Marc Schober, BSc

Generic CPU Self-Tests: Instruction Coverage and Optimizing Compilers

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner

Institute of Technical Informatics

Graz, September 2018

Kurzfassung

Ein Fehler in einem kritischen System kann ernsthafte Auswirkungen haben. Software basierende Selbsttests werden oft verwendet um in ausfallsicheren Systemen die fehlerfreie Funktionalität der Hardware sicherzustellen. Normalerweise werden diese Selbsttests speziell für eine Hardwareplattform in Assembler implementiert. Wenn ein anderer Prozessor verwendet werden soll, erfordert dies eine neue Selbsttest-Implementierung mit hohem Aufwand für die Spezifikation, Implementierung, Prüfung und Zertifizierung. Die Nachfrage nach fehlerfreien Systemen ist in den letzten Jahren unter anderem wegen sogenannten intelligenten Heimgeräten stark gestiegen.

Diese Masterarbeit gibt einen Überblick über kritische und ausfallsichere Systeme im Allgemeinen mit Fokus auf dem Sicherheitsstandard IEC 61508. Selbsttest Technologien werden im Detail präsentiert und bewertet.

Der Hauptteil der Arbeit beschäftigt sich mit dem Design und der Implementierung eines Hardwareunabhängigen Selbsttests in der Programmiersprache C. Dabei wurde eine anwendungsbasierende Herangehensweise gewählt: nur von der Anwendung genutzte Ressourcen müssen getestet werden. Um die Vollständigkeit des Tests zu zeigen, wurde eine Instruktions-Abdeckungs-Analyse verwendet. Mithilfe eines Compilers wird sowohl der Selbsttest als auch die Anwendung in Assemblercode kompiliert. Die Assembler-Instruktionen der Anwendung werden dann mit den Instruktionen des Selbsttests verglichen. Wenn alle Instruktionen und verwendeten Statusregister der Anwendung vom Selbsttest überprüft werden, wird der Test akzeptiert.

Ein weiterer Schwerpunkt dieser Masterarbeit beschäftigt sich mit den Auswirkungen von optimierenden Compiler auf Selbsttests und die dadurch verwendeten und getesteten Instruktionen. Der implementierte Selbsttest wurde mit vier Applikationen getestet, eine dieser Applikationen wurde mit optimierenden Compiler übersetzt, bei den anderen drei wurde auf Optimierungen verzichtet. Bei der Verwendung von optimierenden Compiler war eine volle Instruktions-Abdeckung aufwendig zu erzielen, aber auch diese wurde erreicht.

Abstract

Failures in safety-critical systems can be serious. Software-based self-tests are often used to ensure the correct functionality of the hardware. Usually, those self-tests are implemented in hardware-dependent assembly code. A CPU change needs a new self-test implementation with high effort for the specification, implementation, testing, and certification. Not only but also due to the smart home devices the demand for safety-critical systems is increasing. Also, the requested features of safety-critical systems are rising.

This master thesis gives an overview of safety-critical systems in general and of the IEC 61508 safety standard in particular. Software-based self-test approaches are presented in detail. The main part of this thesis includes the design and implementation of a C-based hardware-independent self-test for the CPU. The approach followed in this paper is application-based, which means that only used resources are getting tested. To show the completeness of the test an instruction coverage analysis was done: The compiler generates the binary and the assembly list files for the application and the tests. In the next step the instructions utilized for the application are compared with those checked by the tests. Only if all instructions and CPU flag checks used by the application are checked by the tests, the self-test is accepted. This thesis also shows the challenges of such an approach when optimizing compilers are utilized. The implemented self-test was checked against four applications, one of which was compiled with compiler optimization techniques and the other three were compiled without any optimization. Full coverage for optimizing compilers was hard to achieve, but the final implementation reached full coverage for all tested applications.

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 indicated all material which has been quoted either literally or by content from the sources used. 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 in cooperation with Siemens Digital Factory, Graz.

At this point I want to thank my girlfriend Christina and friend and business partner Mario for their encouragement and understanding during my study and master thesis.

Special thanks to my master thesis supervisor, Christian Kreiner, for his guidance and support. His passing away is a huge loss.

I am grateful for the support and motivation of Eugen Brenner who took over the supervision. His recommendations were very much appreciated for the organization and theoretical part of my master thesis.

I also want to thank Gerhard Schönfelder and David Ferenczi from Siemens who helped me with technical approaches and suggestions.

Graz, September 2018 Marc Schober

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Safety-critical system may cause harm if they fail. To prevent or at least to decrease the probability of dramatic effects fail-safe systems need to be used. An important part of the implementation of such systems is to ensure the correct functionality of the hardware, which is why a variety of different self-tests have been developed for this purpose. The aim of this thesis is to develop hardware independent C-based self-tests which are suitable for optimized compiler as well. Even the effectiveness of those tests should be shown without any detailed information about the hardware under test.

1.1 Motivation

Years ago safety-critical systems were only needed for specialized applications. But the demand for safety-critical systems was increasing dramatically. Nowadays everyday objects are computer controlled. For instance, a car involves a traditional safety-critical system, but it is by far not the only one. Even some lower priced systems can be seen as safety-critical. What happens if the oven in the kitchen does not work anymore as expected and causes a fire in the apartment? A failure of some of those systems may have dramatic effects, including harm or even death of people.

In processor-based fail-safe systems it is essential to guarantee the correct functionality of the hardware. With the increment of new features testing costs are rising significantly. Hardware testing is one of the significant parts. While development costs per transistor decreased dramatically, the testing costs per transistor were not fallen the same way. In the early 2000s it was not unlikely that the production costs of transistors reached a point where testing was more expensive than producing [WST10].

Even with new test methods, which were introduced to lower the costs of testing, the 2015 edition of the international technology roadmap for semiconductors expected a rise of testing prices with most current approaches. *"The cost of testing continues to rise yet system integrators expect prices to stay constant or lower even with increases in performance and function. Increasing performance and adding more functions requires higher accuracy tests and additional tests, which should normally increase testing costs."* [SIA15]

In the last decades new methods were introduced in order to reduce the development costs, but most of them were still implemented for a specific hardware and not portable to other hardware platforms. The change in new hardware usually requires to rewrite the hardware-based assembly code. From a safety point of view, the work is not done by adapting the software to the new hardware platform. It is also a time consuming and costly job to prove that the tests are sufficient enough for a fail-safe system. Due to this hardware dependencies, the switch to a new platform is very time consuming and expensive. [SS16] [Com11]

1.2 Goals

In 2011 Siemens introduced a new approach to reduce the hardware dependencies. The idea is to implement the test of the CPU in hardware independent C code. The proof is made by comparing the assembly (ASM) instructions utilized by the application with the ASM instructions checked by the test. Only if all used ASM instructions also occur in the test the CPU core test is accepted. This new approach was implemented for a part of the instruction tests, but other instructions were still tested based on hardware-dependent ASM instructions. Especially the instructions which need to be tested with optimizing compilers were not following the new approach and were implemented in ASM or challenging to understand and maintainable C code. The outcome of this master thesis should achieve the following goals or objectives:

- **Improved portability**
  The majority of the tests should be written in C, the use of processor-specific assembly code should be minimized

- **Hardware independent coverage measurement**
  The coverage measurement should be done by analyzing the used instructions.

- **Support for optimizing compilers**
  The coverage of the test should be given even if compiler optimization techniques are utilized.

- **Test slice runtimes shall be balanced**
  The self-tests are executed in test slices. The execution time of each test slice should be balanced and not exceed 150μs.

- **No testing of not utilized instructions**
  To reduce the time and space requirements the instruction test should only cover used instructions.

- **Extensibility for new instruction set architectures**
  When the instruction test is compiled for another hardware platform it is possible that the instruction test has to be extended. This extensions should be supported with well-defined interfaces and helper functions.

Furthermore, an essential part of this master thesis is to analyze the possibilities and drawbacks of a C based hardware independent instruction test, especially when used with optimizing compilers.

1.3 Outline

Chapter 2 gives the technical background about safety-critical systems with a focus on the safety standard IEC 61508. Different test techniques are presented and their advantages and drawbacks are shown. Moreover, the chapter provides a general overview on the approach of software-based self-tests with a particular focus on the safety standard IEC 61508. This part will present techniques of the SBST currently used.

The related work section focuses on the instruction set architecture (ISA) and also includes projects using generic portable instruction tests.

Chapter 3 provides a detailed description of the system requirements. The operations and operands, that need to be tested, are presented and classified. In addition to this, the approach to test the CPU flags will be analyzed. Furthermore, the difficulties of compiler optimizations techniques and generic software-based self-test (SBST) are demonstrated. This chapter also shows the concept of the instruction coverage analysis.

Chapter 4 describes the design and implementation of the SBST. For each type of operand the test design is shown. The chapter will also include additional tests, which are only needed to increase the coverage with optimizing compilers.

Chapter 5 presents the result of this master thesis, focusing on two real-life applications. One of those projects was compiled without the usage of compiler optimizations, for the second project those techniques were used. Here, the analysis will discuss the advantages and drawbacks. The results archived are individually presented and also compared to each other. Furthermore, the experienced difficulties are discussed here.

Chapter 6 summarizes the master thesis and shows possible suggestions for future work.

Chapter 2

Technical background and related work

This chapter gives a general introduction to safety-critical systems. The related work section presents some current existing approaches for SBST.

2.1 Safety-critical systems

Systems are called safety-critical when a malfunction may influence the environment directly or indirectly. A failure or malfunction of such a system may result in one or more of the following outcomes:

- Environmental harm
- Loss or damage of equipment
- Injury or even death to humans

2.1.1 Introduction to fail-safe systems

There is no such thing as zero-risk. No software design, especially when it comes to large projects, can foresee every operational possibility. Also, physical items have a failure rate higher than zero.

A fail-safe system accepts the fact that it may fail. The goal is to detect a fault in a very early stage and bring the system to a safe state. Neither a fail-safe system can guarantee zero-risk, even with the best possible test - not every fault can be detected. The aim of a fail-safe system is to reduce this risk to a tolerable level and to reduce its negative impact. [SS16]

Engineering a fail-safe system involves the identification of specific hazardous failures which may lead to severe consequences and development of techniques and measurements to reduce this risks. Any equipment (with or without software) whose failure may lead to severe effects are likely to be safety-related. [PH09]

2.1.2 IEC 61508

IEC 61508 is an international standard titled *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, which sets out a generic approach and which is used as a base to ensure that the systems are designed, implemented, operated and maintained to provide the required safety integrity level (SIL) for all kinds of industries.

This standard consist of 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 SILs
- **IEC 61508-6**: Guidelines on the application of IEC 61508-2 and IEC 61508-3
- **IEC 61508-7**: Overview of techniques and measures

To archive an IEC 61508 certification it is mandatory to follow the first three parts of the standard. Parts 4 - 7 contains additional information to understand and apply the required parts.

2.1.3 Maximum tolerable risk and safety integrity level

To set a quantified safety integrity target the maximum tolerable risk is needed. In IEC 61508 the fail-safe systems are categorized in safety integrity levels (SILs) based on the demand of the system and the residual risk. The IEC 61508 safety standard differentiates between low demand mode, high demand mode and continues mode. The classification is based on the operational frequency and test frequency. It is called low demand mode if the operating rate is less than once per year and smaller than twice of the test frequency. [SS16]

The levels range from SIL 1 to SIL 4:

- **SIL 1**: Implies good design practice, relatively easy to archive
- **SIL 2**: Requires proper design techniques and operating practice, not much harder to file than SIL 1
- **SIL 3**: Requires sophisticated competence and high effort, costs are significantly higher than for SIL 2
- **SIL 4**: The highest SIL level, onerous to archive. Costs will be extraordinarily high, and competence in all techniques is required

SIL	Continuous and high demand rate (dangerous failures/hr)	Low demand rate (probability of failure on demand)
4	$\geq 10^{-9} to < 10^{-8}$	$\geq 10^{-5} to < 10^{-4}$
3	$\geq 10^{-8} to < 10^{-7}$	$\geq 10^{-4} to < 10^{-3}$
2	$\geq 10^{-7} to < 10^{-6}$	$\geq 10^{-3} to < 10^{-2}$
1	$\geq 10^{-6} to < 10^{-5}$	$\geq 10^{-2} to < 10^{-1}$

Table 2.1: Average failure probability for different SILs

The main difference between the SILs is the quantification of hardware failures and the safe failure fraction rules. The requirements for SIL 1 and 2 levels are similar, as well as those for SIL 3 and 4. Not only the design and implementation of higher SIL levels are costly, but also the verification becomes more onerous. Table 2.1 shows the failure probability for different SILs. Note: If the continuous and high demand rates had been expressed in dangerous failures per year, the number would be numerically similar to the low demand rate probabilities.

The standard lists techniques and methods for all life cycles. Table 2.2 shows an example of techniques and methods recommended for software design and development. Based on the SIL level the techniques and methods are rated in 5 recommendation levels:

- **Mandatory**
  This technique or method has to be used.
- **Highly recommended**
  If this technique or method is not applied, there has to be a good reason why.
- **Recommended**
  This technique or method is recommended.
- **Neutral**
  The standard is not in favor or against the use of this technique or method.
- **Clearly not recommended**
  The technique or method is explicitly not recommended, there has to be a good reason if it is applied.

In some situations, a high SIL level, especially SIL 4, is tough to acquire, which is why additional levels of protection or redundancy could be applied instead. A configuration involving parallel elements may be used to claim an increment of one SIL. The independence of those parallel elements has to be shown by the use of approved techniques (e.g., functional diversity). Figure 2.1 shows the idea to achieve SIL 3 with two SIL 2 elements. [SS16]

Technique / measure	SIL 1	SIL 2	SIL 3	SIL 4
No dynamic objects	Recommended	Highly recommended	Highly recommended	Highly recommended
No dynamic variables	Neutral	Recommended	Highly recommended	Highly recommended
Limited use of pointer	Neutral	Recommended	Highly recommended	Highly recommended
Limited use of recursions	Neutral	Recommended	Highly recommended	Highly recommended

Table 2.2: Software design and development techniques and measures

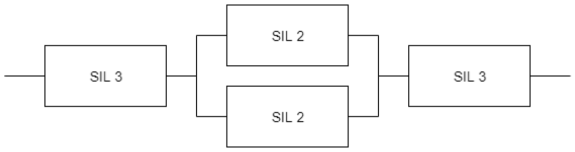


Figure 2.1: Two parallel SIL 2 elements are used to achieve SIL 3

2.1.4 Fault, error, failure

This section describes the keywords fault, error, and failure and explains the differences.

A **fault** is defined as an *"Abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function"* [Com11]. A fault may raise the risk of a system breakdown, but not all faults lead to an error. Whether an error occurs or not depends on the failure resistance and safety management Some systems, for example, may be equipped with redundancy and online error correction. A fault in a hardware part, which is not used at all, will usually not lead to an error.

An **error** is defined as the *"Discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition"* [Com11]. If a fault causes a system malfunction, it is called an error. For example, the system reads a value from a gate where a bit is stuck.

A **failure** is the *"Termination of the ability of a functional unit to perform a required function"* [Com11]. If a system is not able to handle an error and go to a safe state, it may become a failure. A failure may lead to accidents and loss of equipment, to the harm of the environment or injuries and even death of people.

2.1.5 Separation between safety-related and non-safety-related

Any equipment (with or without software) whose failure may be hazardous is likely to be safety-related. The non-safety-related system elements should be separated from the safety-related system elements. If such a separation between the system elements cannot be done the complete system should be treated as safety-related.

SIL 1 and 2 require a precise specification and separation between the safety-related and non-safety-related elements of the system with well-defined electrical/data interfaces.

For SIL 3 physical separation between the safety system and the non-safety-related components is needed. Electrical and data interfaces should be clearly defined and physical

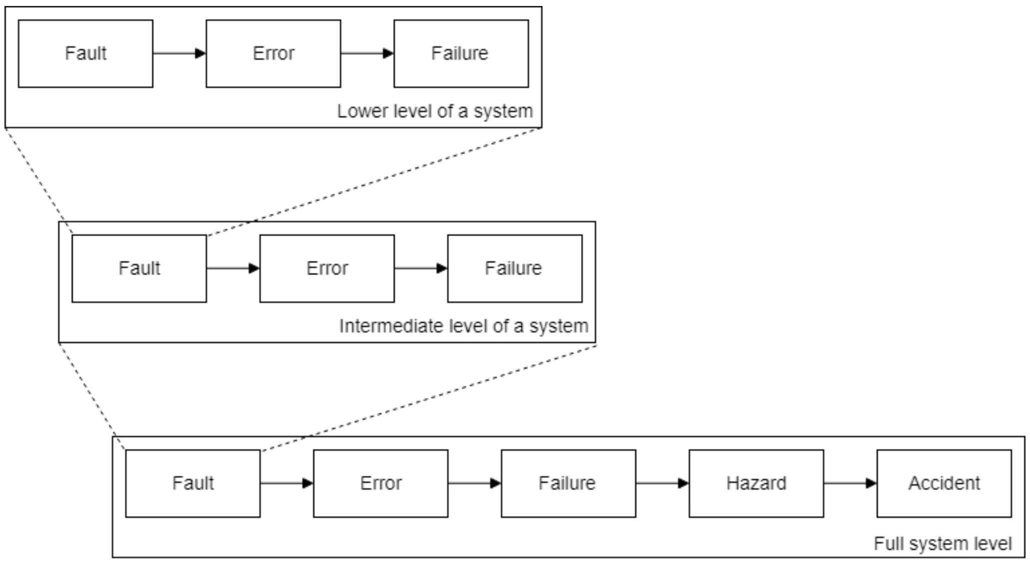


Figure 2.2: A fault which may lead to an error and failure

separation of redundant parts of the systems should be considered.

For SIL 4 a total physical separation between the safety system and the non-safety-related elements is necessary. Here, the redundant components of the safety system have to be physically separated as well.

2.1.6 Techniques and measures

The challenge is to design a safety system which prevents dangerous failures. In case of a fault, the system has to switch to a safe state in order to avoid harm. An essential part of functional safety engineering is the identification of failures which may lead to severe consequences.

Based on this outcome specific designs and mechanisms have to be established and implemented with the purpose to reduce the risk of such failures. A collection of recommended techniques and measures to archive the needed SIL is presented in part 7 of the IEC 61508 safety standard. This part also describes why the technique or measure should be used and how it is applied. References to more detailed information are also provided.

