient Faults* [IEC10]. Permanent faults reflect long-term damage of hardware components and can be modeled by a stuck bit of a binary value. These kind of faults are called *Stuck-at Faults*. An example of such a stuck-at 0 or 1 fault can be a short between positive supply V_{DD} or negative supply V_{ss} and an output pin of a combinational unit or simple a CMOS-Transistor.

A transient fault (or soft-error) is a fault, which appears for a short time at various locations. They are caused by temporary environmental influences such as neutron and alpha particles, power supply and interconnect noise, electromagnetic interference or electrostatic discharge [KML+06]. In contrary to permanent faults, transient faults do not result in long-term damage of the hardware. These type of faults are drastically increasing, because of the decreasing size of circuits and the increasing density of hardware components per area.

[KML+06] defines another third fault type, the so-called *Intermittent Faults*. These faults appear repeatedly and periodically at the same location and produce errors in bursts while they occur. Intermittent faults are caused by unstable hardware, which are mainly due to process variations and manufacturing residuals.

2.4.2 Self-Tests and IEC 61508

As already mentioned in Section 2.1.4, the IEC 61508 safety standard defines four SIL levels, which covers the different probabilities of failures. The highest achievable SIL level is limited by the *Hardware Fault Tolerance (HFT)* and by the *Safe Failure Fraction (SFF)* (see Table 2.7).

SFF	**Hardware Fault Tolerance**		
	0	**1**	**2**
$< 60\%$	-	SIL1	SIL2
$60\% - < 90\%$	SIL1	SIL2	SIL3
$90\% - < 99\%$	SIL2	SIL3	SIL4
$\geq 99\%$	SIL3	SIL4	SIL4

Table 2.7: Maximal allowed SIL for a given Safe Failure Fraction (SFF) depending on the level of Hardware Fault Tolerance [IEC10].

The SFF is a percentaged ratio of failures that does not result in dangerous situations. This ratio can be seen in Equation 2.1, where λ_s are the number of safe failures, λ_{Dd} defines the number of detected dangerous failures and λ_{Du} covers the number of undetected dangerous failures.

$$SFF = \frac{\sum \lambda_s + \sum \lambda_{Dd}}{\sum \lambda_s + \sum \lambda_{Dd} + \sum \lambda_{Du}} \tag{2.1}$$

The maximal achievable SIL depends in addition to the SFF also on the HFT. The HFT

of N defines a minimum number of faults $N + 1$ that can cause the system to lose its safety functionality. For example, a single processing element that computes a safety-critical function ($N = 1$) fails its safety functionality, if two ($N + 1$) failures occur. Table 2.7 shows that the higher the HFT is, the lower is the required SFF in order to achieve the needed SIL.

The goal of SBSTs are to increase the number of dangerous detected failures $\sum \lambda_{Dd}$ and to decrease the number of undetected dangerous failures $\sum \lambda_{Du}$, in order to increase the overall SFF ratio. In order to achieve safety certificates only dangerous failures are important and should be detected by SBSTs. Hence, the SFF can be simplified to a new ratio, the so-called Diagnostic Coverage (DC), which is defined as:

$$DC = \frac{\sum \lambda_{Dd}}{\sum \lambda_{D}}, \tag{2.2}$$

where $\sum \lambda_{Dd}$ is the number of dangerous detected failures and $\sum \lambda_{D}$ is the sum of all dangerous failures ($\sum \lambda_{Dd} + \sum \lambda_{Du}$). The DC (or fault coverage) is a good metric to evaluate or compare different SBSTs. Before a DC value can be calculated, it is important to know all possible faults, which can occur in a component. If this component is a new unknown device, a *Failure Mode and Error Analysis (FMEA)* should be applied to identify all possible faults (FMEA is described in [IEC10]-Part 7). For processor-based systems, the IEC 61508 already defines some fault sources, which have to be handled for achieving a certain SIL. The next two sections will introduce the reader to these lists, which covers the fault sources for the CPU-core elements as well as for the memory modules such as RAM.

CPU Test Requirements defined by IEC 61508

Table 2.8 shows the requirements for the safety mechanisms of the processing elements. These requirements are defined in the IEC 61508 safety standard part 2 and give an overview of the different fault sources for the three DC classes, which should be covered for a specific fault coverage by safety mechanisms. For example, if a system with a HFT value of 1 has to reach SIL 3, then this system requires at least a SFF of 90% according to Table 2.7. If we assume that all failures are critical, then the system requires a DC value of at least 90%. Thus, we have to consider the fault types in the middle column (medium DC) of Table 2.8 and the tests have to detect at least 90% of these faults in order to achieve SIL 3.

Furthermore, it can be seen that the IEC 61508 focuses on components, which are responsible for the execution of instructions and/or for the storage of data in register files or internal memories. In the literature, these components are often grouped to the so-called CPU-core elements. These CPU-core elements are only a part of the whole CPU and do not cover arithmetic units, multiplier or barrel shifters. In the most cases, these components are used to compute outputs based on an algorithm (for example control parameters). Hence, these components have also a significant influence on the safety issues and have to be tested too. In some cases of medium and high DC, a so-called DC fault model has to be considered. This DC fault model is an aggregation of different fault types and covers stuck-at and stuck-open errors (a special case of stuck-at-errors), open circuits or outputs with high impedance as well as shorts between two lines.

CPU	**Requirements for claimed DC**		
	low (60%)	**medium (90%)**	**high (69%)**
registers, internal RAM	Stuck-at for data and addresses	DC fault model for data and addresses Change of information caused by soft-error	DC fault model for data and addresses Change of information caused by soft-error Dynamic crosstalk between memory cells No, wrong or multiple addressing
Coding and execution including flag register	Wrong coding or no execution	Wrong coding or wrong execution	No well-defined fault assumption
Address calculation	Stuck-at	DC fault model Change of addresses caused by soft-errors	No well-defined fault assumption
Program Counter (PC) and 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 2.8: Fault sources defined by the IEC 61508 for CPU-core elements [IEC10].

CPU Test Techniques suggested by IEC 61508

The IEC 61508 suggests a few techniques to achieve different DC values. A listing of these techniques is given in Table 2.9. It can be seen that SBSTs are a proposed solution, but achieves only low or medium DC. However, SBSTs are still able to achieve high DC values, but it has to be shown in a verification of these tests by an appropriate fault injection experiment. The most techniques, which are listed in Table 2.9, depend on hardware redundancy (comparator, majority voter, reciprocal comparison) or require a special design for a failure detection unit (coded processing). This work excludes CPU-core techniques, which requires a modification of or additional hardware. Thus, from the methods shown in Table 2.9, only SBSTs will be covered.

Technique / Measure	Achievable DC value
Comparator	High (99%)
Majority voter	High (99%)
Self-test by software: limited number of patterns (one channel)	Low (60%)
Self-test by software: walking bit (one channel)	Medium (90%)
Self-test supported by hardware (one channel)	Medium (90%)
Coded processing (one channel)	High (99%)
Reciprocal comparison by software	High (99%)

Table 2.9: Techniques/measures for testing CPU-core defined by IEC 61508 [IEC10].

Memory Test Requirements defined by IEC 61508

Table 2.10 shows the requirements for the safety mechanisms of variable memories such as RAMs. These requirements are defined in the IEC 61508 safety standard part 2 and give an overview of the different fault sources for the three DC classes, which should be covered by safety mechanisms. In general, it is the same as for the register files or internal memories of the CPU-core elements shown in Table 2.8.

Memory	**Requirements for claimed DC**		
	low (60%)	**medium (90%)**	**high (69%)**
Variable memory	Stuck-at for data and addresses	DC fault model for data and addresses Change of information caused by soft-error	DC fault model for data and addresses Change of information caused by soft-error Dynamic crosstalk between memory cells No, wrong or multiple addressing

Table 2.10: Fault sources defined by the IEC 61508 for variable memory [IEC10].

Memory Test Techniques suggested by IEC 61508

The IEC 61508 safety standard distinguish between variable and invariable memories. Invariable memories are for example ROMs and Flash memories and variable memories are register files, caches and every type of RAMs. This thesis will focus only on variable

memories (RAMs). The IEC 61508 suggests a few techniques to test RAM-modules, which can be seen in Table 2.11. The detailed description of RAM tests is given in Section 2.4.4. Furthermore, the safety standard suggests techniques, which depend on hardware redundancy like double RAM with hardware or software comparison or uses a special coding like parity bits or monitoring with modified Hamming codes or EDC is not covered in this thesis. This thesis will only focus on test techniques, which can be run as SBSTs.

Technique / Measure	**Achievable DC value**
RAM test *checkerboard* or *march*	Low (60%)
RAM test *Walk-Path*	Medium (90%)
RAM test *Galpat* or *transparent Galpat*	High (99%)
RAM test *Abraham*	High (99%)
Parity bit for RAM	Low (60%)
RAM monitoring with modified Hamming code or detection of data failures with errordetection-correction codes (EDC)	Medium (90%)
Double RAM with hardware or software comparison and read/write test	High (99%)

Table 2.11: Techniques/measures for testing variable memories defined by IEC 61508 [IEC10].

2.4.3 CPU-core Tests

CPU-core tests can be classified in two different categories [PGSR10], [GPZ04]:

Functional testing: The goal of functional testing is not to obtain a high fault coverage for a physical or structural fault model, but rather the testing of a digital circuit for the correctness of all known functions. The main idea is to use the Instruction Set Architecture (ISA) provided by the processor to develop test pattern sets. These test patterns are applied to the processor and are compared with a precomputed output of a correct functioning device (for example, the register value at the end of the computations). The big advantage of this approach is that no low-level details of the hardware are required. Hence, functional testing is easily applicable and more portable than structural testing approaches and hence these techniques can be re-used and have a low development costs. The drawback of functional testing methods is that it is hard to achieve high fault coverage values due to the limited information of the hardware.

Structural testing: In contrary to functional testing, structural testing targets a specific structural fault model and hence, these approaches need low-level details of the hardware. These low-level details could be information on the Register-Transfer-Level (RTL) or on the gate-level of the hardware. Structural tests also use the ISA of processor, but they take further information of the hardware provided by structural information into account. For example, the geometric arrangement of hardware components can be used to increase the fault coverage. It is more likely that two adjacent components influence each other, than components which are physically

further away. Due to these facts, the advantages of structural testing is that a high fault coverage can be achieved and that the test programs are faster, because the instructions are specially tailored to the used hardware. The drawbacks follow from the advantages. Because these methods are specially tailored to a hardware, these approaches are hardly portable and have high development costs. A further disadvantage is that low-level details of the hardware are required.

Furthermore, these CPU tests can be distinguished, if they are deterministic or randomized [GPZ04]:

Deterministic Testing: Deterministic tests apply well-known test sequences, which have a proven minimum achievable test coverage. These tests use always the same test sequences and the same test instructions, which tests always the same components. As an example, pre-developed tests for the most of the functional units of a processor (ALU, multiplier, divider, shifter) exist, which guarantee a high fault coverage independent of the actual hardware architecture [PGK+01]. For such tests it is not necessary to verify the fault coverage by fault injection experiments. The advantages of these approaches are a high fault coverage and short test sequences and the drawbacks are that in general a gate-level or RTL-level model has to be known to develop deterministic tests for a arbitrary component.

Randomized or Pseudorandom Testing: Randomized tests are based on pseudorandom instruction sequences, pseudorandom operands or a combination of both of them. These testing methods have no proven fault coverage and hence they have to be verified by an appropriate fault injection experiment. The advantage of these approaches are that they do not need information about the gate- or RTL-level, but the disadvantages are that they require long test sequences and only achieves low fault coverages with these.

Further details about classification schemes, especially for structural testing, can be seen in [PGSR10].

An effective way to develop SBSTs for processor cores is to divide the CPU into several main components [GPZ04], [KPGZ02]. For each component a set of instructions is defined that test the given component. From this set of instructions the best-observable one is selected and is subsequently applied to develop the SBST for this component. The output of this process is a set of components, which can be tested by the best-observable instruction. These components have to be prioritized according to their importance to quickly reach a sufficient high fault coverage. This is important in context of developing a low-cost SBST with low development effort. The prioritization depends on the component size given in gate-counts (number of gates in a special component) and the accessibility of the components. For this purpose, CPU-core elements can be classified in functional, control and hidden components. *Functional components* are directly and explicitly related to the execution of an instruction. They can be further grouped in *computational functional components* (ALUs, adders, shifters, incrementers, subtracter, divider, multiplier etc.), *storage functional components* (register and internal RAMs) and *interconnect functional components* (multiplexers or tri-state buffer). *Control components* controls either the flow of instructions and data inside the processor or the flow of data from and to external environments like memory subsystems or peripherals. A classical control component is

the control unit of a CPU, which is responsible for the instruction decoding and for the producing of the control signals for the functional components. The control components are not directly related to a specific function or directly implied by the instruction set. Hence these components can only be tested indirectly by performing test for the functional units. *Hidden components* are mainly responsible for increasing the performance and instruction throughput of a processor. Hidden components are for example pipelines or branch prediction schemes and are not visible to the assembly language. In order to quickly achieve a high fault coverage, it is suggested to test the functional units first. They are visible and easier to test by instructions than control or hidden components. Furthermore, functional components are the biggest units according to the gate-counts or chip area on a processor core.

In general, the SBSTs which are considered in this thesis are not based on such an analysis process. The considered SBSTs are a collection of well-known tests for CPU core components and commonly used RAM test routines. The considered tests are functional and not randomized tests. Some of these tests are deterministic tests with a proven fault coverage and for some other tests the fault coverage has to be verified by fault injection experiments.

2.4.4 RAM Tests

The achieved fault coverage and the test length of a memory test depends on the used fault model [HGR02]. In order to define a Functional Fault Model (FFM) it is important to understand the concept of fault primitives (FPs). Functional faults are the deviation of an observed and a specified behavior under a number of performed memory operations. This amount of operations on the memory is called operation sequence and if this operation sequence results in a deviation of observed and expected memory behavior, then this operation sequence is called a *sensitizing operation sequence (S)*. The observed memory behavior that deviates from the expected behavior is called *faulty behavior (F)*. The combination of S, F and the *read result (R)* in case of a memory read operation specifies a certain fault and is called a FP, which is denoted as $< S/F/R >$.

These FPs can be classified according to:

- The number of simultaneous operations required by S into single- and multi-port faults.
- The number of sequential operations required by S into static and dynamic faults.
- The way the FP manifest themselves into simple and linked faults.

Single-port faults are FPs that require one port in order to sensitize a fault. In contrary to single-port faults, multi-port faults are FPs that can only sensitize a fault by performing two or more operations on different ports. Static faults are FPs that can sensitize a fault by performing one operation sequentially and dynamic faults have to be sensitized by more than one operation. Simple faults are faults, which cannot be influenced by other faults and hence no fault masking can occur. Linked faults are faults, which influence the behavior of each other and hence fault masking can occur. In this thesis, only single-port, simple and static as well as dynamic faults with maximal two operations are considered.

RAM Fault Models

Many FFMs for memories have been introduced in the past [HGR02]. The oldest but well-known FFM consists of *Address decoder Faults (AFs)* and *Stuck-at Faults (SAFs)*. Later, these FFMs are refined by introducing the *Data retention fault (DRF)*[1], *Stuck-Open Fault (SOF)*, *Read Destructive Fault (RDF)*, *Deceptive Read Destructive Fault (DRDF)* and *Disturbing Coupling Fault (CFds)*. In 1999, experimental results were applied to many memory chips and memory tests. The results show that not all by these tests detected faults could be described by using the well-known FFMs. Hence, additional FFMs exists and these new introduced FFMs are described in the following section.

FFMs for RAMs can be classified in *Single-cell static FFMs*, *Single-cell dynamic FFMs*, *Two-cell static FFMs* and *Two-cell dynamic FFMs* [AAG01]. Single-cell static FFMs describe faults, which are sensitized by performing one operation to a single cell. As seen in Table 2.12, these faults can be grouped, according to their FPs, in *State Faults* (SF_x), *Transition Faults* (TF_x), *Read Disturb Faults* (RDF_x), *Write Disturb Faults*(WDF_x), *Incorrect Read Faults*(IRF_x) and *Deceptive Read Disturb Faults*($DRDF_x$).

FFMs	FPs
SF_x	$SF_0 =< 0/1/- >, SF_1 =< 1/0/- >$
TF_x	$TF_\uparrow =< 0w1/0/- >, TF_\downarrow =< 1w0/1/- >$
RDF_x	$RDF_0 =< 0r0/1/1 >, RDF_1 =< 1r1/0/0 >$
WDF_x	$WDF_0 =< 0w0/1/- >, WDF_1 =< 1w1/0/- >$
IRF_x	$IRF_0 =< 0r0/0/1 >, IRF_1 =< 1r1/1/0 >$
$DRDF_x$	$DRDF_0 =< 0r0/1/0 >, DRDF_1 =< 1r1/0/1 >$

Table 2.12: List of single-cell static FFMs with the according FPs.

Single-cell dynamic FFMs describe faults, which are sensitized by performing more than one operation to a single cell. There exist 2-operation, 3-operation, and so on dynamic FFMs, but this thesis will focus only on 2-operation dynamic FFMs. For single-cell 2-operation dynamic FFMs, there are 30 different possible FPs, but this number can be reduced to 12 possible FPs, because an isolate write operation may not be sufficient to detect a fault while the cell has not been read in order to detect the stored value set during the write operation. This 12 single-cell 2-operation FPs are used to define *Dynamic Read Disturb Faults* (RDF_{xy}), *Dynamic Incorrect Read Faults* (IRF_{xy}) and *Dynamic Deceptive Read Disturb Faults* ($DRDF_{xy}$),which is visualized in Table 2.13.

Two-cell static FFMs describe faults, which are sensitized by performing one operation to a cell, while observing the effect on another cell. A FP for a two-cell static FFMs can be defined as $< S/F/R >=< S_a; S_v/F/R >_{a,v}$, where S_a and S_v are the sequences performed on the aggressor and the victim cell, respectively (S_a and $S_v \in 0, 1, 0w0, 0w1, 1w0, 1w1, 0r0, 1r1$). As Table 2.14 shows, these sequences result in 36 possibles two-cell static FPs, which can be grouped to *State Coupling Faults* ($CFst$), *Disturb Coupling Faults* ($CFds$), *Transition Coupling Faults* ($CFtr$), *Write Disturb Coupling Faults*

[1]DRFs are caused by defective refresh logic in DRAMS or defective pull-up resistors in SRAMs and result in the loss of data in less than the specified hold time. DRFs can be tested by applying delays between march sequences and subsequently check the cell content for writing/reading an one and a zero. In this thesis DRFs will not be handled.

FFMs	FPs
RDF_{xy}	$RDF_{00} =< 0w0r0/1/1 >, RDF_{11} =< 1w1r1/0/0 >,$ $RDF_{01} =< 0w1r1/0/0 >, RDF_{10} =< 1w0r0/1/1 >$
IRF_{xy}	$IRF_{00} =< 0w0r0/0/1 >, IRF_{11} =< 1w1r1/1/0 >,$ $IRF_{01} =< 0w1r1/1/0 >, IRF_{10} =< 1w0r0/0/1 >$
$DRDF_{xy}$	$DRDF_{00} =< 0w0r0/1/0 >, DRDF_{11} =< 1w1r1/0/1 >,$ $DRDF_{01} =< 0w1r1/0/1 >, DRDF_{10} =< 1w0r0/1/0 >$

Table 2.13: List of single-cell dynamic FFMs with the according FPs.

(CFwd), *Read Disturb Coupling Faults (CFrd)*, *Incorrect Read Coupling Faults (CFir)* and *Deceptive Read Disturb Coupling Faults (CFdr)*.

FFMs	FPs
$CFst$	$CFst_{0,0} =< 0; 0/1/- >, CFst_{0,1} =< 0; 1/0/- >,$ $CFst_{1,0} =< 1; 0/1/- >, CFst_{1,1} =< 1; 1/0/- >$
$CFds$	$CFds_{0w0,0} =< 0w0; 0/1/- >, CFds_{0w0,1} =< 0w0; 1/0/- >,$ $CFds_{1w1,0} =< 1w1; 0/1/- >, CFds_{1w1,1} =< 1w1; 1/0/- >,$ $CFds_{0w1,0} =< 0w1; 0/1/- >, CFds_{0w1,1} =< 0w1; 1/0/- >,$ $CFds_{1w0,0} =< 1w0; 0/1/- >, CFds_{1w0,1} =< 1w0; 1/0/- >,$ $CFds_{0r0,0} =< 0r0; 0/1/- >, CFds_{0r0,1} =< 0r0; 1/0/- >,$ $CFds_{1r1,0} =< 1r1; 0/1/- >, CFds_{1r1,1} =< 1r1; 1/0/- >,$
$CFtr$	$CFtr_{0,\uparrow} =< 0; 0w1/0/- >, CFtr_{0,\downarrow} =< 0; 1w0/1/- >,$ $CFtr_{1,\uparrow} =< 1; 0w1/0/- >, CFtr_{1,\downarrow} =< 1; 1w0/1/- >$
$CFwd$	$CFwd_{0,0} =< 0; 0w0/1/- >, CFwd_{0;1} =< 0; 1w1/0/- >,$ $CFwd_{1;0} =< 1; 0w0/1/- >, CFwd_{1;1} =< 1; 1w1/0/- >$
$CFrd$	$CFrd_{0,0} =< 0; 0r0/1/1 >, CFrd_{0,1} =< 0; 1r1/0/0 >,$ $CFrd_{1,0} =< 1; 0r0/1/1 >, CFrd_{1,1} =< 1; 1r1/0/0 >$
$CFir$	$CFir_{0,0} =< 0; 0r0/0/1 >, CFir_{0,1} =< 0; 1r1/1/0 >,$ $CFir_{1,0} =< 1; 0r0/0/1 >, CFir_{1,1} =< 1; 1r1/1/0 >$
$CFdr$	$CFdr_{0,0} =< 0; 0r0/1/0 >, CFdr_{0,1} =< 0; 1r1/0/1 >,$ $CFdr_{1,0} =< 1; 0r0/1/0 >, CFdr_{1,1} =< 1; 1r1/0/1 >$

Table 2.14: List of two-cell static FFMs with the according FPs.

For the sake of completeness it should be mentioned that 2-operation dynamic two-cell FFMs can be denoted with the FP $< S/F/R >$. For example, $< v(0r0)a(1r1)/1/- >$ defines a FP, where the fault is sensitized by a $0r0$ operation on the victim cell and an $1r1$ operation on the aggressor cell. After applying these sequences, an one is read from the victim cell instead of a zero. Based on the values for S, F and R, 192 possible two-cell 2-operation dynamic FPs can be built. Since these FPs have not been observed in the experiments in [AAG01], they are not further considered in this thesis.

Further RAM Test Definitions

RAM tests can be divided in diagnostic and non-diagnostic testing schemes [BNR00]. *Diagnostic tests* can distinguish between different type of faults (for example between single-cell and multiple cell faults). A common approach is to use different march tests, where every march test can detect one certain fault type. *Non-diagnostic tests* detect fault types, without determining the type of fault.

Furthermore, RAM tests can be developed for *Bit-Oriented Memories (BOMs)* or *Word-Oriented Memories (WOMs)* [GAC02]. RAM tests for BOMs have a word width of a single bit (write or read of one bit) and RAM tests for WOMs can read or write more bits simultaneously. RAM tests for WOMs have the problem that the detection of CFs is more challenging, because CFs have to be distinguished in intra- and inter-CFs. For an intra-CF the victim and the aggressor cell belongs to the same memory word and for an inter-CFs the victim cell and the aggressor cell does not belong to the same cell.

RAM tests can be transparent, which means that the initial content of a memory is preserved [Nic96]. This feature is very suitable for periodic online testing of memories. Furthermore, the authors of [Nic96] have proven that the fault coverage for transparent tests does not decrease for modeled faults. The disadvantage of transparent tests is that a signature prediction phase is needed to decide if a fault has occurred. This prediction phase increases the runtime of these tests.

In order to avoid this additional runtime, caused by the prediction phase of transparent memory tests, [YHW99] introduces a symmetric transparent memory test, which exploits symmetries of the test data sequences to omit the signature prediction and decreases the runtime of transparent memory tests.

RAM Test Methods

This section describes the memory testing methods, which are suggested by and explained in the IEC 61508 part 7 (appendix A) [IEC10].

Checkerboard: The Checkerboard test can detect static bit faults. A checkered (alternating) pattern of zeros and ones is written to the memory. The memory cells are checked in pairs. The first address of a memory pair is variable and can be chosen and the address of the second cell of this pair is build by a bit-wise inverting of the first address. The content of both cells should be the same. The test consists of two cycles. The first cycle of the test accesses the memory cells in an ascending address order and the second cycle is performed in a descending address order. After this, the test is repeated with the inverse initial memory content. The complexity of this test is about 10N memory accesses, where N is the number of checked bits.

March: March tests for RAMs are described in the IEC 61508 as alternative to the Checkerboard RAM tests. It is the most commonly used memory test due to its linear complexity. March tests define a specific order, how a memory cell is accessed, written or read and are classified in the IEC 61508 with a low DC, but the standard considers only the simplest march test. There are many march tests, which are able to detect different types of memory faults. All march tests have a common structure and notation, which is described in Table 2.15. A march test consists of a set of *March elements*, which specifies the addressing order of and the operations

on a memory cell. Every march element is sequentially applied to one cell and after finishing this cell, the test is applied to the next cell.

Symbol	Description
{...}	delimits the beginning and the end of a march test
(...)	delimits the beginning and the end of a march sequence
$\Uparrow$	ascending addressing order (from 0 to $N-1$)
$\Downarrow$	descending addressing order (from $N-1$ to 0)
$\Updownarrow$	either ascending or descending addressing order (not relevant)
$r0$	read the content of a cell with expected value 0
$r1$	read the content of a cell with expected value 1
$w1$	write 1 to a memory cell
$w0$	write 0 to a memory cell

Table 2.15: The commonly used notation of march memory test and their corresponding description.

Walkpath: The Walkpath RAM test is able to detect static and dynamic bit faults as well as crosstalk between memory cells. The Walkpath initializes all memory cells with an uniform pattern (zeros or ones). The first cell is inverted and the remaining memory cells are checked, to guarantee that the data background is correct. Then the first cell is inverted one more time to get the original value and the whole process is continued for the remaining untested memory cells. The second cycle of the walking bit model is started with the inverse initial memory data background. The Walkpath memory test requires $2(N^2 + 3N)$ memory accesses.

Galpat: The Galpat RAM test detects static bit faults and many dynamic coupling faults. The Galpat test starts with an initial phase, where the whole memory is initialized with zeros or ones. The test starts with the inverting of the first cell and subsequently all other cells are checked if their content is changed. After every read access of one of the remaining cells, the inverted cell is checked too. This process is repeated for every cell until every cell is checked. The second cycle is executed with the inverse initial configuration. The Galpat RAM test has a complexity of $2(2N^2 + N)$. It is possible to apply a transparent Galpat test to preserve the memory content. Further details about the transparent Galpat RAM test can be looked up in the IEC 61508 safety standard - part 7 [IEC10].

Abraham: The Abraham test [NTA78] is able to detect all stuck-at errors and coupling faults. This test detects more faults than the Galpat memory test and has the best assessment in the IEC 61508 safety standard (high DC). The Abraham test reads and writes every cell of a memory in ascending and descending addressing order. The test writes either a transition from zero to one or vice versa and is comparable with the march tests. The test requires 30 operation per bit (30N).

2.5 Related Work

This section gives an overview of the current approaches of QEMU-based Fault Injection (QEMU-FI) platforms in the literature. Furthermore, this section covers the current available software-based self-tests, which are used to detect faults on a target system.

