Master Thesis

Design and Implementation of a Fault Emulation Environment for a Java Virtual Machine

Michael Hraschan

Institute for Technical Informatics
Graz University of Technology
Head of the Institute: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Römer

Reviewer: Ass.-Prof. Dipl.-Ing. Dr.techn. Christian Steger

Advisor: Ass.-Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dipl.-Ing. Michael Lackner

Graz, February 2014

Kurzfassung

Java und Java Card Systeme sind oft Ziele von Attacken. Der Hauptgrund dafür sind die oftmals sensiblen Informationen, welche auf der Karte gespeichert sind. Diese Informationen sind meistens Teile von sogenannten Applets, Applikationen, welche in der Java Card Umgebung laufen. Einige dieser Applikationen befinden sich im Bereich von Identifikations- und Bankanwendungen und benötigen deshalb sensible, anwenderspezifische, Informationen. Diese Informationen müssen so gut wie möglich geschützt werden.
Angreifer sollten unter keinen Umständen Zugriff auf diese sensiblen Daten bekommen. Aus diesem Grund wird derzeit in speziellen Forschungsthemen daran gearbeitet, Sicherheitsfeatures zu entwickeln, um genau dies zu verhindern. Diese Sicherheitsfeatures können in Hardware (Security Module, Koprozessoren, ...), Software (Virtual Machine, Operating System (OS), ...) oder einer Kombination aus diesen bestehen. Ein Schwerpunkt der aktuellen Forschungsarbeit basiert auf der Entwicklung von Sicherheitsfeatures zur Vermeidung von sogenannten Fehlerattacken.
Ein Problem, das vor allem Java Card Sicherheitsfeatures betrifft, sind die teils fehlenden Testumgebungen. Sicherheitsfeatures, egal ob in Hardware oder Software entwickelt, müssen auf ihre Funktionsfähigkeit und Funktionstüchtigkeit so einfach wie möglich geprüft werden. Es bestehen bereits diverse Ansätze um Fehler zu simulieren, allerdings setzen diese meist auf niederen Ebenen (Register Transfer Level (RTL)) an. Dies macht es oft schwer, Sicherheitsfeatures auf höheren Ebenen, wie in diesem Fall einer Java Virtual Machine, zu testen. Deshalb werden neue Ansätze benötigt, welche in dieser Arbeit gesucht und entwickelt werden.

Stichwörter: Java, Java Card, Fault, Attack, Fault Attack, Simulation, Virtual Machine, LEON3, FPGA, Co-design, Hardware, Software

Abstract

Java and Java Card systems are often the target of attacks. The main reason for this is that these systems more than often contain sensitive data (or have access to it). This especially holds for Java Card systems which are used in identification and banking environments.
Attackers should not get access to this sensible data by any means. Therefore, current research topics consider the design and development of security features used in Java Card. These security features can either be written in software (Virtual Machine, OS, ...), build into the hardware or are a combination of both. One part of the research focuses on security features to prevent fault injection on Java based systems.
However, only implementation of these security features is not sufficient. These mechanism require an easy and fast way of testing to guarantee their functionality. There are already several approaches on how to simulate fault injections to test such systems. One drawback of these systems is that they do not consider the higher level approach which is used in Java. It is therefore not perfectly suited to test these features. Hence, a new approach is required which is covered in this project.

Keywords: Java, Java Card, Fault, Attack, Fault Attack, Simulation, Virtual Machine, LEON3, FPGA, Co-design, Hardware, Software

STATUTORY DECLARATION

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

....

date (signature)

Danksagung

Diese Diplomarbeit wurde im Studienjahr 2013/2014 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Zuerst möchte ich Ass.-Prof. Dr.techn. Christian Steger, für die Betreuung und Begutachtung dieser Arbeit, danken. Weiters allen Beteiligten des Cocoon Projekts welche diese Arbeit ermöglicht haben. Speziell meinem Betreuer Dipl.-Ing. Michael Lackner, welcher immer Zeit für Diskussionen und Hilfestellungen hatte.

Besonders möchte ich mich bei meiner Familie bedanken, welche mich während des Studiums immer unterstützt hat.