The techniques and measures are divided into five categories: [Com11]

- **Protection against random hardware failures**

  Some of the methods and rules, which are shown in this section, are required by part 2 (hardware description part) of the IEC 61508 safety standard. Examples in this section are: CPU self-tests, RAM tests, hardware redundancy.

- **Protection against systematic failures**

  These techniques are concerned about project management as well as the overall system design and verification. Some examples are checklists, documentation, black box testing, and separation of safety-related systems from non-safety-related systems.

- **Techniques and measures for achieving software safety integrity**

  This part covers techniques and measures related to software development and testing. The primary goal of these structured methods is to ensure the software development quality by taking particular attention to the early stage of the lifecycle. Some examples are coding standards, data flow diagrams, software diversity, and high order logic.

- **A probabilistic approach to determining software safety integrity for pre-developed software**

  This part presents ideas and techniques to determine the SIL of pre-developed software, some of which are statistical tests and complete tests.

- **Techniques and measures for application specific integrated circuits (ASICs)**

  This part covers methods and measures for the design of ASICs as well as for testing ASICs. Some examples are design for testability, validation of the softcore, modularization and defensive programming.

2.1.7 Fault types

Safety-critical systems have to function correctly even in case of faults. It has to be ensured that a (hardware) fault does not become a critical system failure. To reach this goal a fault has to be detected at an early stage to bring the system into a safe state. The IEC 61508 safety standard distinguishes two types of hardware faults: permanent faults and transient faults. The third type of fault, the intermittent faults, are listed in the relevant literature.

- **Permanent faults** can be damaged microcontrollers or communication links. A typical permanent fault is a stuck-at fault where a bit is stuck at a specific value. An example of a stuck-at fault is a flip-flop, whose output is always 0, even if the input is a logical 1.

- **Intermittent faults** are a kind of faults which appear and disappear repeatedly. These faults are caused by several factors, some of them may look random. The more complex a system gets, the greater the like-hood of such a fault. [HKK08]

- **Transient faults** (also called soft-errors) may be caused by single event upsets or electromagnetic interferences. For example, the logical value 0 stored in a flip-flop may be changed by a transient fault to a logical 1. This kind of failures may be caused by alpha particles from the package decay, neutrons, external EMC disturbances or crosstalk [KML+06]. These faults may appear for a short time only at various locations. The typical consequences of transient faults are a single bit-flip or multiple bit-flip errors.

The transient faults are the more common faults and are increasing in numbers due to advanced processor logic and decreasing manufacturing size. Furthermore, the number of affected bits by a single event is going up. Multiple affected bits caused by a single event were not very common with previous silicon technologies, but modern technologies are much more vulnerable to multiple bit upsets (MBU). [IS16]

SFF	Hardware fault tolerance		
	0	1	2
<60%	not allowed	SIL 1	SIL 2
60% - <90%	SIL 1	SIL 2	SIL 3
90% - <99%	SIL 2	SIL 3	SIL 4
>99%	SIL 3	SIL 4	SIL 4

Table 2.3: Maximum allowed SIL depending on the HFT and SFF for type B components

2.1.8 Safe failure fraction and diagnostic coverage

The IEC 61508 safety standard defines the safe failure fraction (SFF) as the sum of potentially dangerous failures revealed by tests together with the failures which lead to a safe state divided by the total number of failures.

$$SFF = \frac{noncritical failures\ +\ revealed\ critical\ failures}{noncritical\ failures\ +\ revealed\ critical\ failures\ +\ undetected\ critical\ failures}$$

A fail-safe example might be a slam shut valve where 90 percent of the failures will lead to a closed lid, and 10 percent will lead to a fail-to-close situation. Even without any additional mechanisms to detect failures, the SFF of that system will be 90 percent.

A combined example might be a system where 60 percent of the shortcomings lead to a safe state, and 80 percent of the remaining failures are revealed. In this case, the SFF would be 92 percent (60 percent + 40 * 0.8 percent) [GPZ04]

Based on the SIL the IEC 61508 safety standard defines the minimum SFF based on the hardware fault tolerance and the type of the components. In IEC 61508 components are classified into two types: Type A components have distinct failure modes and well-defined behavior under fault conditions plus failure data has to be available. All the other elements are called type B components. The hardware fault tolerance (HFT) is defined as the number of tolerated failures. Table 2.3 shows the correlation between the SIL, HFT, and SFF for type B components. If HFT of one is needed redundancy might be used. By adding the same component and using it separately (e.g., calculating the same safety-critical function on two independent CPUs) a single fault in one element cannot cause an overall system failure. [Com11]

2.2 Hardware-based self-tests

Hardware-based self-tests usually require additional test hardware. If those tests are integrated in the system they are called built-in self-tests (BISTs). These tests can be very accurate, resulting in a very high fault coverage. The disadvantage of those tests may be overtesting. Overtesting means that a fault in a CPU is detected which does not affect any operation under normal conditions. Most existing BISTs are not effective in testing time-related faults like crosstalk faults or delays. Testing for this faults significantly complicates the BISTs and may lead to circuit overtesting because many delay faults can never be sensitized in the normal functional operation. On the other hand, the external testers overall timing accuracy does not increase as fast as the clock speed, and this implies yield loss. [SEN05] [WCTI03] At speed testing of gigahertz processors with external testers may not be technically and economically feasible and might not be available for some high-performance chips. [Yan05] [CD01]

In most cases, the BISTs are simple and limited to some functionality blocks. If the commercial CPUs are equipped with a BIST, the products lack precise information about its effectiveness (fault coverage, used algorithms, etc.). [Sos06] As the name indicates, this kind of tests are hardware-based, which are not portable to another system easily.

2.3 Software-based self-tests

Software-based self-tests (SBSTs) do not require additional hardware and can be implemented for existing equipment. The tests are executed by the processor itself, rather than being assigned to specially synthesized hardware modules as it is in hardware-based self-testing. SBST allows at-speed testing; the tests are executed at the actual speed of the processor, which means that timing misbehavior can be detected as well. There are two main drawbacks of a SBST. First, the hidden components can only be tested indirectly and, second, it requires additional processing time resources to execute the SBST. [KPGX05]

2.3.1 Memory self-testing

One of the classical problems of RAM modules are the radiation coming from their impurities in the package or cosmic sources. Since the mid-1970s the reliability of such RAM devices has been approved and the tolerance to radiation has been increased. At the system level, very little information regarding the reliability has been published by the vendors [Jou11]. To fill this gap some data centers analyzed measurements of memory errors in large fleet server farms. According to a study, which involved all Google servers, every server saw a RAM error every 2.5 hours on average. Luckily most of this errors were corrected by modern error correcting code (ECC) technologies. But even with the use of ECC technologies 1.3 percent of all Google servers experienced on average at least one uncorrectable memory error per year. [BCH13] [SPW09]

Table 2.4 shows the memory self-tests which are listed in part 2 of the IEC 61508 safety standard to gather a specific coverage:

Direct current (DC) fault model covers the failure modes stuck-at faults, stuck-open, open or high impedance outputs, and short circuits in signal lines.

Soft-error failures are also known as single event upsets. The adverse effects of such failures can only be controlled by the usage of runtime techniques like parity bits, ECC, or redundancy.

Techniques that reach the requested coverage for the **invariable memory** include different checksums methods and block replication.

Test techniques for **variable memories** are more complex but well defined. To achieve a medium coverage the walk-path test or RAM monitoring with a modified Hamming code may be used. For high coverage, the Galpat and Abraham tests or the RAM doubled storage test is recommended.

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

Table 2.4: Memory test requirements

Galpat test

A number of variants of the galloping patterns (Galpat) test exist.

In one of those the memory under test is set to 0. The content in the lowest address is read before it gets modified to 1. The content of the remaining memory addresses are read and checked for correctness. This process is repeated for all the remaining addresses until it reaches the highest address. The modification of this test includes variants, where a whole column or a whole row is modified in one iteration. [Gro06]

An alternative version is the transparent Galpat test in which the first step (the initialization with 0) is skipped. Instead, it is calculating a signature over the RAM cells before inverting a cell.

Abraham test

The Abraham test aims to detect stuck-at faults and coupling faults. It walks through the whole memory up and down and sets or reads memory cells. It is also known as a variation of the march test under the name Mats. [NTA78]

RAM doubled storage test

This kind of test can only be used if the safety-related RAM is stored twice. It is common to safe the copy of the RAM inverted. The test is executed by comparing the RAM with its duplication. [Com11].

2.3.2 Software-based CPU self-tests and IEC 61508

This section describes the IEC 61508 requirements for CPU self-tests. The second part of this section covers the recommended techniques and measures. The requirements can be found in the second part of the IEC 61508 safety standard, and the recommended techniques are taken from part 7.

IEC 61508 CPU test requirements

Table 2.5 lists fault types for individual parts of the CPU core. The table also shows which faults have to be detected to reach a specific fault coverage. Techniques and measures to detect that faults can be found in the next section.

Component	Requirements for diagnostic coverage claimed		
	Low (60%)	Medium (90%)	High (99%)
Register, internal RAM	Stuck-at for data and addresses	DC fault model for data and addresses Change of information caused by soft-errors	DC fault model for data and addresses Dynamic cross-over for memory cells Change of information caused by soft-errors No, wrong or multiple addressing
Coding and execution including flag register	Wrong coding or no execution	Wrong coding or no execution	No definite failure assumption
Address calculation	Stuck-at	DC fault model	No definite failure assumption
Program counter, 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 2.5: IEC 61508 CPU test requirements

IEC 61508 CPU test techniques and measures

IEC 61508 recommended techniques and measures for CPU tests are shown in table 2.6.

The technique **comparator** is used to detect failures in independent CPUs or the comparator. A hardware comparator cyclically or continuously compares the output of independent CPUs.

A **majority voter** has several inputs from independent CPUs and decides for the majority (2 out of 3, 3 out of 3, m out of n) of the provided results.

For the technique **self-test by software with limited number of patterns** software-based self-test data patterns are used as in input, and the calculation results are checked for correctness. At least two complementary patterns are needed.

Self-test by software with walking bit patterns are applied to test the register and processing unit. At the initialization all bits are set to zero and in each step, one bit

Diagnostic technique / measure	Maximum diagnostic coverage considered achievable
Comparator	High
Majority voter	High
Self-test by software: limited number of patterns (one channel)	Low
Self-test by software: walking bit (one channel)	Medium
Self-test supported by hardware (one channel)	Medium
Coded processing (one-channel)	High
Reciprocal comparison by software	High

Table 2.6: IEC 61508 CPU test diagnostic techniques / measures

is set after another. After each step the register content is compared to an expected value.

Additional special hardware is needed to implement the **self-tests supported by hardware**. The self-test result is made available at a hardware output and specialized equipment could monitor whether the bit pattern occurs cyclically according to the watchdog timer principle.

The **coded processing technique** needs processing units with failure recognizing or correction techniques. These techniques are not very widespread and have been only applied to simple circuits.

For **reciprocal comparison by software** hardware redundancy is needed. Two separate processing units are doing the same operations and exchanging data (including results, intermediate results and test data). Each of the units is comparing the own calculations with the results of the second processing unit.

2.3.3 Software-based CPU self-tests

This section provides an overview of software-based CPU self-test techniques in general (not only from the IEC 61508 safety standard view) and includes the implementation of it in more detail.

Functional testing versus structural testing

The approaches of software-based CPU core tests are classified in two categories: Functional testing and structural testing. [PGSR10] [GPZ04]

Structural tests are based on detailed information about the CPU and targets a specific structural fault model. The implementation of these tests is only possible if a gate-level model of the processor is available. For example, if two registers are physically close to each other on the chip, interference is more likely. The design and implementation of structural tests can use such information to increase the test quality without adding too much overhead. This kind of test leads to higher test coverage and better performance. The drawbacks are the higher development costs, and due to the hardware dependency, they are not portable to any other hardware.

Functional tests aim to test the correctness of all known functions. Functional tests of a CPU only needs the ISA information and no other low-level model of the processor.

Specific instructions are executed and judged based on the outcome. For example, a calculation is done and the result gets verified. The result may be checked internally with external hardware or against a redundant system. The advantage is the portability of test sequences or test programs and lower development costs. The drawback of functional testing is that the tests are not directly related to the actual structure of the processor and, therefore, not directly linked to physical defects which leads to a lower fault coverage. Especially when automatically generated functional tests are used the test programs are quite extensive in space and time-consuming. [GPZ04]

Functional level fault models

Functional level fault models are developed on a high level of abstraction. They include fault models for data processing and control sections: [WST10]

- **Register decoding faults**
  The decoded address of the register is incorrect. Due to this fault, the wrong register or no register at all may be accessed.
- **Instruction decoding and control faults**
  When this fault occurs the processor may execute the wrong instruction or execute no instruction at all.
- **Data storage fault**
  Single stuck-at faults may occur in any cell in any register.
- **Data transfer fault**
  This fault includes that a line in the data transfer path is stuck at 1 or 0, or two lines are coupled.
- **Data manipulation faults**
  An example for this fault is an ALU-control fault. No fault in the rest of the processor can mask an ALU fault. [TA95]
- **Processor functional-level faults**
  An inactive or additional active micro-operation is a mistake from this category. [Pfl03]

Pseudo random versus deterministic testing

A further orthogonal classification for CPU core tests is based on whether the CPU core tests are pseudo random or deterministic.

- **Pseudorandom tests** for CPU core can be based on pseudo random sequences, pseudo random operands or a combination of both. Pseudo random testing has been studied and applied extensively because of its low engineering effort and its low costs. Pseudo random tests may come with some learning mechanism. During the learning phase, the test sequences and operands are adapted based on feedback from a simulation of the CPU. The main disadvantage of this approach is that the test might become very large to reach an acceptable fault coverage. Especially pseudo random instruction sequences are unlikely to achieve a high fault coverage [GPZ04]. The fault coverage of such tests has to be verified, for example, by using a fault-injection framework.

A promising approach for the automatic generation of SBST is presented in [RCS$^+$16], by which a fault injection framework for the generation of the tests can be utilized. It starts with a fault list and tries to find ASM instructions to detect these faults. If instructions are found to detect a fault those instructions are added to the SBST, and the fault is removed from the list. Instructions, which do not improve the fault detection rate, are rejected. Therefore, the final SBST is much more efficient as compared to traditional pseudo random test and the fault coverage might come close to the fault coverage of manual written SBST.

- **Deterministic tests** are using test sequences with a known minimum coverage for the CPU or its components. For most of the functional modules of a processor (like ALU, multipliers, dividers) test sets guaranteeing a specific coverage exist. These test sets are usually implemented in ASM. In comparison to the pseudo random tests, the coverage is much higher and the tests are smaller. Since some test techniques guarantee a minimum coverage a verification (for example with a fault-injection framework) is not needed. The drawback is that for some of this test sets the gate level model of the CPU has to be known. [GPZ04]

A combination of both techniques is not unusual. Components that can be adequately tested with deterministic test methods are covered by deterministic tests and for other components a random approach may be used. Another common practice is to use deterministic patterns in pseudo random sequences. [ST10]

Application driven testing

Another self-test scheme is the application driven testing. Usually, the traditional application-driven tests are only applied for smaller systems, in which testing includes a full test with all possible input scenarios. If the system realizes some fixed algorithm, this test approach may be very efficient. These scenarios only test used resources. [Sos06] The preamble traditional is applied in this master thesis in order to distinguish the two meanings, even it is not common in the relevant literature.

The name application driven testing is also applied if only the needed resources are tested. In this case, the application can be more complex. If the needed resources for the application are already known, it can be effective to create a self-test which is covering those. There is no need to check unused resources. Statistics about affected resources are utilized for this kind of tests. [ST10]

In practice, it is reasonable to deploy a combination of deterministic tests for some modules of the CPU (e.g., ALU) and the application driven approach. [ST10]

Fault coverage and test efficiency

The most important decision when it comes to testing development is the fault coverage and test quality that is required. Common fault coverage levels are 90%, 95%, and 99%.

Another question that has to be raised is the fault model for the test development. Comprehensive, sequential fault models such as the delay fault model may lead to higher test quality and more straightforward combinational fault models such as stuck-at fault models may be easier to implement. Higher fault coverage can be achieved by: [GPZ04]

- More test engineering effort
- Larger test programs
- Longer test application time

Software-based self-testing approaches may not be able to achieve the same fault coverage levels as test approaches based on structured DfT techniques. SBST is capable of detecting faults that are possible in normal operations. By using this approach overtesting can be avoided. Overtesting can happen when a chip is detected as faulty even if the fault is not supposed to happen in a normal operation. [GPZ04]

The test efficiency is defined as the SFF divided by the SBST execution time [GPZ04]. For some applications such as battery powered low-power devices, a very high test efficiency is mandatory. Usually, very high test efficiencies can be achieved by using structural self-testing techniques which require a fault driven test development and detailed information about the CPU under test. [ZZR06]

2.4 CPU components classification

As shown in figure 2.3 the components of a processor can be classified into three different main categories. This classification is done based on their use and contribution to the CPU operation and instructions execution.

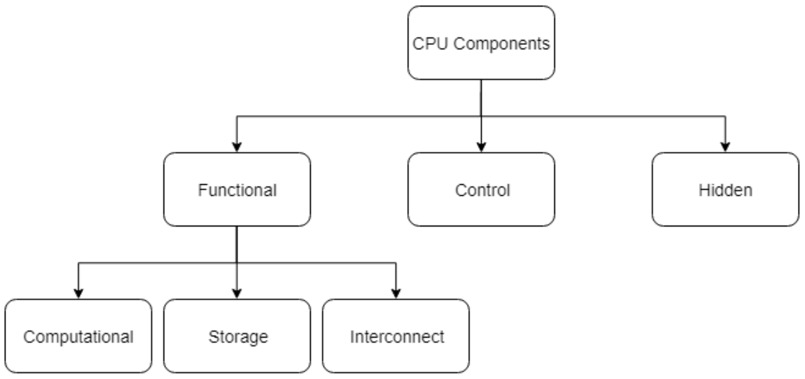


Figure 2.3: CPU components [PGSR10]

2.4.1 Functional components

Functional components are the components which are directly related to the execution of CPU instructions. These components can be seen as visible to ASM developers and therefore the easiest to test with SBST. Functional components belong to one of these sub-classes: [GPZ04]

- **Computational functional components** perform arithmetic or logical operations on data. Such components are arithmetical logical units (ALUs); adders, subtractors, comparators, incrementers, shifters, multipliers, and divisors. As shown in table 2.7, each operation excites a different part of the CPU.
- **Storage functional components** serve as storage elements for data and control information. These components include registers for data and control information as well as several pointers which can be accessed by ASM instructions.
- **Interconnected functional components** implement the interconnection between functional components and control the flow of data. It is mainly the interconnection between the two previously mentioned functional components: the computational and the storage functional components. These components include multiplexers and bus control elements.