2.5.1 QEMU-based Fault Injection Framework

There are many approaches available, which uses an emulator to simulate a closed-source target system, like an ARM9 core for FI. One possible emulator is the Quick EMUlator (QEMU), which provides an opportunity to implement a FI by little code modifications. QEMU is an open source software emulator, which emulates several CPU architecture (x86, ARM, Sparc, Alpha etc.) on several host platforms (x86, PowerPC, ARM etc.) [Bel05]. QEMU is based on the dynamic translation principle, which provides a fast emulation (5 to 20 times slower than native code execution). Further details about QEMU will be presented in the Chapter 3. Due to this fact, QEMU-FI fall into the category of software simulation-based fault injection (SIFI).

The authors of [Chy09] collect program execution statistics with QEMU to improve the efficiency of software-based fault injection. It exploits the dynamic code translation of QEMU, which translates every emulated target instruction to native instructions. This dynamic code translation is the basis for the good performance of the QEMU. The approach sends a signal to the translation procedure before the emulated target instruction translation is executed and dumps the emulated processor state to a statistical module for further analysis. However, this approach only collects program execution statistics, but does not implement a FI based on QEMU.

In [WEL+13] an extension of the QEMU is presented with the purpose of evaluating variability-aware software techniques. The authors use the different processor models provided by QEMU to emulate variations in power consumption (caused for example by dynamic frequency scaling), timings and fault characteristics of multiprocessor systems and to sense and adapt these variations by software. In order to do so, they use a SystemC wrapper, which instantiates a special number of QEMU single processors. Every execution of a processor is executed in a SystemC process, which is necessary for a concurrent execution and an accurate simulation of a multiprocessor system. Furthermore, the wrapper of the framework provides an interface to define and connect other hardware parts like timer or shared memory, which are defined in SystemC. All in all, this approach uses the QEMU emulator only to simulate different processor models, but it is not able to inject faults into memory or registers neither and hence it is not suitable for a FI framework.

The FI framework developed by [XX12] implements a software SIFI based on the QEMU emulator to test board-level Built-In-Test (BIT) software of avionics systems. This paper analyzes functional faults of the memory and defines corresponding fault models and simulates these memory faults. Furthermore, it uses an XML file as fault library, which defines the fault parameters. This XML-approach allows the user to easily define duration, location and type of faults, but this work covers only memory faults and not register faults or faults in CPU functional units.

In [DC07] a QEMU-FI with the name QInject is introduced. QInject is a QEMU emulator, which is extended with FI capabilities to test the behavior of self-healing

operating systems while faults appear. QInject uses the debugging interface of QEMU to inject faults with the GDB debugger into a target back-end. The advantage of this work is that a fault injection experiment could be controlled by a remote GDB session through the network. The drawback of this approach is that the GDB interface to the target back-end heavily limits the access to the state of the target system (faults in network-devices are not possible). Furthermore, this approach take only transient bit-flip faults into account to test exception handling, code reloading or other techniques to recover the system state.

The authors of [BKJ+12] present an approach for mutation-based testing through binary mutation. These mutations are injected at run-time through dynamic translation of the QEMU emulator. They propose a taxonomy of mutation operators for the ARM instruction set. The advantage of using dynamic translation in this approach is that the mutation testing neither relies on source code nor on a certain compiler. Furthermore, this approach is able to inject high-level language faults and target specific faults related to compiler and linker, but it does not cover low-level hardware faults like memory or register faults.

The work in [YPH13] develops a FI framework based on the QEMU emulator, called BitVaSim. BitVaSim aims to test BIT software on embedded development boards, which are equipped with PowerPC or ARM processors. BitVaSim exploits the advantage of simulation-based approaches, which do not harm or interrupt the real hardware or software. Another advantage of this approach is that all simulated parts are reachable and so more fault modes can be achieved. BitVaSim uses abstract key-value pairs, which are defined in an XML format, to describe the functional fault modes (similar to [XX12]). Furthermore BitVaSim configures and simulates the hardware target board, which are defined by a modeling and configuration module and a hardware simulation module. This approach provides an automatic FI by executing fault defined in an XML-file but it also provides a FI interface, which allows configuring and injecting faults on demand at run-time. BitVaSim can inject faults in every process including the kernel of the operating system. For this reason, this FI method can be used to test the behavior of an operating system while faults occur. This approach is based on good basic ideas, but there is no sample-code public available and the authors of this work do not response to emails and due to the sparse evaluation results the correct functioning of this approach can be doubted.

2.5.2 Software-Based Self-Tests (SBSTs) for CPU

The IEC 61508 suggests the walking bit method for testing the registers of a CPU. The authors of [TB06] use march tests to develop SBSTs for the register files. This can be done, because the physical structure of registers is the same as for small SRAMs. Hence, faults in the register files can be modeled with a memory fault model and every memory self-test can be applied to register files.

The authors of [KGPZ02] describes an approach, which tries to minimize the developing effort, code size and runtime of CPU self-tests. This work focuses on developing a low-cost self-test for the ALU and shifter unit of a Parwan CPU. The authors present deterministic tests for these components, which uses the best observable and controllable instruction. These tests are based on a RTL model (VHDL) of the processor and can not be used for an ARM9-CPU, because no RTL level is public available.

In [TP07] an online self-tests for an ARM7 processor in safety related systems is

presented. They apply the walking pattern method, which is described in the IEC 61508, for the RAM testing. For the program memory (flash) the authors use a Cyclic Redundancy Check (CRC) calculation for verifying the data integrity, which is also suggested by the IEC 61508. The registers are checked by a Galpat test (also suggested by IEC 61508) and the ALU, CPU flags and the stack pointer are tested by custom tests. The paper does not describe how the fault coverages are verified, but claims that the tests achieve the required fault coverage for an 1oo2 system architecture (at least 90% for SIL3).

The work in [CD01] introduces tests for ALU, shifter and Program Counter (PC) and the fault coverages of these tests are verified. Furthermore, the fault coverages of other components (accumulator, controller, instruction register unit, status register etc.) are also verified. The results of these experiments are a fault coverage of 99% for ALU and shifter, 90% for the PC and the whole Parwan CPU. This shows that also other parts, which are not directly tested by the tests, can be checked indirectly.

The authors of [PGK+01] describe deterministic SBST for the multiplier, ALU and shifter of a CPU. In general, deterministic tests require details about the hardware (RTL or gate level model) of a CPU, but in this case the authors have proven that the achieved fault coverage for the multiplier is independent of the actual multiplier architecture and is about 99% or higher. The fault coverages for the shifter and ALU are verified for the most commonly used architectures and are about 99.9% for the ALU and 100% for the shifter.

The work of [LCL08] uses a combination of deterministic and randomized test methodologies. The authors use deterministic tests to check hardware components efficiently, where an architecture model is available, with a small test code size and randomized tests for the faults, which can not be easily tested by deterministic tests (hidden components). The fault coverages are verified for different CPUs and have achieved an average fault coverage of 92%. For an ARM9-v4 compatible processor, a fault coverage of 97% has achieved with a combination of deterministic and randomized tests with 5000 basic blocks (5500 instructions).

2.5.3 Software-Based Self-Tests (SBSTs) for RAM

The authors of [LTW05] provide an algorithm, which describes step-by-step how a bit-oriented march test can be converted to a word-oriented march test. Furthermore, the work proves that the fault coverage of the original, bit-oriented march test is preserved by performing the conversation from bit- to word-oriented march test. A further advantage of this algorithm is that the time complexity of the tests are reduced. In comparison to the original transparent BIST theory of [Nic96], the time complexity of a march C- memory test is reduced from 75N to 43N for a 16bit memory interface (N is the number of bits).

The work of [VES13] describes a transparent online memory test for word-oriented memories too, but in addition to [LTW05], this test provides a symmetric approach, which allows skipping the signature prediction phase in order to reduce the time complexity of memory tests. For a march C- test the complexity of a transparent test is about 14N and can be reduced to 10N by using symmetric approaches (N is the number of bits). It is also shown by [YHW99] that the original fault coverage can be increased by applying this methodology.

In [LESO13] an analysis of industrial SRAM tests is presented. They compare 29 linear march tests in order to combine two or more test algorithms to achieve high fault coverages

with low test effort. The experimental results of this work show that 96% fault coverage can be achieved by combining only two out of the 29 considered algorithms. Furthermore, the paper presents an overview of all 29 considered march tests with the corresponding sequences and achieved fault coverages.

In [HGR02] a march test is described, which is able to detect all static simple faults in RAMs. The test has a complexity of 22N (N is the number of bits) and can detect 100% of SF, TF, WDF, RDF, DRDF, IRF and all static CFs.

The authors of [SBS08] focus on tests for DDR-SDRAMs, which can detect all static simple faults in burst-mode operations. The problem in burst-mode accesses of RAMs is that coupling and/or address decoder faults can be masked. In order to overcome this issue, the authors describe two possible solutions. The first explicitly masks the write operation in the memory controller by disabling the data access from memory interface to array cells. The second solution is to use different start addresses of the data patterns in the burst sequence.

The work in [GT98] describes an approach for testing word-oriented memories by converting a bit-oriented march test. The conversion is based on combining a bit-oriented march test for inter-word faults with a test for intra-word faults. This results in a more efficient test for the targeted faults (idempotent CFs, disturb CFs and state CFs). Furthermore, this work gives an example conversion for the march C- and the march LR memory test.

The authors of [AAG01] focuses on the analysis of faulty effects for spot defects in embedded DRAMs (eDRAMs). They perform a simulation with an electrical model of the memory in order to inject defects, which are caused by opens, shorts and bridges. This paper points out that considering only static FFMs like SAF, AF, TF and CFs are not enough to describe opens, shorts and bridges in eDRAMs. The authors introduce new static FFMs (SF_x and WDF_x) and new dynamic FFMs (RDF_{xy}, IRF_{xy} and $DRDF_{xy}$) in order to inject these faults.

Chapter 3

Concept and Design

This chapter covers the concept and design details of the fault injection (FI) and Software-based Self-Tests (SBST) approaches. The FI concept gives an introduction to the used emulator framework (QEMU) and describes the basic functionality of QEMU. Furthermore, the design for all parts of the FI framework (shown in Figure 2.3) is explained.

3.1 System Requirements

This section gives a short recap of the goals for a better understanding of the next sections that are already stated in introduction section (Section 1.2).

1. **SBSTs for ARM9 processors:** A software has to be provided, which tests if the main parts of the CPU (ALU, registers, shifter, adder and multiplier) as well as the RAM works correctly.
2. **Verification of the SBSTs:** In order to verify that the SBSTs satisfy the required test coverages, a special injection framework is needed, which is able to simulate every possible fault (permanent and transient) at every time for the closed-source ARM9 processor.
3. **IEC 61508 compliant SBSTs:** The developed test-methods should satisfy the required IEC 61508 safety standard at least at SIL-3.
4. **Fully-automated simulation:** The Self-Tests and the FI should run without human interaction in order to reduce the run-time of simulation and testing.
5. **Easy-to-use:** The choice of fault as well as the time of injection and other relevant parameters should be easy to set up.

3.2 Fault Injection Framework based on QEMU

QEMU[1] [Bel05] is a fast, open-source and portable software emulator, which is able to simulate various architectures (ARM, SPARC, x86, MIPS, PowerPC, etc.) and runs

[1] Available under: `www.qemu.org`

on several host platforms (x86, PowerPC, ARM). QEMU provides techniques for a fast simulation, which results in a simulation performance of 5 to 20 times slower runtime than native code execution.

QEMU is in many respects a well-suited solution for realizing a FI framework for a closed-source processor like the ARM926EJ-S. First of all, the usage of QEMU is currently the only possible approach to inject faults at low-level (with access to register, memory and other relevant hardware) in a closed-source target without using any hardware. Thus, QEMU makes it possible to test hardware and/or software of a project in an early development stage, where no physical hardware is available. A further advantage is that this approach does not destroy hardware parts, such as hardware-implemented FIs and does not have any dependencies, for example, on operating system APIs (Application Programming Interface), compilers or additional hardware such as JTAG-devices. Furthermore, QEMU supports innately the used ARM instruction set and various ARM machines like the ARM Integrator/CP board, ARM Versatile baseboard and several variants of the ARM RealView baseboard. If there is an undefined machine, it is quite easy to setup or rather define a new machine type, like the Freescale i.MX28 EVK development board.

In comparison to other suitable simulation-based approaches, QEMU is due to its translation process easier to extend with FI capabilities. These are the reasons why QEMU-based FIs are a cheap and low resource costs solution for implementing a FI framework for a closed-source target-platform.

3.2.1 QEMU Overview

QEMU supports two modes, the user mode and the system mode. The primary usage of QEMU happens in the system mode, where the entire target is fully emulated including IO and the complete software stack (boot firmware, operating system, kernel space device driver). The purpose of this mode is to run different operating systems on others, such as Linux on Windows or vice versa. A further usage of this mode is the debugging of software, because the virtual machine (QEMU) can easily be stopped and its state can be inspected, saved and stored.

In the Linux specific user mode, QEMU runs Linux processes for one target PC on another CPU. The primary purpose of this mode is to test the functionality of cross compilers or to test CPU emulators without booting the whole virtual machine.

QEMU supports self-modifying code, precise exceptions and floating point library (software emulation and native host FPU instructions) and consists of the following six subsystems:

- CPU emulator (x86, PowerPC, ARM, SPARC etc.)
- Emulated devices (VGA display, serial port, PS/2 mouse and keyboard, IDE hard disk, network card, etc.)
- Generic devices (block, character and network devices), which connects the target device to the corresponding host devices
- Machine description (target), which instantiates the emulated device
- Debugger
- User interface

QEMU provides various features in the full system emulation mode. Firstly, it uses a full software MMU to guarantee the maximum portability. Furthermore, QEMU can optionally use an in-kernel accelerator, like kvm. This accelerator executes some guest code natively, while continuing to emulate the rest of the instructions. QEMU is able to simulate a symmetric multiprocessor (SMP) system on a single host CPU, but QEMU is currently not able to use all cores of a SMP host system. In this case, QEMU uses only one core fully, because, according to [QEM14], it is too difficult to implement an efficient atomic memory access.

QEMU uses a dynamic translation principle to generate the native code efficiently. In contrast to static binary translation, such as an Instruction Set Simulator (ISS), dynamic translation does not translate every instruction directly. It groups instructions to blocks and executes these blocks only if a branch occurs. Hence, only these instructions are translated, which are executed at runtime. This avoids unnecessary redundancy (for example error handling routines, which are rarely executed) and results in a fast execution of source code. The disadvantage of this method is unfortunately that details of the CPU model's micro architecture get lost, due to the generalization.

QEMU is in comparison to bochs [DAF+14] a faster x86 emulator, because of the usage of dynamic translation. Furthermore, bochs is closely tied to x86-platforms, while QEMU provides different processor types. Valgrind [Val14] is primary a memory debugger, while QEMU does not support this feature. However, Valgrind uses a dynamic translator too, which generates better code than QEMU (see register allocation of QEMU). The disadvantage of Valgrind's dynamic translator is that it is specially tailored to x86-architecture. There are also commercial PC-virtualizer, like VMWare [VMW14], which are faster than QEMU, but they all need specific, proprietary and potentially unsafe host driver. In contrast to emulators, these approaches are not able to provide cycle exact simulations. There are many software approaches available, which are based on QEMU. Commonly known examples are VirtualBox [Ora14], Xen [FCJ+14], KVM [KVM14] and QEMU-SystemC [Gre14]. The last one uses QEMU to simulate a system, where special hardware devices are developed in SystemC. This allows to run native binaries just as on the real hardware. A more detailed comparison to other emulators can be found in [QEM14].

3.2.2 Details about QEMU's Translation Process

The following explanation is referred to QEMU versions above 0.9.1, because previous versions of QEMU implement the dynamic translation by DynGen. The grouped instructions are converted to C code by DynGen and GNU Compiler Collection (GCC) converts this C code into host-specific code. The disadvantage of this approach is that DynGen was tightly coupled to GCC and creates problems, if a newer GCC version is released.

The QEMU translation process is visualized in Figure 3.1. It starts with a check, if the guest instruction has already been translated or not. If it has been already translated, the translated guest instruction can be executed without starting a new fetch-execute cycle by looking up the host instruction in the translation cache. Otherwise, the next instruction is fetched from the binary and is subsequently decoded. The decoding step of the target binary generates so-called micro-operations (micro-ops), which are a kind of non-target-specific intermediate code. This code works on a virtual register set and is assembled for readability. These micro-ops are subsequently stored in a micro-ops buffer

until a branch occurs or the CPU state changes. Thus, guest instructions are grouped to so-called Translation Blocks (TBs), which are translated to host-specific code by the Tiny Code Generator (TCG). An example of such a translation process can be seen in Table 3.1. These translated TBs are additionally stored in a translation cache in order to provide an execution speed close to native execution by avoiding redundant translations. The translation cache has a size of 16 MB (for ARM instruction set), which should make it possible that every instruction is fetched and decoded only once. If the cache is full, the entire cache is flushed instead of using a replacement algorithm. The translation cache uses a hash-function to map the entry address of a target code block to the corresponding address of the TB.

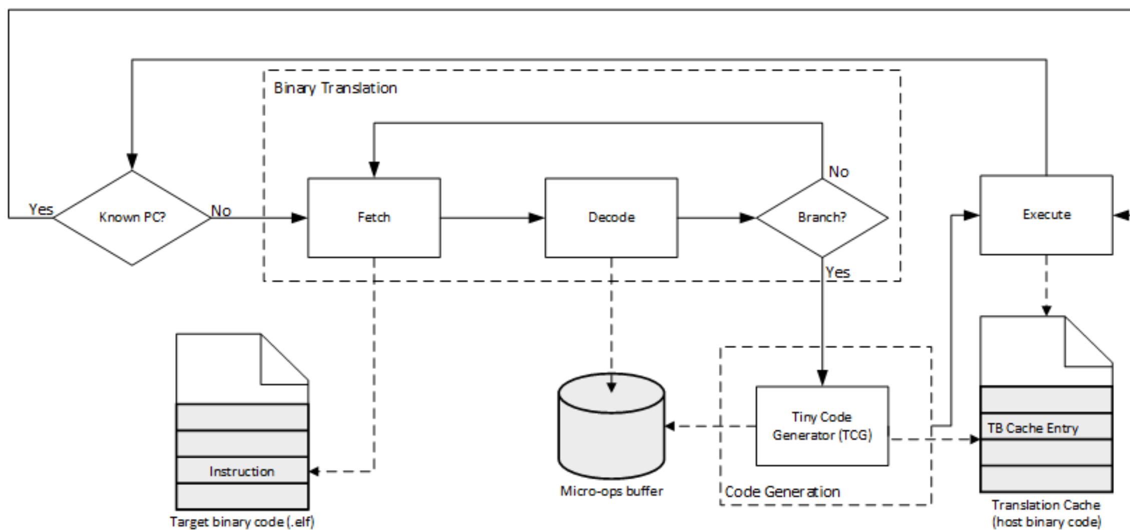


Figure 3.1: This image visualizes the basic steps of the QEMU translation process. If a guest instruction has not already been translated, the instruction is fetched from the binary. After the decoding of this instruction, so-called micro-operations (micro-ops) are generated. This micro-ops are a kind of non-target-specific intermediate code, which is assembled for readability. The translated micro-ops are stored in the micro-ops buffer and translated by the Tiny Code Generator, if a branch occurs. This translated micro-ops are subsequently executed. Furthermore, these instructions are grouped to Translation Blocks (TBs), which are stored in a translation cache for fast subsequent use. This image is adapted from [GFP09].

QEMU uses the chaining of TBs for further optimization of the emulation speed. This technique avoids the unnecessary returning form the translation cache to the QEMU code, which is in general slow. In order to overcome this issue, QEMU chains every TB to the next TB. If a TB is chained to another, the following TB is directly executed after the first TB finished its execution without returning to the QEMU code. If a TB has no follower, the previous TB returns to the QEMU code, which finds, generates and executes the next TB. After the execution of this TB has finished, this TB is chained to the previous one. Hence, if the next time the first TB is executed, the next TB follows immediately on the previous TB without returning to the QEMU code.

guest instructions (ARM)	micro-ops (intermediate code)	host instructions (x86)
`mov r0 r5`	`mov_i32 tmp, r5` `mov_i32 r0, tmp`	`mov 0x14 (%ebp), %ebx` `mov %ebx, 0x0 (%ebp)`

`bl 0xd768`	`movi_i32 tmp, $0xd720`
	`mov_i32 r14, tmp`
	`goto_tb $0x0`
	`movi_i32 pc, $0xd768`
	`exit_tb $0xb550fe60`
		`mov $0xd720, %ebx`
		`mov %ebx, 0x38(%ebp)`
		`jmp 0x601be463`
		`mov $0xd768, %ebx`
		`mov $0xb550fe60, %eax`
		`jmp 0x621de9a8`

Table 3.1: This table shows an example of the binary translation from front-end ARM code to back-end x86 code with the micro-ops generation as intermediate step. In this example the content of register r0 is copied to register r0 and afterwards a branch to address 0xd768 is executed. The names of the register are in general not the same as the names in the front-end side. The micro-ops code uses a virtual register set, but this is neglected for visualization purpose in this example. This example is adapted from [BKJ+12].

3.2.3 Design of Fault Injection Components

The following paragraph explains the design of every FI-component and gives an overview to the basic structure of the whole realized system, including FI-framework, operating system (OS) and SBSTs.

Figure 3.2 shows the basic structure of the FI-framework. A host (physical computer) runs a host OS, like MS Windows or Linux. This host OS executes the QEMU-framework, which includes the necessary FI-modules. The fault injector, which provides the FI-functions, uses the ARM-specific code of QEMU to catch the encoded instructions. The necessary parts of these encoded instructions, such as OP-codes or register addresses, can be subsequently overwritten to simulate register or instruction decoder faults. Furthermore, the content of the registers, which store the operands, can also be overwritten to simulate data storage faults. The exploitation of the target-independent intermediate code of QEMU is not possible, because the abstraction of the hardware does not allow a relation between the passed variable and the registers containing this variable anymore. Hence, the in this work developed FI-framework only supports FI on ARM target platforms. The fault-injector module also parses the XML-file, which specifies the number and parameters of the faults that should be injected (fault library). The controller retrieves this information from the fault injector and decides, if and where a fault should be injected. After a successful fault injection, the controller signals the monitor to print the result to the prompt. Furthermore, a collector- and an analyzer-module is started, which writes the statistics to a file for further analysis. QEMU emulates the used development board on which the target OS is executed. In our case it is a SafeRTOS, which executes the SBST and other additional software.

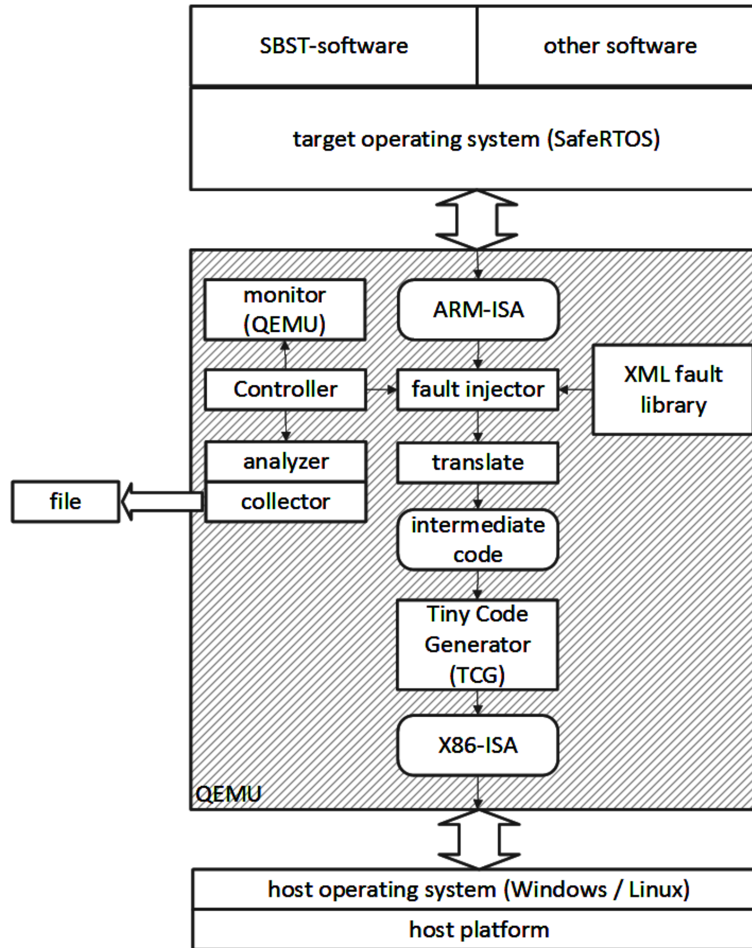


Figure 3.2: This image shows the basic structure of the whole FI-framework. A host platform (the physical computer) runs an operating system, which executes the QEMU-framework including the necessary modules for a FI. The fault injector uses the ARM-specific code of QEMU to inject faults. QEMU emulates the used development board, on which the target operating system is executed. This target operating system executes the SBST and other additional software.

Target System

The whole applications and OS should run on a specific embedded hardware board. In our case this hardware is a development board of the type i.MX28 EVK PCB REV D from Freescale Semiconductor [Fre11]. It contains an i.MX287 application processor including an ARM926EJ-S CPU with 454 MHz, 128 KB low-power on-chip SRAM, 128 KB mask-programmable on-chip ROM and a 16-bit interface to DDR2-SDRAM with a size of 128 MB. Furthermore, the board supports an interface for SPI NAND flash and two sockets for SD/MMC cards, which can used as data storage. The board also supports other hardware, which can be looked up in [Fre11]. The i.MX28 EVK board is not innately supported by QEMU, but a new definition of a hardware machine is not difficult in QEMU and is done by the following steps:

1. **Initializing of CPU cores:** The number and the type of the CPU models can be specified and initialized with the appropriate function call.

2. **Definition of the memory mapping:** The RAM memory regions are initialized and can be further divided into subregions. Furthermore, the initial memory regions for the I/O can be specified, which call specified callback-functions, if an access to these regions happen.

3. **Connection of the devices:** Different devices like LCD-interface, UART, Ethernet, USB, DMA, GPIO, Timer etc. can be wired up with the memory address specified in the memory mapping and the name of the device as well as the address of QEMU IRQ-handler as argument.

4. **Specification of kernel/OS image:** The name and path of the kernel file, the additional kernel arguments, the RAM size and the name and path of the (initial ramdisk) initrd file should be specified here.

The last step is to register the board to QEMU, which is done by one function call with the board info as argument. Hence, the board can be specified at startup of QEMU with the board name and the CPU architecture as argument.

Workload Generator and Workload Library

The SBSTs are developed and compiled for a specified CPU model and run on the target OS executed by QEMU, while faults are injected in the target platform. The SBSTs should discover faults, which occur in RAM and in the main parts of the CPU, which is required by Requirement 1.

Monitor

A few features of the monitor are already pre-implemented by the QEMU framework and can be called by the *[Ctrl]+[Alt]+[2]* shortcut. The QEMU monitor pre-defines commands like *info registers*, which shows the current content of all ARM-registers (see Figure 3.3). A full list of all commands of the QEMU monitor can be listed by typing the command *help* into the monitor console. The features of this monitor have to be extended with commands to load the fault library or to list the fault statistic.

Data Collector

The data collector redirects the output of the monitor to a file and saves the information or statistics for further analysis. This allows human-interaction-less fault injection experiments, which is required by Requirement 4.