Graz, im Februar 2014 Michael Hraschan

Contents

1 Introduction **10**
1.1 Motivation . . . 11
1.2 Main Goals of this Thesis . . . 12
1.3 Structure of this Work . . . 14

2 Related Work **15**
2.1 Attacks on Java Card . . . 15
2.1.1 Java Type Confusion Attack . . . 16
2.1.2 Combined Attacks . . . 17
2.2 Fault Models and Types . . . 17
2.2.1 Mutant . . . 18
2.2.2 Saboteur . . . 19
2.2.3 Simulator Commands . . . 22
2.3 VFIT - VHDL-Based Fault Injection Tool . . . 22
2.3.1 Fault Injection Techniques . . . 23
2.3.2 Design . . . 23
2.4 MEFISTO-L: VHDL Based Fault Injection Tools . . . 26
2.4.1 Framework . . . 26
2.4.2 Design . . . 26
2.5 MFI - Modular Fault Injector . . . 29
2.5.1 Introduction . . . 29
2.5.2 Design . . . 30
2.6 Similarities to the Java Card Fault Injection Unit . . . 32

3 Design of the Fault Injection Unit **35**
3.1 FIU Design Approaches . . . 36
3.1.1 Software Only FIU (Byte-Code Layer) . . . 36
3.1.2 Software Only FIU (Memory Layer) . . . 37
3.1.3 Hardware-Software Co-designed FIU . . . 38
3.1.4 Hardware Only FIU . . . 39
3.1.5 Final Design . . . 41
3.2 Fault Injection Hardware Unit . . . 43
3.3 FIU Software Module . . . 45
3.4 FIU PC Host . . . 45

4 Implementation of the Fault Injection Unit **47**
4.1 Used Tools 49
4.2 GRLIB LEON3 IP Library 52
4.2.1 LEON3 Processor 53
4.3 SimpleRTJ 54
4.3.1 Structure of SimpleRTJ 54
4.4 FIU Hardware Unit 55
4.4.1 Memory Area Register 56
4.4.2 Configuration Register 56
4.4.3 AHB Bus Controller Unit 57
4.4.4 Fault Trigger Unit 59
4.4.5 Saboteur Unit 61
4.4.6 FIU Assembly 61
4.4.7 Hardware Testing 64
4.4.8 Working Example 64
4.5 FIU Software Module 67
4.5.1 Client Module 68
4.5.2 Communication Interface Module 72
4.5.3 Data Provider Module 74
4.6 FIU PC Host 75
4.6.1 User Interface 76
4.6.2 Communication Interface 78
4.6.3 Fault Campaign Support 79

5 Results **80**
5.1 Platform Setup 80
5.2 VHDL Synthesis Results 82
5.3 Simulation Results 83
5.4 Fault Injection Performance 85
5.5 Attack Scenario 86
5.6 Drawbacks 89

6 Conclusions and Future Work **90**
6.1 Conclusions 90
6.2 Future Work 91

A Code Examples **93**
A.1 Client Socket Communication 93
A.2 Server Socket Communication 94
A.3 FIU Hardware Register Usage 96
A.4 Fault Injection Unit SimpleRTJ Provider 96
A.5 Fault Campaign Sekelton 97
A.6 C Program Testing Fault Injection 98
A.7 Java Program for Fault Testing 99

Bibliography **104**

List of Figures

1.1 Basic Java Card Environment . . . 11
1.2 Project Overview . . . 13

2.1 VHDL-Based Fault Injection Techniques [BGGG05] . . . 18
2.2 Types of Saboteurs. (a) Serial. (b) Parallel [BGGG05] . . . 21
2.3 Possible Fault Models for Fault Injection Systems [GKS+11] . . . 22
2.4 VFIT Block Diagram [BGGG05] . . . 25
2.5 Framework of Testing Fault Tolerance with MEFISTO-L [BPC98] . . . 27
2.6 Structure of MEFISTO-L [BPC98] . . . 27
2.7 Scheme of the Modular Fault Injection System [GKS+11] . . . 30
2.8 Schematic View of the Fault Controller of the Modular Fault Injector (MFI) [GKS+11] . . . 31