Operation	**ALU part used**
Addition	Arithmetic part (adder)
Subtraction	Arithmetic part (subtracter)
AND	Logic part (AND)
OR	Logic part (OR)
NOR	Logic part (NOR)
XOR	Logic part (XOR)
Set on less than unsigned	Arithmetic part (subtracter)
Set on less than signed	Arithmetic part (subtracter)

Table 2.7: Operations of the MIPS ALU [GPZ04]

2.4.2 Control components

Control components are responsible for the flow of instructions and data inside the processor. These components are also used to control the flow of data to and from the external environment (e.g., memory subsystem). One of those components implements the decodation of instructions and production of control signals of the functional components. The size of the control components on a CPU is usually much smaller as compared to the functional components. In case of a malfunction of the control components, it is unlikely that any instruction of the CPU can be executed correctly. [GPZ04]

2.4.3 Hidden components

The hidden components are usually to increase the performance and instruction throughput. Some examples of this components are the one that implements pipelining or instruction level parallelism. Some of these hidden components, such as the branch prediction units, are hard to test. Due to the inherent self-correcting nature of branch prediction units, a fault is not directly observable and only leads to performance degradation. [SR14]

2.4.4 Component size and contribution to fault coverage

"Component-level self-test routines development should give higher priority to large components that contain a large number of faults." [GPZ04] In general, large processor components contain large numbers of faults and should be assigned to a higher priority in order to improve the overall fault coverage. For example, developing a self-test with a fault coverage of 90% for a component that occupies 30% of the processor gate count and faults number contributes 30% * 90% = 27% to the total fault coverage. Developing a self-test with a fault coverage of 99% for a component hat only occupies 5% of the processor gate count and faults number contributes only 5% * 99% = 4.95% to the total fault coverage.

Functional components are in almost all cases the largest in size in comparison to the control components and hidden components. Among the functional components, the computational ones are usually the largest. Since the functional components, especially the computational components, are the easiest to test and most of the faults are expected in these components, it is recommended to give those tests the highest priority and weighting. [GPZ04]

Even if the SBST is only focused on a high fault coverage for the more accessible to test functional components, the other parts get tested too. If the control parts fail, it is unlikely that the functional elements can deliver correct results. (see 2.5.4)

2.5 Related work

This section covers related work on generic CPU fault tests and indirect CPU fault tests.

2.5.1 Instruction set architecture for a Cortex-M3

Even if the goal was to design and implementation a portable SBST, the ideas, design, and implementation was focused on specific hardware.

The ISA is required in order to implement a functional self-test. In the case of the Cortex-M3, the number of instructions is quite large and the instruction set of a Cortex-M4 CPU is even larger. In general, the ARM Cortex-M processors are designed as reduced instruction set computing (RISC) processors. Due to its characteristics such as the rich instruction set and mixed instructions sizes some might argue that these processors are closer to the complex instruction set computing (CISC) architecture. But as processor technologies advance, the instruction set of most RISC is getting more powerful, and the traditional differences between RISC and CISC can no longer be applied. [Yiu13] Some of the modern CPUs have taken the best aspects of both architectures. [AMH07]

The ISA lists more than 180 different instructions. A full list of the instructions can be found in [ARM10] and [Yiu13], most of it can be used with an optional code suffix and a flexible second operand. The optional code suffix allows conditional execution based on the CPU flags. In total, there are 14 conditional execution suffixes as shown in table 2.8. Most of the instructions also allow the second parameter to be flexible. A flexible operand can be a constant value or a register with an optional shift. The allowed shift expressions for the flexible operand are shown in listing 2.9.

As a result of an instruction in most cases a result value is stored in the output register and the CPU flags are set accordingly. The Cortex-M3 CPU has 4 CPU flags:

- **N** This flag is set if the result is **negative**.
- **Z** If the result was **zero**, this flag is set.
- **C** If the operation resulted in **carry**, this flag is set.
- **V** This flag indicates that the operation caused an **overflow**.

Most of the standard instructions can be called to modify the flags based on the outcome or to not touch the flags. Some instructions can be executed to set only the flags and not change any values in the registers.

Suffix	**Flags**	**Meaning**
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed
AL or empty	any	Always. Default when no suffix is specified.

Table 2.8: Conditional execution code suffixes for Cortex-M3 and Cortex-M4 [ARM10]

Shift Expression	**Meaning**
ASR #n	Arithmetic shift right n bits
LSL #n	Logical shift left n bits
LSR #n	Logical shift right n bits
ROR #n	Rotate right n bits
RRX	Rotate right one bit and extend
	if omitted no shift occurs

Table 2.9: Optional shift expressions for flexible operands on Cortex-M3 [Yiu13]

2.5.2 Deterministic tests for CPU components

A complete deterministic test without any limitations would need to cover all possible instructions described in 2.5.1. With all the shift operations and conditional flags, this implies thousands of individual instructions, each of which needs to be tested with with several different values and compared to expected values. Luckily, most of the compilers and compiler settings use only a small extract of the theoretically available instructions. [ST10]

2.5.3 C based instruction tests

Most of the CPU core software-based self-tests presented in the literature are written in ASM. Due to the hardware dependency of ASM code, these tests cannot be easily ported to another CPU architecture. In some of the more modern literature deterministic tests for some parts of CPU were already written in a higher language and examined. With the usage of most of the compilers the expected ASM instructions were generated and the expected parts were tested. [PKH$^+$13] [WXT11]. Additional hardware-based tests are often recommended in order to reach a higher coverage.

Generic CPU safety test

In [Pre14] and [Sch15] a full generic CPU self-test written in C++ was implemented and examined. The SBSTs were solely focused on the larger elements of a CPU to reach an acceptable coverage. The generation of the self-tests was focused on general ideas of how to test the ALU and some various operations like Quick Sort to test hidden parts. To proof the effectiveness of the implemented tests a fault injection framework was used. Although a full coverage of all instructions was not the goal, the reached fault-coverage was very promising.

The main difference between this approach and the methods applied in this master thesis were that the effectiveness was not shown based on the compiled instructions and the implementations of the SBST were not meant to be used with compiler optimizations. The usage of compiler optimization algorithms would have replaced the tests and checks with constants and this would have let to a low fault coverage (see 3.4).

2.5.4 Indirect CPU testing

SBST are based on ASM instructions (or higher level implementations which are compiled to ASM instructions). Components, which are invisible to the assembly language programmer, cannot be tested directly (see 2.4). However, a test for a functional component, like the ALU, is also testing other parts. This approach testing non-visible components is called indirect CPU testing and can be quite capable. [GPZ04]

In [CD01] SBST were used to test the ALU and shifter only. No tests were generated for other components because they were not readily accessible trough instructions. An indirect test of not-directly tested components was expected. The tests reached around 99% fault coverage for the directly tested components and more than 91% coverage for the whole CPU.

Chapter 3

Concept and design

This chapter describes the system requirements in detail. The operations and operands, which need to be tested, are presented and classified. The approach to test the CPU flags is shown as well. Furthermore, the difficulties of compiler optimizations techniques and generic C SBST are demonstrated. The concept of showing the instruction coverage is presented as well.

3.1 System requirements

- **Application driven approach**

  The approach is to test all instructions which are used or likely to be used. Instructions, which are not used, do not need not be tested. Instructions which are unlikely to be used should not be tested.

- **C-based self-tests**

  The instruction tests should be implemented in C.

- **Additional assembly based self-tests**

  Only particular instructions, which cannot be tested with C code, should be implemented in ASM.

- **Portability**

  The tests should be as hardware-independent as possible. Due to some specific tests, which cannot be covered in C, a complete hardware independent instruction test which includes all used ASM instructions is not reachable. Nevertheless, the instructions, which need to be adapted in case of a hardware change, must be clearly marked and easily adaptable.

- **Test in slices and verification of the SBST results**

  The tests have to be executed in test slices. The outcome of each test slice is a 32 bit CRC. If all tests were successfully executed, this CRC has to have a specific value which is known at compile time.

- **Balanced test slices**

  The self-tests are executed in test slices. The execution time of each of those test slices should be around 100μs and not exceed 150μs.

- **RAM and register tests**

  The Abraham test, the RAM comparison test and the RAM doubled storage test are used for testing the RAM functionality. The general purpose registers are tested with the usage of the Galpat test. These tests were already in use, the implementation of this tests is not part of this master thesis.

- **Coverage measurement**

  This is done by comparing the outcome of the compiler. The instructions used in the application are compared with the instructions tested in the self-tests. Furthermore, the usage of CPU flags after each instruction is analyzed. If the status of a specific CPU flag is used in the application code, the test has to check the CPU flag for that particular instruction too. This test coverage analysis tool was already implemented but reviewed.

- **Coverage with enabled compiler optimizations**

  The provided self-test needs to have a full coverage, even if specific compiler optimizations are used.

- **Integration of additional self-tests**

  If additional self-tests are required later on (for instance, if it does not include supported compiler optimizations), the implementation of these tests and the integration into the test system needs to be supported.

3.2 General self-tests

General tests are used to check the memory and registers:

- **Register testing: Galpat test**

  This test is used to check the addressing of the general purpose registers. Since the test includes checks against expected values, the instructions for this test can be seen as proven and added to the diagnostic coverage.

- **Data access: Abraham test**

  This test accesses the whole RAM of the controller and checks its content for expected values. Therefore, the applied instructions will be added to the diagnostic coverage as well.

- **Memory addressing: RAM doubled storage test**

  This test compares the safety-related RAM which is stored twice (normal and inverted). Since the test includes a comparison with values also the executed instructions for this test can be seen as tested.

3.3 Application driven approach

Application code and the test code is compiled together to ASM instructions. Therefore, it should be possible to create an instruction test in the same programming language as the application code that covers most the instructions used by the application.

The visible functional parts are the one which can be tested. Therefore, the common operations are listed first. Followed by a section where the idea to test the CPU flags is presented. Common data values, which are often applied in the tests, are shown as well.

3.3.1 Common operations

Relational operators

Relational operators are extensively used in all applications and need to be tested.

- Equal to
- Not equal to
- Greater than
- Less than
- Greater than or equal to
- Less than or equal to

Bitwise operations

Tests for bitwise operations need to include:

- NOT operator
- AND operator
- OR operator
- Exclusive or (XOR) operator
- Logical and arithmetical bit shift

Also, a combination of those operators have to be tested (e.g., NOT AND) since there might be special instructions available which do both operations in one step.

Basic arithmetic functions

Common operations, which need to be tested, are the essential mathematical functions:

- Addition
- Subtraction
- Multiplication
- Division
- Modulo

Data access operations

Data access operations include:

- Data access to different memory sections (see 3.3.4)
- Read a single element from an array
- Write a single element into an array

Memory functions

The memory set, copy and compare functions are common and need to be tested. All three tested memory functions need to be executed on all different storage types (see 3.3.4):

- Memory set with local memory destination
- Memory set with global memory destination
- Memory copy with local source and global memory destination
- Memory copy with global source and local destination memory
- Memory copy with constant source and global destination memory
- Memory copy with constant source and local destination memory

More library functions

Not only the memory functions are widely used, but also other library functions need to be tested. Standard library functions, which need to be tested, include but are not limited to long jump and string functions.

3.3.2 Test of the CPU status flags

Most of the instructions can also affect the flags. The status of those flags may have an influence on the correct processing and have to be tested as well. To support the testing of the CPU status flags (intermediate) results should be checked against expected values with all of these relational operators. Due to different compilers and compile settings, these C instructions may lead to flag checks or to compare instructions.

- **Equal to** checks the **zero** flag status
- **Not equal to** checks the **zero** flag status
- **Greater than** checks the **carry** flag
- **Less than** checks the **carry** flag
- **Greater than or equal to** checks the **carry** and **zero** flag
- **Less than or equal to** checks the **carry** and **zero** flag

- **Is equal to zero** checks the **zero** flag
- **Is unequal to zero** checks the **zero** flag
- **Greater than zero** checks the **sign** and **zero** flag
- **Greater or equal than zero** checks the **sign** and **zero** flag
- **Less than zero** checks the **sign** and **zero** flag
- **Less or equal than zero** checks the **sign** and **zero** flag

3.3.3 Data types

The operations have to be tested with the standard C data types:

- Unsigned 8 bit integer
- Signed 8 bit integer
- Unsigned 16 bit integer
- Signed 16 bit integer
- Unsigned 32 bit integer
- Signed 32 bit integer
- Unsigned 64 bit integer
- Signed 64 bit integer

3.3.4 Storage type

The data types can be stored in different memory sections:

- RAM (global memory)
- Stack (local memory)
- Program memory (constant data)

3.3.5 Data values

The values, which are used to execute the tests, have a tremendous influence on the fault detection. Universal test values are defined in this section.

Walking bit patterns

For most of the tests walking bit values are used which is why lists with all possible values for the supported data types are required.

Lower bits are set

Another set of common bit patterns for testing are those, in which all bits are set to a specific index, but the higher bits are not set. As an example the 8 bit values are shown here:

- 00000001
- 00000011
- 00000111
- 00001111
- 00011111
- 00111111
- 01111111
- 11111111

Prime values

Mainly but not limited to the division and modulo tests lists with low and high primes for each supported data type are needed.

Values with alternating bits

Widely used are also the values with alternating bits (such as 0xAA = 1010 1010 and 0x55 = 0101 0101).

Alternating bits alternating in each byte

To continue the idea of alternating bits the first bit of each byte can be swapped too. For example: 0xA5A5 = 1010 0101 1010 0101

Minimum and maximum values

It is common and practical to use the maximum values of each data type. For signed data types the minimum value is also helpful.

Values with patchy bit patterns

Values with patchy bit patterns are also useful for testing. An example for such a pattern is 0xDEADBEEF = 1101 1110 1010 1101 1011 1110 1110 1111.

3.4 Compiler optimizations

Compiler optimizations have a massive impact on the development of SBST. A SBST implemented in C, which is not supposed to be used with optimizing compilers, will lose most of their abilities to detect glitches. Even if the same SBST compiled without any optimizations may have a good coverage, shallow coverage is expected when used with optimizing compilers. Among of the large list of optimization techniques two of those make the implementation of self-tests challenging:

- **Optimization by reduction of already known results**

  SBST are calculations, in which the outcome is already known in advance and is checked against the expected value. When optimizations are turned on (no matter if the optimization goal is to save memory or runtime), the compiler is looking for pre-known outcomes. The calculations and the checks are replaced by constants, in some cases, a method call with many tests is replaced by just one line of code. [Man03]

- **Optimization by choosing the fastest instructions**

  Most of the compilers are using a limited number of different instructions, only if the compiler is trying to optimize the code instructions which are doing more than one calculation in one step are chosen. Even if it may look straight forward in some situations which instructions are used, the topic of instruction selection is quite complex and in some situations the selected instruction may look incomprehensible. [Bli16].

The first issue (replacing the tests with constants) may be solved by:

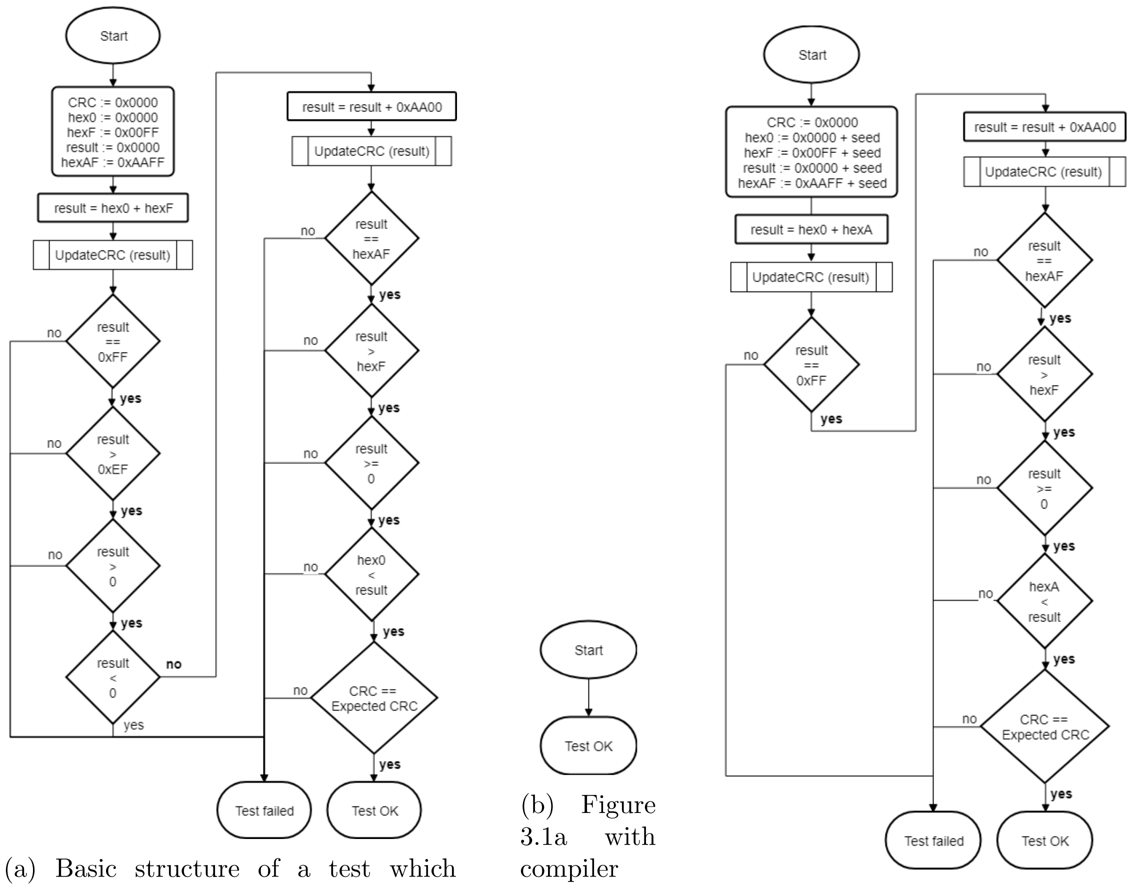
- **Turning off the optimization techniques for the self-tests**

  The first issue may be solved by turning off the optimization techniques just for the SBST and by using the optimization algorithm only for the application code. This fix will make the SBST not useless, but also not as useful as it should be. The application will use instructions which are generated with turned on optimizations only, and these instructions will never be generated for the test code and therefore not tested. Due to these limitations, this method was not further explored in this master thesis.