Data Analyzer

The data analyzer counts the total number of injected faults and compares this number with the discovered faults by the SBSTs. It provides statistics for the fault injection experiments.

```
QEMU
(qemu-fi) info registers
R00=00000020 R01=0000000c R02=00000003 R03=0000ad55
R04=00000006 R05=c41c8c00 R06=ad55ad55 R07=00000000
R08=ffc42110 R09=c42263d8 R10=00000002 R11=c0095a9c
R12=c42263df R13=c0095a18 R14=ffc42180 R15=c011faac
PSR=20000193 --C- A svc32
s00=00000000 s01=00000000 d00=0000000000000000
s02=00000000 s03=00000000 d01=0000000000000000
s04=00000000 s05=00000000 d02=0000000000000000
s06=00000000 s07=00000000 d03=0000000000000000
s08=00000000 s09=00000000 d04=0000000000000000
s10=00000000 s11=00000000 d05=0000000000000000
s12=00000000 s13=00000000 d06=0000000000000000
s14=00000000 s15=00000000 d07=0000000000000000
s16=00000000 s17=00000000 d08=0000000000000000
s18=00000000 s19=00000000 d09=0000000000000000
s20=00000000 s21=00000000 d10=0000000000000000
s22=00000000 s23=00000000 d11=0000000000000000
s24=00000000 s25=00000000 d12=0000000000000000
s26=00000000 s27=00000000 d13=0000000000000000
s28=00000000 s29=00000000 d14=0000000000000000
s30=00000000 s31=00000000 d15=0000000000000000
FPSCR: 00000000
(qemu-fi)
```

Figure 3.3: This image shows the sample output of the command *info registers* in the QEMU monitor console. This command prints all available ARM-registers to the monitor output.

Fault Library

It is important to decouple the testing from the implementation part. A person, who wants to run a FI experiment does not have to know development- or implementation-specific information. Hence, a complexity reduction is important for a successful FI framework usage. A well-suitable solution, is to use an XML-based formalization for the fault definitions. XML is a markup language that defines a set of rules for the encoding of a file. This encoding is a format that is readable by humans and machines and allows an easy definition of faults and an automated simulation of the FI experiment, which is required by Requirement 4 and Requirement 5.

Fault Injector

The fault injector contains a set of functions, which provide the capability to inject different types of faults in a special location (for example a permanent fault in a specific memory cell). This module takes the memory address or parts of the encoded instruction (OP-code or register address) as input and overwrites the address or the content, pointed by the address. The modified instructions, addresses or register contents are translated to intermediate code and are subsequently passed to the TCG, which translates this intermediate code to the host-specific ISA. Furthermore, this module parses the XML-file and holds this information in an appropriate structure. This information is used by the controller, which decides based on this information, which fault (function) should when, where and how long been injected (triggered). The Tables 3.2 and 3.3 show, how different faults in different locations have been realized by the fault injection method. It should be mentioned that these modelings are targeted on the external data paths (signal lines) and not on the internal structure of modules. It can be seen that using only stuck-at faults are not sufficient to model all faults for SIL3 as well as for SIL4 requirements. Only under consideration

of coupling faults, bit-flips as well as writing new values all faults can be modeled. The FFMs, which are presented in the technical background section, describe the internal faults of a RAM-module and are also realized in the fault injection framework.

Controller

The controller includes the basic logic to control the whole FI experiments. It uses the information of the fault library stored in the fault injector to decide, which function should be called in the fault injector module. The controller uses a timer to specify when a transient or intermediate fault should be triggered or stopped. A timer is provided by the QEMU framework and returns the number of real ticks as timer value. This timer can also provide this value in milli-, micro or nanoseconds. The controller needs also the information about the current program counter (PC) for the PC-based fault trigger and the memory address, which is being accessed for the access-based fault trigger. The PC is stored in the virtual register set of QEMU, which is an easy-accessible structure. The monitoring, if a special memory address is accessed by the guest OS or another hardware/software can be easily achieved in QEMU. QEMU provides a memory API, which allows the modeling of memory controllers that can dynamically reroute or overwrite physical memory addresses. The controller also specifies, whether a result should be written to the monitor output or whether the analyzer- and the collector-module should be activated. The basic process of a fault injection experiment and the interaction between all these above-mentioned modules is visualized in Figure 3.4.

Fault description	Fault type	Component	Target	Mode	Parameter	Trigger	Type	Note
Open in address line	Address decoder fault (AF)	RAM	Address decoder	Stuck-at-0	Address	At beginning	Permanent	
Open decoder	Address decoder fault (AF)	RAM	Address decoder	Stuck-at-0/1	Address	At beginning	Permanent	SIL4 requirement
Wrong cell are accessed	Address decoder fault (AF)	RAM	Address decoder	Stuck-at-0/1, bit-flip	Address	Address accessed, at beginning	Permanent, intermittent, transient	SIL4 requirement
No access	Address decoder fault (AF)	RAM	Address decoder	Complete new value / random value	Address	At beginning	Permanent	SIL4 requirement
Multiple access	Address decoder fault (AF)	RAM	Address decoder	Coupling Faults	Data	Address accessed, at beginning	Permanent, transient, intermittent	SIL4 requirement
Address line stuck	Address decoder fault (AF)	RAM	Address decoder	Stuck-at-0/1	Address	Address accessed, at beginning	Permanent	
Cell can be set to 0 and not to 1 (or vice versa), Special case of SAF	Transition Faults (TF)	RAM	Memory cell	Stuck-at-0/1 (or Transition Faults)	Data, address	At beginning	Permanent	
Content of a cell is influenced by its neighbors	Neighborhood Pattern Sensitive Faults (NPSF)	RAM	Memory cell	Coupling Faults	Data	Address accessed	Transient	SIL4 requirement
Stuck cell	Stuck-at-faults (SAF)	RAM	Memory cell	Stuck-at-0/1	Data	At beginning	Permanent	
Driver stuck (sense amplifier)	Stuck-at-faults (SAF)	RAM	Read/write logic	Stuck-at-0/1	Data	At beginning	Permanent	
Read/write line stuck	Stuck-at-faults (SAF)	RAM	Read/write logic	Stuck-at-0/1	Data	At beginning	Permanent	
Chip-select stuck	Stuck-at-faults (SAF)	RAM	Read/write logic	Stuck-at-0/1	Data	At beginning	Permanent	
Data line stuck	Stuck-at-faults (SAF)	RAM	Memory cell	Stuck-at-0/1	Data	At beginning	Permanent	
Open in data lines	Stuck-at-faults (SAF)	RAM	Memory cell	Stuck-at-0	Data	At beginning	Permanent	
Crosstalk between data lines	Coupling Faults (CF)	RAM	Memory cell	Coupling Faults	Data	Address accessed	Permanent	SIL4 requirement
Shorts in data lines	Coupling Faults (CF)	RAM	Memory cell	Stuck-at-0/1, Coupling Faults	Data	At beginning, address accessed	Permanent, transient, intermittent	

Table 3.2: This table visualizes, how different faults in different RAM locations have been realized. The relationships between functional faults (fault descriptions) and fault models (fault types) are based on the information presented in [SBS08] and [GV90].

Fault description	**Fault type**	**Component**	**Target**	**Mode**	**Parameter**	**Trigger**	**Type**	**Note**
Open in address line	Register decoder fault (RF)	CPU	Register decoder	Stuck-at-0	Address	At beginning	Permanent	
Open decoder	Register decoder fault (RF)	CPU	Register decoder	Stuck-at-0/1	Address	At beginning	Permanent	SIL4 requirement
Wrong cell are accessed	Register decoder fault (RF)	CPU	Register decoder	Stuck-at-0/1, bit-flip	Address	Address accessed, at beginning	Permanent, intermittent, transient	SIL4 requirement
No access	Register decoder fault (RF)	CPU	Register decoder	Complete new value / random value	Address	At beginning	Permanent	SIL4 requirement
Multiple access	Register decoder fault (RF)	CPU	Register decoder	Coupling Faults	Data	Address accessed, at beginning	Address accessed, at beginning	SIL4 requirement
Address line stuck	Register decoder fault (RF)	CPU	Register decoder	Stuck-at-0/1	Address	Address accessed, at beginning	Permanent	
Open in address line	Instruction decoder fault IF)	CPU	Instruction decoder	Stuck-at-0	Address	At beginning	Permanent	
Open decoder	Instruction decoder fault IF)	CPU	Instruction decoder	Stuck-at-0/1	Address	At beginning	Permanent	SIL4 requirement
Wrong cell are accessed	Instruction decoder fault IF)	CPU	Instruction decoder	Stuck-at-0/1, bit-flip	Address	Address accessed, at beginning	Permanent, intermittent, transient	SIL4 requirement
No access	Instruction decoder fault IF)	CPU	Instruction decoder	Complete new value / random value	Address	At beginning	Permanent	SIL4 requirement
Multiple access	Instruction decoder fault IF)	CPU	Instruction decoder	Coupling Faults	Data	Address accessed, at beginning	Address accessed, at beginning	SIL4 requirement
Address line stuck	Instruction decoder fault IF)	CPU	Instruction decoder	Stuck-at-0/1	Address	Address accessed, at beginning	Permanent	
Wrong instruction is executed	Control function fault (CFF)	CPU	Instruction execution	Bit-flip, Stuck-at-0/1	Address (op code oder pc)	Address accessed, at beginning	Permanent, transient, intermittent	
Additional instruction is executed	Control function fault (CFF)	CPU	Instruction execution	not possible	Address	Address accessed	transient, intermittent	SIL4 requirement
No instruction is executed (interruption or exception)	Control function fault (CFF)	CPU	Instruction execution	Complete new value (NOP address)	Address (op code oder pc)	Address accessed, at beginning	Permanent, transient, intermittent	
Stuck-at for registers (R/W line, cell,)	Data storage fault (DSF)	CPU	Register	Stuck-at-0/1, bit-flip	Data	Address accessed, at beginning	Permanent, intermittent, transient	
A line in the transfer path is stuck	Data transfer fault (DTF)	CPU	Register	Stuck-at-0/1	Data, address	At beginning	Permanent	
Two lines in the transfer path are coupled (shorts, crosstalk)	Data transfer fault (DTF)	CPU	Register	Stuck-at-0/1, bit-flip, Coupling Faults	Data	Address accessed, at beginning	Permanent, intermittent, transient	
function dependent (wide variety in existing designs)	Data manipulation faults (DMF)	CPU	Interrupt, ALU,...					

Table 3.3: This table visualizes, how different faults in different CPU locations have been realized. The relationships between functional faults (fault descriptions) and fault models (fault types) are based on the information presented in [TA80] and [SPS+94].

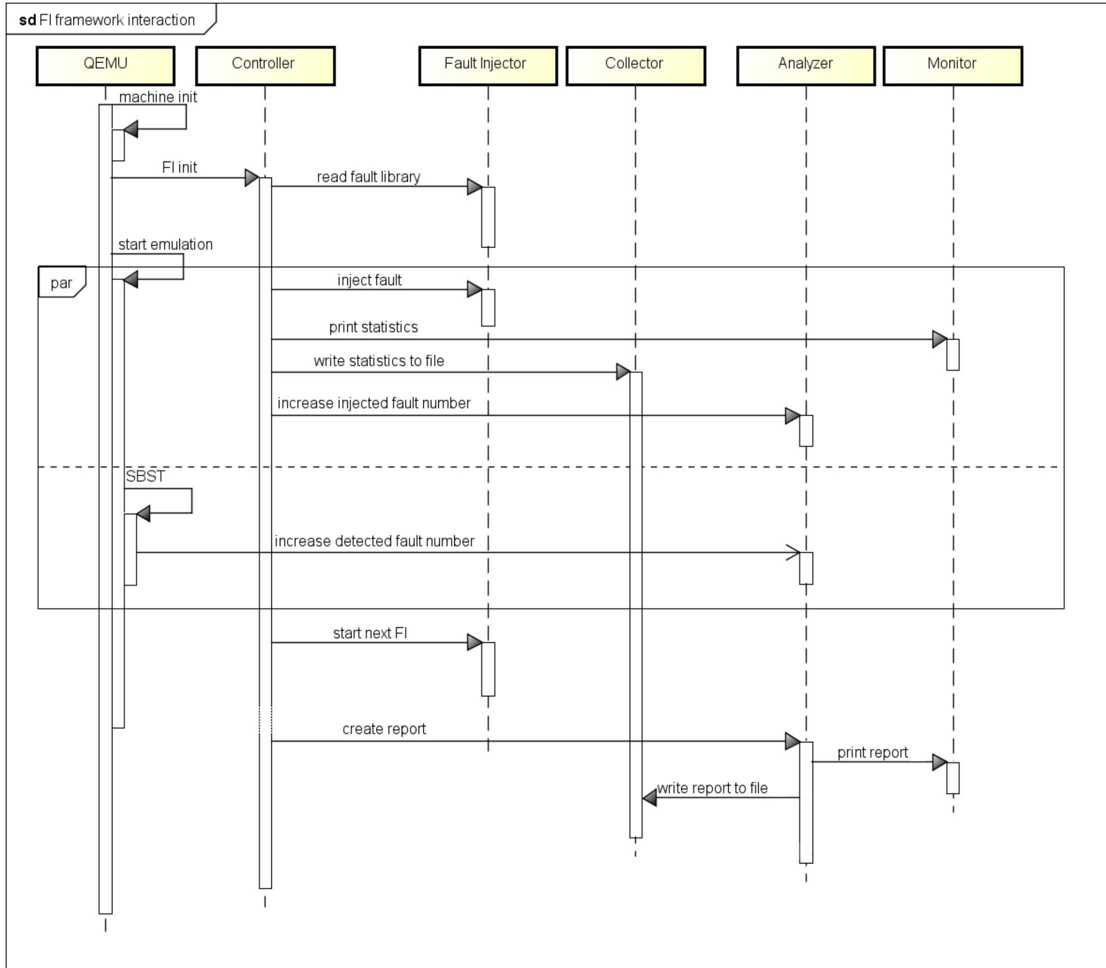


Figure 3.4: This image shows the basic process of a FI experiment. It starts with the initializing of the target platform, followed by the parsing of the XML-fault library. Then, the emulation process starts and the controller is instantiated. If the controller decides to perform a fault injection, it calls the appropriate function in the Fault Injector module and signals the monitor to print a fault statistic to the console. Furthermore, the controller increases the number of injected faults in the analyzer module. The SBST runs concurrently with the fault injection process and increases the number of detected faults in the analyzer, if the fault is detected by SBST. These processes are continued until all faults are triggered once. At the end of the experiment the controller signals the analyzer to create a final report and saves this report also to the file held by the collector.

3.3 Software-Based Self-Tests (SBSTs)

This section describes the SBSTs for the CPU-core elements and the main memory. The CPU-core tests consist of deterministic tests with proven or at least experimentally known high fault coverage, which are taken from literature. The basic tests for the main memory

are taken from literature too, but they are adapted to increase the performance and/or the fault coverage ratio of the SBSTs. Furthermore, a comparison between different SBSTs against test lengths and theoretical detectable FFMs for the main memory is given in this section.

3.3.1 SBSTs for CPU-core Elements

As explained in [GPZ04], it is advisable to test large processor components first in order to quickly achieve a sufficient high fault coverage. This is important for low-cost self testing, because the development effort can be kept as small as possible, while the self-test programs can be kept small and fast. One possible method for the assessment of the processor component size is the so-called gate-count. The gate-count is defined as the number of gates, which are used in the electronic device (the processor in our case). The issue for an ARM-processor is that these gate-counts are not available, because the ARM-processors are closed source projects. [PKK11] assumes that the gate-counts for an ARM926EJ-S core is comparable with an modern pipeline processor and results in 42% for the register banks, 39% for the multiplier and adder, 3% for the barrel-shifter, 2% for the ALU and 14% for other components, like multiplexer or pipeline. Due to these gate-counts, the following testing order should applied for the CPU-core elements:

1. register banks
2. multiplier and adder
3. barrel-shifter
4. ALU

The 14% of the other components are not tested explicitly in this thesis, but they can be tested indirectly with the other tests. The test coverage for these indirect tests are not exactly known and it is suggested by the IEC 61508 to assume a Safe Failure Fraction (SFF) of 50%. However, it can be assumed that in general a high fault coverage ratio is achieved too.

The tests for adder/ALU, shifter/divider and multiplier are taken from [PGK+01]. This work presents deterministic tests, which have a mathematically proven high and architectural independent fault coverage for the multiplier unit as well as high fault coverages for the residual components. All fault coverages are greater or equal 99% and are verified for an Intel 8051 processor. The exact SFFs for the components can be seen in Table 3.4. The test programs are translated from Assembler to C-code and are subsequently executed according to their prioritization sequence. The following paragraphs describe the used test programs:

Multiplier The test is visualized in Figure 3.5a. For a N×M multiplier, the test patterns have the form $X = X_{N-1}...X_1X_0 = (c_7c_6c_5c_4)...(c_7c_6c_5c_4)(c_7c_6c_5c_4)$ and $Y = Y_{M-1}...Y_1Y_0 = (c_3c_2c_1c_0)...(c_3c_2c_1c_0)(c_3c_2c_1c_0)$. This results in a maximal number of 2^8 test patterns, which are able to detect 99% of Stuck-at and Coupling Faults [PKK11]. The results of 256 multiplications are accumulated in a signature register and compared with a pre-computed value at the end of the test. If this value

is equal, the test was successful and no faults are detected. Otherwise, a fault has occurred and is detected by this test.

Divider/Shifter The divider/shifter test can be seen in Figure 3.5b. The test uses a binary division, which means that a division by two is the same as a bit-shift-right operation, to test the divider unit. At first, an initial value (one) is multiplied 32 times and subsequently divided 32 times. After each operation, the result of the shift operation is compared with the mathematical operation result. If one of the 64 results differs, the test fails and hence faults are detected [PKK11].

ALU The test for the ALU is visualized in Figure 3.6a. The ALU can be tested with or without a multiplier. In this thesis, the test is performed without the usage of a multiplier by applying the basic logical operations (AND, OR, XOR and NOT) on different test patterns. The test patterns are generated by building all possible combination of one-bit operations ([0,0],[0,1],[1,0] and [1,1]). This testing methods are targeted on testing Stuck-at and Coupling Faults [PKK11].

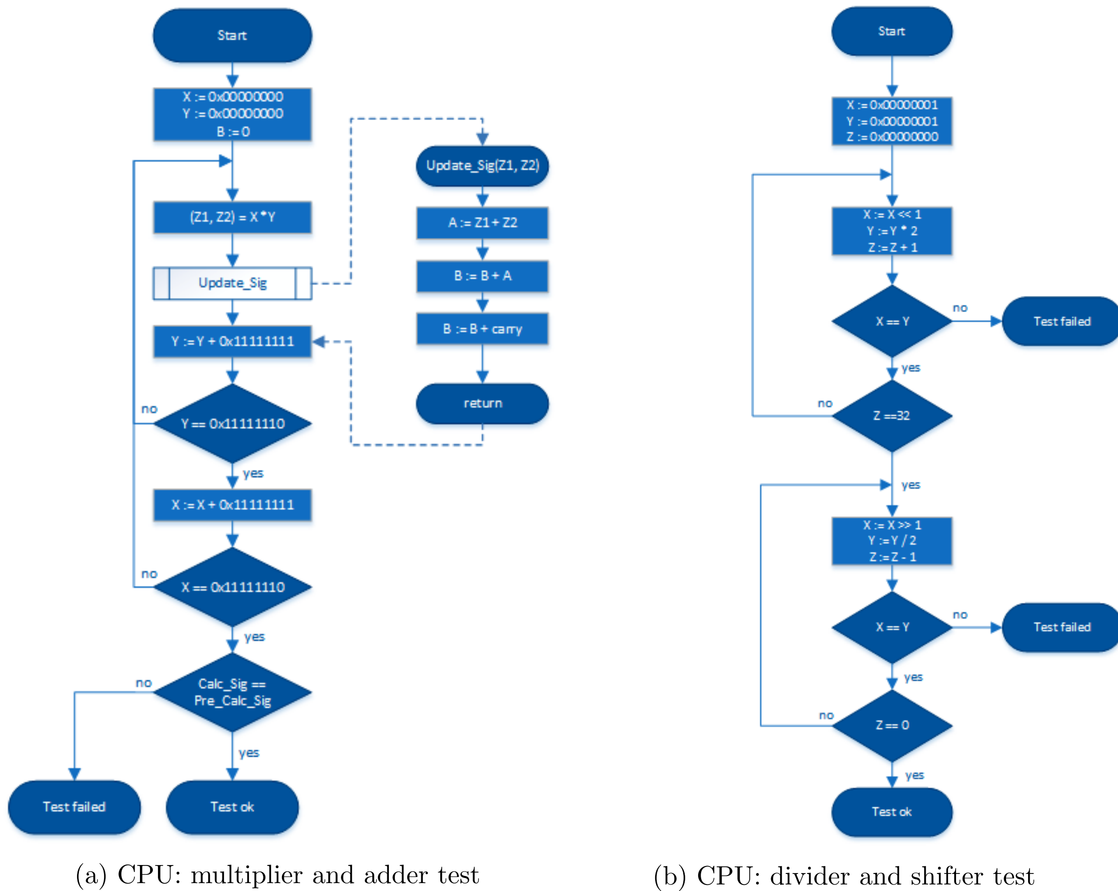
Adder The test for the adder is included in the test for the multiplier and is visualized in Figure 3.5a (signature update routine). After calculation of the signature for all 256 multiplications, the adder of the ALU is completely tested and achieves a test coverage of 99%, which is verified for various architectures. The test vectors aim on the detection of Stuck-at and Coupling Faults [PKK11].

SBSTs for Condition Code Flags

The condition code flags of the ARM9-CPU can be tested by setting and reseting the condition flags. An ARM9-CPU has the following five condition flags: Negative/Less than (N), Zero (Z), Carry/Borrow/Extend (C), Overflow (V) and Sticky overflow (Q), which can be set by arithmetic and logical operations. These flags are stored in the program status registers (CPSR and SPSR). Furthermore, the basic structure of this test is visualized in Figure 3.6b.

SBSTs for Registers and Register Banks

As already mentioned in the related work section, the registers have the same physical structure as small SRAMs and hence every RAM memory test can be applied in order to test the registers and register banks. In this thesis, the by the IEC 61508 proposed methods walking bit and Abraham memory test are compared against two different march tests (transparent, symmetric march C- and march SS). The reason for this is that the walking bit method is suggested by IEC 61508 to test the registers, but this test achieves only a medium fault coverage (about 90%). The Abraham test for variable memories is suggested by the IEC 61508 too and achieves the best test result (fault coverages $\geq$ 99%). The march tests in contrast, only achieve low fault coverages (about 60%), but the IEC 61508 only considers simple march tests. Furthermore, the well-rated Abraham test is only a modified march test sequence too. Due to these different fault coverages, the SFF rates for the register can vary, which is visualized in Table 3.4 with a greater than sign.



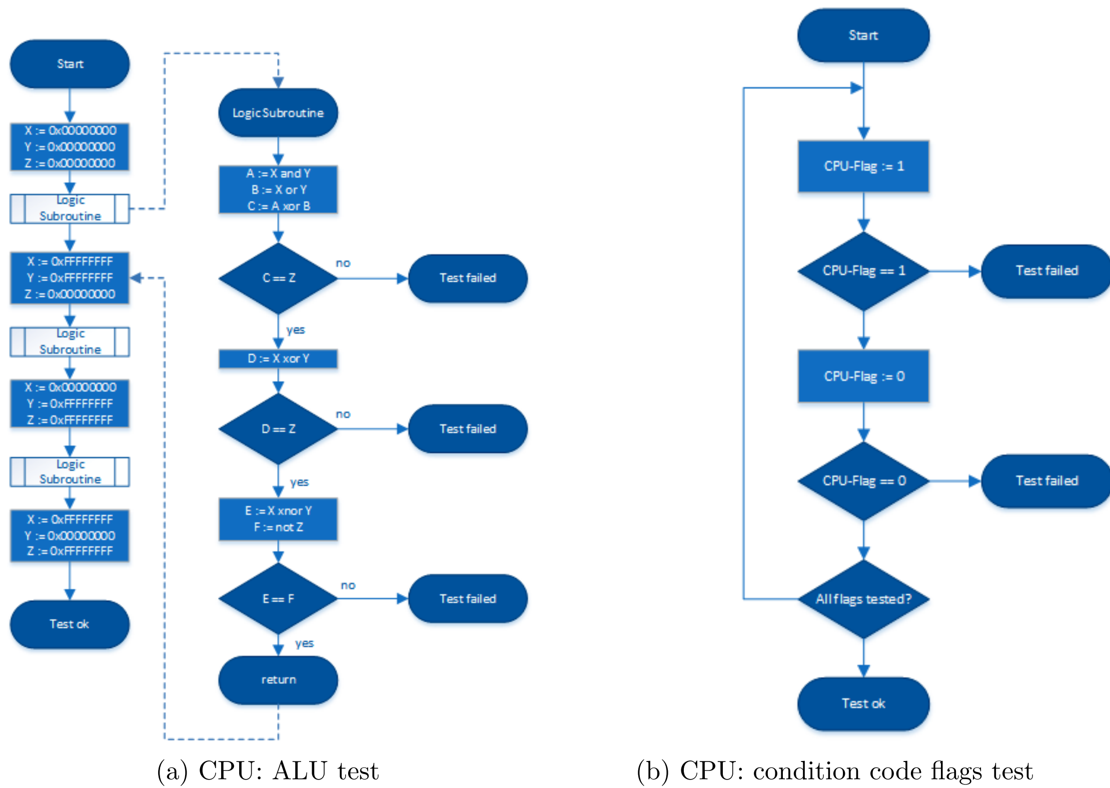
(a) CPU: multiplier and adder test

(b) CPU: divider and shifter test

Figure 3.5: Basic structure of the multiplier and adder as well as the divider and shifter test. This image is adapted from [PKK11].

The walking bit method tests all registers, which are accessible by instructions, by shifting a bit from Least-significant to most significant bit (walking bit). The basic structure of this test is shown in Figure 3.7. The different march tests as well as the Abraham test are described in the following section (SBST for RAMs).

Furthermore, the registers R13 to R15 in an ARM-CPU are reserved for special purpose. R13 contains the linking register, which stores the return address for a branch, R14 stores the stack-pointer and the R15 contains the Program Counter (PC). These registers are hard to test, because a modification of these registers can crash the program. For the linking register and the stack-pointer, one possible solution is to disable all interrupts, while a transparent test checks these registers. This can be done, because this registers are only needed during a branch and not during the execution of a subroutine. The R15 register, which contains the PC, can not be tested directly by a software, but this register is implicitly tested with every executed instruction.



(a) CPU: ALU test (b) CPU: condition code flags test

Figure 3.6: Basic structure of the ALU as well as the condition code flags test. This image is adapted from [PKK11].

Module	assumed gate-count	SFF
ALU	2%	99.9%
Adder and multiplier	39%	99%
Barrel-shifter	3%	100%
Register banks	42%	$\geq$ 90%
Other components	14%	50%
Overall	100%	$\geq$ 88.4%

Table 3.4: This table shows the SFFs for individual as well as for all tested CPU-core elements.