3.1 Byte-Code Layer Design (Software (SW) Only) . . . 36
3.2 Memory Layer (SW Only) . . . 37
3.3 Hardware-Software Co-Design . . . 38
3.4 Hardware Only . . . 39
3.5 Schematic View of the Serial Injection on the Bus and Parallel Computation 41
3.6 Schematic View of the Complete Design . . . 42

4.1 Schematic View of the Implementation . . . 48
4.2 Architecture of the Eclipse RCP [ECLa] . . . 50
4.3 Used Tools (Red Rectangles) During the Implementation . . . 51
4.4 Development FPGA Board GR-XC3S-2000 Block Diagram [Gaib] . . . 51
4.5 LEON3 Design for the GR-XC3S-2000 Development Board [Gaib] . . . 52
4.6 LEON3 Core Components [Gai01] . . . 53
4.7 Overview of the Memory Structure of SimpleRTJ [Com] . . . 55
4.8 Schematic View of the Modules in the FIU Hardware Unit . . . 56
4.9 Simple AHB Transfer [ARM] . . . 57
4.10 Internal State Machine of the Bus Controller . . . 59
4.11 Flow Chart of the Internal Trigger Logic . . . 60
4.12 Schematic View of the Serial Implementation of the Hardware Design . . . 62
4.13 APB Configuration Register . . . 63
4.14 Schematic View of the Hardware Test Setup . . . 64
4.15 Schematic View of the Modules in the FIU Software Module . . . 67
4.16 Overview of the Fault Injection Unit (FIU) Integration into SimpleRTJ and the Initialization Routine . . . 70

4.17 Overview of the FIU Command Dispatch Routine and the Execution Evaluation . . . 71
4.18 Command Structure of the FIU Communication Protocol . . . 73
4.19 Dialog for the FIU Adapter Configuration . . . 76
4.20 User Interface of the FIU Host . . . 78

5.1 FIU Setup within the xconfig Tool . . . 81
5.2 Example Workflow of a Fault Setup in the FIU Host . . . 88

List of Tables

2.1 Communication Interface Methods [GKS+11] 21

4.1 APB Memory Ranges 65
4.2 Communication Interface Methods 72

5.1 APB Memory Ranges 81
5.2 Overview of the Space Requirements on a Spartan-3 FPGA for the FIU Units 82
5.3 Space Requirements of Different Hardware Setups with the Overhead to the Original Design 82
5.4 Results of the Different Simulation Scenarios using ModelSim 85
5.5 Results of the Time Measurements for Different Fault Setups using ModelSim and Real Hardware 85

Chapter 1

Introduction

Java Card is a technology used in a variety of security relevant applications. In the beginning, the technology was mainly used in smart cards or integrated circuit cards (ICCs). Today, the technology is also used in mobile phones, embedded in secure elements (SEs), set top boxes and other security critical devices. The main advantage of Java Card based applications is that they can be used in a variety of systems. When a system supports or provides a Java Card Runtime Environment (JCRE), it can install and run Java Card applications, or as they are called in a Java Card environment, applets.

The basic architecture of a Java Card environment is displayed in Figure 1.1. Java Card systems provide the possibility to support multiple applications installed in one environment. This makes it possible, especially in smart cards, to avoid carrying multiple hardware devices. This allows to store, for example, a banking application on the same card as an identification applet. There is a wide range of applications that is available on Java Card platforms. Some use cases for the operation of Java Card smart card applets are:

1. Passports with image and biometric data verification
2. Payment applications like VISA and MasterCard
3. Identification cards with personal information and verification
4. Access control to buildings, computers and others

Depending on the use case, more or less sensible data is stored in this file structure. For example, in a banking application, that may be certificates to validate the authenticity of the card. Not every person, especially harmful attackers, should get access to this sensible data. There are different approaches to protect this data against different access methods.
First of all, the applications themselves protect the data from unwanted access. Not every attacker, who is in possession of a card reader should be capable to retrieve information from the card. Therefore, the applications use protection mechanism to fulfill authenticity, availability and confidentiality.
Another illegal mean to get access to the sensible data on the device is by fault attacks. By using these faults, an attacker can change the behavior of how data is processed inside