- **Adding values to the tests, which are not known by the compiler**

  The second possibility is to hide the fact from the compiler that the result will always be the same. The constant behavior can be hidden by adding the content of a variable to one of the constant values of a calculation. In normal operation this value will always be the same (preferably 0) and does not change the result of a calculation. Due to performance improvements, it is preferred to XOR the constant value with the seed variable instead of adding it together.

Figure 3.1a shows the basic structure of a test which could be used for testing the addition instructions. However, as soon as compiler optimizations are utilized, the whole test get replaced with no tests and checks at all (see figure 3.1b). Figure 3.1c shows the



(a) Basic structure of a test which would have worked without compiler optimizations

(b) Figure 3.1a with compiler optimizations

(c) Figure 3.1a with a seed variable and compiler optimizations

Figure 3.1: Simple structure of a test compiled with compiler optimizations

same additional test 3.1a with the addition of the variable seed which will always be 0, but this constant behavior of the seed is not known by the compiler. As shown in figure 3.1, even with the usage of a seed variable to hide the constant outcome the fault coverage is lower than it would have been if compiler optimizations are not used at all. Some checks, which are implemented to test the CPU flags, are not executed. A simple reordering of the checks (for example to check the intermediate result against the constant value at the end of the first group of flag checks) does not change the applied instructions. Compilers are usually allowed to change the order of the instructions as long as the functionality of methods is not affected. [Man03]

In many tests the values are taken from predefined and constant arrays. To hide this constant behavior from the compiler without adding the seed variable to each value it is done in a less performance reducing way by adding the seed variable to the array pointer.

Additional tests focusing on optimizations are needed to improve the number of covered instructions.

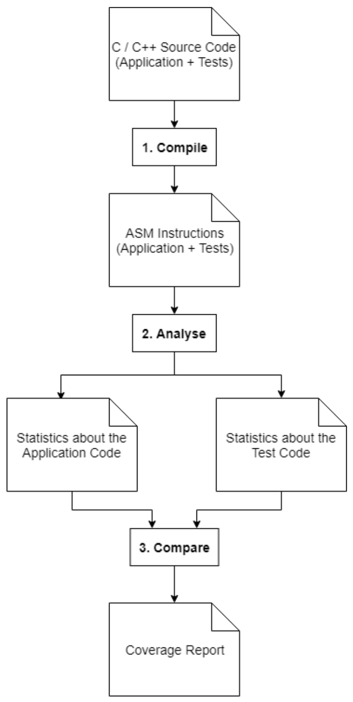

C / C++ Source Code (Application + Tests)
1. Compile
ASM Instructions (Application + Tests)
2. Analyse
Statistics about the Application Code
Statistics about the Test Code
3. Compare
Coverage Report

Figure 3.2: Three steps to measure the coverage

3.5 Coverage measurement

An essential part of a test is to show its effectiveness and completeness. In this section, the coverage measurement is described. As shown in figure 3.2 the coverage is done in three steps: the source code is compiled, analyzed, and compared.

3.5.1 List of ASM instructions

The application source code and the self-test source code are compiled together and assembler listings are generated. This step needs to be done with the compiler and compiler settings which will be used for the release of the application.

3.5.2 Analysis of ASM instructions

In the first step non-relevant information like comments are removed. With the usage of a language recognition tool, the instructions are reduced to its prototype and the usage of CPU flags gets analyzed. The coverage measurement tool knows which listings belong to the self-tests. All ASM listings, which do not belong to the test code, are part of the application and need to be tested. The outcome of this step are two lists with instructions, one of which is the list of instructions used in the application, the second one is the test instructions list.

Furthermore, there may be cases where the compiler generates calls to functions in system libraries (e.g., C standard library). Such calls are located and also expected to be tested if such calls are made from the application.

3.5.3 Comparison

In the last step of the coverage analysis, the two statistical files are compared with each other. All instructions used in the application need to be covered by the test code as well. If flags have been set and checked in the application code, the flags for that instructions need to be checked in the test code as well.

The outcome of this step are statistical information about the used instructions of the application and test environment. Based on this outcome a coverage report is generated.

3.5.4 Coverage report

The report is not only showing whether the coverage is given or not, it also gives useful statistical information. By each ASM instruction, it is showing how often it is applied in the application code and how often this instruction is tested. It also shows by instruction if and which flags were evaluated in the application and test code. Instructions, which are neither used in the application nor in the test code, are not shown here.

Only if all instructions, system calls and flag checks, which are utilized in the application, are found in the test code, the coverage is given. If any instruction or flag check is missing, the report is showing and highlighting the deviations.

Chapter 4

Implementation

This chapter describes the design and implementation of the SBST. For each type of operand the test design is shown and additional tests, which were needed to increase the coverage in case of compiler optimizations being used, are shown here as well.

4.1 Design of the class diagram

In this chapter, the C++ test classes for all SBST are introduced.

The SBST will be divided into several test classes, each of which is focused on a kind of operation:

- Relational operators
- Bitwise operations
- Basic arithmetic functions
- Data access operations
- Bit operations
- Memory functions
- More library functions
- Hardware specific instructions

Each of these classes will hold the belonging tests. According to the requirements, the tests have to be executed in slices and should not exceed a specific maximum execution time. To fulfill this requirement each test class can execute the tests in one test slice or may use several test slices to respect the maximum test execution time per test slice.

Software-based self-test base class

The base class defines the interfaces to execute the test slices and provides useful helper functions and often used test values. The base class supports the tests with the following attributes:

- The interface to call the tests
- The seed variable
- Several often used test values, as shown in 3.3.5, are available for the tests as constants
- CRC calculation method

Seed variable

The seed variable is an unsigned integer variable which is used to hide the constant behavior of some values. Its value will always be 0, but this behavior has to be hidden from the compiler. It is extensively applied, so that compiler optimizations do not eliminate specific calculations.

Interface to call the tests

The base class defines the method "doExecuteSlice". This method is defined as abstract and has to be implemented in each test class. When first called this method executes the first test slice. The outcomes of this call are the calculated CRC value as well as the expected CRC value. If used in a single system implementation, those two values have to be compared with each other. If more than one systems are in use, the value may be exchanged with another parallel system and the calculated CRC value from the other system can be compared for accuracy as well. At the end of each test slice the slice counter is increased, and the next call of this method is executing the next test slice until all test slices are called. Each test class also provides the number of test slices. To start again with the first test slice, a reset method has to be implemented.

4.2 Design of the software-based self-test

This chapter describes the structure of the SBST used in this master thesis. Simply speaking, all C language constructs, which might be affected by the application, need to be used and tested within the C based instruction test. To check the flag status the code sequence will also have checks which implicitly checks the flags. To support optimizing compilers the seed variable is utilized repeatedly and additional tests are added. The seed variable will always be 0 and will be just added in order to avoid that the compiler already knows the result at compile time.

The general idea behind each SBST is to add (intermediate) results to a 32 bit CRC. During each test cycle in most of the tests, the intermediate results are also compared with constant or dynamic values. Depending on the success of the comparison, additional values are added to the CRC. At the end of each test slice the calculated CRC will be compared with the expected CRC.

4.2.1 Relational operators

Relational operators are extensively used in all applications and are also essential for the SBST to check calculated values and CPU flags.

All the relational operations shown in 3.3.1 with all combinations of different data types listed in 3.3.3 need to be tested here. The basic structure, which is utilized to test the relational operators, is shown in figure 4.1a: With the usage of shift operations, each result is slightly changing the 32 bit result variable. To make sure that the compiler is executing each test without any modifications due to optimization techniques the result of each comparison is additionally added to the CRC value.

Another beneficial test, which is implemented in this section, is the calculation of the maximum and minimum of two values.

Relational operator tests for optimizing compiler

As already described in the previous section 4.2.1, the intermediate result was added to the CRC in order to test the instructions which are supposed to be tested. However, it also makes sense to include some tests, in which the intermediate result is not added to the CRC. In this case, the compiler may use different instructions which are only changing the CPU flags but not changing any registers (see figure 4.1b)

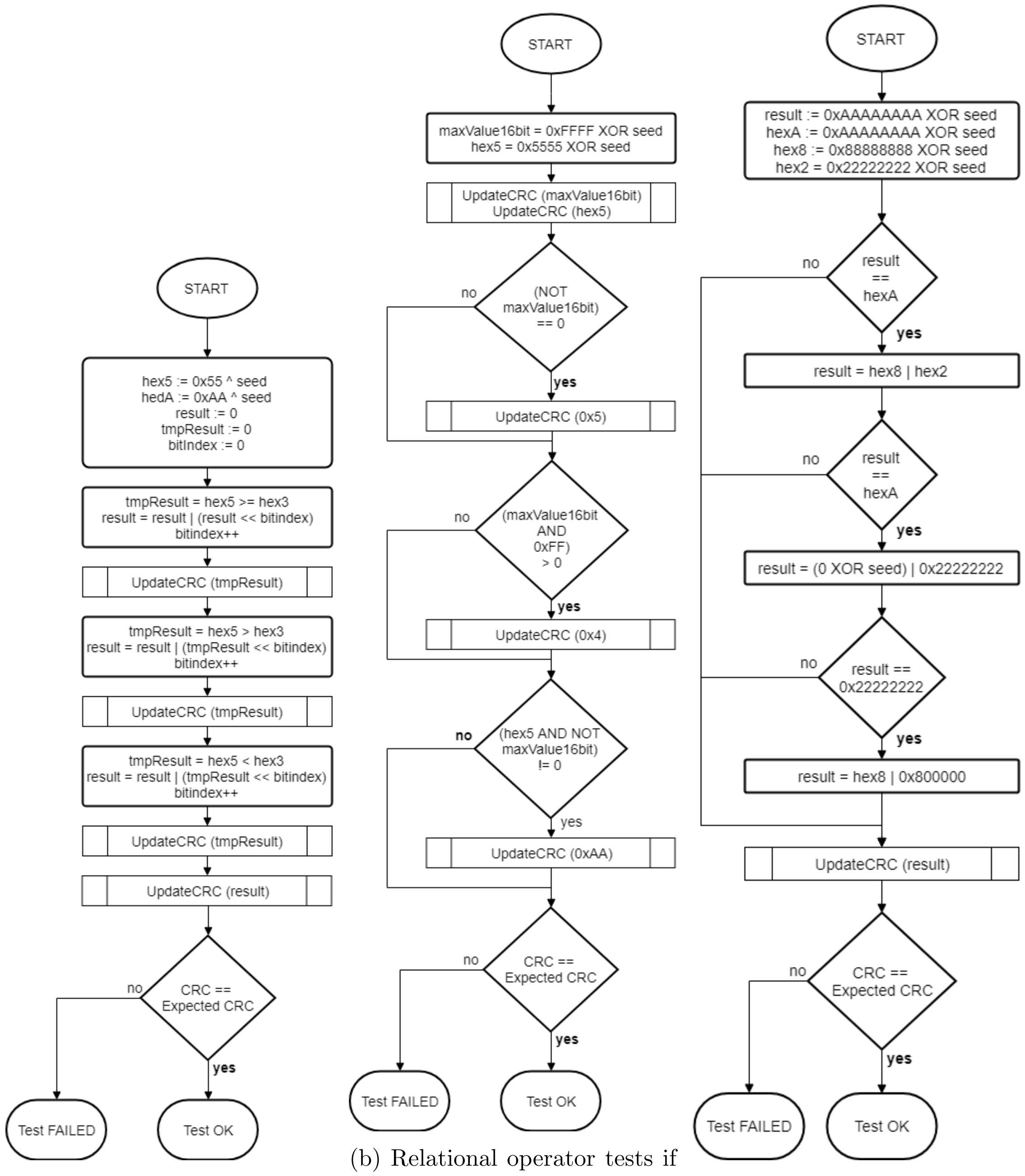
Another set of instructions, which are only applied in case of optimizations are turned on, are the IT (if-then) instructions with more than one conditional instruction. This instruction makes up to four following instructions conditional. The conditions can be all the same (then), or some of then can be the logical inverse (else) [Mah13]. If compiler optimizations are not turned on, the IT (if-then) instruction is used to find the maximum of two values. The implementation of a test in C, which leads to an if-then-then-then-then instruction, can be seen in figure 4.1c. Such a long if-then combination is scarce. A change in one of the values or the use of different compiler settings may result in different instructions and the ITTTT instruction is not tested anymore.

4.2.2 Bitwise operations

Bitwise operations are tested in their SBST but also indirectly tested in a lot of other tests.

The tests for the bitwise operations NOT, AND, OR and XOR is shown in figure 4.2a: A bitwise operation, in this example the NOT, is executed (the XOR is only used to bring in the seed variable). The result of that operation is added to the CRC value. The result is checked against values, some of those checks are supposed to be true, some are not. Depending on the result a specific value is added to the CRC.

Bitshift instructions are also tested here. For these tests, patchy bit patterns, such as the hex value 0xDEADBEEF or 0x4EADBEEF are applied. To cover all shift operations an unsigned and signed value have to be shifted right and left. The test of the bitshift instructions is shown in figure 4.2c. To cover the instructions, which change the CPU

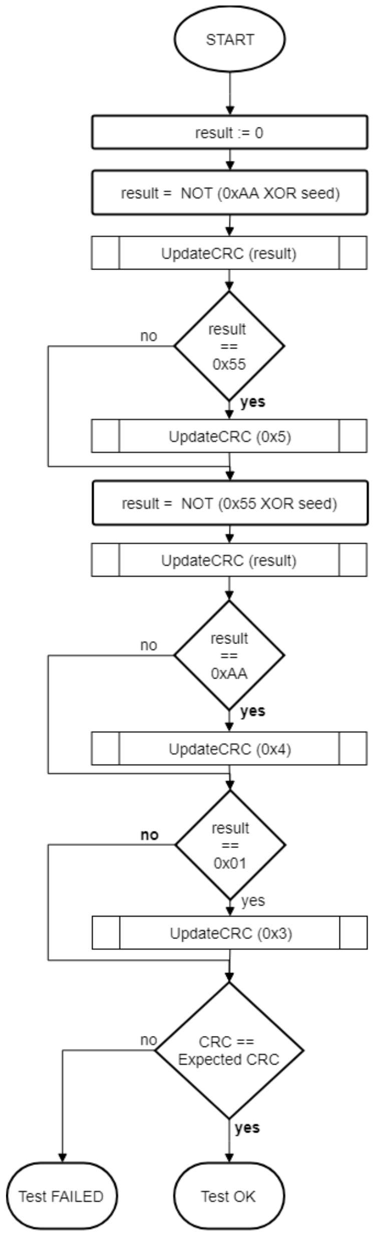


(a) Basic structure of the relational operators test

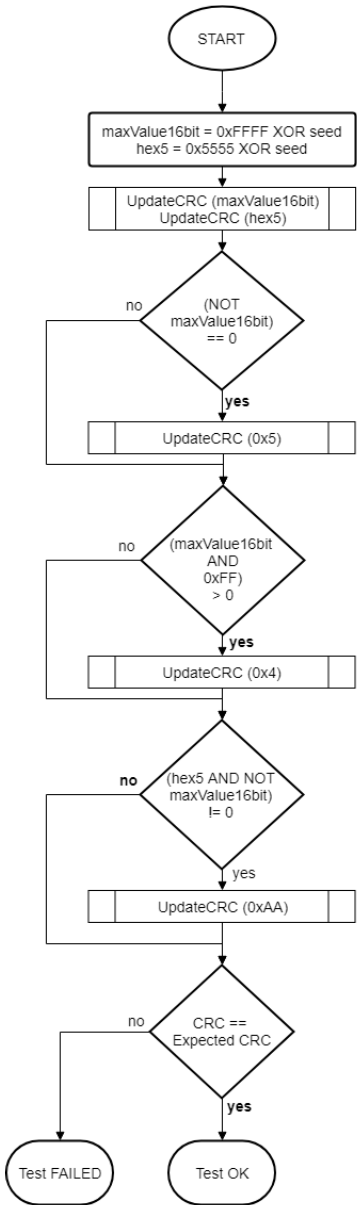
(b) Relational operator tests if the result is not required and only the flags are used

(c) Test for the ITTTT instruction

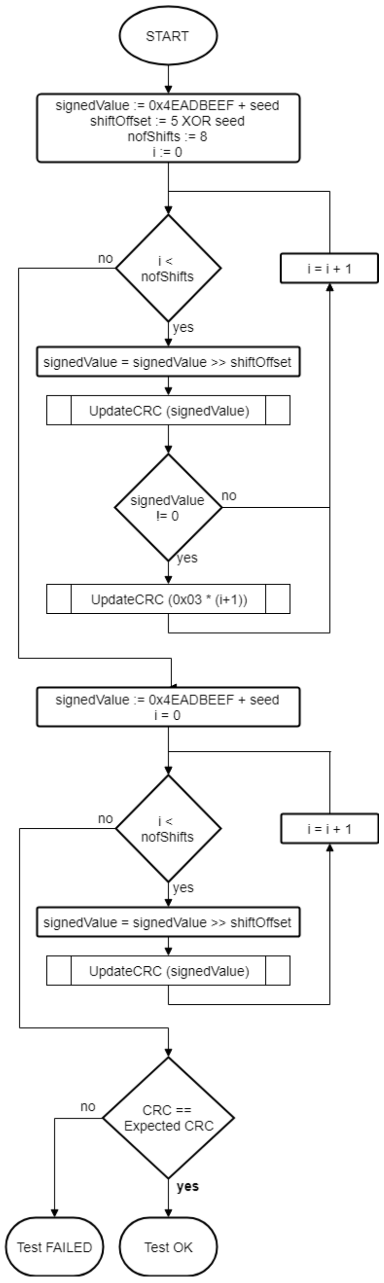
Figure 4.1: Tests for relational operators



(a) Basic structure of the bitwise operations test



(b) Test structure of bitwise tests if the detailed result is not important and only the flags are used



(c) Test structure of the bitshift test

Figure 4.2: Tests for bitwise operations

flags and also those which do not affect the CPU flags, the shift tests are executed with a comparison of the intermediate results and also without that check.