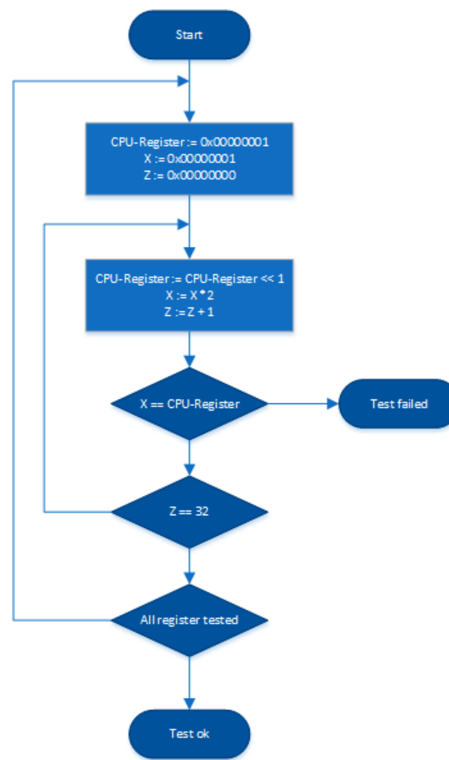


Figure 3.7: Basic structure of the Walking Bit method for testing the registers. This image is adapted from [PKK11].

3.3.2 SBSTs for RAMs

The issue by testing RAMs, especially DDR-SDRAMs, are the increasing memory sizes. Test lengths with n^2, where n is the number of bits, is not feasible for applying these tests in real-time systems. For example, a DDR-SDRAM with 128 MB and a test length of n^2, has an execution time of a few years. In order to overcome this problem, different march tests have been developed. In this section, the descriptions of the implemented march tests as well as the Abraham test is given and furthermore compared against their capability of detecting different FFMs.

Table 3.5 shows the three implemented memory tests with corresponding operation

sequences in the common march notation as well as the corresponding test lengths. It can be seen that the march C- is the simplest memory test, followed by the march SS and the complexest one is the abraham memory test. Table 3.6 shows that the complexest memory test does not yield the best fault coverage. The best fault coverage is achieved by the simpler march SS test and the march C- achieves the worst fault coverage ratio, but for transparent march C- tests it is also possible to detect unmodeled FFMs, because the memory content can randomly chance with every test cycle.

Name	**Algorithm**	**Test length**
March C-	$\{\Updownarrow(w0);\Uparrow(r0,w1);\Uparrow(r1,w0);\Downarrow(r0,w1);\Downarrow(r1,w0);\Updownarrow(r0)\}$	10n
March SS	$\{\Updownarrow(w0);\Uparrow(r0,r0,w0,r0,w1);\Uparrow(r1,r1,w1,r1,w0);\Downarrow(r0,r0,w0,r0,w1);\Downarrow(r1,r1,w1,r1,w0);\Updownarrow(r0)\}$	22n
Abraham (algorithm A)	$\{\Uparrow(w0);\Uparrow(r0,w1);\Downarrow(r1);\Uparrow(r1,w0);\Downarrow(r0);\Downarrow(r0,w1);\Uparrow(r1);\Downarrow(r1,w0);\Uparrow(r0);\Uparrow(r0,w1,w0);\Downarrow(r0);\Downarrow(r0,w1,w0);\Uparrow(r0);\Uparrow(w1);\Uparrow(r1,w0,w1);\Downarrow(r1);\Downarrow(r1,w0,w1);\Uparrow(r1)\}$	30n

Table 3.5: This table compares the different march algorithms as well as the abraham test and visualizes the corresponding operation sequences and test lengths.

FFM	**March C- (%)**	**March SS (%)**	**Abraham - Algorithm A (%)**
SF	100	100	100
TF	100	100	100
WDF	0	100	0
RDF	100	100	100
DRDF	0	100	100
IRF	100	100	100
CFst	100	100	100
CFds	66	100	75
CFtr	100	100	100
CFwd	0	100	75
CFrd	100	100	100
CFdrd	0	100	75
CFir	100	100	100

Table 3.6: This table shows the detected fault coverages for different FFMs for the march tests and the abraham test. The values for the march tests are taken from [HGR02]. The results for the tests are theoretical values and are not verified by fault injection experiments. The values are based on the comparison to other tests and operation sequences, which are theoretically needed to detect different FFMs.

Data Pattens for Inter- and Intra-word Coupling Faults

The most memory tests are designed and optimized for bit-oriented memories and hence they can not be applied to word-oriented memories. One possible solution in order to detect intra- and inter-word CFs is to apply a march test more than one time using different data backgrounds or data patterns. This results in inefficient execution times and limited fault coverage. In order to overcome these issues, the approach proposed by [GT98] is used and adapted for a 16bit memory interface in this thesis. For inter-word CFs, the word-wide organization of the memory chip will not influence the fault coverage and hence every inter-word CF can be detected, which is also detected by a bit-oriented memory test. Intra-word CFs can not be detected by applying bit-oriented march tests and hence special data patterns are needed. This data patterns are shown in the Tables 3.7 and 3.8 for intra-word idempotent Coupling Faults (CFids) and intra-word state Coupling Faults (CFsts). Intra-word inversion Coupling Faults (CFins) can not be masked and hence they can be detected by applying a word-oriented march test using any data pattern, because the value in the victim cell will be the inverse of the value expected by the read operation. For the detection of intra-word CFdsts, it is important to apply the data patterns in a special read/write sequence. These operation sequences with the corresponding data patterns are shown in Table 3.9.

At first sight, the number of applied data patterns are huge, but the execution time can be kept low, because the data patterns includes redundancy. For example the data patterns, which are needed to detect CFsts are already included in the data patterns, which are needed for the detection of CFdsts. For example the execution time for the march C-memory tests for detecting CFids changes from $14n$ to $(10 + 6 \cdot \lceil log_2 B \rceil) \cdot n/B$, where B is the word width of the memory and n the memory size in bits. For a 16-bit memory interface, the test lengths for the march C- to detect CFids is about $34n/16$.

Number	Data pattern (HEX)	Level	Number	Data pattern (HEX)	Level
1	0x0000	0	2	0xFFFF	0
3	0x0000	0	4	0x5555	0
5	0xAAAA	0	6	0x5555	0
7	0x3333	1	8	0xCCCC	1
9	0x3333	1	10	0x0F0F	2
11	0x00FF	2	12	0xF0F0	2
13	0xFF00	2	14	0x0FFF	2
15	0x000F	2			

Table 3.7: This table shows the used data patterns for detecting intra-word idempotent Coupling Faults (CFids) with a 16-bit memory interface. The Level-column shows the different cell pair levels (level 0 means each adjacent cell, level 1 each second adjacent cell and so on), which are used in [GT98] in order to detect all CFids.

Number	Data pattern	
	Normal	Inverse
1	0x0000	0xFFFF
2	0x5555	0xAAAA
3	0x3333	0xCCCC
4	0x0F0F	0xF0F0
5	0x00FF	0xFF00

Table 3.8: This table shows the used data patterns for detecting intra-word state Coupling Faults (CFsts) with a 16-bit memory interface.

Number	Operation	Data pattern (HEX)	Level	Number	Operation	Data pattern (HEX)	Level
1	w	0x0000	0	2	w	0xFFFF	0
3	r	0xFFFF	0	4	r	0xFFFF	0
5	w	0x0000	0	6	r	0x0000	0
7	r	0x0000	0	8	w	0x5555	0
9	w	0xAAAA	0	10	r	0xAAAA	0
11	r	0xAAAA	0	12	w	0x5555	0
13	r	0x5555	0	14	r	0x5555	0
15	w	0x3333	1	16	w	0xCCCC	1
17	r	0xCCCC	1	18	r	0xCCCC	1
19	w	0x3333	1	20	r	0x3333	1
21	r	0x3333	1	22	w	0x0F0F	2
23	w	0x00FF	2	24	r	0x00FF	2
25	r	0x00FF	2	26	w	0xF0F0	2
27	r	0xF0F0	2	28	r	0xF0F0	2
29	w	0xFF00	2	30	w	0x0FFF	2
31	r	0x0FFF	2	32	r	0x0FFF	2
33	w	0x000F	2	34	r	0x000F	2
35	r	0x000F	2				

Table 3.9: This table shows the operation sequences for detecting intra-word disturb Coupling Faults (CFdsts) with a 16-bit memory interface. The Level-column shows the different cell pair levels (level 0 means each adjacent cell, level 1 each second adjacent cell and so on), which are used in [GT98] in order to detect all CFdsts.

Transparent March Tests

In this thesis, the algorithm proposed in [LTW05] is used to convert a bit-oriented march test to a conventional transparent word-oriented one. The result of these conversions are shown in Table 3.10. The notation is the same as in that work, where D is the initial memory cell content and D_a is the bit-wise xor operation applied on the initial data and the data patterns a. The pattern a is for word-oriented memories a non-empty set of m data patterns and T_{m+1} is used for restoring the initial data of the memory cell.

Name	Algorithm
March C-	$T_0 : \{\Uparrow (rD_{a_0}, wD_{\bar{a}_0}); \Uparrow (rD_{\bar{a}_0}, wD_{a_0}); \Downarrow (rD_{a_0}, wD_{\bar{a}_0}); \Downarrow (rD_{\bar{a}_0}, wD_{a_0}); \Updownarrow (rD_{a_0})\}$ $T_1 : \{\Updownarrow (wD_{a_1}); \Uparrow (rD_{a_1}, wD_{\bar{a}_1}); \Uparrow (rD_{\bar{a}_1}, wD_{a_1}); \Downarrow (rD_{a_1}, wD_{\bar{a}_1}); \Downarrow (rD_{\bar{a}_1}, wD_{a_1}); \Updownarrow (rD_{a_1})\}$ ... $T_m : \{\Updownarrow (wD_{a_m}); \Uparrow (rD_{a_m}, wD_{\bar{a}_m}); \Uparrow (rD_{\bar{a}_m}, wD_{a_m}); \Downarrow (rD_{a_m}, wD_{\bar{a}_m}); \Downarrow (rD_{\bar{a}_m}, wD_{a_m}); \Updownarrow (rD_{a_m})\}$ $T_{m+1} : \{\Updownarrow (wD_{a_0})\}$
March SS	$T_0 : \{\Uparrow (rD_{a_0}, rD_{a_0}, wD_{a_0}, rD_{a_0}, wD_{\bar{a}_0}); \Uparrow (rD_{\bar{a}_0}, rD_{\bar{a}_0}, wD_{\bar{a}_0}, rD_{\bar{a}_0}, wD_{a_0}); \Downarrow (rD_{a_0}, rD_{a_0}, wD_{a_0}, rD_{a_0}, wD_{\bar{a}_0}); \Downarrow (rD_{\bar{a}_0}, rD_{\bar{a}_0}, wD_{\bar{a}_0}, rD_{\bar{a}_0}, wD_{a_0}); \Updownarrow (rD_{a_0})\}$ ... $T_m : \{\Updownarrow (wD_{a_m}); \Uparrow (rD_{a_m}, rD_{a_m}, wD_{a_m}, rD_{a_m}, wD_{\bar{a}_m}); \Uparrow (rD_{\bar{a}_m}, rD_{\bar{a}_m}, wD_{\bar{a}_m}, rD_{\bar{a}_m}, wD_{a_m}); \Downarrow (rD_{a_m}, rD_{a_m}, wD_{a_m}, rD_{a_m}, wD_{\bar{a}_m}); \Downarrow (rD_{\bar{a}_m}, rD_{\bar{a}_m}, wD_{\bar{a}_m}, rD_{\bar{a}_m}, wD_{a_m}); \Updownarrow (rD_{a_m})\}$ $T_{m+1} : \{\Updownarrow (wD_{a_0})\}$
Abraham	$T_0 : \{\Uparrow (wD_{a_0}); \Uparrow (rD_{a_0}, wD_{\bar{a}_0}); \Downarrow (rD_{\bar{a}_0}); \Uparrow (rD_{\bar{a}_0}, wD_{a_0}); \Downarrow (rD_{a_0}); \Downarrow (rD_{a_0}, wD_{\bar{a}_0}); \Uparrow (rD_{\bar{a}_0}); \Downarrow (rD_{\bar{a}_0}, wD_{a_0}); \Uparrow (rD_{a_0}); \Uparrow (rD_{a_0}, wD_{\bar{a}_0}, wD_{a_0}); \Downarrow (rD_{a_0}); \Downarrow (rD_{a_0}, wD_{\bar{a}_0}, wD_{a_0}); \Uparrow (rD_{a_0}); \Uparrow (wD_{\bar{a}_0}); \Uparrow (rD_{\bar{a}_0}, wD_{a_0}, wD_{\bar{a}_0}); \Downarrow (rD_{\bar{a}_0}); \Downarrow (rD_{\bar{a}_0}, wD_{a_0}, wD_{\bar{a}_0}); \Uparrow (rD_{\bar{a}_0})\}$ ... $T_m : \{\Uparrow (wD_{a_m}); \Uparrow (rD_{a_m}, wD_{\bar{a}_m}); \Downarrow (rD_{\bar{a}_m}); \Uparrow (rD_{\bar{a}_m}, wD_{a_m}); \Downarrow (rD_{a_m}); \Downarrow (rD_{a_m}, wD_{\bar{a}_m}); \Uparrow (rD_{\bar{a}_m}); \Downarrow (rD_{\bar{a}_m}, wD_{a_m}); \Uparrow (rD_{a_m}); \Uparrow (rD_{a_m}, wD_{\bar{a}_m}, wD_{a_m}); \Downarrow (rD_{a_m}); \Downarrow (rD_{a_m}, wD_{\bar{a}_m}, wD_{a_m}); \Uparrow (rD_{a_m}); \Uparrow (wD_{\bar{a}_m}); \Uparrow (rD_{\bar{a}_m}, wD_{a_m}, wD_{\bar{a}_m}); \Downarrow (rD_{\bar{a}_m}); \Downarrow (rD_{\bar{a}_m}, wD_{a_m}, wD_{\bar{a}_m}); \Uparrow (rD_{\bar{a}_m})\}$ $T_{m+1} : \{\Updownarrow (wD_{a_0})\}$

Table 3.10: This table shows the transparent march C-, march SS and abraham test.

Transparent Symmetric March Tests

In this thesis, the algorithm proposed in [YH99] is used to convert a transparent word-oriented march test to a symmetric one. The result of these conversions are shown in Table 3.11. The advantage of a symmetric approach is that the signature prediction phase can be skipped and hence the execution time of these tests can be decreased. The signatures for the transparent memory tests, presented in Table 3.11, are about $m \cdot \#read_operations$, which as example results in $5 \cdot m$ additional read operations for the march C-. The abraham test can not be easily converted to a symmetric approach, because this test does not innately contain symmetries. However, the abraham test is only used as a reference to the IEC 61508 and is not considered to be implemented in a real system.

Name	Algorithm
March C-	$T_0 : \{\Uparrow (rD_{\bar{a}_0}); \Uparrow (rD_{a_0}, wD_{\bar{a}_0}); \Uparrow (rD_{\bar{a}_0}, wD_{a_0}); \Downarrow (rD_{a_0}, wD_{\bar{a}_0}); \Downarrow (rD_{\bar{a}_0}, wD_{a_0}); \Downarrow (rD_{a_0})\}$ $T_1 : \{\Uparrow (rD_{a_1}, wD_{\bar{a}_1}); \Uparrow (rD_{\bar{a}_1}, wD_{a_1}); \Downarrow (rD_{a_1}, wD_{\bar{a}_1}); \Downarrow (rD_{\bar{a}_1}, wD_{a_1}); \Downarrow (rD_{a_1})\}$ ... $T_m : \{\Uparrow (rD_{a_m}, wD_{\bar{a}_m}); \Uparrow (rD_{\bar{a}_m}, wD_{a_m}); \Downarrow (rD_{a_m}, wD_{\bar{a}_m}); \Downarrow (rD_{\bar{a}_m}, wD_{a_m}); \Downarrow (rD_{a_m})\}$ $T_{m+1} : \{\Updownarrow (wD_{a_0})\}$
March SS	$T_0 : \{\Uparrow (rD_{\bar{a}_0}); \Uparrow (rD_{a_0}, rD_{a_0}, wD_{a_0}, rD_{a_0}, wD_{\bar{a}_0}); \Uparrow (rD_{\bar{a}_0}, rD_{\bar{a}_0}, wD_{\bar{a}_0}, rD_{\bar{a}_0}, wD_{a_0}); \Downarrow (rD_{a_0}, rD_{a_0}, wD_{a_0}, rD_{a_0}, wD_{\bar{a}_0}); \Downarrow (rD_{\bar{a}_0}, rD_{\bar{a}_0}, wD_{\bar{a}_0}, rD_{\bar{a}_0}, wD_{a_0}); \Downarrow (rD_{a_0})\}$ ... $T_m : \{\Uparrow (rD_{\bar{a}_m}, wD_{\bar{a}_m}); \Uparrow (rD_{a_m}, rD_{a_m}, wD_{a_m}, rD_{a_m}, wD_{\bar{a}_m}); \Uparrow (rD_{\bar{a}_m}, rD_{\bar{a}_m}, wD_{\bar{a}_m}, rD_{\bar{a}_m}, wD_{a_m}); \Downarrow (rD_{a_m}, rD_{a_m}, wD_{a_m}, rD_{a_m}, wD_{\bar{a}_m}); \Downarrow (rD_{\bar{a}_m}, rD_{\bar{a}_m}, wD_{\bar{a}_m}, rD_{\bar{a}_m}, wD_{a_m}); \Downarrow (rD_{a_m})\}$ $T_{m+1} : \{\Updownarrow (wD_{a_0})\}$

Table 3.11: This table shows the transparent symmetric march C- and march SS test.

Chapter 4

Implementation

This chapter covers the implementation details about the fault injection framework as well as the details about the Software-Based Self-Tests (SBSTs). Furthermore, a fault injection example is stated, which shows the workflow of executing a fault injection experiment including the fault detection by SBST.

4.1 Fault Injection Framework

This section gives an overview to the QEMU implementation and explains how each fault injection component is integrated in the QEMU-framework and how the interact among themselves.

4.1.1 Implementation Overview of QEMU

QEMU is an open-source project, where the core-components of QEMU consist of more than 1000 and the whole project more than 6000 files. This section aims to deliver insight into the implementation of QEMU's simulation process, which is needed in the next section to understand the integration of the fault injection components.

The QEMU's root directory (symbolized by /) contains the major source files for the start of the execution (*/vl.c*, */cpus.c*, */exec-all.c*, */exec.c*, */cpu-exec.c*). The code for the hardware emulation is localized in the directory */hw/*. The target (or guest)-specific files, which is the currently emulated processor architecture, can be found in */target-xyz*, where *xyz* is the target architecture (arm). The host (TCG)-specific files are stored in */xyz/*, where *xyz* is the host architecture (i386 or x86_64).

The following paragraph lists the major files, which are used for the emulation process, and briefly describes their functionality.

/vl.c: Includes the main-function, which initializes the virtual machine with the given specifications and starts the CPU execution.

/cpu-exec.c: Implements the functions for finding and executing the next Translation Block (TB). Starts the generation of a new TB, if the next TB cannot be found.

/target-xyz/translate.c: Converts the guest ISA to architecture independent TCG-operations (instruction decoding).

/tcg/tcg.c: Includes the main functions for the functionality of the TCG.

/tcg/xyz/tcg-target.c: Converts architecture independent TCG-operations to host-specific ISA.

QEMU implements a Software Memory Management Unit (SoftMMU), which supports a fast finding of the mapping between Guest Physical Address (GPA) and Host Virtual Address (HVA). If the guest operation system (OS) tries to access a virtual address, QEMU has to translate this Guest Virtual Address (GVA) to GPA. Then, QEMU has to find the corresponding physical page descriptor in the page table (page table walk). This returns a physical offset, which is added to GVA to get the HVA. The main idea of the SoftMMU is to use a Translation Look-aside Buffer (TLB), which stores the corresponding physical offset of a GVA. If a translation from GVA to HVA is needed, QEMU searches in the TLB for the corresponding physical offset first. If the entry is found, QEMU can add this offset to GVA to get the resulting HVA. Otherwise, QEMU has to start a page table walk to find the corresponding offset. The TLB is accessed by some bits of the GVA and hence the TLB entries have to be flushed if a process is switched. The same idea is used for the I/O-emulation, where a separate IOTLB stores the corresponding index of the I/O-emulation function for the accessed GVA. The function calls of fetching code from guest memory, would look as follows: *cpu_exec(...)* → *tb_find_fast(...)* → *tb_find_slow(...)* → *get_phys_addr_code(...)* → (if TLB miss) *ldup_code(...)*{*softmmu_header.h*} → *__ldl_mmu(...)*{*softmmu_template.h*} → *tlb_fill(...)* → *cpu_arm_handle_mmu_fault(...)* → *tlb_set_page(...)* → *tlb_set_page_exec(...)*.

The execution trace of QEMU is visualized in Figure 4.1 with the notation *function_name(...)*{*/directory/file_name*}. As already mentioned, the whole emulation process starts in *main(...)*{*/vl.c*}. In here, the arguments are parsed and the virtual machine is set up. After this, main calls *main_loop(...)*{*/vl.c*}, which creates an execution thread. This thread runs a do-while loop and can be stopped by calling *qemu_vmstop_requested()*, killed by *qemu_shutdown_requested()* and so on. After a few function calls, which are not relevant for the basic understanding and which are not mentioned here, the function *qemu_tcg_cpu_thread_fn(...)*{*/cpus.c*} is called. This function calls *tcg_exec_all(...)*{*/cpus.c*} in an infinity loop. *tcg_exec_all* selects the active CPU in a round-robin fashion and due to this reason, QEMU cannot emulate a real multi-core processor, because all cores are sequentially executed. After this, QEMU chooses the corresponding *struct CPUState*{*/target-xyz/cpu.h*} structure and parses it to *tcg_cpu_exec(...)*{*/cpus.c*}. This structure holds information about the CPU state (standard registers, exceptions, interrupt handling, processor features and emulator specific internal variables and flags). Then, tcg_cpu_exec basically calls *cpu_exec(...)*{*/cpu-exec.c*}. *cpu_exec* is the main execution loop, which handles exceptions and generates the Translation Blocks (TB). This function calls *tb_find_fast(...)*{*/cpu-exec.c*}, which initiates the search for the next TB. The TBs are stored as *struct TranslationBlock*{*/include/exec/exec-all.h*}, which contains the pc, flags corresponding to the TB, variables for finding the chained TBs and a pointer to the TB that jumps into this TB. *tb_find_fast* gets the program counter (PC) by calling *cpu_get_tb_cpu_state()*, which parses the value from the CPUState structure (env). The PC-value is used to search the index of the TB in a hash table (*tb_jmp_cache[]*). This TB index is subsequently verified and if the TB index is invalid, the function calls *tb_find_slow(...)*{*/cpu-exec.c*}. Otherwise, the TB index is returned and executed by calling the functions *cpu_tb_exec(...)*{*/cpu-exec.c*} and *tcg_qemu_tb_exec(...)*{*/cpu-exec.c*}. If *tb_find_fast* fails that means the TB

is not stored in the TB cache, *tb_find_slow* has to find the TB by searching through physical memory mappings. For this, a physical PC of the Guest OS is delivered by calling *get_page_addr_code(...)*{*/cputlb.c*}, which is used to get the TB by searching in a hash table (*tb_phys_hash(tb_phys_hash_func(pc))*. If the TB is not found or is invalidated, QEMU has to generate a new TB by calling *tb_gen_code(...)*{*/translate-all.c*}, otherwise the TB is added to the TB cache (*tb_jmp_cache[]*) and is subsequently executed. *tb_gen_code* allocates a new TB, which corresponds to the physical PC stored in the CPUState structure. Subsequently, the function *cpu_gen_code(...)*{*/translate-all.c*} is called, which calls the target-specific function *gen_intermediate_code(...)*{*/target-arm/translate.c*}. *gen_intermediate_code* initializes the target-specific instruction decoding and calls *gen_intermediate_code_internal(...)*{*/target-arm/translate.c*}, which calls *disas_arm_insn(...)*{*/target-arm/translate.c*}. *disas_arm_insn* implements the functionality for decoding target-specific instructions to TCG-Ops in the arm-mode (a long switch-case). Similar functions are available for arm-thumb mode decoding and for the NEON-extension. *gen_intermediate_code_internal* groups the decoded instruction in the TCG code buffer till a branch or an interrupt occurs. Afterward, QEMU returns to *cpu_gen_code*, which calls *tcg_gen_code(...)*{*/tcg/tcg.c*}. *tcg_gen_code* performs the conversation from TCG-Ops to host-specific code, which are subsequently executed, grouped in the new TB and stored in the TB-cache.

The next paragraph explains how the QEMU-monitor can be extended with new commands. For this, QEMU defines the QEMU Machine or Monitor Protocol (QMP) and the Human Monitor Protocol (HMP), which are two internal protocols of QEMU. If the user calls a command from monitor by typing in a command (string), the monitor handler decides, which function should be called in the HMP. All functions have *hmp_* as prefix and are stored in *hmp.c* (*hmp_*(...)*{*/hmp.c*}). The HMP functions should be converted to use internal QMP commands (QEMU coding requirements). These QMP functions have the prefix *qmp_* and are principally stored in *qmp.c* (*qmp_*(...)*{*/qmp.c*}). The QMP can be seen as a kind of interface to QEMU, which implements the required functionality (for example the querying of register content, which is finally implemented in the target-specific part). Instead of taking a string like the HMP, QMP interacts with objects, which are automatically generated by data structures defined in a JSON format. The process of including a new command to QEMU, would be done by following steps:

1. Define the command- and type-specification in the QAPI schema file ({/qapi-schema.json}).
2. Implement the QMP command in */qmp.c* or in another QEMU subdirectory.
3. Write the HMP command in */hmp.c*, which calls the QMP-command.

A detailed example about implementing a new command is given in Section 4.1.2 - Monitor.

4.1.2 Implementation of the Fault Injection Components in QEMU

This section gives a detailed explanation, how every FI-component is implemented and how they interact among themselves. Furthermore, this interaction is visualized in Figure 4.2 and the sequence of a fault injection example is shown in Figure 4.3.

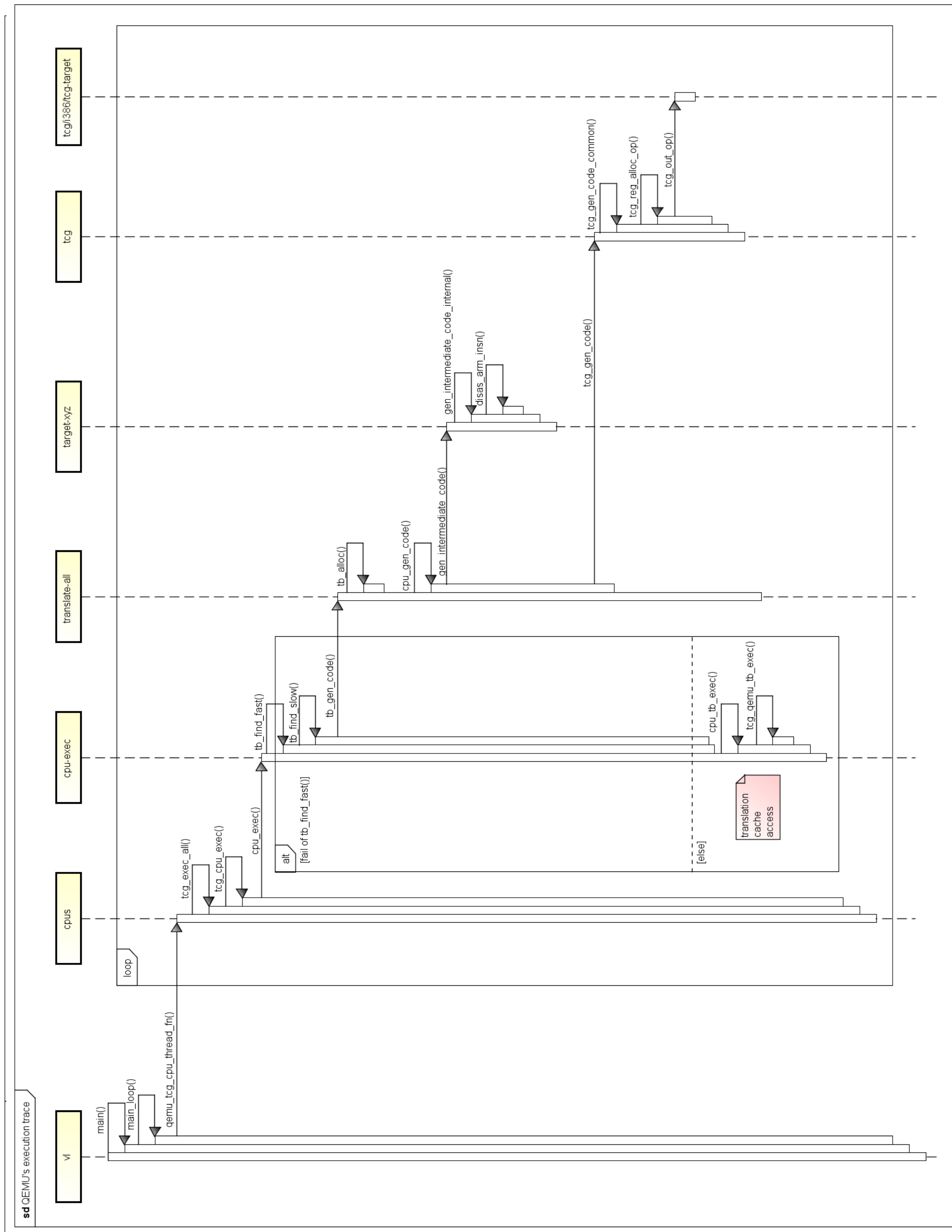


Figure 4.1: This figure shows the simplified emulation process of QEMU for an ARM-setup (based on QEMU version 1.7.0).