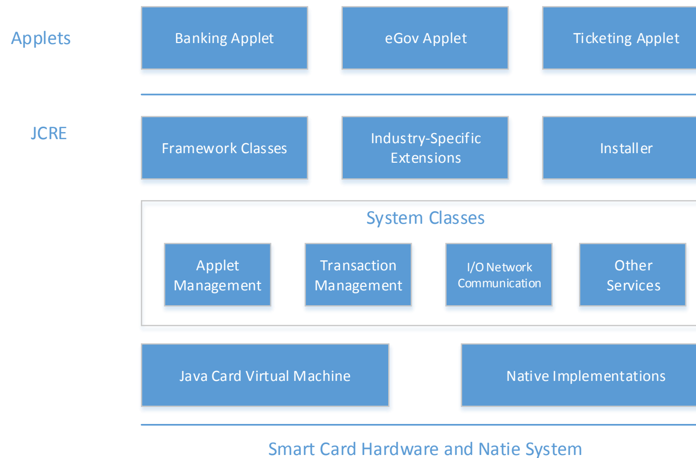


Figure 1.1: Basic Java Card Environment

the card. Furthermore, he can change the control flow and, for example, skip security checks. There are different kinds of faults that a possible attacker could try to get access to the security critical data of the device. The types of faults possible on a Java Card system and how they operate are described later in this work.

1.1 Motivation

Java Card systems can contain sensible information which, in most cases, should not be accessed by anyone, especially not by potential attackers. To avoid this, different security measures are developed to protect this data. Current research topics focus on hardware and software security features, especially at the Institute of Technical Informatics at the Technical University of Graz [ITI].

The main goal of new security features is to protect the data not only by software means, but also design and create hardware security modules. Software security units sometimes lack the possibilities of hardware units, especially when it comes to fault attacks. While software security units can suffer the same drawbacks regarding fault attacks as the software they are added to (e.g. instruction skipping), this is far more complex to be used with hardware security units. Software units also face the disadvantage of adding an additional overhead to the execution time. Hardware units, when designed properly, can be executed in parallel to the software and require a smaller overhead than a software equivalent. Additionally, they can get access to information that software units do not have access to.

In a previous work [LBL+13], such hardware security units were implemented and tested in a smart card prototype. This prototype implemented several security features as described in [LBH+13]. The security features were especially designed to protect Java Card systems against fault attacks and implemented by four protection units: An integrity protection unit, a bound protection unit, a control flow and a type protection unit. The integrity

protection unit is used to detect faults injected on a bus between the processor and the memory. The bound protection unit prevents memory access outside of defined memory ranges. The control flow protection unit checks if the current fetched byte-codes are inside the code area.
All processor systems require a main memory module to store interim results and data. This is also true for Java Card systems. More complex systems could have a bus system which connects several components to each other, for example the processor with a memory controller. This memory controller could be responsible for the memory access of the system. When a Java Card virtual machine (VM) accesses a certain memory area, the access is routed through the bus. An attacker could try to introduce faults on this bus system, to change the data which is read and written into the memory. Most attack scenarios introduced in Java Card systems focus on fault injection into the local variable, operand stack and byte-code area of the memory.
A problem that occurs when designing and implementing hardware units is the testability. While there are a lot of tools to help test software units, starting from simulators to debuggers for real hardware, it is hard to test hardware units. Especially when it comes to fault protection units.
Faults are not part of the original system design and therefore need to be somehow simulated in the design. To do this fault simulation, it is often required to get a deep knowledge of the underlying system. Getting this knowledge is in most cases a very time consuming task. Another option would be to test the system against real faults. The drawback with this approach is that real faults are hard to introduce, require a lot of setup time and expensive equipment. Especially when it comes to precise faults, a real setup is very complex and hard to perform.
There are several existing approaches to simulate fault injections. The problem with these approaches is that they attach themselves on the RTL level. These approaches are set up by specifying a number of clock cycles from the start until the fault is injected. These fault simulators are hard to use when it comes to security features which work on a higher level. For example, the bound protection unit described in [LBL+13] checks if accessed data is inside the range of reserved VM memory. To easily test this security feature, it would require a fault setup where a fault condition could be provided, when the program accesses a specific memory range. Meaning, when the program accesses memory location A, located in memory range B, a fault should be introduced. However, this approach is not possible with the existing fault simulation models.
Therefore, a new fault injection approach is required, which allows to specify advanced fault conditions. This would allow the user of the fault simulation to specify fault trigger conditions other than a specific point in time dependent on the cycles performed in the system. This approach should focus on parameters of the executed program, in case of this work, the memory ranges accessed by a VM. When one of the specified memory ranges is accessed, a fault should be introduced into the system.