Bitwise operation tests for optimizing compiler

To support instructions, which are generally used only if compiler optimizations are deployed, the combination of AND and NOT in one operation has to be tested as well. Additionally, it is also essential to test the operation when only the status of the flags is relevant, and the value itself is not important. An example of such a test can be seen in figure 4.2b. The step to add the initial values (max value and hex5) to the CRC is only done to avoid further compiler reductions.

4.2.3 Basic arithmetic functions

The arithmetic functions are tested with the following parameters:

- Constant value
- Value from an array
- Walking bit

Intermediate results are checked against constants, values from a constant array and against values, of which the compiler does not know that the value is constant.

Addition and subtraction tests

Whereas the idea behind the testing of addition tests is shown in figure 4.3, the method of subtraction testing is illustrated in figure 4.4. The order of the operations with constant values, values from an array and walking bit varies. The shown steps are repeated until an overflow for the additions or underflow for the subtraction tests is reached. Even when the overflow or underflow is reached two more test steps are executed.

Addition tests for optimizing compiler

If compiler optimization techniques are utilized, specific switch-case statements may be compiled in order to add instructions with specific parameters which are not applied in normal situations. The add instruction which is generated in this situation is only working with the first 8 bit of both 32 bit parameters for the operation and is using the status of the flags to exclude some branches. An example of a switch-case statement that leads to such special instructions is shown in figure 4.5. It seems that it has something to do with the bit structure of the case expressions. Further tests have shown that these instructions are only applied with specific compiler optimization flags. Even if the same compiler is used with different optimization flags, this optimization is often not utilized anymore.

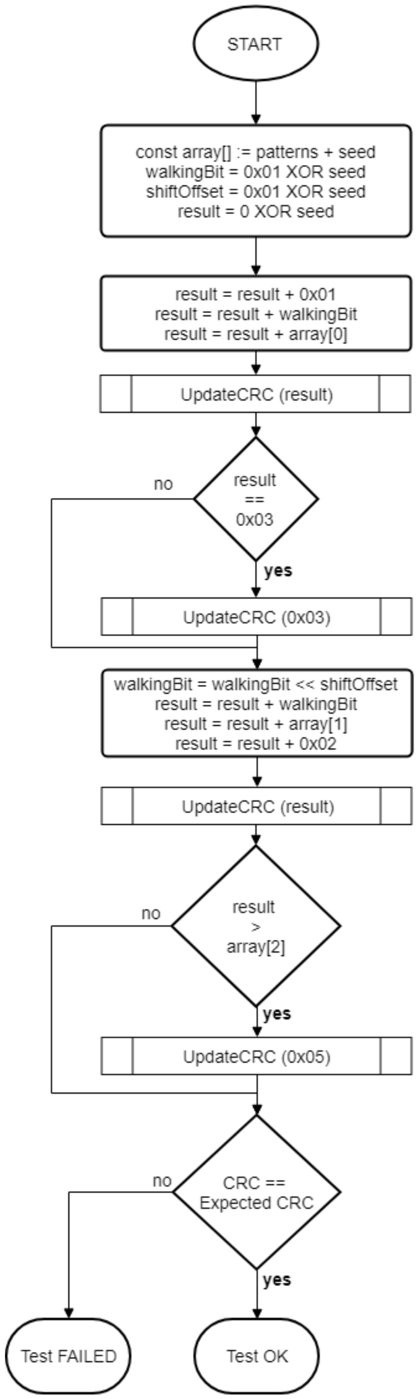

START
const array[] := patterns + seed
walkingBit = 0x01 XOR seed
shiftOffset = 0x01 XOR seed
result = 0 XOR seed
result = result + 0x01
result = result + walkingBit
result = result + array[0]
UpdateCRC (result)
result == 0x03
no
yes
UpdateCRC (0x03)
walkingBit = walkingBit << shiftOffset
result = result + walkingBit
result = result + array[1]
result = result + 0x02
UpdateCRC (result)
result > array[2]
no
yes
UpdateCRC (0x05)
CRC == Expected CRC
no
yes
Test FAILED
Test OK

Figure 4.3: Test structure of the addition test

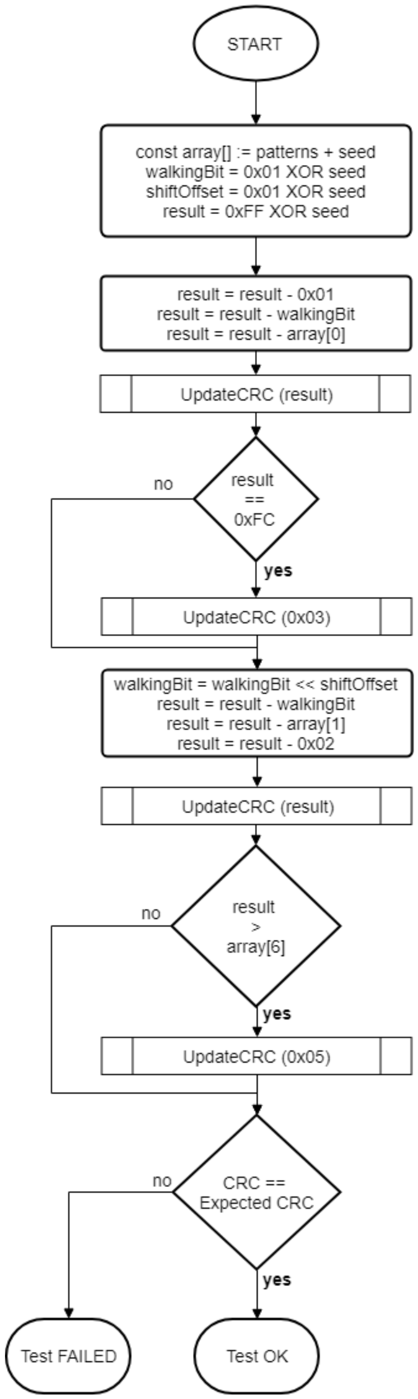

START
const array[] := patterns + seed
walkingBit = 0x01 XOR seed
shiftOffset = 0x01 XOR seed
result = 0xFF XOR seed
result = result - 0x01
result = result - walkingBit
result = result - array[0]
UpdateCRC (result)
result == 0xFC
no
yes
UpdateCRC (0x03)
walkingBit = walkingBit << shiftOffset
result = result - walkingBit
result = result - array[1]
result = result - 0x02
UpdateCRC (result)
result > array[6]
no
yes
UpdateCRC (0x05)
CRC == Expected CRC
no
yes
Test FAILED
Test OK

Figure 4.4: Test structure of the subtraction test

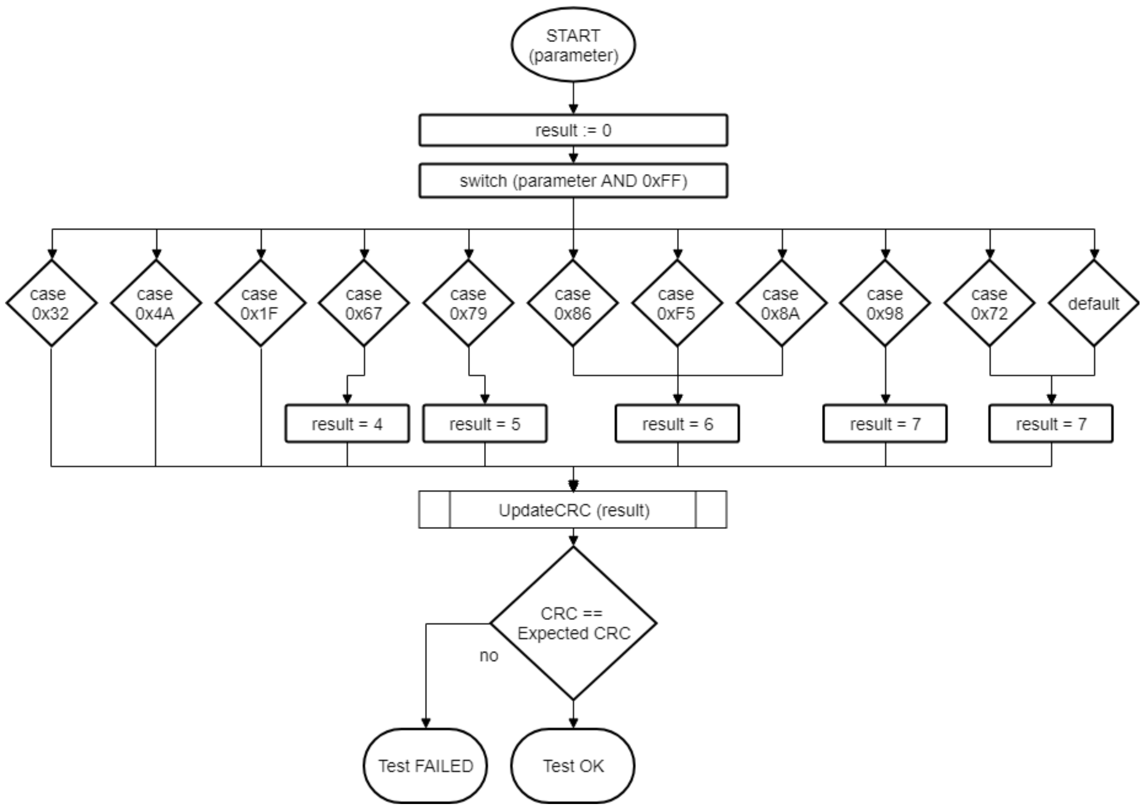


Figure 4.5: Switch-case construct which leads to uncommon addition ASM instructions

Multiplication

The multiplication test is done again according to the arithmetic function tests presented in 4.2.3, which can be seen in figure 4.6. Borderline tests (e.g., multiplication with 0 or high values which result in an overflow) are necessary as well.

Division and modulo tests

Division and modulo tests are shown in figure 4.7 and are mostly done based on low and high primes. Most of those values are taken from the arrays which were prepared in advance and available for all tests, but some of those values were directly entered into the source code to make the compiler aware of the constant behavior.

Some essential tests are also executed with non-primes and borderline values.

Multiplication and division tests for optimizing compiler

Compiler optimizations did not lead to any unexpected instructions. In some cases, bit shiftings are used instead of the multiplication, but these instructions were already covered in 4.2.2.

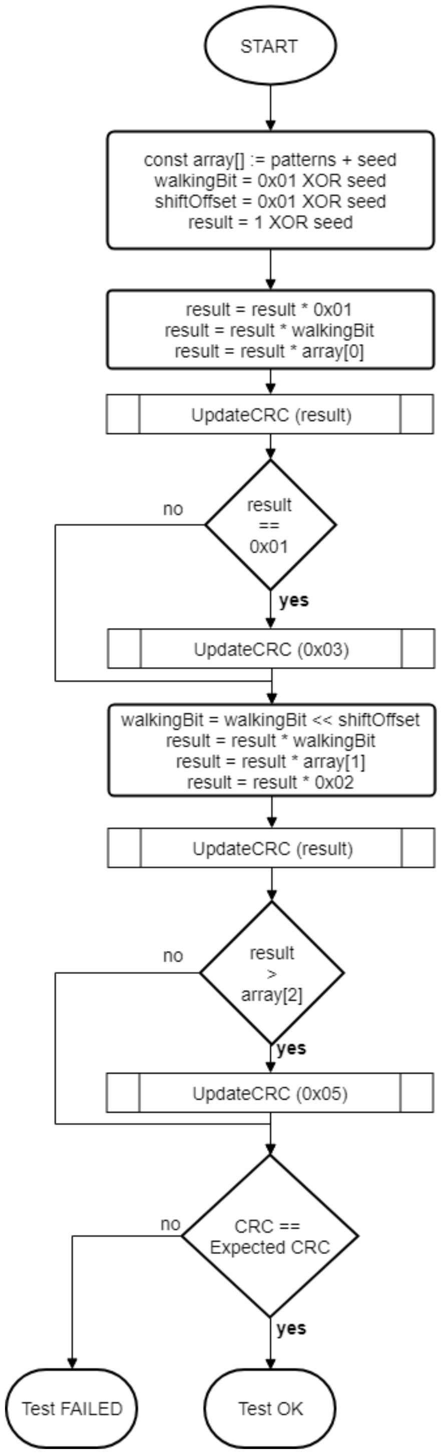

START
const array[] := patterns + seed
walkingBit = 0x01 XOR seed
shiftOffset = 0x01 XOR seed
result = 1 XOR seed
result = result * 0x01
result = result * walkingBit
result = result * array[0]
UpdateCRC (result)
result == 0x01
no
yes
UpdateCRC (0x03)
walkingBit = walkingBit << shiftOffset
result = result * walkingBit
result = result * array[1]
result = result * 0x02
UpdateCRC (result)
result > array[2]
no
yes
UpdateCRC (0x05)
CRC == Expected CRC
no
yes
Test FAILED
Test OK

Figure 4.6: Basic structure of the multiplication test

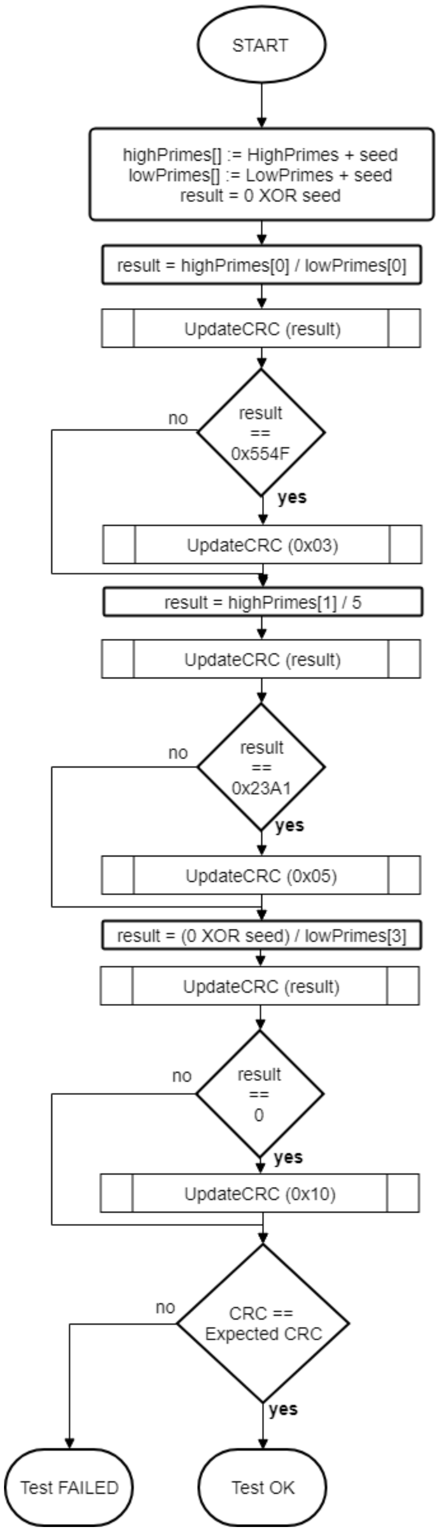
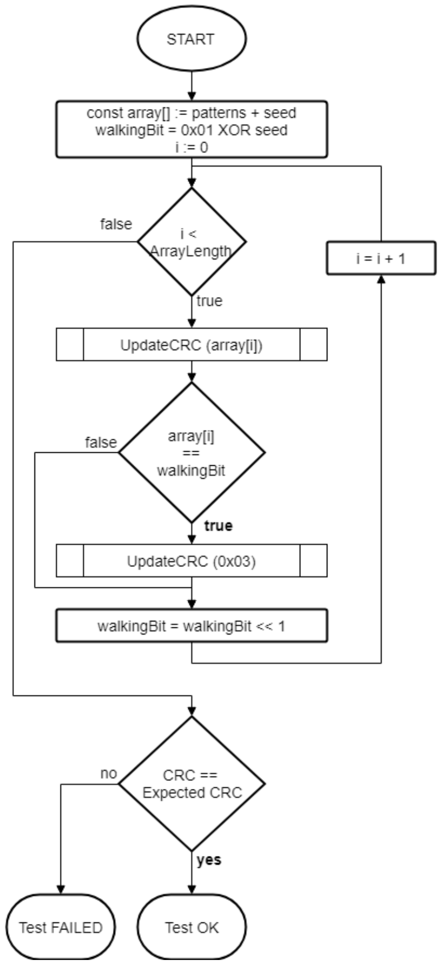


Figure 4.7: Basic structure of the division and modulo test

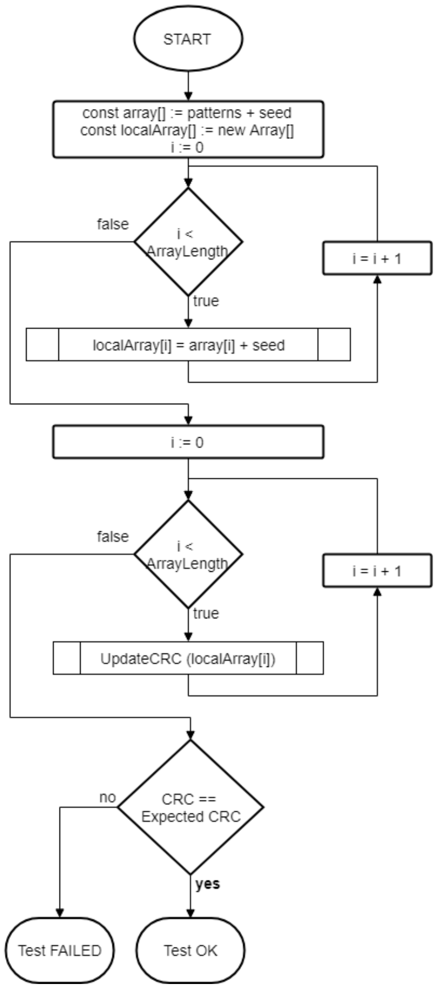
4.2.4 Data access operations

The first data access test is writing a value to a global variable, modifying the data and checking at the end if the value is correct. In the next step, the data access to arrays is tested. The test structure for reading values from an array is shown in figure 4.8a. An array with all bit shift values for the tested data type is used and compared with a freshly calculated bit shift value.

Figure 4.8b shows the test, which is utilized to test the instructions, used to store data into an array. The values from an existing global array are copied to an uninitialized local array. To evaluate the storing functionality all elements of the local array are added to the CRC.



(a) Basic structure to test the load array operations



(b) Basic structure to test the operations used to store information in an array

Figure 4.8: Tests for array operations

Data access tests for optimizing compiler

If compiler optimizations are used, the instructions for data access operations are variable. In addition to the instructions, which are generated based on the load and store array tests, more instructions are generated in more complex application code.