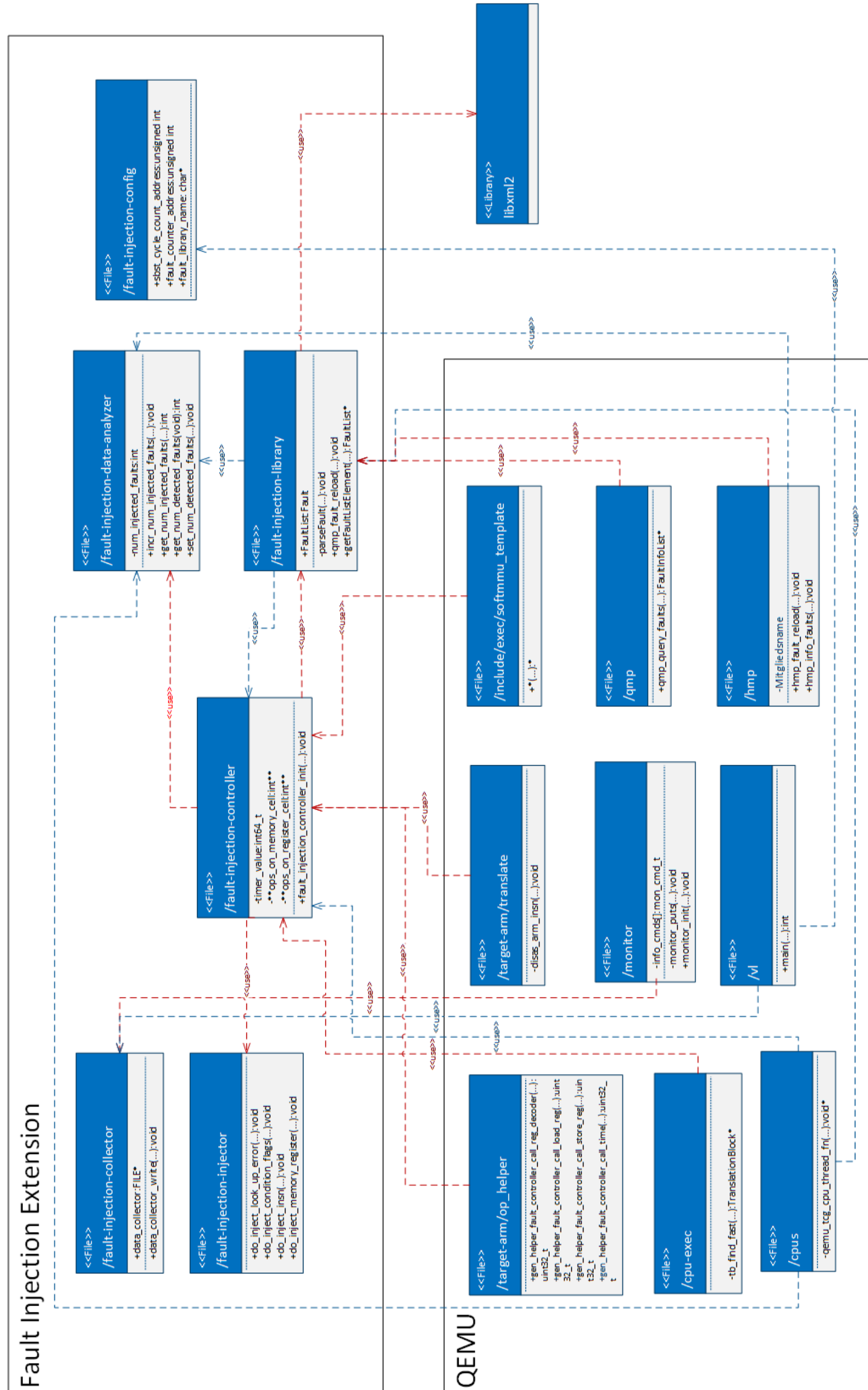


Figure 4.2: This figure shows the dependencies of FIES based on a UML-class diagram, which is modified for visualizing c-code. The classes represent c-files and the private modifier of methods represent static functions. The class members are static variables for private class members and global variables symbolize public class members. The use-arrows model the dependencies of each file, which are basically includes of c-header-files.

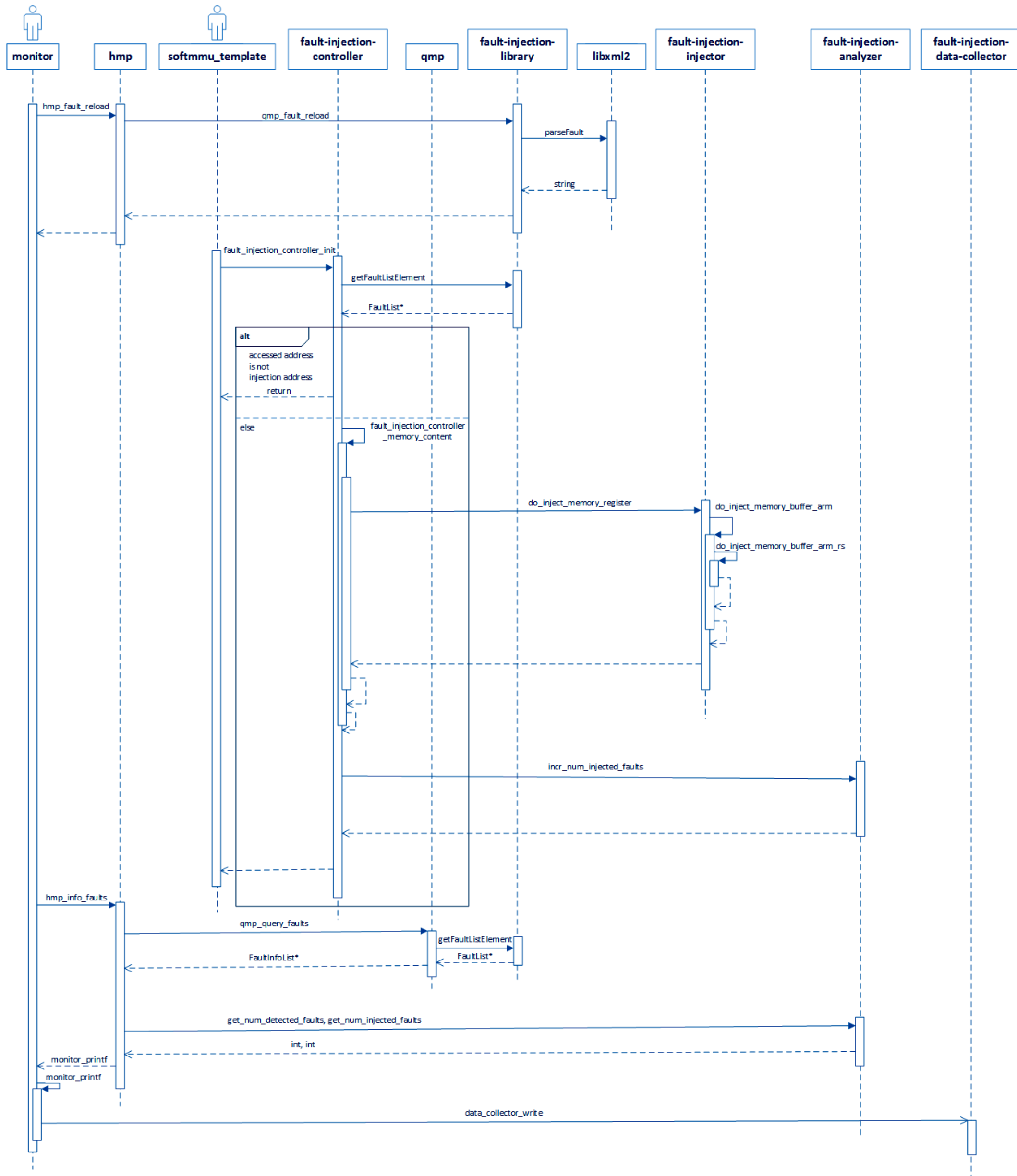


Figure 4.3: This figure visualizes the function call sequence of a fault injection into a memory cell. The experiment starts with the loading of a fault library initiated by the user through the monitor. Afterwards, the memory fault is triggered, if an access of the memory address occurs. The controller decides with the help of the fault library, which fault should be injected and calls the appropriate function in the fault-injector component. After returning from fault injector, the fault controller increments the injected fault counter in the analyzer module. If the user queries the fault statistic by typing the appropriate command in the monitor, FIES collects the fault parameters from fault library and the number of injected and detected faults from fault analyzer, computes the fault coverages and writes these results to the monitor, which visualizes the information.

Target System

The target system is responsible for the simulation of the emulated hardware. In this thesis the i.MX28 EVK PCB REV D from Freescale Semiconductor [Fre11] is used, but this board is not supported by QEMU yet. Hence, the board has to be defined and further hardware-specific modules have to be simulated in software. First of all, the CPU-type and the number of cores have to be defined, which is simply calling *cpu_arm_init* with the appropriate cpu-type as string (*arm926* in this case). The next step is the definition of the memory mappings for SRAM, RAM, ROM and SRAM alias. This is realized by calling the function *memory_region_init_ram(...){/memory.c}*, which defines the name, owner and the size for SRAM or RAM. Next the memory region is registered in the virtual machine as global with the function *vmstate_register_ram_global(...){/savevm.c}* and subsequently added to the system memory by calling *memory_region_add_subregion(...){/memory.c}*, which defines the addresses for accessing the given memory region. The same process is done for all other memory regions (SRAM, RAM, ROM and SRAM alias) with the exception that the memory region for ROM is set to read-only by calling *memory_region_set_readonly(...){/memory.c}* and the SRAM alias is initialized by calling *memory_region_init_alias(...){/memory.c}* instead of *memory_region_init_ram(...){/memory.c}*.

Then, the Digital Control module of Freescale has to be added to QEMU, which is done by writing *imx28_digcntr_write(...){/hw/misc/imx28_DigCntr.c}* and *imx28_digcntr_read(...){/hw/misc/imx28_DigCntr.c}*. This module is basically a collection of configuration registers, which are used for reading and modifying the parameters for the Default First Level Page Table (DFLPT) module implemented by Freescale board[1].

The next included module is the DFLPT, which is a hardware-based approach of Freescale to increase the performance and decrease the power consumption of the board for the fetching of the first-level page table. The basic idea is to save the first-level page table in this module instead of saving it in the small onchip-SRAM. The corresponding function for the functionality of this DFLPT are localized in **(...){/hw/misc/imx28_dflpt.c}*.

These two components are created by calling *qdev_create(...){/hw/core/qdev.c}* and *qdev_init_nofail(...){/hw/core/qdev.c}* and are subsequently added to the system memory with their corresponding memory addresses by calling *sysbus_mmio_map(...){/hw/core/sysbus.c}*.

Furthermore, the components for the Global Interrupt Controller (GIC) and the Timer have to be added to QEMU. The address of the GIC is defined with the help of *sysbus_create_varargs(...){/hw/core/sysbus.c}*. The GIC-implementation itself is implemented at **(...){/hw/intc/imx28_gic.c}* and defines a set of registers, which defines if a special interrupt is active, where the vector address of the active interrupt can be found, status about the current interrupt and the global configuration of the interrupts. Furthermore, this component sets the interrupt active by calling *qemu_set_irq(...){/hw/core/irq.c}*, which calls the appropriate implementation of the current IRQ-handler. Last, this component sets the register status values for the active interrupt. *sysbus_create_varargs* returns a DeviceState structure, which is used to bind special IRQ-GPIO-pins to the corresponding IRQ-handler. This is done for the debug and application UART and for the timer component, which are explained next.

[1] In general, it contains configuration registers for other components too, but they are not necessary for the basic functionality of SafeRTOS and hence they are not implemented in this work.

The realization of the timer can be found in **(...){/hw/core/timer/imx28_rotary_decoder.c}*, which is only used for writing and reading a configuration register, and **(...){/hw/core/timer/imx28_timer.c}*, which is the main file. In here, the function for reading and writing the timer-value or other timer-specific configuration and status registers are implemented as well as the functions for incrementing the timer ticks and updating the timer status by calling the functions *qemu_irq_raise(...){/hw/core/irq.c}* and *qemu_irq_lower(...){/hw/core/irq.c}* for raising and lowering the IRQ level.

The debug and application UART as well as the four timers are bounded to their corresponding GPIO-pin by calling *sysbus_create_simple(...){/include/hw/sysbus.c}* with the name of the module, the MMIO-address and the corresponding GPIO-pin (*qdev_get_gpio_in(dev, gpio_num)*) as argument. The UART-module is already implemented in QEMU and hence it has not to be included to QEMU.

The last step for the emulation of the hardware, is the specification of the kernel parameters, which is done in a *struct arm_boot_info{/include/hw/arm/arm.c}* structure. After parsing the RAM size, number of CPUs, kernel filename, kernel cmdline, initrd filename and board id from QEMU startup arguments, this structure is parsed to *arm_load_kernel(...){/include/hw/arm/boot.c}*, which loads the kernel.

Workload Generator and Workload Library

The workload generator and library are realized by the applications (SBSTs), which are scheduled by the target OS (SafeRTOS) and hence the implementation details for these components are stated in the next section (Section 4.2).

Data Collector

The data collector holds the file pointer, where the whole monitor in- and output is stored to. Furthermore, this component implements the function for writing to this file and can be found under *data_collector_write(...){/fault-injection-collector.c}*. The file is opened in the init-function of the monitor (*monitor_init(...){/monitor.c}*) and the data_collector_write function is called every time when *monitor_puts(...){/monitor.c}* is called.

Data Analyzer

The data analyzer is a container, which holds the number of injected and detected faults and provides the functions for incrementing, setting, reseting and getting these variables. The corresponding code can be found under **(...){/fault-injection-data-analyzer.c}*. The number of injected faults is incremented by the fault controller and reseted by the fault-library if a new fault experiment is loaded with the corresponding monitor command. The number of detected faults is also reseted by the fault-library and incremented in the SBSTs executed by SafeRTOS. This variable is stored in a certain RAM-address, which is parsed as argument at the start of QEMU. This RAM-address is read from QEMU, if the fault analyzer is requested to deliver the number of detected faults.

Fault Library

The fault library (**(...){/fault-injection-library.c}*) holds the fault parameters in a linked list, which is parsed from a XML-encoded file (fault configuration). The fault configuration is parsed with the help of an external library (libxml2). This component also provides a function, which checks the configuration file for correct defined data types (for example if the component-tag contains a predefined keyword), but it does not check if all needed parameters are specified. Furthermore, this file contains the implementation of the command for loading the fault configuration file (*qmp_fault_reload(...){/fault-injection-library.c}*), which is called from HMP.

The XML-encoded fault configuration file supports the following XML-tags, which are used to describe faults:

<*injection*>: Defines the start and end of the fault configuration list.

<*fault*>: Defines the start and end of a fault description.

<*id*>: Defines the fault id (positive, non-zero integer value).

<*component*>: Defines the target component of a fault (CPU, RAM or REGISTER).

<*target*>: Defines the target point of a fault (for CPU: INSTRUCTION DECODER, INSTRUCTION EXECUTION or CONDITION FLAGS; for REGISTER: ADDRESS DECODER or REGISTER CELL; for RAM: ADDRESS DECODER or MEMORY CELL).

<*mode*>: Defines the fault mode (

- **Condition Flags:** VF, ZF, CF, NF, QF
- **General fault modes:** NEW VALUE, SF, BIT-FLIP
- **Operation-dependent, static faults:** TF0, TF1, WDF0, WDF1, IRF0, IRF1, DRDF0, DRDF1, RDF0, RDF1
- **Operation-dependent, dynamic faults:** RDF00, RDF01, RDF10, RDF11, IRF-00, IRF01, IRF10, IRF11, DRDF00, DRDF01, DRDF10, DRDF11
- **Coupling faults:** CFST00, CFST01, CFST10, CFST11, CFTR00, CFTR01, CFTR-10, CFTR11, CFWD00, CFWD01, CFWD10, CFWD11, CFRD00, CFRD01, CFRD10, CFRD11, CFIR00, CFIR01, CFIR10, CFIR11, CFDR00, CFDR01, CFDR10, CFDR11, CFDS0W00, CFDS0W01, CFDS0W10, CFDS0W11, CFDS-1W00, CFDS1W01, CFDS1W10, CFDS1W11, CFDS0R00, CFDS0R01, CFDS-1R10, CFDS1R11

).

<*trigger*>: Defines the triggering method (ACCESS, TIME, or PC)

<*timer*>: Defines the start time for intermittent and transient faults (string, that contains a positive, non-zero time value in ms, us or ns - e.g. 1000MS).

<*type*>: Defines the fault type (can be TRANSIENT, PERMANENT or INTERMIT-TEND).

<*duration*>: Defines the duration for intermittent and transient faults (string that contains a positive, non-zero time value in ms, us or ns - e.g. 1000MS).

<*interval*>: Defines the interval for intermittent faults (string that contains a positive, non-zero time value in ms, us or ns - e.g. 1000MS).

<*params*>: Defines the start and end of the parameter description.

> **<*address*>:** Contains the register or memory address (hexadecimal value!) for access- or time- triggered faults. In the case of a PC-triggered fault, this tag contains the PC-value that triggers the fault.
>
> **<*mask*>:** Defines the mask for determining the position, where the fault should become active. In the case of a NEW VALUE-mode, this tag contains the new value, which should be written to the destination target.
>
> **<*cf_address*>:** Defines the coupling cell address for register and memory coupling faults (hexadecimal value!). If the address- and the cf_address-tag are equal, the fault configuration defines a intra-coupling fault, otherwise it describes a inter-coupling fault.
>
> **<*instruction*>:** Defines the instruction number, which should be replaced for CPU-INSTRUCTION DECODER and CPU-INSTRUCTION EXECUTION faults. 0xDEADBEEF injects a NOP operation for simulating a „no-execution“. In the case of a PC-triggered fault, this tag contains the faulty memory or register address (hexadecimal value!), because the address tag contains the PC-value for triggering the fault.
>
> **<*set_bit*>:** Defines if a by the mask selected bit should be set or reseted in a state fault (for example, a set_bit of 0x2 and a mask of 0x3 defines that the first bit is set to zero and the second bit is set to one). In the case of an intra-coupling fault, this tag defines the aggressor-bit.

For the correct behavior, it is mandatory to correctly define a fault, which is described in the next itemize:

- All strings have to be upper-case.
- The id-, component-, target-, mode-, trigger- and type-tag must be defined.
- If the mode is „PC“, timer, duration, type and interval must not be defined.
- If the mode is „TIME“or „ACCESS“, type must be defined.
- If the type is „TRANSIENT“or „INTERMITTEND“, timer and duration must be defined.
- If the type is „INTERMITTEND“, interval must be defined.
- The address- and mask-tag must be defined for register and memory faults.
- The cf_address must be defined for register- and memory-coupling faults.

- If the fault is configured as PC-triggered fault or the fault is a CPU-instruction decoding and execution faults, the instruction-tag must be defined.
- For state or intra-coupling faults, the set_bit tag must be defined.
- The mask-tag must be defined for faults with „NEW VALUE“ mode.

An example of a fault configuration file is given in Appendix A.7

Monitor

The monitor is responsible for the visualization of the by commands requested data, which is already pre-implemented in QEMU, and the correct decoding and forwarding of monitor-commands. The monitor is extended with two commands in this thesis: The command for loading a fault library (*fault_reload <path-to-fault-configuration>*) and an extension of the pre-implemented info-command for querying the fault statistics (*info faults*). A further useful command for the fault injection is *info registers*, which shows the register content in the monitor output (already implemented by QEMU).

For the fault_reload command, the implementation of the function *hmp_fault_reload(...)* {*/hmp.c*} has to be added and has to made available to the monitor by adding the command in {*/hmp-commands.hx*}. The content of this file defines the needed command parameters (in *Haxe*) for the automatic generation of source-code. It defines the name of the command, which should be typed in the monitor, the name and data type of the arguments, the string, which should be shown when *help <command>* is called and the name of the function, which should be called in the HMP-file. The corresponding code can be seen in Listing A.1

The hmp_fault_reload function gets the filename of the fault configuration file by using *qdict_get_str(...)*{*/qobject/qdict.c*}, which returns the by a keyword („filename“) requested variable as defined argument type. This returned filename is parsed to *qmp_fault_reload(...)* {*/fault-injection-library.c*}, which loads the fault configuration.

The implementation of *info faults* is more difficult, because this function has to return a data type. At first, the user-defined data type has to be added to {*/qapi-schema.json*}, which is shown in Listing A.2. The keyword „type“ defines a new QAPI data type and the keyword „data“ defines the corresponding data names and types. In this case, a structure with the name *FaultInfo* is generated, which contains the members *component* as string (str), *is_active* as integer (int), a nested structure with the name *params* and so on. The next step is to define the name and the return type of the implemented function *qmp_query_faults(...)*{*/qmp.c*}, which is visualized in Listing A.3. The keyword „returns“ is used to define the data return type of the command and the keyword „command“ determines the name of the generated function, which is added to the qmp_-prefix. In this example the function name is *qmp_query_faults(...)* implemented in {*/qmp.c*} and returns a *FaultInfoList* (linked list of *FaultInfos*, which is shown in Listing A.4). In this function, the whole fault status content, stored in a linked list in the fault library, is parsed to the FaultInfoList. An important note is that the strings as well as the FaultInfo-elements have to be dynamically allocated. The head of the FaultInfoList is subsequently returned by this function. The corresponding HMP-function (*hmp_info_faults(...)*{*/hmp*}) calls this function, iterates through this list and writes this content to the monitor output. This function computes also the fault coverages and parses the rest of the needed variables and

the statistic to the monitor output. The corresponding source code can be seen in Listing A.5. *qapi_free_FaultInfoList(...)*{*/qapi-types.c*} is called at the end of this function, which is a auto-generated function for freeing the in the qmp_query_faults dynamically allocated variables. Furthermore, „info“ commands are not specified in the {*/hmp-commands.hx*}, they are specified in the same way in *mon_cmd_t info_cmds[]*{*/monitor.c*} as subcommands. Notice that {*/hmp-commands.hx*} should be also updated for these subcommands with the corresponding help-string.

Fault Injector

The fault-injector component contains the functions for the injection of faults, which are called from fault controller. The main functions are *do_inject_memory_register(...)*{*/fault-injection-injector.c*} for injecting faults in a register or memory, *do_inject_insn(...)*{*/fault-injection-injector.c*} for injecting faults in the decoding and execution path of the CPU, *do_inject_condition_flags(...)*{*/fault-injection-injector.c*} for injecting faults into a condition flag and *do_inject_look_up_error(...)*{*/fault-injection-injector.c*} for modifying the PC-value.

The main functions check the type of the simulated target[2] and call the appropriate, target-specific functions. Furthermore, this component defines a *FaultInjectionInfo* structure, which is used to decide which kind of fault should be injected in which component (memory or register level). This structure is set by the controller and parsed to the main functions of the fault injector. This structure contains the following flags:

fault_on_address: Is used to decide, if the fault injection should be performed on an address or on the content of a cell (register or memory). In the case of an injection on address, the address value is parsed by reference and modified according to the decision of the controller. In the case of injecting a fault on the content of a cell, the appropriate functions are called, which use *cpu_memory_rw_debug(...)*{*/exec.c*} for overwriting or modifying memory content or *((CPUARMState*)env)->regs[regno]* for overwriting or modifying the register content as well as *cpsr_write(...)/cpsr_read(...)*{*/target-arm/helper.c*} for modifying the content of the Current Processor Status Register (CPSR).

fault_on_register: Decides, if the called function aims to modify a register or a memory.

bit_flip: Defines, if a bit-flip is performed as fault injection.

injected_bit: Determines the bit position, on which a fault should be injected (for example 0x1 defines the first bit, 0x2 defines the second bit and so on).

bit_value: Is used for defining, if a bit should be set or reseted in the case of State Faults (SFs) or contains a new value in the case of „NEW VALUE“-mode.

new_value: Defines, if a new value should be written to the specified target.

access_triggered_content_fault: Is set for performing faults, which are access-triggered and do not modify the content of a cell immediately. Especially, in dynamic-operation-depended fault models, the content is parsed from SoftMMU or access-triggered

[2] Only ARM-targets are supported in this thesis

register functions and the possibly-modified content value is returned, which is written to the cell by subsequently executed functions. In the case of time-triggered faults, the access-type can not be determined and the faulty values have to be written to the cells immediately. Hence, operation-depended faults (WDF, TF, RDF,...) can not be injected in time-triggering mode.

do_inject_insn just overwrites the decoded instruction number by modifying the by reference parsed instruction number and hence it makes only sense to call this function for access-triggered faults. In the case of time-triggering, the function *do_inject_look_up_error* should be used. This function sets the PC to another specified PC-value to simulate an instruction or decoding fault. This is the only possibility to simulate instruction or decoding fault in time-triggering mode, because the decoding-function of QEMU is not called in this execution step.

do_inject_condition_flags is the function for overwriting the ARM condition flags. This function uses string-comparing to decide, which of the five ARM condition flags should be set or reseted.

Fault Controller

The fault controller decides if, when and where a fault should be injected and subsequently calls the corresponding function in the fault injector module. *fault_injection_controller_init(...)*{*/fault-injection-controller.c*} is the main function, which is called from SoftMMU (**(...)*{*/softmmu_template.h*}) for the injection of access-triggered memory faults and from *disas_arm_insn(...)*{*/target-arm/translate.c*} for injecting access-triggered instruction decoding or execution faults. The call of the fault controller for the injection of access-triggered register faults and time-triggered faults is more complicated. They are defined as helper functions (*HELPER(fault_controller_call_time)(...)*{*/target-arm/op_helper.c*}, *HELPER(fault_controller_call_reg_decoder)(...)*{*/target-arm/op_helper.c*}, *HELPER(fault_controller_call_store_reg)(...)*{*/target-arm/op_helper.c*} and *HELPER(fault_controller_call_load_reg)(...)*{*/target-arm/op_helper.c*}), which are invoked after every instruction decoding (*gen_intermediate_code_internal(...)*{*/target-arm/translate.c*}) and if a register is accessed (*load_reg(...)/store_reg(...)*{*/target-arm/translate.c*}). This helper functions can be seen as a wrapper function for the *fault_injection_controller_init(...)* function, which is translated in intermediate code and afterwards in host-specific code. This means that the fault controller is dynamically called at runtime, if a register is accessed or after every executed instruction. The controller has to know, which function calls the fault_injection_controller_init to decide which fault can be injected. For example, an access-triggered memory faults can only be injected, if the address is really accessed (caller-function is SoftMMU). In order to realize this, the fault controller defines a *enum InjectionMode*{*/fault-injection-controller.h*}, which is parsed to the init function. The fault controller can distinguish, based on this variable, if a memory address decoder fault, a memory content fault, a register address decoder fault, a register content fault, an instruction decoder or execution fault or another time-triggered fault (for example condition flags) could be injected.