1.2 Main Goals of this Thesis

All the previously described requirements should be fulfilled in the finished design and implementation of this work. Therefore, the result of this project should be an easy to

use fault injection framework. It should be possible, to use the designed and implemented features in existing projects without huge implementation efforts. Additionally, different design approaches should be evaluated. All of these approaches should be considered and evaluated against different criteria. These criteria are defined beforehand. The configuration of faults should be easily understandable and fast to use. The design should consider a modular and reusable approach, so that existing modules can easily be reused, modified and integrated into existing projects.
Additional hardware units, required by the design, have to be synthesizable on a field-programmable gate array (FPGA) development board. As defined before, hardware fault injection units should be designed and implemented in a modular way. Another goal for the hardware fault injection unit is that it has to work with zero-delay. This means that every computation, and the fault injection itself, have to be performed in one cycle. This is required so that existing hardware designs are influenced as less as possible.
The finished fault injection design should provide several fault modes which are described in Chapter 2. This includes random and precise faults, as well as a library of fault models. Additionally, multi-bit faults have to be supported. Multiple bits have to be affected by a single introduced fault.
The final design should at least consist of two components: An FPGA board, consisting of the processor and other hardware components representing the embedded device, including all fault injection relevant modules, and a host tool, to setup the fault configurations. This host tool should parse Java files and allow the user to easily inject faults by choosing the fault position in the source code. Then the user can specify in which memory area this fault can be introduced. An overview of the project structure is given in Figure 1.2.

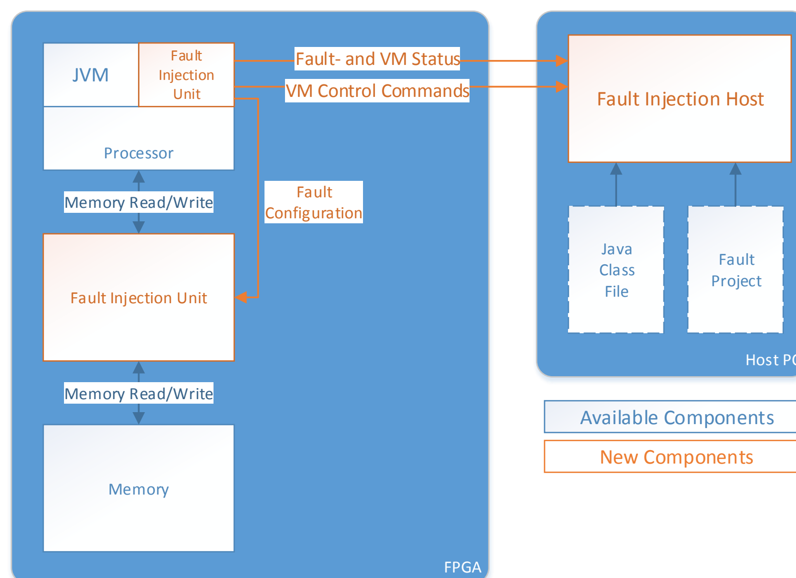


Figure 1.2: Project Overview

1.3 Structure of this Work

In Chapter 2 the related work to this project is described. It contains information on current research in the field of fault emulation and current appropriate fault models. Additionally, the related work chapter gives an overview on state of the art fault attacks against the Java Card system. Furthermore, it describes a modular fault injector which is implemented at the RTL layer. Chapter 3 presents the design of the project and all the separate modules. It shows how the hardware module, VM and host is designed and how they work. The implementation is presented in Chapter 4. It describes some of the specific implementation details of all the modules and an evaluation design on a LEON3 development board. The results of the work and the respective overheads for each module are shown in Chapter 5. In Chapter 6 a conclusion of this thesis and possible future work are presented.