In some code snippets in which the compiler detects that a part of the array up to a specific index is not relevant anymore and in which no register for storing the index is easily available, the array pointer may be modified. This modification of the array pointer is sometimes done in one instruction together with reading or writing of a value. This optimization technique is applied only if most of the registers are used for other more variable operations, and there is no register available for the index variable. During the implementation of the SBST it was found out that this behavior was only shown when a lot of other operations and calculations were added to the test. In order to avoid a test of becoming to large, another test was found and implemented, in which the compiler was pushed in the direction to use the instructions which are reading or storing information in the array at the same time as modifying the array pointer. Whereas the idea of testing the load operation can be seen in figure 4.9, the test of the store instruction is illustrated in figure 4.10. The check if the index i is lower than a specific constant is only added to satisfy the static code analysis tool. Without those checks many warnings would have shown up.

Another instruction which is only applied when compiler optimizations are used is the load word instruction. This instruction loads the first 8 bit of a 32 bit element from an array. This is only done if the array is constant and holds 32 bit data types, but all the elements in there would have fit into an 8 bit data type. The compiler recognizes this behavior and reads only the first 8 bit of an element of the array. This procedure has to be done in a loop and the loaded element has to be compared with a 32 bit value within the loop. This test is illustrated in figure 4.11a.

More efficient instructions are also used if only one bit of the value to be read is essential. In this case, the compiler is doing a left shift operation together with the load operation in one instruction. After this load instruction, there is a right shift operation which sets the status of the CPU flags followed by a compare instruction which is a fast and straightforward check of the zero flag. The regular data access test would not cover this instructions, therefore a specific test was implemented which is presented in figure 4.11b.

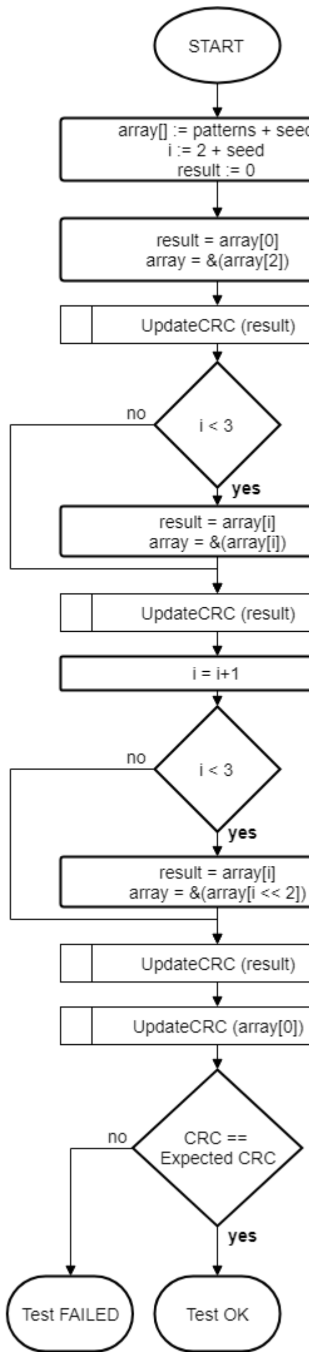


Figure 4.9: Structure to test data access to an array element while modifying the array pointer in the same instruction

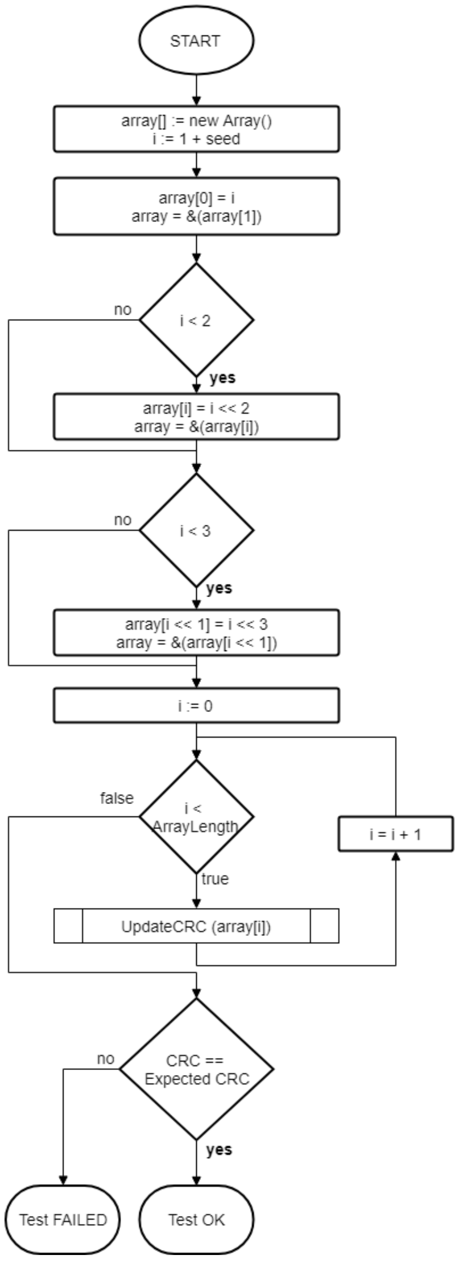
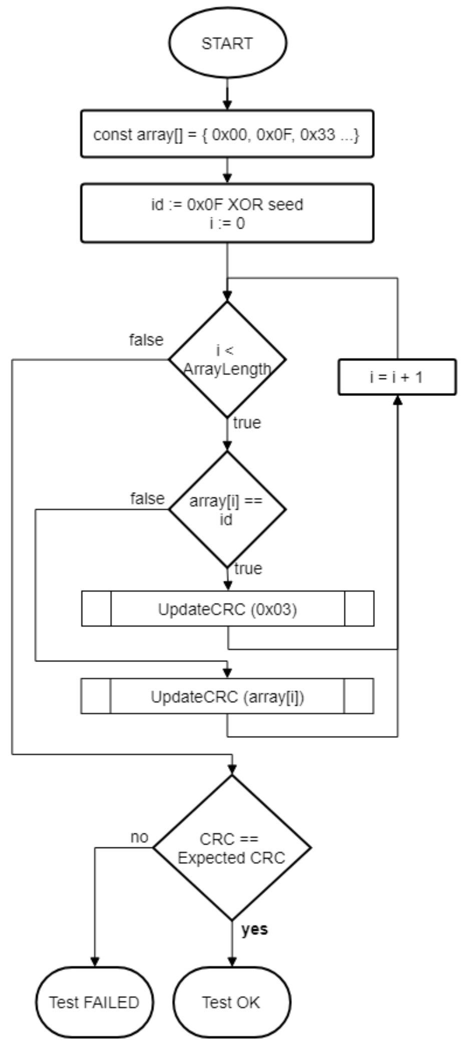
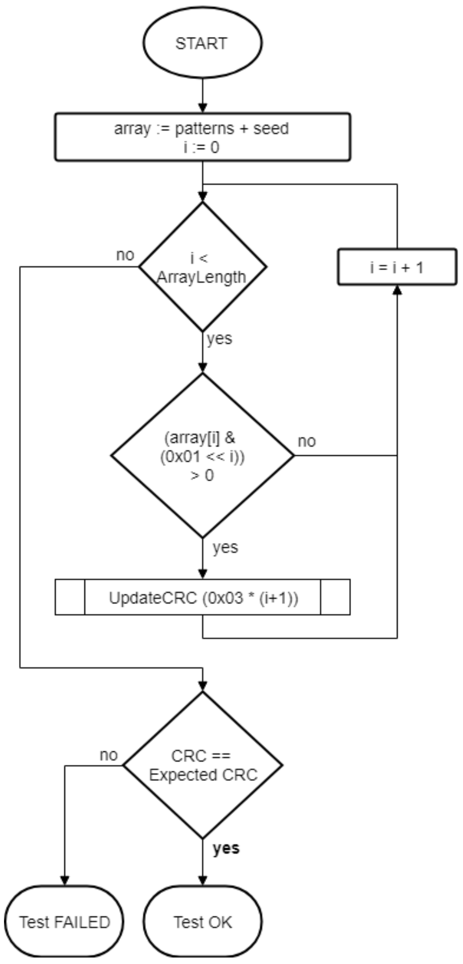


Figure 4.10: Structure to test the modifying of an element in the array while changing the array pointer in the same instruction



(a) Data access test if 8 bit values are stored in a 32 bit array



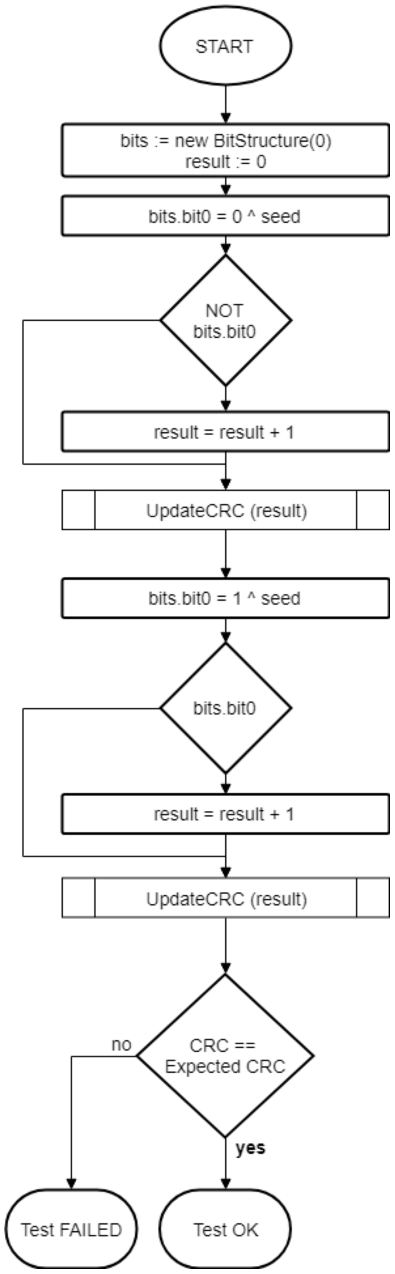
(b) Test structure to test only one bit of each value

Figure 4.11: Special data access tests

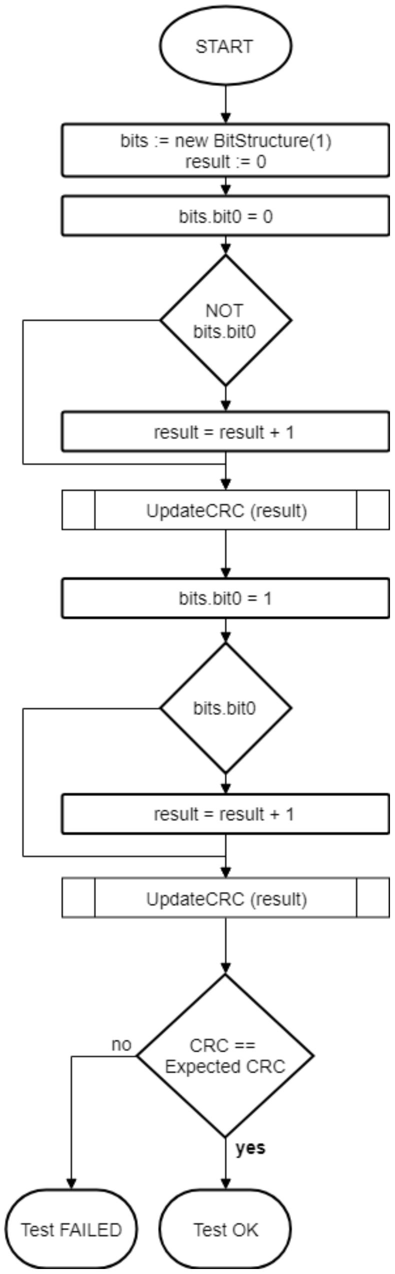
4.2.5 Bit operations

In this section the bit operations are tested. In order to gain access to the individual bits a struct with 32 bit fields is used. Each bit field has a size of only one bit. As shown in figure 4.12a, the bits are first initialized with 0, followed by setting the bit to a value which is not known by the compiler but which is always 0. After checking this bit it is changed to 1 and checked again. This procedure is repeated for each transit bit (for every first and last bit of each byte).

In the second step, which is shown in figure 4.12b, the procedure is almost the same, but in this test the compiler knows when the bit is set or cleared.



(a) Bit test where the value which is saved in a bit is unknown to the compiler



(b) Bit test where the value which is saved in a bit is known by the compiler

Figure 4.12: Tests for the bit operations

4.2.6 Memory functions

The library functions to set, copy and compare memory are widely used and need to be tested. As already specified in 3.3.1, these operations have to be tested with global constant, global and local memory. Therefore, a global constant and a global array of unsigned 32 bit integer were created.

The memory set test is executed in three steps (figure 4.13a):

- Copy the content of an array with different values to the destination array.
- Use the memory set function to override a part of the destination array with a constant value.
- The values of the destination array are added to the checksum and checked against the expected values.

The execution of the memory copy test is similar (figure 4.13b):

- Copy the content of an array with different values to the destination array.
- Use the memory copy function to override a part of the destination array with the content of the source array.
- The values of the destination array are added to the checksum and checked against the expected values.

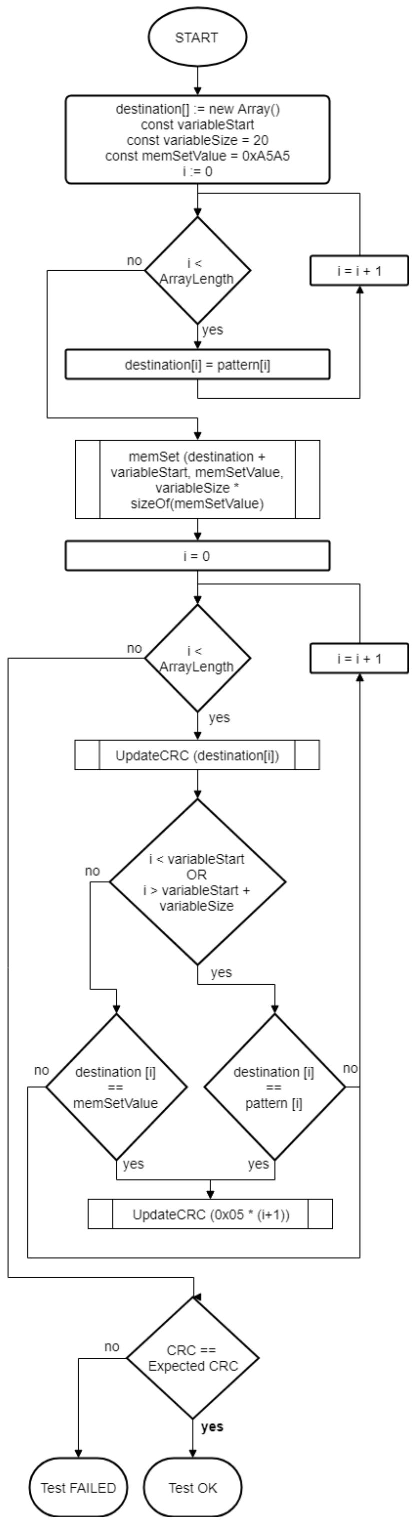
The memory compare function test is shown in figure 4.14. The first step is to execute the memory compare function with two arrays which have the same patterns. The detection of equality is expected. The second step is to execute the memory compare function with two different arrays of the same size but different content. The result of both compares is added to the CRC.

4.2.7 More library functions

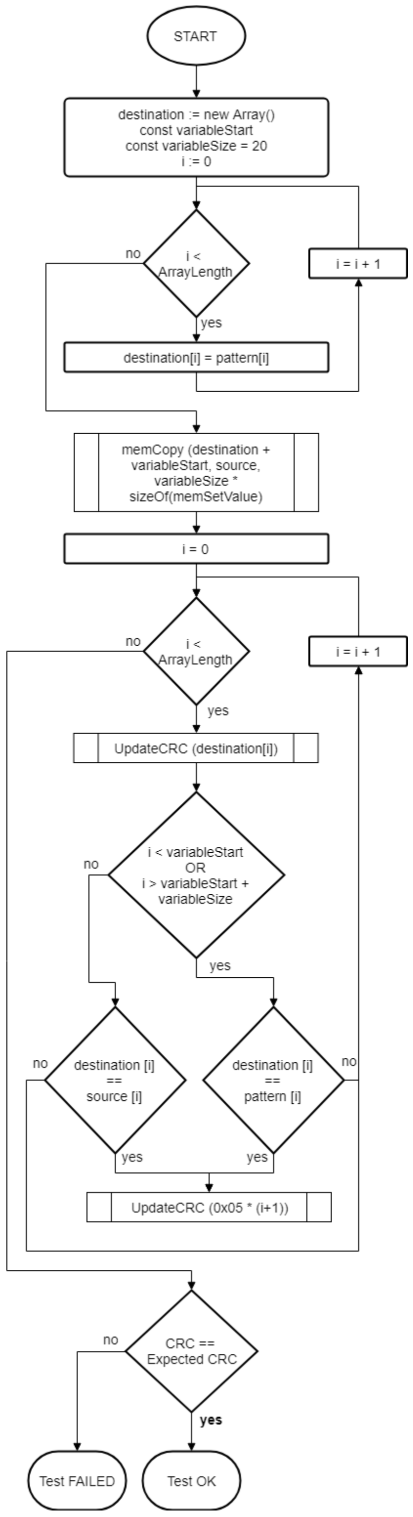
Another library function, which needs to be tested, is the long jump. The test is shown in figure 4.15. Set jump returns 0, if it is not called from long jump. Therefore, the result is 0, and the long jump with value 1 is initiated. The long jump returns to the position where set jump is called and returns with the given parameter 1. If the CRC value at the end is correct, both paths (the true and false path) are executed once.

4.2.8 Hardware specific instructions

As already discussed before, full hardware independent SBST with a 100% instruction coverage is not feasible. In this category instructions, which cannot be tested in C at all or only with much overhead and effort, are tested here. Especially with compiler optimizations some of the C operations are only compiled to specific instructions if all the side-conditions such as the number of already used instructions are fulfilled. In some cases, it does not make sense to have hundreds of lines of code and operations to simulate the side conditions to test one instruction. The instructions to enable and disable interrupts are tested here together with some unusual instruction combinations and carry flags.