Furthermore, the controller has a further *enum AccessType*{*/fault-injection-controller.h*}, which is used for determining the access type. This is important for the implementation of operation-dependent faults like the WDF, RDF, TF and so on. The access type is set by the caller-function and can be *read_access_type*, *write_access_type* or *exec_access_type*.

After the correct function is determined, the fault controller calls the appropriate function, which iterates through the fault configuration list (fault library) and checks if the configured parameters (address or trigger mode) matches. If yes, the fault type is determined from fault configuration and the appropriate function is called. This function prepares the variables in the FaultInjectionInfo, which is needed from the fault injector component, decides if the fault parameters matches with the current execution state (timer or PC-values), sets the fault active and increments the number of injected faults in the data analyzer module.

For dynamic, operation-dependent faults, it is important to know the previous operation types on the cell to trigger the fault. This is realized by the controller with a two-dimensional array. This array has a length of *number_of_ids* × *memory_bandwidth* and contains one of the four possible operations, which change the memory content, in each entry. The four possible operations (*OPs_0w0*, *OPs_0w1*, *OPs_1w0* and *OPs_1w1*) are defined in *enum CellOps{/fault-injection-controller.c}*.

The controller implements also a few helper-functions, which are shown in the following itemization:

***ends_with(...){/fault-injection-controller.c}*:** Is used for extracting the timer unit, which are returned as string(ms, μs, ns).

***timer_to_int(...){/fault-injection-controller.c}*:** Converts the timer value defined as string to an equivalent integer value.

***fault_injection_controller_getTimer(...) {/fault-injection-controller.c}*:** Returns the current timer value (QEMU_CLOCK_VIRTUAL - is stopped when QEMU-emulation is stopped)

***fault_injection_controller_initTimer(...){/fault-injection-controller.c}*:** Sets the start-timer value when the fault library is loaded and this value is substracted from the current timer value every time when fault_injection_controller_getTimer is called.

***time_normalization(...) {/fault-injection-controller.c}*:** Normalizes the different timer units to ns.

4.1.3 Compiling and Executing FIES

The following manual gives an overview how FIES can be compiled and subsequently executed with the right arguments. The whole manual is based on a linux-based operating system. FIES is an extension of the QEMU emulator and hence it has the same dependencies as QEMU has. Furthermore, FIES needs the libxml2-library, which has to be linked to QEMU. The following enumeration shows the needed steps, which should be executed in a linux bash, for compiling FIES:

1. Install the needed libraries (*libffi*, *libiconv*, *gettext*, *python*, *pkg-config*, *glib*, *sdl*, *zlib*, *pixman*, *libfdt*, *libxml2*)

2. Run configure in the FIES-root directory (*./configure –target-list=arm-softmmu –extra-cflags=„<library-include-path>“–extra-ldflags=„<library-binary-path>“ –enable-sdl*). The paths for cflag and ldflag can be determined by using the commands

xml2-config –cflags and *xml2-config –libs*. *–target-list=* defines the target architecture, which is in this case an arm-platform with a SoftMMU. *–enable-sdl* defines that the SDL-library should be used, which is needed for the VGA-ouput.

3. Running *make* and *make install*

After successful compiling, FIES can be started by executing the command *qemu-system-arm -M imx28evk -m 128 -kernel <path-to-SafeRTOS-image> -fi <fault-counter-address>, <path-to-fault-library>, <sbst-cycle-count-address>* in the linux bash. The parameters after *-M* defines the target system, which is in this case the implemented Freescale development board. The number after the argument *-m* defines the number of physical RAM in MB, which is in this case 128MB. The *-kernel* defines the path and the name of the image, which should be executed by FIES. In this case, the image is the SafeRTOS, which contains the SBSTs. The last argument *-fi* activates the data-collector to write the content of the monitor to a predefined file. Furthermore, this argument requires at least one parameter, which is the address of the fault counter address (the RAM-address where the number of detected faults are stored). The second argument defines the path and the name of the fault library. If this argument is not used, the default value is taken, which is stored in *fault-config.h*. The last parameter is used for the automatic test process. It defines the RAM-address, where the number of SBST cycles are stored. This is used by FIES to decide, when a fault can be injected or when the virtual machine can be terminated and the next fault injection experiment can be loaded. This variable can be seen as a software-to-hardware communication, which signalizes the hardware (FIES) if the system (SafeRTOS) is successfully booted and the SBST is executed at least on time. The number of SBST-cycles, before the virtual machine is terminated, can also be defined in *fault-config.h*.

After FIES is started, a window appears, which is visualized in Figure 4.4. It is possible to switch between the serial output and the monitor console of FIES with the shortcuts *[CTRL]+[ALT]+[1]* and *[CTRL]+[ALT]+[2]* (can be seen in Figure 4.4a and 4.4b). In the FIES monitor console, it is possible to use various commands, which are described in the following paragraph:

***info faults*:** Shows information and statistics about faults

***info registers*:** Prints the register content

***fault_reload <path-to-XML-fault-config-file>*:** Loads the parameters for a fault experiments and starts it

***q*:** Terminates FIES

***c*:** Continues the emulation process

***stop*:** Stops the emulation process

***help*:** Lists all commands with a short description[3]

The bash script for the automatic test process can be seen in Appendix A.6. This script reads the elf-file (SafeRTOS image) and extract the addresses of the global-defined variables

[3] The FIES monitor can be scrolled by using the shortcuts *[CTRL]+[↑]* and *[CTRL]+[↓]*.

(fault counter and sbst cycle count). The SafeRTOS image, which runs the SBST-method as well as the corresponding fault configuration file should be stored in a file per line and separated by commas. The path and the name of this file should be parsed to the bash script via argument. Every pair of image and fault configuration is sequentially executed in a do-while loop and the corresponding data-collector file is stored in a ascending way (data_collector_1.txt till data_collector_n.txt) to the directory, where the bash script is executed.

(a) The monitor output after loading a fault configuration

(b) The serial output for the execution of a SBST (register test)

Figure 4.4: The two consoles of FIES

4.2 Software-Based Self-Tests (SBSTs)

This section gives a short introduction about the used OS (SafeRTOS) and its components and describes how the SBSTs are included into it. Furthermore, a manual is given, which should simplify the compiling and the execution of current and further SBSTs.

4.2.1 Overview of SafeRTOS

SafeRTOS is a by WITTENSTEIN high integrity systems [WIT12a] modified version of FreeRTOS, in order to achieve a SIL3 certified real-time OS. SafeRTOS implements a preemptive real-time scheduler, which decides based on priorities if a task will be scheduled or not. Tasks with equal priorities are scheduled in a time sliced round robin fashion. Furthermore, tasks can block for a fixed period or for a specified time with a specified timeout period. This OS uses queues for sending data between tasks and for sending data between tasks and interrupt service routines.

SafeRTOS uses a Memory Management Unit (MMU) for isolating the kernel from the user task activities and for isolating the user tasks from each other [WIT12b]. The OS only uses level 1 page table entries and hence the MMU regions have to be defined in blocks of 1MB (performance issue). This 1MB-blocks can be shared among tasks in so-called *task_groups*, because the most tasks use less than 1MB RAM. This task_blocks have to be declared in the linker control file. Every task can join a certain task group by placing its stack in the appropriate memory region and declaring access in the task parameter block. Every task within this task_group has full access to variables that belongs to a task in

this task_group. Furthermore, SafeRTOS uses a flat memory model for the translation of virtual to physical address (virtual addresses = physical addresses).

SafeRTOS provides so-called *hook functions*, which are called if a special event occurs (callback functions). The OS provides these hook functions for error handling (error hook), for task deleting (task delete hook), for entering the idle task (idle hook) and for executing the tick handler (tick hook). The error hook function is called, if a corruption within the scheduler data structures or a potential stack overflow, while performing a context switch, occurs. This function can be used to implement a specific error handling to ensure that a certain safe state can be reached. The task delete hook is called if a task is deleted and can be used to tell the host application that the allocated memory is freed and is available for other purposes. The tick hook function is called every time, when the tick handler is called and can be used to execute a periodic task (for example an application timer). The idle hook function is called, if the idle task is currently executed and can be used to execute a low priority application specific background tasks. Such a task can be a function, which gets the processor in a power sleep mode or in our case it can be the execution of a preemptive SBST.

Figure 4.5 gives an overview of the by SafeRTOS used components. It can be seen that SafeRTOS uses the Timer0 interrupt for measuring the time. This time are measured in tick units, where a tick or a tick period is the time between two consecutive timer interrupts. The number of milliseconds between these ticks can be defined by the member *ulTickRateHz* of the *xPORT_INIT_PARAMETERS* structure, which is passed in the initializing function of the scheduler.

In order to create tasks, SafeRTOS provides the API function *xTaskCreate(...)* and the API function *xTaskDelete(...)* for deleting a task. A task code is executed in an infinity loop and should never return to its caller after termination. If a task has to delete itself, the function *xTaskDelete(NULL)* has to be called before reaching the end of the task code function. Furthermore, the task code function allows a void pointer as parameter for passing one arbitrary parameter or a pointer to a structure for more than one parameter to the task code function. The xTaskCreate function needs two parameters: *pxCreatedTask*, which returns a handle for referencing a certain task (for example, to change the priority of a task) and *pxTaskParameters*, which is a pointer to a structure with the following members:

***pdTASK_CODE pvTaskCode*:** Function pointer to the application code

***const signed portCHAR* pcTaskName*:** Name of the task (for debug purpose)

***xTCB* pxTCB* :** Pointer to the Task Control Block (TCB) of the application

***signed portCHAR* pcStackBuffer*:** Base-pointer of the stack

***unsigned portLONG ulStackDepthBytes*:** The stack size in bytes (must be greater than *configMINIMAL_STACK_SIZE*)

***void* pvParameters*:** The task function parameter as void*

***unsigned portBASE_TYPE uxPriority*:** The priority of a task, which can be between 0 and *configMAX_PRIORITIES* - 1 (the lower the value of the priority the lower the priority of the task)

***xMMUTaskParameters xMMUParameters*:** Structure that contains the MMU related task parameters

***unsigned portBASE_TYPE uxPrivilegeLevel*:** The privilege level of the task (*portUNPRIVILEGED_TASK* or *portPRIVILEGED_TASK*

***xMMUMemoryRegion xRegions[mmuNUM_CONFIGURABLE_REGIONS]*:** Structure for each of the configurable MMU regions available to the task

***unsigned portLONG ulRegionAddress*:** Start address of the memory region (must be aligned to 1MB boundaries)

***unsigned portLONG ulLengthInBytes*:** The length of the region in bytes (size must be a power of two and not greater than 128MB)

***unsigned portLONG ulAccessPermissions*:** Permission settings of the region (*mmu ACCESS_PRIV_RO_USER_RO*, *mmuACCESS_PRIV_RW_USER_NA*, *mmuACCESS_PRIV_RW_USER_RO* or *mmuACCESS_PRIV_RW_USER_RW*)

***unsigned portLONG ulCacheSettings*:** Cache settings for the memory region (*mmuCACHE_NOCACHE_NOBUFFER*, *mmuCACHE_NOCACHE_BUFFER*, *mmuCACHE_WRITE_THROUGH* or *mmuCACHE_WRITE_BACK*)

xTaskCreate returns *pdPASS* in the case of successful task creation or an error number, otherwise. Further details can be found in [WIT12a].

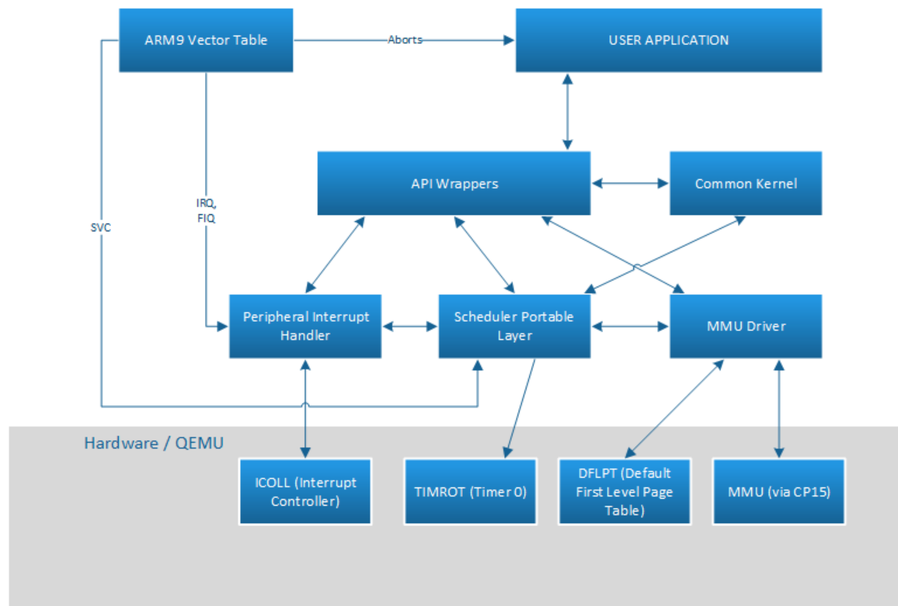


Figure 4.5: This figure shows the context diagram of the used Operating System (SafeRTOS) with the necessary hardware parts (ICOLL, TIMROT and DFLPT), which are simulated in the FIES framework. The image is adapted from [WIT12b].

4.2.2 Implementation of SBSTs in SafeRTOS

The SBST tasks are created with the function *xCreateSBSTTask(...)*{*/source/Software-BasedSelfTests.c*} and are called in the main-file before *xTaskStartScheduler* is called. In order to decide which SBST should be created, an *enum SBSTMethod*{*/include/SoftwareBasedSelfTests.h*} is parsed in xCreateSBSTTask. In xCreateSBSTTask a function is called, which sets the xTaskParameters structure, based on enum SBSTMethod, to the appropriates values. After creating this structure the task is created and placed into the *Ready State* by calling *xTaskCreate*. The corresponding source code can be found in B.1. It should be mentioned that *mmuACCESS_PRIV_RW_USER_RW* and *mmuCACHE_NOCACHE_NOBUFFER* are set in the xTaskParameters structure. This is important to allow the SBST (memory tests) the access to all memory addresses on the one hand and to guarantee that every memory access is targeted to the real memory and not to the cache, on the other hand.

Furthermore, functions for writing data to the debug UART are provided in **(...)* {*/source/DebugUtils.c*}. These functions delete, allocate or rather reallocate buffers, in which the data are stored and can be flushed later. These functions are implemented in order to avoid the blocking UART access, which yields the task and activates the scheduler to choose another task and hence interrupts the current execution of the SBST.

The current implementation of the SBSTs in SafeRTOS are configured with the same stack and hence it is not possible to create two or more tasks, which run simultaneously. If this simultaneously execution of SBSTs it is wanted, the user has to define a separate stack for each SBST. Furthermore, it is suggested to compile SafeRTOS without any optimization level (*O0-flag*), because the instruction order of the single SBSTs are well thought-out and should not be reordered or other optimized by the compiler.

March Tests

In this thesis, three different march tests (march c-, march ss and abraham) are implemented in three different ways. The first way is the basic non-transparent march test, which overwrites the content in a certain memory cell. In order to make this tests transparent and restore the initial memory content, the initial memory content is temporally saved and restored after the successful execution of the memory test. Due to the fact that this kind of tests are not really transparent, they are named *WOMPseudoTransparent<name of the test>(...)*{*/source/MemorySelfTests.c*} in the implementation. These pseudo transparent tests write the data patterns, which are listed in the Section 3.3.2, to the memory cells. The corresponding source code for the pseudo transparent march c- test can be found in Appendix B.2.

The second test method is the original transparent memory test (march ss in this case), which uses a Multiple Input Signature Register (MISR). The MISR has a period of 65535 and can be used for a RAM with a 16bit data interface. The usage of a MISR is a possible approach how SBSTs can be implemented in practice and is shown in Appendix B.3. At the beginning of the test, every RAM cell is initialized with random data. This random data is recovered at the end of the SBST execution, because this test uses only the initial and the inverse initial memory content. The MISR is initialized with the inverse initial memory content and for every iteration through the memory, the MISR is feed with the current memory content. For an ascending iteration thought the memory, *misr_encode(...)*

is called and for a descending iteration through the memory, *misr_decode(...)* is called. For a symmetric march test, the signature after execution of the SBST is independent from the initial memory content and hence it is in every test cycle the same value.

The last test methods are memory tests, which are modified according to the transformation rules presented in [LTW05]. A corresponding implementation of a modified abraham test can be found in Appendix B.4. This test performs a transparent version of abraham first and afterwards it executes an additional transparent test, which applies four data patterns which are xor-linked with the original memory content.

CPU-core Tests

In this thesis, SBSTs for the CPU-core elements are implemented for ALU, condition flags, divider/shifter and multiplier/adder. All core tests excepting the condition flag testing are adapted from [Pre13] and the corresponding source code are listed in Appendix B.5.

The SBST for the ALU passing three variables to the test routine, which tests all possible binary operations with these variable values. In order to simulate a fault in the ALU, FIES overwrites the passed parameters or the return value, which contains the resulting signature. These parameters are passed by reference, to ensure that the used input is read from the same memory value and not from the address, which contains the local copy of the variable. Furthermore, these values are declared as volatile to ensure that the compiler knows that these values could be overwritten by hardware (FIES) and has to be reloaded at every access of these variables.

The SBST for the divider or shifter component is simply a shift-left with a subsequently shift-right through all bits of the register, which are in every step checked for the right value.

The test for the condition flags uses inline-assembler statements to perform special operations (addition instructions) with special register values, which forces the CPU to set the appropriate condition flags. These condition flags are encoded in the CPSR-register of ARM, and can only be read in the privileged mode. In order to get in this privileged mode, a software interrupt (*svc 3/4*) is generated, which informs the OS to enter or exit this mode. Furthermore, it is important to add the character *s* to the end of the assembler instruction to activate the setting of condition flags (*add* → *adds*). It should also be noticed that the QF-condition flag has to be reseted by software (*MSR*).

The fourth and last SBST is aimed to test the multiplier or adder unit of the CPU. This test generates 256 data patterns for the input of the multiplier and compares the signature at the end of the test with a pre-computed signature. If these signatures differ, a fault has occurred.

Register Tests

The classical walking bit method and a modified march ss test (according to [LTW05]) is implemented as register tests. The walking bit method simply shifts a one through the register and compares the register content with a golden signature value, which is generated by multiplication of the initial value (one) with two. Hence, it is suggested to test the multiplier before executing this register test. The test patterns are written to the register by using the inline-assembler statement in combination with the mov-instruction. The corresponding source code is visualized in Appendix B.6.

The march ss test for registers is the same as the tests for memories with the exception that the read- and write-operations are different. As Appendix B.6 shows, are these operations implemented as separate c-macros instead of function calls as in walking bit method. This is necessary, because in this case the subsequently read and write can not be executed due to the fault checking between single read or write operations. If the read and write operation to registers are implemented as functions, the register content will be overwritten when calling these functions.

Chapter 5

Results

This chapter covers the single results for all components under test and gives an overview on the simulation time for a certain number of injected faults of a fault injection experiment. Furthermore, a summary about the validation of the system requirements is shown at the end of this section.

5.1 Test Results

The following sections list the test results for the memory tests as well as for the CPU-core elements and the register banks. All test runs are aimed to show, which kind of faults are detectable by which self-tests. Hence only permanent faults are injected, because if a test is able to detect a permanent fault, the same test is able to detect transient faults of the same type (assumption: the duration of the fault occurrence is greater than the runtime of the self-test, otherwise the detection of transient faults are unlikely or simply pure coincidence).

5.1.1 Memory Test Results

A fault injection experiment is applied to the transparent versions of march c- (TMarch C-), abraham (TAbraham) and march ss (TMarch SS), to the pseudo transparent versions of march c- (Pseudo TMarch C-), abraham (Pseudo TAbraham) and march ss (Pseudo TMarch SS) as well as to the modified versions, according to the transformation rules presented in [LTW05], of march c- (TWM TA March C-), abraham (TWM TA Abraham) and march ss (TWM TA March SS).

The corresponding test results are compared in Table 5.1 and Table 3.6 shows the theoretical detection rates for different FFMs for march c-, march ss and abraham memory test. The comparison of these two tables shows that the results of static FFMs (SFs, TFs, RDFs and IRF) match for all tests. The WDFs are not detected by the transparent version of march c- and abraham, because these tests do not contain the needed write/read sequence to trigger these faults and hence they are not detected by these tests. The modified versions of these tests (Pseudo TAbraham, Pseudo TMarch C-, TWM TA March C- and TWM TA Abraham) extend these tests with additional write/read sequences and additional data patterns for the purpose of detecting coupling faults. These extensions also trigger WDFs, which result in a detection of these faults. These additional sequences are

also responsible for the detection of DRDFs in Pseudo TMarch C- and TWM TA March C- memory tests. TMarch C- is also able to detect DRDFs, because it reads the memory content of a cell before every write operation and uses this content for the next march steps by inverting it. Hence, after a normal read operation an additional read is executed, which detects the DRDF.

For the detection of dynamic FFMs (RDF_dyn, IRF_dyn, DRDF_dyn) all tests, except the transparent version of abraham (TAbraham) and march c- (TMarch C-), achieve the full detection rate. TAbraham and TMarch C- do not contain all possible sequences to trigger dynamic faults.

Coupling faults (CF) are split in intra- and inter CFs. All tests detect all intra and inter CFs with the exception of intra CFtrs and inter and intra CFwds. The detection of intra CFs depends not only on the read/write sequences of the memory tests, but also on the used data patterns as well as the initial memory content of transparent memory tests. For the detection of inter CFwds the right operation sequences are needed, which are not fulfilled by the transparent march c- and abraham test. Pseudo TAbraham, Pseudo TMarch C-, TWM TA March C- and TWM TA Abraham partly fulfill these needed operation sequences by adding additional sequences and data patterns.

All in all, the pseudo transparent memory tests deliver the best results for the given fault experiment. The best result achieves the Pseudo TMarch SS followed by the Pseudo TMarch C- and the Pseudo TAbraham. An interesting fact is that the march ss and even the march c- test deliver better results with shorter runtimes than the abraham test. The fourth place reaches the TWM TA March SS followed by the transparent march ss test (TMarch SS), but it should be kept in mind that the detection rate for intra CFs of transparent tests depend on the initial memory content. TWM TA Abraham and TWM TA March C- achieves the same detection rate and as before, the TWM TA March C- has a shorter runtime than the TWM TA Abraham. The last place is shared by the transparent versions of march c- (TMarch C-) and abraham (TAbraham).

FFM (#faults)	**TMarch C- (%)**	**Pseudo TMarch C- (%)**	**TWM TA March C- (%)**	**TAbraham (%)**	**Pseudo TAbra-ham (%)**	**TWM TA Abra-ham (%)**	**TMarch SS (%)**	**Pseudo TMarch SS (%)**	**TWM TA March SS (%)**
SF (2)	100	100	100	100	100	100	100	100	100
TF (2)	100	100	100	100	100	100	100	100	100
WDF (2)	0	100	100	0	100	100	100	100	100
RDF (2)	100	100	100	100	100	100	100	100	100
IRF (2)	100	100	100	100	100	100	100	100	100
DRDF (2)	100	100	100	100	100	100	100	100	100
inter CFst (4)	100	100	100	100	100	100	100	100	100
intra CFst (4)	100	100	100	100	100	100	100	100	100
inter CFds (12)	100	100	100	100	100	100	100	100	100
intra CFds (12)	100	100	100	100	100	100	100	100	100
inter CFtr (4)	100	100	100	100	100	100	100	100	100
intra CFtr (4)	50	100	75	50	75	75	50	100	75
inter CFwd (4)	0	75	50	0	75	50	100	100	100
intra CFwd (4)	0	100	0	0	100	0	50	100	50
inter CFrd (4)	100	100	100	100	100	100	100	100	100
intra CFrd (4)	100	100	100	100	100	100	100	100	100
inter CFir (4)	100	100	100	100	100	100	100	100	100
intra CFir (4)	100	100	100	100	100	100	100	100	100
inter CFdr (4)	100	100	100	100	100	100	100	100	100
intra CFdr (4)	100	100	100	100	100	100	100	100	100
RDF_dyn (4)	50	100	100	50	100	100	100	100	100
IRF_dyn (4)	50	100	100	50	100	100	100	100	100
DRDF_dyn (4)	50	100	100	50	100	100	100	100	100
Overall DC	**78.26**	**98.91**	**92.39**	**78.26**	**97.82**	**92.39**	**95.65**	**100**	**96.73**

Table 5.1: This table compares the detection rates for the implemented memory self-tests. The first six rows cover static faults, the next 14 rows contain intra and inter coupling faults and the last three rows show the dynamic faults. The brackets after every FFM contains the number of injected faults.

5.1.2 CPU-core and Register Test Results

The diagnostic coverages (DCs) for ALU, adder, multiplier and shifter are determined by deterministic test and hence the minimum DCs are proven values. The only DC variability for CPU-core elements is given by the DC value for the register and register banks. If it is possible to increase this value at least to 90%, SIL3 can be achieved for the tested CPU-core elements and the given Hardware Fault Tolerance of one.

Table 5.2 presents these values for the TWM TA March SS and for the by IEC 61508 suggested Walking Bit test. The test runs are only performed on the ARM R10 general purpose cpu register and hence Table 5.2 only considers intra CFs (no inter CFs). In order to determine the DC values for inter CFs, the given SBSTs have to be extended with further adjacent register under test (e.g. R9 or R11). The results show that the march test achieves a much higher DC value than the Walking Bit method. Furthermore, Table 5.2 shows that march tests can be applied on registers and achieve the equivalent results as for memories. In comparison to Table 5.1, the TWM TA March SS achieves a smaller DC rate for intra CFtr faults. The reason for this can be explained by the structure of this test. A TWM TA-modified test consists of a transparent march test and an extension of this march test sequences. It is possible that this CFtr is detected by the transparent march test part in combination with the random initial memory content of the TWM TA March SS.

FFM (#faults)	**TWM TA March SS (%)**	**Walking Bit (%)**
SF (2)	100	100
TF (2)	100	100
WDF (2)	100	0
RDF (2)	100	100
IRF (2)	100	100
DRDF (2)	100	0
CFst (4)	100	100
CFds (12)	100	66.67
CFtr (4)	50	0
CFwd (4)	50	50
CFrd (4)	100	75
CFir (4)	100	75
CFdr (4)	100	0
RDF_dyn (4)	100	50
IRF_dyn (4)	100	75
DRDF_dyn (4)	100	0
Overall DC	**93.75**	**55.72**

Table 5.2: This table compares the detection rates for the implemented register self-tests. The first six rows cover static faults, the next seven rows contain intra coupling faults and the last three rows show the dynamic faults. The brackets after every FFM contains the number of injected faults.