Chapter 2

Related Work

The related work of this thesis mainly focuses on three parts: State of the art attacks on Java Card, fault types and methods used in related work, and already implemented fault tools. Attacks on Java Card are described to understand how attacks affect a Java Card system. This is required, so the possible fault options in a fault injection system can be described.
Fault methods and types are analyzed to evaluate which of these methods should be implemented for a VM based fault injection system. These fault types are compared to each other and brought into implementation.
Finally, already existing fault injection tools are examined. The evaluation of these tools will help during the design phase of the new fault injection tool.

2.1 Attacks on Java Card

The FIU proposed in this thesis, focuses on the simulation of fault injections. Fault attacks are basically done by attackers which are trying to get access to a system. This chapter focuses on analyzing which faults on Java Card exist and on the general nature of fault attacks.
One major security drawback of the Java Card architecture is that programs can be stored in writable memory (e.g. Electrically Erasable Programmable Read-Only Memory (EEPROM) or Flash). This writable memory is more vulnerable against fault attacks than read-only memory (ROM). When a potential attacker has knowledge of the position of the memory and knows how to attack it, he may be able to change data, stored in the memory. There are several attacks which were successfully performed on Java Card which resulted in a behavioral change of the program. Some of these attacks are [Ver06]:

1. **Physical Attacks**: Classified as an active side-channel attack. Uses tampering of the underlying hardware. Possible attacks are micro probing on the system bus or using optical methods to read the ROM or similar means.

2. **Fault Attacks**: Classified as an active side-channel. Injects physical faults into the system to tamper the environment and therefore change the programs behavior. Fault attacks can have a variety of causes, such as supply voltage variation in ranges that are not supported by the hardware, temperature variation out of the operation

scope, external radiation, clock manipulation or any other mean to change the operating conditions. To efficiently perform fault attacks, these attacks have to be setup so that they can be performed multiple times with the same operating conditions.

3. **Observation Attacks**: Classified as a passive side-channel attack. These kind of attacks observe different operating conditions which are then used to get information on the program flow or the hardware. A possible test setup could measure the different power consumptions of various input data provided to a system. Depending on the power measurements, the attacker could trace the data back to program behavior. Another channel that could be exploited, is the timing behavior of the hardware.

4. **Malicious Code**: This attack method uses possible bugs or malfunctions of the executing hardware. The attacker introduces malicious code into the system which is then executed by the hardware. If the hard- or software is not working properly, certain effects could occur during execution of the program that are not intended. These kind of attacks are also used efficiently on Java Card systems where malicious applets are installed and executed on a Java Card environment.

The attack methods described above are ordered by the degree of invasiveness: the physical attack as the most invasive and the malicious code insertion with the least invasive attack possibilities. In most attack scenarios, these attacks are combined to perform even more powerful attacks and to achieve the desired goals. In the referenced papers [Ver06], these attacks are also referred as „combined attacks“ which will be described later. Combined attacks are also often used to attack Java Card systems.

2.1.1 Java Type Confusion Attack

Type confusion is an attack scenario for Java Card systems. Type confusion means that an application which is executed in the VM, tries to store or read data with an illegal type that is not valid. A Java Card applet may try to store an object reference as an integer. When this scenario occurs, the application illegally gets the memory location where an object is stored. In general, a VM should provide protection mechanism to protect the application against type confusion attacks. Attack scenarios which use this type confusion approach, are typically used in combination with malicious code injections and fault attacks to get access to the system.
Java Card executes the installed applets in a sandboxed environment. This means that every application got its own address space and the VM prohibits applications to access memory ranges of other applications in the system. The sandboxed execution should also prohibit the applications from accessing illegal memory areas which should not be accessed by the applet.
Another problem that most Java Card environments face, is the static-only validation of the applications. This static validation can either happen on-card or off-card. When performed on-card, the application is validated at install time of the applet. The off-card verification is performed in a secure and authorized environment outside the card. After a successful off-card verification, the applet is signed with a secret key. This signature is checked during the installation process to guarantee a good-natured applet. The problem

with the on- or off-card approach is when an attacker changes the program during execution, this protection mechanisms do not work. A possible attack scenario which uses this drawback could be the attacker somehow modifying the code, after the validation has been performed. This would give him the opportunity to break the sandbox of the executed applet. One attack that could be used to break this sandboxed environment, is a buffer overflow attack. Buffer overflows are described in more details in [LBL+13], [Ver06] or [BICL11].