START
destination[] := new Array()
const variableStart
const variableSize = 20
const memSetValue = 0xA5A5
i := 0
i < ArrayLength
no
yes
i = i + 1
destination[i] = pattern[i]
memSet (destination + variableStart, memSetValue, variableSize * sizeOf(memSetValue)
i = 0
i < ArrayLength
no
yes
i = i + 1
UpdateCRC (destination[i])
i < variableStart OR i > variableStart + variableSize
no
yes
destination [i] == memSetValue
no
yes
destination [i] == pattern [i]
no
yes
UpdateCRC (0x05 * (i+1))
CRC == Expected CRC
no
yes
Test FAILED
Test OK

(a) Basic structure of the memory set function test


START
destination := new Array()
const variableStart
const variableSize = 20
i := 0
i < ArrayLength
no
yes
i = i + 1
destination[i] = pattern[i]
memCopy (destination + variableStart, source, variableSize * sizeOf(memSetValue)
i = 0
i < ArrayLength
no
yes
i = i + 1
UpdateCRC (destination[i])
i < variableStart OR i > variableStart + variableSize
no
yes
destination [i] == source [i]
no
yes
destination [i] == pattern [i]
no
yes
UpdateCRC (0x05 * (i+1))
CRC == Expected CRC
no
yes
Test FAILED
Test OK

(b) Basic structure of the memory copy function test

Figure 4.13: Tests for the memory library functions copy and set

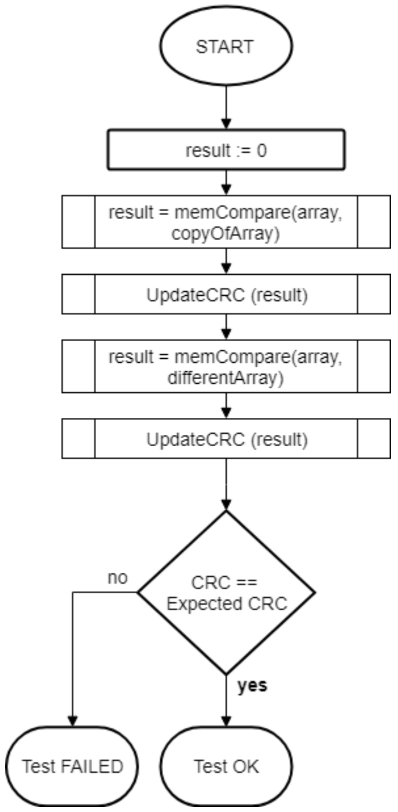


Figure 4.14: Basic structure of the memory compare function test

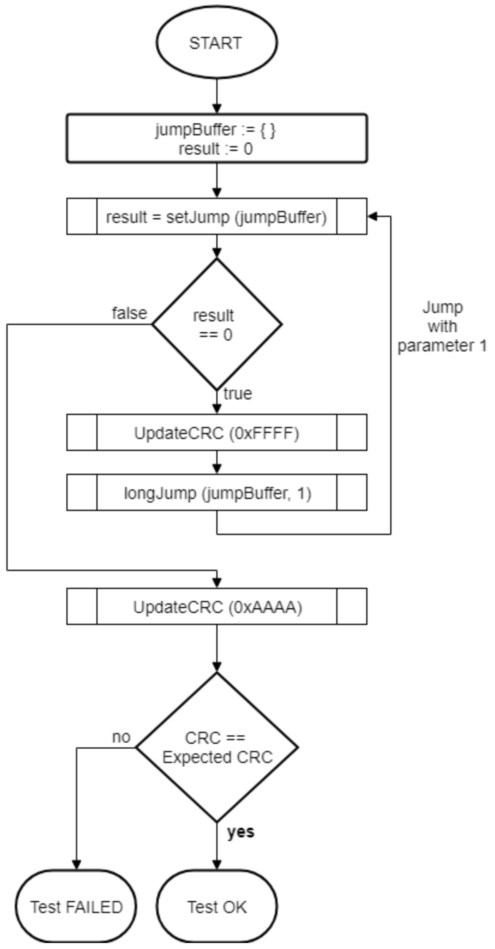


Figure 4.15: Test of the library function long jump

Chapter 5

Results and evaluation

This chapter presents the evaluation results of the SBST implementation. The used compilers and target CPU, which were used for the tests, are shown here as well. Furthermore, the results of the requirements shown in chapter 3.1 are evaluated.

5.1 Target CPU, compiler, and application

Target CPU

The SBST used target CPU for all these tests was the ARM Cortex-M3. The ARM Cortex-M is intended for microcontroller use and has been used in tens of billions of devices [ARM18]. Within the Cortex-M series, the Cortex-M3 is the most widely used. Not only because it was the first released and has the longest time on the market, but also because it meets the requirement for a general-purpose microcontroller. This means a right balance between high performance, low power consumption, and low cost. [Yiu13]

Some features of this CPU are [Yiu13] [OTUoLK17]:

- 13 general purpose registers and three special purpose registers
- Three-stage pipeline design
- Supports a mixture of 16 bit and 32 bit instructions
- 32 bit addressing, supporting 4 GB of memory space
- Support for bit-data accesses in two specific memory regions using a feature called bit band

As shown in 2.5.1 the ISA of this CPU provides a wide range of instructions [Yiu13]:

- General data processing, including hardware divide instructions
- Memory access instructions supporting 8 bit, 16 bit, 32 bit, and 64 bit data, as well as instructions for transferring multiple 32 bit data
- Instructions for bit field processing
- Multiply accumulate (MAC) and saturate instructions

Used compiler

The compiler used to evaluate the SBST was the Green Hills Arm 2013.5.4. According to their own statements, Green Hills Software is the leader in embedded optimizing compilers. *"... Green Hills Compilers can improve speed and reduce size by at least a 20% compared to the both GNU and LLVM compilers."* [Gre18]

Application code

Since the coverage is checked based on a comparison of an application with the SBST the tested applications are important as well. The SBST was compiled together and compared with four full applications which are actually in use. The application together with the SBST were compiled with the same compiler settings which were used for the application delivery. One system was compiled with optimizations turned on; the other three were compiled without the use of compiler optimizations.

5.2 Test results

This section shows the results of the SBST instruction coverage tests based on real applications. As already mentioned, the SBST was checked against four applications. In this chapter, the tests with the two larger application projects are shown in detail. One of this applications was compiled with compiler optimizations; the other one was not meant to be optimized.

5.2.1 Test slice execution times

As shown in table 5.1 the test slices are within the given range. The average execution time is 118 μs. All test slices executed one after another would take in total 0.004 seconds.

Test slice	μs
Relational operators	99
Bitwise operators NOT, AND NOT	118
Bitwise operators AND	131
Bitwise operators OR	146
Bitwise operators XOR	140
Bitwise operators bitshifts	103
Bit operations	100
Data access global variables	55
Data access volatile instructions	128
Data access load from array	142
Data access store in array	127
Addition unsigned 32 bit	125
Addition unsigned 64 bit	111
Addition signed	117
Subtraction unsigned	149
Subtraction signed	121
Multiplication unsigned 64 bit	132
Multiplication unsigned 16 + unsigned 32 bit	134
Multiplication 8 bit, signed 16 bit	102
Multiplication signed 32 bit	115
Division	131
Modulo	129
Memory functions set local	128
Memory functions set global	128
Memory functions compare	64
Memory functions copy local to local	142
Memory functions copy global to local	142
Memory functions copy global to global	141
Memory functions copy global to local	141
Memory functions copy const to local	143
Memory functions copy const to global	142
Library functions	51
Tests in ASM	46

Table 5.1: Test slices with execution times

5.2.2 Test results when optimization techniques are not used

Table 5.2 shows the number of instructions generated by the test codes. Most of those instructions are implemented in this SBST, but also instructions from other tests such as the flash or RAM test as well as tests for other components are included. The list does not show the number of different instructions; it shows the total number.

Test source	**# of instructions**
Instruction test environment	128
Addition tests	963
Subtraction tests	846
Multiplication tests	1713
Division tests	1184
Bit operation tests	499
Bitwise operands tests	2250
Data access tests	989
Relational operators tests	519
Memory library functions tests	375
Other library functions tests	105
Tests in assembly	85
Register tests	395
Flash tests	209
Ram tests	753
Other tests	898
Total number of instructions	**11911**

Table 5.2: Number of instructions generated from test code

The total number of instructions used in the application is more than 40000.

Number of different instructions

In total, 109 different instructions (utilizing the parameter) are tested. The most significant number of different instructions were used for data access. While 16 instructions were applied to load the data, seven instructions were needed to store the information.

From those 109 different instructions 101 are used in the application. There are no instructions utilized in the application which are not tested by the SBST. A detailed list of the instructions tested is shown in A.1.

System calls

In addition to the memory library system calls and the long jumps, which are explicitly implemented in the self-test, there were several additional system calls used by the application. Only four of those system calls utilized by the application were not tested. Analysis has shown that those system calls are only executed on startup and when the system is shutting down (destruction). These system calls cannot be tested and were covered by the system designer: such a malfunction would not lead to critical situations.

CPU flags

The tested CPU flags are shown in 5.3. "Yes" means that a flag check is covered in the tests. An empty field means that the CPU flag status was affected by an instruction but not checked by the tests and also not used by the application. A "No" would mean that the flag status is used by the application but not checked by the tests. Only instructions, which are being utilized at least once, are shown in this list.

Instruction	**Tested flags**			
	Zero	**Carry**	**Negative**	**Overflow**
adds	yes	yes		
cmp	yes	yes	yes	yes
movs				
msr	yes	yes	yes	yes
sbcs		yes		
subs	yes	yes		
tst	yes			

Table 5.3: CPU flag coverage

Coverage

As shown above, all instructions used in the application are covered by the test, as well as the needed system calls are tested, and the functionality of the flags is verified in the tests as well. Therefore, the instruction coverage is given, and the test is successful.

5.2.3 Test results with the usage of optimization techniques

Table 5.4 shows the number of instructions generated by the test code. As already described in 5.2.2, for the test with optimizing compilers the instructions applied for other tests are also added to the list of approved ASM instructions.

The number of total instructions used by the application is around 27 thousand.

Number of different instructions

In total, 184 different instructions (utilizing the parameter) are tested. Also, in case of optimizations the largest number of different instructions were used for data access. While 31 instructions were applied to load the data, 22 instructions were needed to store the information.

From those 184 different instructions 156 are used in the application. There are no instructions utilized in the application which are not tested by the SBST. A detailed list of the instructions tested is shown in A.2.

System calls

In addition to the memory library system calls and the long jumps, which are explicitly implemented in the self-test, there were several additional system calls used by the appli-

Test source	**# of instructions**
Instruction test enviroment	125
Addition tests	752
Subtraction tests	697
Multiplication tests	1241
Division tests	938
Bit operation tests	307
Bitwise operands test	1634
Data Access tests	731
Relational operators tests	310
Memory library functions tests	360
Other library functions tests	90
Tests in assembly	85
Register tests	343
Flash tests	153
Ram tests	569
Other tests	693
Total number of instructions	**9028**

Table 5.4: Number of instructions generated from test code when compiler optimizations are used

cation, some more in comparison to the non-optimized intermediate code. Only four of those system calls used by the application were not tested. The analysis has shown that these system calls are only executed on startup and when the system is shutting down (destruction). As described before, those system calls cannot be tested and were covered by the system designer: such a malfunction would not lead to critical situations.

CPU flags

The tested CPU flags are shown in 5.5. "Yes" means that a flag check is covered in the tests. An empty field means that the CPU flag status was affected by an instruction but not checked by the tests and also not used by the application. A "No" would mean that the flag status is utilized by the application but not checked by the tests. Only instructions which are being used at least once are shown in this list.

Coverage

As presented above all instructions used in the application as well as the needed systems calls are covered by the test. Moreover, the functionality of the flags is verified in the tests. Therefore, the instruction coverage is given, and the test is successful.

Instruction	**Tested flags**			
	Zero	**Carry**	**Negative**	**Overflow**
adcs				
adds	yes	yes		
adds.w	yes			
ands	yes			
asrs	yes			
bics	yes			
cmp	yes	yes	yes	yes
eors	yes			
lsls	yes		yes	
lsrs	yes			
movs	yes			
movs.w			yes	
msr	yes	yes	yes	yes
muls				
mvns				
negs				
orrs	yes			
sbcs		yes		
subs	yes	yes	yes	
teq	yes			
tst	yes			

Table 5.5: CPU flag coverage if optimization techniques are used

5.2.4 Other tested applications

The test was also executed with two more applications which were compiled without optimizations. Since those two projects were smaller projects with less functionality, it was no surprise that the coverage was given.

5.2.5 Discussion of the results

This section discusses the differences between the two test results. Even if the two projects do not have the same size a comparison is done here. The second project, which was compiled with an optimizing compiler, is a little bit larger in terms of application code size than the other one.

As shown in table 5.6, the total number of instructions is lower when compiler optimizations are utilized and the number of different instructions increased. This is the expected behaviour - the total number of instructions used by the application was reduced by 35% while the reduction for the test code reached only 24%. One reason for these differences might be that most of the tests were implemented in a way to avoid too many optimizations: the list of tested instructions includes many operations which are only used to bring in the seed variable. Those instructions do not improve the coverage but are needed to avoid the compiler optimizations as shown in 3.4.

		Tests		**Application**	
Without compiler optimizations					
	# Instructions	11911		40775	
	# Different instructions	109		101	
	# Instructions flag checks	7		3	
Compiler optimizations					
	# Instructions	9028		26613	
	# Different instructions	184		157	
	# Instructions flag checks	21		13	
Difference		#	percent	#	percent
	# Instructions	-2883	-24%	-14162	-35%
	# Different instructions	+75	+69%	+56	+55%
	# Instructions flag checks	+16	+300%	+10	+333%

Table 5.6: Comparison of the instructions

5.3 Discussion of the system requirements

The following section includes a list of the system requirements and a discussion of how well the requirements are satisfied

- **Application driven approach**

  As verified in the result section, the application instructions are covered and tested. The number of additional tested instructions, which are not needed for the application, is less than 20%. This value can be easily reduced if separate SBST are used for optimizing and non-optimizing compilers.

- **C based self-tests**

  All SBST, which can be reasonably implemented in C, are done so.

- **Additional assembly based self-tests**

  The number of instructions, which are implemented in ASM, is low. Only seven instructions, which could not be tested in C, needed to be tested in ASM. Due to the preparation of the registers and the check of the result, in total 85 lines of ASM code were used for the implementation of those tests.

- **Portability**

  All C instructions tests are portable to another hardware platform and compiler. Only the specific ASM instructions, which are grouped in one class, need to be adapted when another hardware should be used.

- **Test in slices and verification of the SBST results**

  All tests are executed in slices, none of which is exceeding the maximum execution time. The outcome of each test slice are the calculated CRC value as well as the correct expected CRC value.

- **RAM and register tests**

  Those tests were already implemented before. The implementation was verified and the used ASM instructions were added to the coverage analysis.

- **Coverage measurement**

  Coverage was given for all four projects which were available for this test.

- **Coverage with enabled compiler optimizations**

  Even if compiler optimizations were turned on the coverage was given.

- **Integration of additional self-tests**

  Additional self-tests can be easily implemented by deriving from the base class which contains helpful features and test patterns.

5.4 Difficulties

The difficulties during the design and implementation of this master thesis were mostly concerned about the support of compiler optimizations.

The first implementation of the SBST did not consider optimizing compilers and already had a good coverage. After further investigation on the first prototype and the comparison with the instructions generated from an application was done, the SBST tests were improved and additional tests were added. Uncovered instructions were located and reasonable tests were found for most of them. Only a few instructions needed to be tested in ASM (like turning the interrupts on and off).

This behavior changed dramatically when the support for compiler optimization techniques was added. The need for the seed variable to almost any initialization and other operations added a lot of overhead, which did not improve the test coverage. An addition or exclusive or operation, in which one parameter has the value 0, does not improve the test quality, especially if this test is done dozens of times in each test slice. Another disadvantage with compiler optimizations is that only one SBST delivery was requested at the end. The SBST was supposed to work no matter if compiler optimizations were used or not. This led to a lower test efficiency especially if optimizing compilers are not applied.

On the other hand, it was quite a challenge to find a test code in C that covers all the instructions which were not covered but needed for compiler optimizations. When seeking for a C test code in order to test a specific ASM instruction is often uncomplicated, it is not the same with optimizing compilers. Some instructions are only used in certain circumstances, e.g., if a specific amount of registers are already in use and other instructions are executed after a specific operation. Nevertheless, since it was mandatory to use as less ASM code as possible and cover as much as possible in C a lot of effort and time (weeks) was invested to find related tests in C. For most of those instructions (in total there were 15 instructions for which it was hard to find corresponding C tests) a test was found, but some of those tests are not very stable. A few of these tests do not test the requested ASM instructions anymore when the compiler settings are changed, or the source code is changed a little bit. Sometimes the coverage was already broken when the code was changed in so far that it was compatible with the coding guidelines. In case of a compiler or hardware change, a change in the generated instructions is expected.

Chapter 6

Conclusion

This master thesis gave an overview of fail-safe systems with a particular focus on the IEC 61508 safety standard. Different methods to test hardware components were given. An overview of current approaches to test CPUs with the focus on the software-based self-tests was presented. Furthermore, the analysis discussed the advantages and disadvantages of the different testing methods. This thesis also gave insight into the development of a generic software-based self-test of which the coverage analysis was based on the used ASM instructions. Furthermore, the profound impact of using optimizing compilers was shown.

6.1 Discussion of the results

An appropriate generic SBST was developed within the scope of this thesis. Since the primary goal of this master thesis was the portability to other hardware platforms the tests were implemented in C. Not only the easy to test components get tested with hardware independent C code, the goal was full coverage of the used ASM instructions. An instruction coverage analysis tool was applied to determine the completeness of the SBST. Therefore, the currently existing generic approaches, in which only established tests with known fault coverage are used to test specific hardware elements, did not go far enough. In theory, simplified all C language constructs, which might be applied in the application, need to be tested within the C based instruction test. The application as well as the self-test are compiled to assembly listings. Only if all application ASM instructions together with its flags are analyzed in the self-test, the coverage is given. If one instruction or flag check is missing the self-test is rejected. A full coverage is important because some instructions, which might rarely be utilized in the application, may only be executed in exceptional situations. The emergency stop switch can be one of those specific situations. If a rare instruction is only used in case of an emergency shutdown, a failure in that functionality can be hazardous. Only a limited number instructions, which cannot be tested in C code, are approved with ASM instructions.