Table 5.3 visualizes the DCs for each single CPU-core element. The overall DC values result in 53.01% for the applied walking bit method and in 89.98% for the applied march test as register test. The march test yields a quite near value to the needed 90% for

achieving SIL3 and the walking bit method achieves in this test experiment just SIL1. As a conclusion, neither the walking bit nor the march test can achieve the needed 90% for the CPU-core elements, but it is possible to use the Pseudo TMarch SS as register test, which will results in 100% test coverage and an overall DC of 92.60%. In order to achieve higher DC values than 92.60%, it is necessary to test the other components like multiplexer or pipeline directly.

Module	assumed gate-count (%)	SFF (%)
ALU	2	99.9
Adder and multiplier	39	99
Barrel-shifter	3	100
Register banks (march)	42	93.75
Register banks (walking bit)	42	55.72
Other components	14	50
Overall with march	**100**	**89.98**
Overall with walking bit	**100**	**53.01**

Table 5.3: This table shows the achieved Diagnostic Coverages (DCs) for the CPU-core elements for the fixed DC values for ALU, adder, multiplier and shifter as well as the DC values for the registers tested by march and walking bit test.

SBSTs, which are developed for a certain target, can also be applied to other different target points. This indirect testing is performed with the SBSTs for the CPU-core elements and they are applied to the memory. The achieved DC values are listed in Table 5.4. An interesting fact is that all tests yield a better result as the walking bit method. The test for multiplier and adder achieves even a high DC of 98.55%, which is comparable with the result of a march ss test. The reason for this can be explained by the test structure of this SBST. The multiplier test applies 256 (16×16) data patterns to test the functionality of adder and multiplier. This is the same strategy that memory tests or march tests apply to test intra- and inter CF. But it should be kept in mind that the execution time for applying the multiplier test to a memory cell is greater than applying march sequences to it.

FFM (#faults)	ALU test (%)	Divider/shifter test (%)	Multiplier/adder test (%)
SF (2)	100	100	100
TF (2)	100	100	100
WDF (2)	0	50	100
RDF (2)	100	100	100
IRF (2)	100	100	100
DRDF (2)	100	100	100
inter CFst (4)	100	100	100
intra CFst (4)	100	100	100
inter CFds (12)	66.67	50	66.67
intra CFds (12)	83.34	83.34	100
inter CFtr (4)	50	50	100
intra CFtr (4)	50	25	100
inter CFwd (4)	50	25	100
intra CFwd (4)	0	50	100
inter CFrd (4)	100	100	100
intra CFrd (4)	50	75	100
inter CFir (4)	100	100	100
intra CFir (4)	50	75	100
inter CFdr (4)	100	50	100
intra CFdr (4)	50	75	100
RDF_dyn (4)	50	75	100
IRF_dyn (4)	50	75	100
DRDF_dyn (4)	50	75	100
Overall DC	**69.56**	**72.10**	**98.55**

Table 5.4: This table visualizes the results of achieved DC values for the CPU-core element self-tests, which are applied on memory cells. The first six rows cover static faults, the next 14 rows contain intra and inter coupling faults and the last three rows show the dynamic faults. The brackets after every FFM contains the number of injected faults.

5.2 Simulation Time

This section covers the evaluation of the simulation times for a fault injection experiment. The fault injection experiment is executed on a Intel® Core™ i7-3770 CPU @ 3.40GHz × 8 with 16GB RAM and Linux Debian Wheezy with kernel 3.2.0-4-amd64 as operating system. For this purpose, every implemented test (15 tests) are executed three times and the memory tests are applied to an 1MB portion. These evaluation runs inject a certain number of transient and permanent faults and the times, measured by the linux command *time*, are determined with serial output for visualizing faults to the serial prompt and without. The results, presented in Table 5.5, are split in the injection of state faults and in the injection of operation dependent faults like TF, WDF, RDF and so on. The simulation times show an exponential relation, because the faults are detected by several march elements of a march test and every march element accesses the fault counter variable to increase the number of detected faults and writes an output to a buffer, which is flushed at the end of the test to the serial output. Furthermore, QEMU/FIES has a decreasing performance with the increasing of simulation time, which results also in a larger simulation time for a greater number of injected faults. The operation dependent faults have a shorter

simulation time than the state faults, because state faults are detected by every memory test with 100% DC and operation dependent faults (static and coupling faults) are not detected by every memory test. Hence, not every march element needs to access the fault counter variable and writes something to the serial buffer, which results in a shorter simulation time. This is also the reason, why pseudo memory tests take the largest portion of the whole simulation time, because they have a larger complexity on the one hand, but also detect more faults on the other hand.

Number of state faults	**Simulation time with serial output**	**Simulation time without serial output**
0	18s	10s
240	5min 47s	5min 28s
1158	26h 30min 10s	26h 34min 22s
Number of operation dependent faults	**Simulation time with serial output**	**Simulation time without serial output**
0	18s	8s
240	4min 7s	4min 17s
2157	10h 11min 52s	10h 15min 40s

Table 5.5: This table shows the simulation times for a fault injection experiment, which injects a certain number of state and operation dependent faults. The results are evaluated with the usage of a serial output and without using it.

5.3 Validation of the System Requirements

The following enumeration summarizes the system requirements and discusses how well every single goal is satisfied.

SBSTs for ARM9 processors: Component-specific SBSTs for the ALU, adder, multiplier, shifter and the condition flags were developed for an ARM9-processor. Furthermore, different tests for the memory as well as for the CPU registers were developed and evaluated according to their DC rates. The main part of the self-tests were written in C, except the reading and writing from or to registers were implemented in assembler code.

Verification of the SBSTs: The SBSTs were verified with the help of fault injection experiments. For every target component (RAM and CPU-core elements), several different kind of faults were injected to several bits within a memory-word (intra CFs) and over different memory-words (inter CF).

IEC 61508 compliant SBSTs: Faults were injected into CPU registers and memory cells, which covers the by IEC 61508 required DC-model for data and address lines. The standard requires the injection of transient and permanent faults. Although, the injection of transient faults is possible with FIES, it does not make sense, because a

reliable detection of transient faults is only possible if the duration of these faults are longer than the runtimes of the self-tests and hence transient faults are not injected for the evaluation of these self-tests. For a reliable detection of transient faults, a diverse calculation is required. Furthermore, the injection of faults in the coding and executing path including the condition flags as well as the injection in stack pointer (SP), linking register (LR) and program counter (PC) is possible by FIES, but it is hard to evaluate these faults with the corresponding SBSTs. The reason for this is that a modification of these target points means that the operating system (SafeRTOS) enters an error handler and the SBSTs are not further executed. All in all, this work showed that the needed DC rates for SIL3 and a Hardware Fault Tolerance of one can be achieved with the right choice of SBSTs methods for the CPU-core elements and the memory. Furthermore, the results proved that methods, which are higher ranked by the IEC 61508, achieved worser results than methods, which have a lower rank in the standard.

Fully-automated simulation: The whole simulation process can be configured before starting the fault injection experiment, which was done by calling the execution script. This script parsed the needed parameters and executed the configured self-tests with the corresponding fault parameters. The script ran every self-tests subsequently until every test was finished. The operator can simply read the corresponding files, which contains the statistics about the fault injection experiment. Hence, a human interaction between a fault injection experiment is unnecessary.

Simulation time: The simulation for executing all implemented tests (15 tests) three times on a RAM size of 1MB needed less than 20 seconds. For the injection of just a few specific faulty bits, which is needed to show if a SBST is able to detect some kind of faults, FIES needs just a few seconds till minutes (depends on the number of injected faults and SBST). For a greater number of injected faults, the simulation time increases exponentially. The injection of 1158 state faults (stuck-at faults) and the execution of the corresponding SBSTs needed a few hours. This increase of simulation time depends on the complexity of the executed SBSTs on the one hand, but also on the implementation of QEMU/FIES. All in all, the simulation time is still acceptable, because fault injection experiments can run without human interactions.

Easy-to-use: A fault injection experiment does not require a modification of the QEMU source code. The configuration of faults can be easily done by defining the necessary fault parameters in an XML-encoded fault library file. Thus, the operator does not need the understanding of the QEMU-implementation.

Chapter 6

Conclusion

This thesis gave an overview of the IEC 61508 safety standard, the test methods according to the standard, as well as the basic architecture of hardware components for an ARM9 processor. Furthermore, this work explained the classifications of Fault Injection frameworks, showed the corresponding advantages and disadvantages and gave some examples of fault injection frameworks in the literature for each classification. This thesis also gave insight into software-based self-tests for testing memory and CPU components.

An appropriate fault injection framework, called FIES (Fault Injection framework for the Evaluation of Self-tests), which is an extension of the QEMU emulator, was developed within the scope of this thesis. A short overview of the main parts, the basic functionality as well as the current implementation of QEMU was given in this thesis. Based on this knowledge about QEMU, the implementation of FIES was explained and important relations were visualized in appropriate diagrams. FIES is able to inject transient, permanent and intermittent faults in the instruction decoding and execution path as well as in the condition flags of a CPU, in the CPU registers, including stack pointer, linking register and program counter, and in the address decoding and directly in the memory cells of RAMs. The fault injection can be triggered by time, which means that after each instruction a fault can be injected, and by access, which means that a fault can only be injected if the corresponding memory or register cell is accessed. This allowed us to implement further operation-dependent faults for memory and register, which was shown in [AAG01] that these faults are important for the simulation of shorts, open circuits and bridging faults in a dynamic or static RAM.

Furthermore, a few suitable SBST methods were chosen from a huge pool, presented in the literature, for testing the CPU-core elements and the memory. For testing ALU, multiplier, adder and shifters special deterministic tests with proven minimal Diagnostic Coverages (DCs) were used. For testing memory cells and CPU registers, two march tests (march c- and march ss) and the Abraham test, suggested by IEC 61508 were used. Each of these three memory tests was implemented in a transparent version, which allows to restore the initial memory or register content after testing, in an modified version presented in [LTW05], which uses further write/read sequences and special data patterns for the detection of further faults, and in a pseudo transparent version, which backups the initial content and applies a non-transparent march test with special data patterns for the detection of further faults.

The SBSTs were executed by a real-time operation system (SafeRTOS), which ran on a

Freescale i.MX28 EVK development board. FIES simulated this board and injected faults into it, if necessary. The results of these fault injection experiments showed that march ss and even a much simpler march c- memory test can perform better or at least equal results than the by IEC 61508 best-rated Abraham test. Furthermore, the evaluation pointed out that memory tests, especially march tests, can also be applied to registers and register banks and they also achieved equivalent results. A further interesting insight was that CPU-core tests applied on memory cells also achieved high DC values by indirect testing.

In summary, this thesis pointed out that by IEC 61508 worser-rated march tests can achieve better results than the best-rated Abraham memory test and hence parts of the safety standard, especially the testing methods, are outdated and should be revised.

6.1 Further Work

There are many ideas which could improve this work:

Support of other CPU architectures and multi-core platforms: The current implementation of FIES supports only one hardware board (i.MX28 EVK) and one CPU architecture (single-core ARM9). It would be of great interest to emulate further hardware boards with different CPU architectures. Furthermore, it would be interesting to extend FIES with multi-core support and evaluate the SBSTs on a multi-core platform.

Extensions of target points: FIES/QEMU supports many hardware components and hence FIES could be extended with further fault injection targets like communication components, which are heavily used in embedded systems (e.g. SPI, USB, ethernet, etc.). A fault injection into memory storages like SD-card modules or in the Direct Memory Access (DMA) controller is also possible.

Optimization of the simulation time: A shorter simulation time is always desirable and could be achieved for example by checking if a time-triggered fault should be injected. If not, the call of the fault controller after each instruction could be skipped and hence the simulation time could be decreased.

More self-tests/operating systems: A further improvement could be the usage of other SBST methods, which run on other operating systems than SafeRTOS (maybe also non-safety certified ones like FreeRTOS). SBSTs, which are able to test hidden components (pipelines, multiplexer, etc.) directly, could further improve the diagnostic coverage of a processor core.

SBST runtime evaluation: FIES/QEMU operates on a high abstraction level of the hardware and is optimized for a fast simulation. Hence, it is not possible to say, how much time a certain SBST needs on a real hardware. It would be interesting to implement the SBSTs on real hardware board, measure the execution time and optimize the SBST if necessary.

Appendix A

FIES Source Code

Listing A.1: Parameter definitions for the *fault_reload* command.

```
    {
        .name       = "fault_reload",
        .args_type  = "filename:s",
        .params     = "file",
        .help       = "load the config file",
        .mhandler.cmd = hmp_fault_reload,
    },

STEXI
@item fault_reload @var{file}
@findex fault_reload
load the config file from @var{file}.
ETEXI
```

Listing A.2: Definition of the user-defined data types in JSON format

```
##
# @FaultInfo:
#
# A description of the faults.
#
# @component:                  defines the component, where a fault should be injected
#
# @mode:                       defines which fault should be injected
#
# @target:                     defines the target point
#
# @type:                       defines if the fault is permanent, transient or intermittend
#
# @params.mask:                defines bits, which should be changed
#
# @params.address:             the address of the bit, which should be changed
#
# @params.instruction:         defines the address of the replaced instruction
#
# @is_active:                  shows if the fault is currently active
#
# Since: 1.7.0
##
{ 'type': 'FaultInfo',
  'data': {'params': {'mask': 'int', 'address': 'int', 'cf_address': 'int',
                      'instruction': 'int', 'set_bit': 'int'},
           'component': 'str',
           'mode': 'str',
           'target': 'str',
           'type': 'str',
           'duration': 'str',
           'interval': 'str',
           'id': 'int',
           'trigger': 'str',
           'timer': 'str',
           'is_active': 'int'} }
```

Listing A.3: Definition of the *info faults* command in JSON format

```
##
# @query-faults:
#
# Returns the fault informations.
#
# Returns:  A @FaultInfoList object describing the injected faults.
#
# Since: 1.7.0
##
{ 'command': 'query-faults', 'returns': 'FaultInfoList' }
```

Listing A.4: Source code for *qmp_query_faults(...)*

```
FaultInfoList *qmp_query_faults(Error **err)
{
    FaultInfoList *head = NULL, *cur_item = NULL;
    FaultList *fault;
    int element = 0;

    for (element = 0; element < getNumFaultListElements(); element++)
    {
      FaultInfoList *info;
      fault = getFaultListElement(element);

      info = g_malloc0(sizeof(*info));
      info->value = g_malloc0(sizeof(*info->value));
      info->value->component = g_strdup(fault->component);
      ...
      info->value->is_active = fault->is_active;

      if (!cur_item)
         head = cur_item = info;
      else
      {
         cur_item->next = info;
         cur_item = info;
      }
    }

    return head;
}
```

Listing A.5: Source code for *qmp_query_faults(...)*

```
void hmp_info_faults(Monitor *mon, const QDict *qdict)
{
   //initializing of variables
   ...

   fault_list = qmp_query_faults(NULL);

    for (fault = fault_list; fault; fault = fault->next)
    {
      monitor_printf(mon, "id: %d\n", (int) fault->value->id);
      //output of all variables and statistics
      ...
      monitor_printf(mon, "active: %d\n", (int) fault->value->is_active);

    }

    if (fault_list == NULL)
      return;

    //in- and output of variables, computation of fault coverage
    ...

    qapi_free_FaultInfoList(fault_list);
}
```

Listing A.6: bash script for the automatic testing

```
#!/bin/bash

if [ "$#" -ne 1 ]
then
  echo "Usage: ./startTests.sh <path-to-kernel-and-fault-library-config-files>"
  exit 1
fi

DATA_COLLECTOR_PATH="data_collector.txt"
```

```
CONFIG_FILE=$1

counter=1
while read line
do
    KERNEL_PATH=$(echo -e "$line\n" | cut -f1 -d,)
    FAULT_LIBRARY_PATH=$(echo -e "$line\n" | cut -f2 -d,)

    FAULT_COUNTER_ADDRESS=$(readelf $KERNEL_PATH -s | grep fault_counter)
    FAULT_COUNTER_ADDRESS=$(echo $FAULT_COUNTER_ADDRESS | cut -f2 -d :)
    FAULT_COUNTER_ADDRESS=$(echo $FAULT_COUNTER_ADDRESS | cut -f1 -d ' ')
    echo "FAULT_COUNTER_ADDRESS: $FAULT_COUNTER_ADDRESS"

    SBST_CYCLE_COUNT=$(readelf $KERNEL_PATH -s | grep sbst_cycle_count)
    SBST_CYCLE_COUNT=$(echo $SBST_CYCLE_COUNT | cut -f2 -d :)
    SBST_CYCLE_COUNT=$(echo $SBST_CYCLE_COUNT | cut -f1 -d ' ')
    echo "SBST_CYCLE_COUNT: $SBST_CYCLE_COUNT"

    qemu-system-arm -M imx28evk -m 128 -kernel $KERNEL_PATH -fi \
    $FAULT_COUNTER_ADDRESS,$FAULT_LIBRARY_PATH,$SBST_CYCLE_COUNT
    DATA_COLLECTOR_OUT="data_collector_$counter.txt"
    cat $DATA_COLLECTOR_PATH > $DATA_COLLECTOR_OUT
    counter=$((counter+1))
done <$1

echo "Fault injection experiment finished"
```

Listing A.7: An example for a fault configration file with XML-encoding

```
<?xml version="1.0" encoding="UTF-8"?>

<injection>
  <!--DRDF0, access-triggered, permanent register cell fault-->
  <fault>
    <id>1</id>
    <component>REGISTER</component>
    <target>REGISTER CELL</target>
    <mode>DRDF0</mode>
    <trigger>ACCESS</trigger>
    <type>PERMANENT</type>
    <params>
      <address>0xa</address>
      <mask>0x00FF</mask>
    </params>
  </fault>

  <!-- time-triggered, permanent condition flag (NF) fault-->
  <fault>
    <id>2</id>
    <component>CPU</component>
    <target>CONDITION FLAGS</target>
    <mode>NF</mode>
    <trigger>TIME</trigger>
    <type>PERMANENT</type>
    <params>
      <set_bit>0x0</set_bit>
    </params>
  </fault>

  <!--Access-triggered, permanent intra-coupling state fault in the memory cell-->
  <fault>
    <id>3</id>
    <component>RAM</component>
    <target>MEMORY CELL</target>
    <mode>CFST00</mode>
    <trigger>ACCESS</trigger>
    <type>PERMANENT</type>
    <params>
      <address>0x40200ab6</address>
      <cf_address>0x40200ab6</cf_address>
      <mask>0xF003</mask>
      <set_bit>0x1</set_bit>
    </params>
  </fault>

  <!--PC-triggered, instruction execution fault, which injects a NOP-->
  <fault>
    <id>4</id>
    <component>CPU</component>
    <target>INSTRUCTION EXECUTION</target>
    <mode>NEW VALUE</mode>
    <trigger>PC</trigger>
    <params>
      <address>0x40003b8c</address>
      <instruction>0xDEADBEEF</instruction>
    </params>
```

```
    </fault>

    <!--time-triggered, intermittent bit-flip fault in the memory cell-->
    <fault>
        <id>5</id>
        <component>RAM</component>
        <target>MEMORY CELL</target>
        <mode>BIT-FLIP</mode>
        <trigger>TIME</trigger>
        <type>INTERMITTEND</type>
        <timer>1000MS</timer>
        <duration>20000MS</duration>
        <interval>2000MS</interval>
        <params>
            <set_bit>0x1</set_bit>
        </params>
    </fault>

    <!--time-triggered, transient inter-coupling read disturb fault-->
    <fault>
        <id>6</id>
        <component>RAM</component>
        <target>MEMORY CELL</target>
        <mode>CFRD00</mode>
        <trigger>ACCESS</trigger>
        <type>TRANSIENT</type>
        <timer>1000MS</timer>
        <duration>10000MS</duration>
        <params>
            <address>0x40201688</address>
            <cf_address>0x40003ba8</cf_address>
            <mask>0xFF</mask>
        </params>
    </fault>
</injection>
```

Appendix B

SBST Source Code

Listing B.1: Code for creating a SBST task in SafeRTOS

```
/* Define the priority at which the task is to be created. */
#define SBST_TASK_PRIORITY ( unsigned portBASE_TYPE )1
/* Declare the TCB of the task that is to be created and tell
the linker to locate it in a section reserved for kernel access only. */
static xTCB xSBSTTaskTCB __attribute__ ( ( section ( "lnkUserDefKernelData" ) ) ) = { 0 };
/* Declare reference to linker defined symbols. */
extern unsigned portBASE_TYPE _lnkStartTaskGroup0;
/* Declare the buffer to be used by the task's stack and ensure that
it is located in the memory region set up by the linker. */
#define SBST_STACK_SIZE 4096
static signed portCHAR cSBSTTaskStack[ SBST_STACK_SIZE ]
__attribute__ ( ( section ( "lnkTaskGroup0" ), aligned ( 8 ) ) ) = { 0 };

static xTaskParameters createTransMarchCMinus( void )
{
  xTaskParameters xNewTaskParameters =
  {
      WOMTransparentMarchCMinus,
      /* The function that implements the task being created. */
      ( signed portCHAR * ) "Transparent_March_C-",
      /* The name of the task being created. */
      &xSBSTTaskTCB,
      /* The TCB for the task. */
      cSBSTTaskStack,
      /* The buffer allocated for use as the task stack. */
      SBST_STACK_SIZE,
      /* The size of the buffer allocated for use as the task stack. */
      NULL,
      /* The task parameter will be initialised later. */
      SBST_TASK_PRIORITY,
      /* The priority to be assigned to the task being created. */
      {
        portUNPRIVILEGED_TASK,
        {
          { ( unsigned portLONG ) &_lnkStartTaskGroup0,
            mmuSMALLEST_ACTUAL_REGION_SIZE,
            mmuACCESS_PRIV_RW_USER_RW, mmuCACHE_NOCACHE_NOBUFFER,
          },
          { 0UL, 0UL, 0UL, 0UL },
          { 0UL, 0UL, 0UL, 0UL },
          { 0UL, 0UL, 0UL, 0UL }
        }
      }
  };

  return xNewTaskParameters;
}

//... xTaskParameters for other SBSTs ...

void xCreateSBSTTask( SBSTMethod method )
{
  xTaskParameters xNewTaskParameters;

  switch(method)
  {
  case MEMORY_TRANS_MARCH_C_MINUS:
    xNewTaskParameters = createTransMarchCMinus();
    break;
```

```
  //... other tests ...

  default:
    vDebugUartPutString( "Unimplemented SBST!\r\n" );
    break;
  }

  if( xTaskCreate( &xNewTaskParameters, (xTaskHandle *)NULL ) != pdPASS )
    vDebugUartPutString( "Task creation failed\r\n" );
  else
    vDebugUartPutString( "Task creation successful\r\n" );
}
```

B.1 Pseudo Transparent March C- Test

Listing B.2: Code for pseudo transparent MarchC-

```
static unsigned short data_patterns_cfid[NUMB_OF_CFID_PATTERN] =
                                  {0x0000, 0xFFFF, 0x0000, 0x5555,
                                  0xAAAA, 0x5555, 0x3333, 0xCCCC,
                                  0x3333, 0x0F0F, 0x00FF, 0xF0F0,
                                  0xFF00, 0x0FFF, 0x000F};

static unsigned short data_patterns_cfdst[NUMB_OF_CFDST_PATTERN] =
                                  {0x0000, 0xFFFF, 0x0000, 0x5555,
                                   0xAAAA, 0x5555, 0x3333, 0xCCCC,
                                   0x3333, 0x0F0F, 0x00FF, 0xF0F0,
                                   0xFF00, 0x0FFF, 0x000F};

static void MarchCMinus(int pattern)
{
  //... initializing ...

  /* M0 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
    baseAddress[offset] = pattern;

  /* M1 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    if (baseAddress[offset] != (unsigned short)pattern)
    {
      //increment fault counter and visualize fault
    }
    baseAddress[offset] = ~pattern;
  }

  /* M2 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    if (baseAddress[offset] != (unsigned short)~pattern)
    {
      //increment fault counter and visualize fault
    }
    baseAddress[offset] = pattern;
  }

  /* M3 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
    if (baseAddress[offset] != (unsigned short)pattern)
    {
      //increment fault counter and visualize fault;
    }
    baseAddress[offset] = ~pattern;
  }

  /* M4 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
    if (baseAddress[offset] != (unsigned short)~pattern)
    {
      //increment fault counter and visualize fault
    }
    baseAddress[offset] = pattern;
  }

  /* M5 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
```

```
    if (baseAddress[offset] != (unsigned short)pattern)
    {
      //increment fault counter and visualize fault
    }
  }
}

static void CFdsTest(void)
{
  //... initializing ...

  /* M0 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
    baseAddress[offset] = data_patterns_cfds[0];
    baseAddress[offset] = data_patterns_cfds[1];
    if (baseAddress[offset] != data_patterns_cfds[1])
    {
      //increment fault counter and visualize fault
    }
  }

  do
  {
    /* r-w-r (initial i = 1)*/
    for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
    {
      if (baseAddress[offset] != data_patterns_cfds[i])
      {
        //increment fault counter and visualize fault
      }

      baseAddress[offset] = data_patterns_cfds[i+1];

      if (baseAddress[offset] != data_patterns_cfds[i+1])
      {
        //increment fault counter and visualize fault
      }
    }

    /* r-w-w-r */
    for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
    {
        if (baseAddress[offset] != data_patterns_cfds[i+1])
        {
          //increment fault counter and visualize fault
        }

        baseAddress[offset] = data_patterns_cfds[i+2];
        baseAddress[offset] = data_patterns_cfds[i+3];

        if (baseAddress[offset] != data_patterns_cfds[i+3])
        {
          //increment fault counter and visualize fault
        }
    }

    i += 3;
  }while(i < (NUMB_OF_CFDST_PATTERN - 4));

  /* r-w-r */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    if (baseAddress[offset] != data_patterns_cfds[i])
    {
      //increment fault counter and visualize fault
    }

    baseAddress[offset] = data_patterns_cfds[i+1];

    if (baseAddress[offset] != data_patterns_cfds[i+1])
    {
      //increment fault counter and visualize fault
    }
  }

  /* Mend */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
    if (baseAddress[offset] != data_patterns_cfds[i+1])
    {
      //increment fault counter and visualize fault
    }
  }
}

void WOMPseudoTransparentMarchCMinus( void * pvParameters )
{
```

```
  //... initializing and saving intial memory content ...

  while(1)
  {
    for (pattern_num = 0; pattern_num < NUMB_OF_CFID_PATTERN; pattern_num++)
      MarchCMinus(data_patterns_cfid[pattern_num]);

    for (pattern_num = 0; pattern_num < NUMB_OF_CFDST_PATTERN; pattern_num++)
      CFdsTest();

    //... restoring intial memory content and preparing end of test...
  }
}
```