2.1.2 Combined Attacks

Combined attacks use multiple attack approaches at once to get unintended access to the system [BTG10]. As described above, one combination of a successful attack, is to use physical attacks and malicious code. An example of this approach is given in [Ver06]. In this example, the possible attacker has, in a first instance, written valid Java code which was also verified by the Java byte-code verifier. This creates an installable file which can then be installed in a Java Card environment. However, this program is written so that a simple memory error leads to an error in the Java type system. This error is then introduced by physical faults and allows the program to break out of the sandbox. When the program is not sandboxed anymore, it can start to execute arbitrary code.
A concrete example using this approach is given in [Ver06]. The basic idea behind the attack scenario is:

1. Create the valid program. This program has to be written so that simple changes in the byte-code could lead to a type confusion attack. However, the original program has to be valid and verified so it can be installed into the system.

2. Perform a power consumption analysis to see when the critical instructions are executed.

3. Use an fault attack to skip the instructions. This skipping allows the program to break the sandbox.

2.2 Fault Models and Types

To design and implement a fault injection unit for Java systems, existing fault injection systems were analyzed. In addition to this task, the possibilities on how to inject faults and what type of faults exist were studied. Figure 2.1 gives an overview of Very High Speed Integrated Circuit Hardware Description Language (VHDL) based fault injection techniques. A fault injection system can be categorized into two types: either they use *saboteurs* to inject faults, or use *mutants* to introduce errors into the system. Another approach, which cannot be implemented directly in hardware, but is used in several VHDL fault injectors, are *simulator commands*.

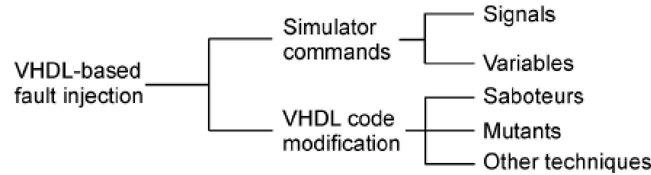


Figure 2.1: VHDL-Based Fault Injection Techniques [BGGG05]

2.2.1 Mutant

Mutants are the first form to inject faults into a system. Mutants are manipulated logic elements or modified submodules. Every hardware design, normally consists of several modules which interact with each other. Each of these submodules have tasks to perform and several input and outputs. When the behavior of one of these submodules is changed, and in response faulty values are generated, they are defined as mutants.
In normal operating conditions, mutants should not affect the system behavior. When they are activated, they should behave like a faulty variant of the original. For mutants to work, they have to be replaced by its mutant submodule. An example for this type of fault system could be a memory controller which returns invalid data on specific memory accesses. A mutant can be generated in three ways [BGB+08]:

1. Adding saboteurs to the structural model description
2. Modifying structural descriptions by replacing sub-components (e.g. exchange a NOR gate by a NAND gate)
3. Modifying syntactical structures of behavioral descriptions

Normally, a model can contain a lot of possible mutations. Therefore, there has to be a subset of representative faults considered at the RTL layer. Possible fault models that can be used in accordance with mutants are:

1. **Stuck-Then**: If condition is changed to be always true.
2. **Stuck-Else**: If condition is changed to be always false.
3. **Assignment Control**: Changing or disturbing the assignment of a signal/value/-variable.
4. **Dead Process**: Elimination of the sensibility list in a module/process.
5. **Dead Clause**: Elimination of a clause in a case.
6. **Micro-Operation**: Changing or disturbing an operator.
7. **Local Stuck-Data**: Changing or disturbing the value of a variable/constant/signal in an expression.
8. **Global Stuck-Data**: Elimination of all modifications of a variable/signal in an architecture.

Some of these faults do not represent a physical fault. However, they can be used to generate a somewhat erroneous behavior in a submodule.