Using optimizing compilers has serious consequences when using generic self-tests. As shown in this thesis, SBST, which are not supposed to be used with such compilers will lose most of their abilities to detect glitches. The outcome of a SBST is already known at compile time. If this behavior is discovered by the compiler, the test code is seen as unnecessary and will be removed. Even after the tests are modified in order to hide the constant behavior the coverage of such tests is not satisfying. Optimizing compilers are not only transforming the C code into ASM instructions, amongst other things the environment is analyzed and computations are reordered. The instruction selection is also an extensive topic and not always comprehensible. Despite or even because of this fact, a full instruction coverage was still requested which made the implementation for optimizing compilers challenging.

Software certification, which is mandatory in safety-critical systems, is often related to source code. A compiler usually only transforms the source into machine code. But an optimizing compiler also modifies or even rewrites the program in order to be more efficient. [Sch12] High levels of compiler optimizations may not be suitable for safety-critical systems, even the IEC 61508 safety standard in part 3 include warnings about optimizing compilers. [Hob17] [Com11] [Sch12] Nevertheless, the implementation of the SBST provides full coverage with the given compiler and application shown in chapter 5.1. Due to the vast possibilities and freedoms, which are grant to an optimizing compiler, the ASM outcome for all optimization techniques and source codes is very hard to predict. Nevertheless, the coverage is judged based on that outcome and therefore full coverage cannot be guaranteed if the hardware platform changed or another compiler is used.

In sum, the approach with C based SBST and coverage analysis based on the compiled ASM instructions works great as long as optimizing compilers are not used. For the usage with optimizing compilers a full coverage is complex and adjustments might be necessary for each hardware change and compiler change. Even a change in the application may lead to additional instructions which are not tested. Therefore, another SBST approach might suit better for optimizing compilers.

6.2 Future work

There are many ideas which could improve this work:

- **Fault injection framework**
  A fault injection framework should be used to realize the SFF. Only faults in utilized resources should be tested. Resources, which are not needed by the application do not need to be covered and such undetected faults should not reduce the SFF.

- **Test separation between optimizing and non-optimizing compilers**
  The requirements for this master thesis were clear that only one SBST has to be implemented and this one has to have a full coverage for optimizing as well as non-optimizing compilers. As expected and confirmed during the development, the SBST could be much smaller, more efficient and more portable if the support for optimizing compilers is only given in a separate implementation.

- **More coverage tests on different hardware and compilers**
  As shown in chapter 5 the implementation was tested on four applications. Especially when optimizing compilers are used the evaluation with more applications would be of interest.

- **More test of hidden parts**
  Tests for some hidden parts such as pipelining or branch prediction units would be of

interest. Since a malfunction of some of those techniques only leads to performance degradation they are not detected by the currently used tests.

Appendix A

List of tested instructions

In this appendix the tested instructions are shown with the compiler specified in 5.1.

A.1 Tested instructions if compiler optimization techniques are not used

Instruction	**# of usages**
adc register,register,register	35
adc register,register,numeric	4
add register,register,numeric	1289
add register,numeric	51
add register,register,register	260
add register,register	11
add register,register,register shift	8
adds register,register	4
adds register,register,register	35
adds register,numeric	5
adds register,register,numeric	3
adr.w register,label	2
and register,register,numeric	94
and register,register,register	41
asr register,register,register	8
b label	102
beq label	90
bfc register,numeric,numeric	56
bge label	10
bgt label	7
bhi label	33
bhs label	27
bic register,register,register	6
bic register,register,numeric	1
bl label	1083

ble label	10
blo label	71
bls label	24
blt label	8
blx register	12
bne label	210
bx register	1
cbnz register,label	116
cbz register,label	91
cmp register,register	252
cmp register,numeric	194
cpsid iflags	1
cpsie iflags	1
cpy register,register	1549
eor register,register,numeric	162
eor register,register,register	70
it condition	1
ite condition	59
ldmfd [register]!,registerlist	36
ldmfd [register],registerlist	1
ldr register,[register,numeric]	612
ldr register,[register]	228
ldr register,label	1
ldr.w register,[register]	78
ldr.w register,[register,numeric]	225
ldr.w register,[register],numeric	6
ldr.w register,label	7
ldrb register,[register]	106
ldrb register,[register,numeric]	56
ldrh register,[register]	60
ldrh register,[register,numeric]	4
ldrsb register,[register]	4
ldrsh register,[register]	16
ldrsh register,[register,numeric]	1
lsl register,register,register	50
lsr register,register,register	9
mov register,numeric	1117
mov register,register shift	219
mov register,register	90
movs register,register shift	1
movt register,condition(label)	367
movt register,numeric	154
movw register,condition(label)	367

movw register,numeric	154
mrs register,register	4
msr register,register	4
mul register,register,register	114
mvn register,numeric	42
mvn register,register	28
orr register,register,register	118
orr register,register,numeric	36
orr register,register,register shift	3
pop registerlist	99
push registerlist	99
ret	15
rsb register,register,numeric	3
rsb register,register,register shift	8
sbc register,register,register	16
sbc register,register,numeric	5
sbcs register,register,register	26
sdiv register,register,register	31
stmea [register],registerlist	1
stmfd [register]!,registerlist	23
stmia [register]!,registerlist	13
str register,[register,numeric]	362
str register,[register]	125
str register,[register,numeric]!	6
strb register,[register,numeric]	30
strb register,[register]	19
strh register,[register,numeric]	3
strh register,[register]	8
sub register,numeric	35
sub register,register,register	82
sub register,register,numeric	61
subs register,register,register	39
subs register,numeric	29
subs register,register,numeric	4
sxtb register,register	15
sxth register,register	26
tst register,numeric	24
ubfx register,register,numeric,numeric	1
udiv register,register,register	41
uxtb register,register	143
uxth register,register	104

Table A.1: Tested instructions without compiler optimization techniques

A.2 Tested instructions if compiler optimization techniques are used

Instruction	**# of usages**
adc register,register,numeric	8
adc register,register,register	1
adcs register,register	30
add register,numeric	59
add register,register,numeric	896
add register,register,register shift	71
add register,register,register	7
add register,register	4
adds register,register,numeric	16
adds register,numeric	103
adds register,register,register shift	1
adds register,register	4
adds register,register,register	92
adds.w register,register,numeric	2
adr.w register,label	2
and register,register,numeric	47
and register,register,register	15
ands register,register	18
ands register,register,numeric	1
asr register,register,register	1
asrs register,register	7
asrs register,register,numeric	12
b label	122
beq label	120
bfc register,numeric,numeric	8
bfi register,register,numeric,numeric	16
bge label	7
bgt label	6
bhi label	16
bhs label	29
bic register,register,register	2
bic register,register,numeric	5
bics register,register	6
bl label	1048
ble label	9
blo label	73
bls label	14
blt label	8
blx register	11
bmi label	3

bne label	180
bpl label	3
bx register	2
cbnz register,label	56
cbz register,label	35
cmp register,register	263
cmp register,numeric	172
cmp register,register shift	1
cpsid iflags	1
cpsie iflags	1
cpy register,register	807
eor register,register,numeric	147
eor register,register,register	34
eor register,register,register shift	6
eors register,register,numeric	5
eors register,register	19
eors register,register,register shift	1
it condition	98
ite condition	36
itee condition	3
iteee condition	1
itt condition	7
itte condition	2
ittee condition	1
ittt condition	15
ittte condition	1
itttt condition	1
ldmfd [register]!,registerlist	75
ldmfd [register],registerlist	10
ldmia [register]!,registerlist	5
ldr register,[register,numeric]	466
ldr register,[register]	19
ldr register,label	1
ldr register,[register_offset]	2
ldr.w register,[register,numeric]	216
ldr.w register,[register_offset_shift]	32
ldr.w register,label	7
ldr.w register,[register],numeric	5
ldr.w register,[register,numeric]!	2
ldr.w register,[register]	4
ldr.w register,[register_offset]	2
ldrb register,[register_offset]	36
ldrb register,[register]	10

ldrb register,[register,numeric]	48
ldrb register,[register],numeric	4
ldrb register,[register_offset_shift]	1
ldrb register,[register,numeric]!	1
ldrh register,[register,numeric]	33
ldrh register,[register],numeric	2
ldrh register,[register_offset]	20
ldrh register,[register,numeric]!	3
ldrh register,[register]	3
ldrsb register,[register]	1
ldrsb register,[register,numeric]	3
ldrsh register,[register],numeric	2
ldrsh register,[register_offset]	3
ldrsh register,[register,numeric]	10
ldrsh register,[register]	1
lsl register,register,register	5
lsls register,register,numeric	65
lsls register,register	15
lsr register,register,register	1
lsrs register,register,numeric	26
lsrs register,register	8
mla register,register,register,register	16
mls register,register,register,register	21
mov register,numeric	279
mov register,register	76
mov register,register shift	1
movs register,register	47
movs register,numeric	512
movs.w register,register shift	2
movt register,condition(label)	180
movt register,numeric	162
movw register,condition(label)	180
movw register,numeric	162
mrs register,register	4
msr register,register	4
mul register,register,register	19
muls register,register	71
mvn register,numeric	39
mvn register,register	1
mvns register,register	23
negs register,register	1
orr register,register,numeric	44
orr register,register,register	21

orr register,register,register shift	27
orrs register,register,register	16
orrs register,register,numeric	3
orrs register,register	21
pop registerlist	99
push registerlist	98
ret	26
rsb register,register,numeric	3
rsb register,register,register shift	15
sbc register,register,numeric	4
sbcs register,register,register	18
sbcs register,register,numeric	8
sbcs register,register	17
sdiv register,register,register	19
smull register,register,register,register	12
stmea [register],registerlist	44
stmea [register]!,registerlist	5
stmfd [register]!,registerlist	18
stmia [register]!,registerlist	8
str register,[register,numeric]	215
str register,[register_offset_shift]	5
str register,[register]	11
str register,[register,numeric]!	4
str register,[register],numeric	2
str register,[register_offset]	1
strb register,[register,numeric]	27
strb register,[register_offset]	3
strb register,[register_offset_shift]	1
strb register,[register],numeric	6
strb register,[register,numeric]!	2
strb register,[register]	1
strh register,[register,numeric]	5
strh register,[register_offset]	3
strh register,[register]	3
strh register,[register],numeric	1
strh register,[register,numeric]!	1
sub register,numeric	39
sub register,register,numeric	35
sub register,register,register	9
sub register,register,register shift	8
subs register,register,register	90
subs register,numeric	37
subs register,register,numeric	11

subs register,register,register shift	2
subs.w register,register,numeric	1
sxtb register,register	5
sxth register,register	5
teq register,numeric	3
teq register,register	2
teq register,register shift	1
tst register,numeric	42
tst register,register	3
tst register,register shift	2
ubfx register,register,numeric,numeric	4
udiv register,register,register	20
umull register,register,register,register	34
uxtb register,register	60
uxth register,register	53

Table A.2: Tested instructions if compiler optimization techniques are used

Appendix B

Abbreviations

ALU arithmetic logic unit

ASIC application-specific integrated circuit

ASM assembly

BIST build-in self-test

CISC complex instruction set computing

CRC cyclic redundancy check

ECC error correcting code

EMC electromagnetic compatibility

Galpat galloping patterns

GNU complex instruction set computing

HFT hardware fault tolerance

IEC International Electrotechnical Commission

ISA instruction set architecture

MBU multiple bit upsets

MIPS microprocessor without interlocked pipeline stages

RISC reduced instruction set computing

SBST software-based self-test

SFF safe failure fraction

SIL safety integrity level

XOR exclusive or

Bibliography

[AMH07] Reza Adhami, III Peter M. Meenen, and Denis Hite. *Fundamental Concepts in Electrical and Computer Engineering with Practical Design Problems*, volume 2. Universal Publishers, 2007.

[ARM10] ARM Limited. *Cortex-M3 Devices Generic User Guide.* 1 edition, 2010.

[ARM18] ARM Limited. Cortex-M - ARM, 2018. http://www.arm.com/products/processors/cortex-m.

[BCH13] Luiz A. Barroso, Jimmy Clidaras, and Urs Holzle. *The datacenter as a computer: an introduction to the design of warehouse-scale machines.* 2013.

[Bli16] G.H. Blindell. *Instruction selection: Principles, methods, and applications.* 01 2016.

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

[Com11] International Electrotechnical Commission. *Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme.* 2011.

[GPZ04] Dimitris Gizopoulos, Antonis Paschalis, and Yervant Zorian. *Embedded processor-based self-test*, volume 28. Kluwer, Dordrecht [u.a.], 2004.

[Gre18] Green Hills Software Ltd. The most advanced optimizing compiler technology, 2018. https://www.ghs.com/Benchmarks.html.

[Gro06] Ian A. Grout. *Integrated circuit test engineering: Modern techniques.* 2006.

[HKK08] David Hutchison, Takeo Kanade, and Josef Kittler. *Computer Safety, Reliability, and Security : 27th International Conference, SAFECOMP 2008 Newcastle upon Tyne, UK, September 22-25, 2008 : Proceedings*, volume 5219. Springer Berlin Heidelberg, Berlin/Heidelberg, 2008.

[Hob17] C. Hobbs. *Embedded Software Development for Safety-Critical Systems.* CRC Press, 2017.

[IS16] Thomas Kaegi-Trachsel Igor Schagaev. *Software Design for Resilient Computer Systems.* Springer International Publishing, first;1; edition, 2016.

[Jou11] Norman P. Jouppi. Technical perspective: Dram errors in the wild. *Association for Computing Machinery. Communications of the ACM*, 54(2):99, 2011.

[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 *Proceedings of the Design Automation Test in Europe Conference*, volume 1, pages 1–6, March 2006.

[KPGX05] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis. Software-based self-testing of embedded processors. *IEEE Transactions on Computers*, 54(4):461–475, 2005.

[Mah13] Vincent Mahout. *Assembly Language Programming: ARM Cortex-M3*. Iste, US, 2012;2013;.

[Man03] C. J. H. Mann. The compiler design handbook - optimisation and machine code generation. *Kybernetes*, 32(9/10):1562, 2003.

[NTA78] Nair, Thatte, and Abraham. Efficient algorithms for testing semiconductor random-access memories. *IEEE Transactions on Computers*, C-27(6):572–576, 1978.

[OTUoLK17] Oluwole O. Oyetoke, Department of Electronic The University of Leeds, and Electrical Engineering; LS2 9JT; Leeds; United Kingdom. A practical application of arm cortex-m3 processor core in embedded system engineering. *International Journal of Intelligent Systems and Applications*, 9(7):70–88, 2017.

[Pfl03] Matthias Pflanz. *On-line error detection and fast recover techniques for dependable embedded processors*, volume 2270. Springer, 2003.

[PGSR10] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo S. Reorda. Microprocessor software-based self-testing. *IEEE Design & Test of Computers*, 27(3):4–19, 2010.

[PH09] Seong Poong-Hyun. *Reliability and Risk Issues in Large Scale Safety-critical Digital Control Systems*. Springer-Verlag London, 2009.

[PKH$^+$13] Christopher Preschern, Nermin Kajtazovic, Andrea Holler, Christian Steger, and Christian Kreiner. Verifying generic iec 61508 cpu self-tests with fault injection. pages 1–2. IEEE, 2013.

[Pre14] Christopher Preschern. Design and implementation of a fault injection system for verifying generic cpu safety-tests. Master's thesis, 2014.

[RCS$^+$16] Andreas Riefert, Riccardo Cantoro, Matthias Sauer, Matteo S. Reorda, and Bernd Becker. A flexible framework for the automatic generation of sbst programs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(10):3055–3066, 2016.

[Sch12] Dietmar Schreiner. Optimizing compilers for safety-critical robotic systems. *Proceedings of the 'Austrian Robotics Workshop 2012 (ARWS2012)'*, page 6, 2012.

[Sch15] Gerhard Schönfelder. Fies: a fault injection framework for the evaluation of self-tests. Master's thesis, 2015.

[SEN05] Saeed Shamshiri, Hadi Esmaeilzadeh, and Zainalabdein Navabi. Instruction-level test methodology for cpu core self-testing. *ACM Trans. Des. Autom. Electron. Syst.*, 10(4):673–689, October 2005.

[SIA15] SIA Semiconductor Industry Association. The International Technology Roadmap for Semiconductors 2.0. Technical report, Semiconductor Industry Association, 2015.

[Sos06] Janusz Sosnowski. Software-based self-testing of microprocessors. *Journal of Systems Architecture*, 52(5):257–271, 2006.

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. volume 37, pages 193–204. ACM, 2009.

[SR14] Ernesto Sanchez and Matteo S. Reorda. On the functional test of branch prediction units. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(9):1675–1688, 2015;2014;.

[SS16] David J. Smith and Kenneth G. L. Simpson. *The Safety Critical Systems Handbook: A Straightforward Guide to Functional Safety: IEC 61508 (2010 Edition), IEC 61511 (2015 Edition) and Related Guidance.* Butterworth Heinemann, Oxford, fourth;4; edition, 2016.

[ST10] J. Sosnowski and L. Tupaj. Cpu testability in embedded systems. pages 108–112. IEEE, 2010.

[TA95] S. M. Thatte and J. A. Abraham. A methodology for functional level testing of microprocessors. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pages 326–, Jun 1995.

[WCTI03] Baosheng Wang, Cho, Tabatabaei, and Ivanov. Yield, overall test environment timing accuracy, and defect level trade-offs for high-speed interconnect device testing. volume 2003-, pages 348–353. IEEE, 2003.

[WST10] Laung-Terng Wang, Charles E. Stroud, and Nur A. Touba. *System-on-Chip Test Architectures.* Morgan Kaufmann, 1 edition, 2010.

[WXT11] Chang Wei, Bao Xiaohong, and Zhao Tingdi. A study on compiler selection in safety-critical redundant system based on airworthiness requirement. *Procedia Engineering*, 17:497–504, 2011.

[Yan05] Laurence Tianruo Yang. *Embedded Software and Systems: Second International Conference, ICESS 2005, Xian, China, December 16-18, 2005. Proceedings*, volume 3820. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[Yiu13] Joseph Yiu. *The definitive guide to ARM Cortex-M3 and Cortex-M4 processors*. Newnes/Elsevier, Amsterdam [u.a.], 3.;third;3; edition, 2014;2013;.

[ZZR06] R. Zhang, Z. Zilic, and K. Radecka. Energy efficient software-based self-test for wireless sensor network nodes. volume 2006, pages 6 pp.–191. IEEE, 2006.