B.2 Transparent March SS Test with Multiple Input Signature Register

Listing B.3: Code for transparent MarchSS with the usage of a Multiple Input Signature Register (MISR based on the implementation in [PKK11])

```
static unsigned short current_signature = 42;

static void misr_encode(unsigned short value)
{
  unsigned short carry;

  carry = (current_signature ^
          (current_signature >> 2) ^
          (current_signature >> 3) ^
          (current_signature >> 5)) & 1;

  current_signature = current_signature >> 1;
  current_signature = current_signature ^ value;
  current_signature = current_signature ^ (carry << 15);
}

static void misr_decode(unsigned short value)
{
  unsigned short carry;

  current_signature = current_signature ^ value;
  carry = ((current_signature >> 15) ^
           (current_signature >> 4) ^
           (current_signature >> 2) ^
           (current_signature >> 1)) & 1;

  current_signature = current_signature << 1;
  current_signature = current_signature | carry;
}

static void TSMarchSS(const unsigned short *initial_memory_content)
{
  //... initializing ...

  /* M0 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
    misr_encode(~baseAddress[offset]);

  /* M1 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    misr_encode(baseAddress[offset]);
    misr_encode(baseAddress[offset]);
    baseAddress[offset] = baseAddress[offset];
    misr_encode(baseAddress[offset]);
    baseAddress[offset] = ~baseAddress[offset];
  }

  /* M2 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    misr_encode(baseAddress[offset]);
    misr_encode(baseAddress[offset]);
    baseAddress[offset] = baseAddress[offset];
    misr_encode(baseAddress[offset]);
    baseAddress[offset] = ~baseAddress[offset];
  }
```

```
  /* M3 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
    misr_decode(baseAddress[offset]);
    misr_decode(baseAddress[offset]);
    baseAddress[offset] = baseAddress[offset];
    misr_decode(baseAddress[offset]);
    baseAddress[offset] = ~baseAddress[offset];
  }

  /* M4 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
    misr_decode(baseAddress[offset]);
    misr_decode(baseAddress[offset]);
    baseAddress[offset] = baseAddress[offset];
    misr_decode(baseAddress[offset]);
    baseAddress[offset] = ~baseAddress[offset];
  }

  /* M5 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
    misr_decode(baseAddress[offset]);

  if (current_signature != PRECOMPUTED_MARCHSS_SIGNATURE)
  {
    //signalizing fault occurence
  }
  else
    current_signature = 42;
}

void WOMTransparentMarchSS( void * pvParameters )
{
  //... initializing ...

  while(1)
  {
    TSMarchSS(initial_memory_content);

    //... preparing end of test...
  }
}
```

B.3 TWM-TA-modified Abraham Test

Listing B.4: Code for TWM-TA-modified Abraham Test

```
static void TSAbraham(const unsigned short *initial_memory_content)
{
  //... initializing ...

  /* M1 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    if (baseAddress[offset] != (unsigned short) initial_memory_content[offset])
    {
     //increment fault counter and visualize fault
    }
    baseAddress[offset] = ~baseAddress[offset];
  }

  /* M2 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
    if (baseAddress[offset] != (unsigned short) ~initial_memory_content[offset])
    {
     //increment fault counter and visualize fault
    }
  }

  /* M3 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    if (baseAddress[offset] != (unsigned short) ~initial_memory_content[offset])
    {
     //increment fault counter and visualize fault
    }
    baseAddress[offset] = ~baseAddress[offset];
  }
```

```
/* M4 */
for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
{
  if (baseAddress[offset] != (unsigned short) initial_memory_content[offset])
  {
   //increment fault counter and visualize fault
  }
}

/* M5 */
for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
{
  if (baseAddress[offset] != (unsigned short) initial_memory_content[offset])
  {
   //increment fault counter and visualize fault
  }
  baseAddress[offset] = ~baseAddress[offset];
}

/* M6 */
for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
{
  if (baseAddress[offset] != (unsigned short) ~initial_memory_content[offset])
  {
   //increment fault counter and visualize fault
  }
}

/* M7 */
for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
{
  if (baseAddress[offset] != (unsigned short) ~initial_memory_content[offset])
  {
   //increment fault counter and visualize fault
  }
  baseAddress[offset] = ~baseAddress[offset];
}

/* M8 */
for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
{
  if (baseAddress[offset] != (unsigned short) initial_memory_content[offset])
  {
   //increment fault counter and visualize fault
  }
}

/* M9 */
for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
{
  if (baseAddress[offset] != (unsigned short) initial_memory_content[offset])
  {
   //increment fault counter and visualize fault
  }
  baseAddress[offset] = ~baseAddress[offset];
  baseAddress[offset] = ~baseAddress[offset];
}

/* M10 */
for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
{
  if (baseAddress[offset] != (unsigned short) initial_memory_content[offset])
  {
   //increment fault counter and visualize fault
  }
}

/* M11 */
for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
{
  if (baseAddress[offset] != (unsigned short) initial_memory_content[offset])
  {
   //increment fault counter and visualize fault
  }
  baseAddress[offset] = ~baseAddress[offset];
  baseAddress[offset] = ~baseAddress[offset];
}

/* M12 */
for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
{
  if (baseAddress[offset] != (unsigned short) initial_memory_content[offset])
  {
   //increment fault counter and visualize fault
  }
}

/* M13 */
```

```
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    baseAddress[offset] = ~baseAddress[offset];
  }

  /* M14 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    if (baseAddress[offset] != (unsigned short) ~initial_memory_content[offset])
    {
     //increment fault counter and visualize fault
    }
    baseAddress[offset] = ~baseAddress[offset];
    baseAddress[offset] = ~baseAddress[offset];
  }

  /* M15 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
    if (baseAddress[offset] != (unsigned short) ~initial_memory_content[offset])
    {
     //increment fault counter and visualize fault
    }
  }

  /* M16 */
  for (offset = (MEMORY_REGION_SIZE - 1); offset >= 0; offset--)
  {
    if (baseAddress[offset] != (unsigned short) ~initial_memory_content[offset])
    {
     //increment fault counter and visualize fault
    }
    baseAddress[offset] = ~baseAddress[offset];
    baseAddress[offset] = ~baseAddress[offset];
  }

  /* M17 */
  for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
  {
    if (baseAddress[offset] != (unsigned short) ~initial_memory_content[offset])
    {
     //increment fault counter and visualize fault
    }
    baseAddress[offset] = ~baseAddress[offset];
  }
}

static void ATMarch(const unsigned short *initial_memory_content)
{
  short patterns[4] = {0x5555, 0x3333, 0x71C7, 0x0F0F};
  //... initializing ...

  for (i = 0; i < 4; i++)
  {
    pattern = patterns[i];

    /* M1 */
    for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
    {
      baseAddress[offset] = initial_memory_content[offset] ^ pattern;
      baseAddress[offset] = initial_memory_content[offset] ^ ~pattern;

      if (baseAddress[offset] != (unsigned short)(initial_memory_content[offset] ^ ~pattern))
      {
      //increment fault counter and visualize fault
      }

      baseAddress[offset] = initial_memory_content[offset] ^ pattern;
      if (baseAddress[offset] != (unsigned short)(initial_memory_content[offset] ^ pattern))
      {
      //increment fault counter and visualize fault
      }
    }

    /* M2 */
    for (offset = 0; offset < MEMORY_REGION_SIZE; offset++)
      baseAddress[offset] = initial_memory_content[offset];
  }
}

void WOM_TWM_TA_Abraham( void * pvParameters )
{
  //... initializing and saving intial memory content ...

  while(1)
  {
    TSAbraham(initial_memory_content);
    ATMarch(initial_memory_content);
```

```
    //... restoring intial memory content and preparing end of test...
  }
}
```

B.4 CPU-core Tests

Listing B.5: Code for CPU-core element tests (adpated from [Pre13])

```
// ...

int logic_test_sub(volatile unsigned int *x, volatile unsigned int *y, volatile unsigned int *z)
{
  unsigned int  a = 0, b = 0, c = 0, d = 0, e = 0, f = 0;

  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;
}

void CPUCoreTestALU( void * pvParameters )
{
  // ... initializing ...

  while (1)
  {
    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))
    {
      //increment fault counter and visualize fault
    }
   //... preparing end of test ...
  }
}

void CPUCoreTestDividerShifter( void * pvParameters )
{
  // ... initializing ...

  while (1)
  {
    x = 1;
    y = 1;
    z = 0;

    while (z != 32)
```

```
    {
      x <<= 1;
      y *= 2;
      z++;

      if (x != y)
         //increment fault counter and visualize fault
   }

    while (z != 0)
    {
      x >>= 1;
      y /= 2;
      z--;

      if (x != y)
         //increment fault counter and visualize fault
    }
   //... preparing end of test ...
  }
}

static inline unsigned int read_cpsr(void)
{
  unsigned int cpsr;

  asm volatile("svc 3\n\t"
               "mrs     %0, CPSR\n\t"
               "svc 4\n\t"
      :  "=r" (cpsr) : );

  return cpsr;
}

static inline void write_cpsr(unsigned int cpsr_val)
{
  asm volatile("svc 3\n\t"
               "msr     cpsr, %0\n\t"
               "svc 4\n\t"
               : : "r" (cpsr_val));
}

void CPUCoreTestConditionFlags( void * pvParameters )
{
  // ... initializing ...

  while (1)
  {
    /*
     * set CF and ZF; reset NF, QF and VF
     */
    asm volatile(
        "ldr     r1, =0xffffffff\n\t"
        "ldr     r2, =0x00000001\n\t"
        "adds    r0, r1, r2\n\t");

    cpsr = read_cpsr();
    //check values of condition flags and, if necessary, increment fault
    //counter and visualize fault


    /*
     * set NF and VF; reset CF, ZF and QF
     */
    asm volatile(
        "ldr     r1, =0x7fffffff\n\t"
        "ldr     r2, =0x00000001\n\t"
        "adds    r0, r1, r2\n\t");

    cpsr = read_cpsr();
    //check values of condition flags and, if necessary, increment fault
    //counter and visualize fault

    /*
     * set QF;
     */
    asm volatile(
        "ldr     r1, =0x7fff7fff\n\t"
        "ldr     r2, =0x00010001\n\t"
        "qadd    r0, r1, r2\n\t");

    cpsr = read_cpsr();
    //check values of condition flags and, if necessary, increment fault
    //counter and visualize fault

    /* clear QF (it remains set until explicitly
     * cleared by an MSR instruction writing to
     * the CPSR.
```

```
      */
     cpsr &= ~(1 << QF);
     write_cpsr(cpsr);

   //... preparing end of test ...
  }
}

void CPUCoreTestMultiplierAdder( void * pvParameters )
{
  // ... initializing ...

  while (1)
  {
    x = 0;
    y = 0;
    return_value = 0;

    while(x != 0x11111110)
    {
      y = 0;
      while(y != 0x11111110)
      {
        m_res = (unsigned long long)x * (unsigned long long)y;
        z1 = (unsigned int)m_res;
        z2 = (unsigned int)(m_res >> 32);

        a_res = z1 + z2;
        temp = (unsigned long long)return_value + (unsigned long long)a_res;
        return_value = (unsigned int)temp;
        carry = (temp >> 32);
        return_value = return_value + carry;

        y = y + 0x11111111;
      }
      x = x + 0x11111111;
    }

    if(return_value != 0xffffffdc)
     //increment fault counter and visualize fault

   //... preparing end of test ...
  }
}
```

B.5 Register Tests

Listing B.6: Code for the Register Tests

```
#define TEST_REGISTER_NAME    r10
#define TEST_REGISTER_NUM     10
#define TEST_REGISTER_AUX(X)  #X
#define TEST_REGISTER(X)      TEST_REGISTER_AUX(X)

unsigned int inline setAndTestRegister(unsigned int value)
{
  unsigned int ret = 0;

  asm volatile("mov  "TEST_REGISTER(TEST_REGISTER_NAME)", %1\n\t"
               "mov  %0, "TEST_REGISTER(TEST_REGISTER_NAME)"\n\t"
               : "=r" (ret) :  "r" (value) : TEST_REGISTER(TEST_REGISTER_NAME) );
  return ret;
}

void RegisterTestWalkingBit( void * pvParameters )
{
  // ... initializing ...

  while (1)
  {
     x = 1;
     z = 0;
     register_value = 1;

     while (z != 32)
     {
       read = setAndTestRegister(register_value);

       if (x != read)
        //increment fault counter and visualize fault
```

```
        register_value <<= 1;
        x *= 2;
        z++;
     }
   //... preparing end of test ...
  }
}

#define READ_REGISTER(X)                    \
     asm volatile("mov  %0, "TEST_REGISTER(TEST_REGISTER_NAME)"\n\t" \
                  : "=r" (X) :: TEST_REGISTER(TEST_REGISTER_NAME) );

#define WRITE_REGISTER(X)                   \
     asm volatile("mov  "TEST_REGISTER(TEST_REGISTER_NAME)", %0\n\t" \
                  :: "r" (X) : TEST_REGISTER(TEST_REGISTER_NAME) );

static void TSMarchSS_SingleCell(const unsigned int initial_register_content)
{
  unsigned int temp = 0;

  /* M1 */
  READ_REGISTER(temp);

  if (temp != initial_register_content)
    //increment fault counter and visualize fault

  READ_REGISTER(temp);

  if (temp != initial_register_content)
    //increment fault counter and visualize fault

  temp = initial_register_content;
  WRITE_REGISTER(temp);
  READ_REGISTER(temp);

  if (temp != initial_register_content)
    //increment fault counter and visualize fault

  temp = ~initial_register_content;
  WRITE_REGISTER(temp);

  // ... other march elements ...
}

static void ATMarch_SingleCell(const unsigned int initial_register_content)
{
  unsigned int patterns[4] = {0x55555555, 0x33333333, 0xC71C71C7, 0x0F0F0F0F};
  unsigned int pattern = 0, temp = 0;
  int i = 0;

  for (i = 0; i < 4; i++)
  {
    pattern = patterns[i];
    temp = initial_register_content ^ pattern;
    WRITE_REGISTER(temp);

    temp = initial_register_content ^ ~pattern;
    WRITE_REGISTER(temp);
    READ_REGISTER(temp);

    if (temp != (initial_register_content ^ ~pattern))
      //increment fault counter and visualize fault

    temp = initial_register_content ^ pattern;
    WRITE_REGISTER(temp);
    READ_REGISTER(temp);

    if (temp != (initial_register_content ^ pattern))
      //increment fault counter and visualize fault

    WRITE_REGISTER(initial_register_content);
  }
}

void RegisterTestMarchSS( void * pvParameters )
{
  // ... initializing ...

  while (1)
  {
    TSMarchSS_SingleCell(initial_register_content);
    ATMarch_SingleCell(initial_register_content);

   //... preparing end of test ...
  }
}
```

Bibliography

[AAA+90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *Transactions on Software Engineering (TSE'90)*, 16(2):166–182, February 1990.

[AAG01] Z. Al-Ars and A. J. van de Goor. Static and dynamic behavior of memory cell array opens and shorts in embedded DRAMs. In *Design, Automation and Test in Europe*, pages 496–503, 2001.

[ARM02] ARM Limited. *ARM9EJ-S - Technical Reference Manual*. ARM Limited, Cambridge, England, United Kingdom, r1p2 edition, September 2002.

[ARM08] ARM Limited. *ARM926EJ-S - Technical Reference Manual*. ARM Limited, Cambridge, England, United Kingdom, r0p5 edition, June 2008.

[AVFK03] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. GOOFI: Generic Object-Oriented Fault Injection Tool. In *International Conference on Dependable Systems and Networks (DSN'03)*, pages 668–668, June 2003.

[BDL96] J. M. Bieman, D. Dreilinger, and Lijun Lin. Using fault injection to increase software test coverage. In *International Symposium on Software Reliability (ISSRE'96)*, pages 166–174, Oct 1996.

[Bel05] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Annual Conference on USENIX Annual Technical Conference*, ATEC'05, pages 41–46, 2005.

[BGGG00] J.-C. Baraza, J. Gracia, D. Gil, and P. Gil. A prototype of a VHDL-based fault injection tool. In *International Symposium on Defect and Fault Tolerance in VLSI Systems(DFT'00)*, pages 396–404, October 2000.

[BKJ+12] M. Becker, C. Kuznik, M. M. Joy, Tao Xie, and W. Mueller. Binary mutation testing through dynamic translation. In *International Conference on Dependable Systems and Networks (DSN'12)*, pages 1–12, June 2012.

[BNR00] T.J. Bergfeld, D. Niggemeyer, and E.M. Rudnick. Diagnostic testing of embedded memories using BIST. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 305–309, 2000.

[BPRR98] A. Benso, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. EXFI: A Low-cost Fault Injection System for Embedded Microprocessor-based Boards. *Transactions on Design Automation of Electronic Systems (TODAES'98).*, 3(4):626–634, October 1998.

[CD01] Li Chen and S. Dey. Software-based self-testing methodology for processor cores. *Computer-Aided Design of Integrated Circuits and Systems*, 20(3):369–380, Mar 2001.

[Cho92] G. S. Choi. Focus: an experimental environment for fault sensitivity analysis. 41(12):1515–1526, December 1992.

[Chy09] S. Chylek. Collecting program execution statistics with Qemu processor emulator. In *International Multiconference on Computer Science and Information Technology (IMCSIT'09)*, pages 555–558, October 2009.

[CMS98] J. Carreira, H. Madeira, and J. G. Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *Transactions on Software Engineering (TSE'98)*, 24(2):125–136, February 1998.

[DAF+14] Bryce Denney, Greg Alexander, Todd Fries, Donald Becker, and Tim Butler. Bochs IA-32 Emulator Project. `http://bochs.sourceforge.net/`, 2014. last accessed: 2014-03-19.

[DC07] F. M. David and R. H. Campbell. Building a Self-Healing Operating System. In *International Conference on Dependable, Autonomic and Secure Computing (DASC'07)*, pages 3–10, September 2007.

[ELO01] L. Entrena, C. Lópe, and E. Olías. Automatic Generation of Fault Tolerant VHDL Designs in RTL. In *Forum on Design Languages (FDL'01)*, September 2001.

[Exi06] Exida. IEC 61508 Overview Report - A Summary of the IEC 61508 Standard for Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. Technical Report Version 2.0, Exida, January 2006.

[FCJ+14] Keir Fraser, Ian Campbell, Ian Jackson, Jan Beulich, and Tim Deegan. The Xen hypervisor. `http://www.xenproject.org/`, 2014. last accessed: 2014-03-19.

[Fre11] Freescale Semiconductor. *i.MX28 EVK Hardware User's Guide*. Freescale Semiconductor, Austin, Texas, USA, 0 edition, July 2011. Document Number: 924-76415.

[FSK98] P. Folkesson, S. Svensson, and J. Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *International Symposium on Fault-Tolerant Computing (FTCS'98)*, pages 284–293, June 1998.

[Fuc96] E. Fuchs. An Evaluation of the Error Detection Mechanisms in MARS Using Software-Implemented Fault Injection. In *European Dependable Computing Conference on Dependable Computing (EDCC'96)*, EDCC-2, pages 73–90, 1996.

[GAC02] A. J. van de Goor, M.S. Abadir, and A. Carlin. Minimal test for coupling faults in word-oriented memories. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 944–948, 2002.

[GFP09] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation. In *Hardware/Software Codesign and System Synthesis*, CODES+ISSS'09, pages 71–80, 2009.

[GPZ04] D. Gizopoulos, A. Paschalis, and Y. Zorian. *Embedded Processor-Based Self-Test*. Kluwer Academic Publishers, 2004.

[Gre14] GreenSocs Ltd. QEMU-SystemC, a hardware co-simulator. `http://www.greensocs.com/projects/QEMUSystemC`, 2014. last accessed: 2014-03-19.

[GS95] J. Güthoff and V. Sieh. Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method. In *International Symposium on Fault-Tolerant Computing (FTCS'95)*, FTCS'95, pages 196–206, June 1995.

[GT98] A. J. van de Goor and I. B. S Tlili. March tests for word-oriented memories. In *Design, Automation and Test in Europe*, pages 501–508, Feb 1998.

[GV90] A. J. van de Goor and C. A. Verruijt. An Overview of Deterministic Functional RAM Chip Testing. *ACM Comput. Surv.*, 22(1):5–33, March 1990.

[HGR02] S. Hamdioui, A. J. van de Goor, and M. Rodgers. March SS: A Test for All Static Simple RAM Faults. In *Memory Technology, Design and Testing (MTDT 2002)*, pages 95–100, 2002.

[HSR95] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: an integrated software fault injection environment for distributed real-time systems. In *International Computer Performance and Dependability Symposium (IPDS'95)*, pages 204–213, April 1995.

[HTI97] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[IEC10] International Electrotechnical Commission. Functional safety of electrical/-electronic/programmable electronic safety-related system. Norm IEC 61508, 2010.

[Kal05] D. Kalinsky. Architecture of safety-critical systems. `http://www.embedded.com/design/prototyping-and-development/4006464/Architecture-of-safety-critical-systems`, 2005. last accessed: 2014-02-23.

[KGPZ02] N. Kranitis, D. Gizopoulos, A. Paschalis, and Y. Zorian. Instruction-based self-testing of processor cores. In *VLSI Test Symposium (VTS 2002)*, pages 223–228, 2002.

[KKA92] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: a tool for the validation of system dependability properties. In *International Symposium on Fault-Tolerant Computing (FTCS'92)*, pages 336–344, July 1992.

[KLD+94] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *Micro*, 14(1):8–23, February 1994.

[KML+06] N. Kranitis, A. Merentitis, N. Laoutaris, G. Theodorou, A. Paschalis, D. Gizopoulos, and C. Halatsis. Optimal Periodic Testing of Intermittent Faults In Embedded Pipelined Processor Applications. In *Design, Automation and Test in Europe (DATE'06)*, volume 1, pages 1–6, March 2006.

[Kni02] J. C. Knight. Safety critical systems: Challenges and directions. In *International Conference on Software Engineering (ICSE'02)*, pages 547–550, May 2002.

[KPGZ02] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian. Effective software self-test methodology for processor cores. In *Design, Automation and Test*, pages 592–597, 2002.

[KVM14] KVM Developers. Kernel Based Virtual Machine (KVM). `http://valgrind.org/`, 2014. last accessed: 2014-03-19.

[LCL08] Tai-Hua Lu, Chung-Ho Chen, and Kuen-Jong Lee. A Hybrid Software-based Self-testing Methodology for Embedded Processor. In *ACM Symposium on Applied Computing*, SAC'08, pages 1528–1534, 2008.

[LESO13] M. Linder, A. Eder, U. Schlichtmann, and K. Oberlander. An Analysis of Industrial SRAM Test Results - A Comprehensive Study on Effectiveness and Classification of March Test Algorithms. volume PP, pages 1–1, 2013. This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

[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 in Europe Conference and Exhibition*, volume 1, pages 574–579, March 2005.

[MRMS94] H. Madeira, M. Rela, F. Moreira, and J. G. Silva. RIFLE: A General Purpose Pin-level Fault Injector. In *European Dependable Computing Conference (EDCC'94)*, volume 852 of *Lecture Notes in Computer Science*, pages 197–216, June 1994.

[Nic96] M. Nicolaidis. Theory of transparent BIST for RAMs. *Computers*, 45(10):1141–1156, October 1996.

[NTA78] R. Nair, S.M. Thatte, and J.A. Abraham. Efficient Algorithms for Testing Semiconductor Random-Access Memories. *Computers*, C-27(6):572–576, June 1978.

[Ora14] Oracle Corporation. The VirtualBox PC virtualizer. `http://virtualbox.org/`, 2014. last accessed: 2014-03-19.

[PGK+01] 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*, pages 92–96, 2001.

[PGSR10] M. Psarakis, D. Gizopoulos, E. Sanchez, and M.S. Reorda. Microprocessor Software-Based Self-Testing. *IEEE Design Test of Computerss*, 27(3):4–19, May 2010.

[PKK11] Christopher Preschern, Nermin Kajtazovic, and Christan Kreiner. Software Based Self Tests - Literature Research and Conceptual Test Programs. Technical report, Institute for Technical Informatics, November 2011.

[Pre13] C. Preschern. Verifying Generic CPU Safety-Tests with Fault Injection. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2013.

[Pul01] L. L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Artificial Intelligence. Artech House computing library, 2001.

[QEM14] QEMU developers. QEMU Internals. `http://qemu.weilnetz.de/qemu-tech.html`, 2014. last accessed: 2014-03-19, frequently updated.

[RPB+01] C. Rousselle, M. Pflanz, A. Behling, T. Mohaupt, and H. T. Vierhaus. A register-transfer-level fault simulator for permanent and transient faults in embedded processors. In *Design, Automation and Test in Europe (DATE'01)*, pages 811–, 2001.

[SBS08] André Borin Soares, Alexsandro Cristovão Bonatto, and Altamiro Amadeu Susin. A New March Sequence to Fit DDR SDRAM Test in Burst Mode. In *Integrated Circuits and System Design*, SBCCI'08, pages 28–33, 2008.

[SPS+94] V. Sieh, A. Pataricza, B. Sallay, W. Hohl, J. Hönig, and B. Benyó. Fault Injection Based Validation of Fault-Tolerant Multiprocessors. In *Symposium on Microcomputer and Microprocessor Application*, volume 1, pages 85–94, October 1994.

[STB97] V. Sieh, O. Tschache, and F. Balbach. VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions. In *International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 32–36, June 1997.

[SV+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 *International Symposium on Fault-Tolerant Computing (FTCS'88)*, pages 102–107, June 1988.

[TA80] S. M. Thatte and J. A. Abraham. Test Generation for Microprocessors. *IEEE Trans. Comput.*, 29(6):429–441, June 1980.

[TB06] M. Tuna and M. 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*, pages 67–72, Buenos Aires, Argentina, 2006.

[TI95] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE fault injection tool. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems*, volume 977 of *MMB '95*, pages 26–40, 1995.

[TP07] T. Tamandl and P. Preininger. Online Self Tests for Microcontrollers in Safety Related Systems. In *Industrial Informatics*, volume 1, pages 137–142, June 2007.

[TUE14a] TUEV Rheinland. Functional safety. `http://www.certipedia.com/fs-products/functional-safety`, 2014. last accessed: 2014-02-20.

[TUE14b] TUEV SUED. Functional safety. `http://www.tuv-sud.com/activity/focus-topics/functional-safety`, 2014. last accessed: 2014-02-19.

[Val14] Valgrind Developers. Valgrind, an open-source memory debugger for x86-GNU/Linux. `http://valgrind.org/`, 2014. last accessed: 2014-03-19.

[VES13] I. Voyiatzis, C. Efstathiou, and C. Sgouropoulou. Symmetric transparent online BIST for arrays of word-organized RAMs. In *Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 122–127, March 2013.

[VMW14] VMWare Developers. The VMWare PC virtualizer. `http://www.vmware.com/`, 2014. last accessed: 2014-03-19.

[WEL+13] L. Wanner, S. Elmalaki, L. Lai, P. Gupta, and M. Srivastava. VarEMU: An emulation testbed for variability-aware software. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*, pages 1–10, September 2013.

[WIT12a] WITTENSTEIN HighIntegritySystems. *SAFERTOS USER MANUAL FOR THE GCC I.MX28 - PRODUCT VARIANT*. WITTENSTEIN HighIntegritySystems, Browns Court, Bristol, England, 1.0 edition, January 2012.

[WIT12b] WITTENSTEIN HighIntegritySystems. *USING THE SAFERTOS I.MX28 GCC DEMO - APPLICATION NOTE: #34-172-AN-015*. WITTENSTEIN HighIntegritySystems, Browns Court, Bristol, England, 3.0 edition, January 2012.

[XX12] Jun Xu and Ping Xu. The Research of Memory Fault Simulation and Fault Injection Method for BIT Software Test. In *Instrumentation, Measurement, Computer, Communication and Control (IMCCC'12)*, pages 718–722, December 2012.

[YH99] V.N. Yarmolik and S. Hellebrand. Symmetric transparent BIST for RAMs. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 702–707, March 1999.

[YHW99] V.N. Yarmolik, S. Hellebrand, and H.J. Wunderlich. Symmetric transparent BIST for RAMs. In *Design, Automation and Test*, pages 702–707, March 1999.

[YPH13] Li Yi, Xu Ping, and Wan Han. A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing. In *International Symposium on Computer, Communication, Control and Automation (ISCCCA-13)*, 2013.

[ZAV04] H. Ziade, R. Ayoubi, and R. Velazco. A Survey on Fault Injection Techniques. *The International Arab Journal of Information Technology (IAJIT'04)*, 1(2):171–186, July 2004.