Mutant Generation

Fault injection using mutants is more complex than other injection techniques such as saboteurs. This is due to the spatial overhead required by the generation of mutants [BGGG05], [GGBG03].
One approach for automatic, mutant-based fault injection is the generation of multiple copies of the modules used in the architecture. Each of the copies had a modification (or mutation) in the behavioral code [GGBG03]. The mutant modules were created using a pre-defined configuration. This approach allowed the selection of the mutated model which was created. All the generated mutants used a static approach, meaning that all faults were permanent and started from the beginning of the execution of the program.
Another approach which was developed, had the goal to fix the problem with the static nature of the mutants [GGBG03]. This dynamic approach uses guarded signals in addition to the configuration mechanism. This makes it possible to stop the execution of the original model, used in the architecture, and use a faulted model instead. The approach used simulation commands to save the status of the simulation, save it to a configuration file, and then use this setup to continue the simulation in another model. With this dynamic approach, permanent, transient and intermittent faults can be used. However, this implementation has some drawbacks [BGGG05]: The costs required to store the status of the simulation were enormous. This simulation style was 100 times slower than using other approaches due to the synchronization required between simulations [BGGG00].
A third approach towards mutant generation was proposed in [BGGG05]. This approach gets rid of the synchronization issues by a brute force implementation. For this approach, a mutated version of each model in the architecture is created. Therefore, a mutant is generated for each possibility, defined in the configuration. The modifications mainly include *if* and *case* statements, but other possibilities are described. The goal of this approach is to allow the selection of the wrong and correct statements. For this to work, a new signal is used which selects a particular mutated version. This new approach reduces the temporal overhead, required by the simulation. Additionally, the overall size requirements are shrunk, since only one architectural modified version is required which contains all modifications.

Automatic Mutant Generation

From the mutant generation procedures described above, only the last one is feasible to be used with automatic systems. To perform this, a potential tool only requires a VHDL parser. This parser has to find the statements that should be modified, replace them with new ones and add the required selection signals [BGGG05], [BGB^{+}08].

2.2.2 Saboteur

Saboteurs are the other form of how faults can be injected into a system. They are small circuit elements which change the system behavior. When not active, saboteurs should not change the behavior of the system. When activated, they can disturb internal signals

or directly inject faults into a submodule. In contrast to mutants, saboteurs do not require the submodules to be changed. This has the advantage that the function of the submodule does not need to be known. A saboteur is placed between two modules and changes the signals depending on the trigger state of the saboteur.
In general, saboteurs can be differentiated between serial simple, serial complex and parallel insertion [GGBG03], [BGGG05], [BGB+08]. This depends on how the saboteurs are inserted into the system. Another differentiation criterion for saboteurs is if they are either uni- or bidirectional. Unidirectional saboteurs work only one-way, while bidirectional saboteurs can work in read and write mode. A last classification that can be performed for saboteurs are if they either work on a single signal (also referenced to as single-bit) or multiple signals (also referenced to as multi-bit). The difference with single-bit saboteurs is that each saboteur only affects one signal/bit at once. When multiple bits need to be triggered, multiple saboteurs have to be added to the system. In contrast, multi-bit saboteurs can change multiple signals/bits at one time. With this approach, bus attacks can be simulated. With these types of classification, the following saboteurs can be placed in a system [GGBG03]:

1. **Serial Simple Saboteur** - Placed between a source and a sink. Intercepts the signal and modifies the value.

2. **Serial Simple Bi-Directional Saboteur** - Contains two input and output signal. Also requires a read/write input that defines the fault direction.

3. **Serial Complex Saboteur** - Interrupts the connection of two outputs and their inputs, modifying the received value.

4. **Serial Complex Bi-Directional Saboteur** - Consists of four input and output signals and an additional signal to define the fault direction.

5. **n-Bit Unidirectional Simple Saboteur** - Used for unidirectional buses of n bits. Consists of n serial simple saboteurs.

6. **n-Bit Bi-Directional Simple Saboteur** - Used for bi-directional buses of n bits. Consists of n bi-directional simple saboteurs.

7. **n-Bit Unidirectional Complex Saboteur** - Used for unidirectional buses of n bits. Consists of $n/2$ serial complex saboteurs.

8. **n-Bit Bi-Directional Complex Saboteur** - Used for bi-directional buses of n bits. Consists of $n/2$ bi-directional complex saboteurs.

A serial saboteur is placed between the source and the sink of a module, whereas a parallel saboteur is used as an additional source of a given signal. This behavior of serial and parallel saboteurs is illustrated in Figure 2.2.
The internal architecture of saboteur units can either be behavioral or structural. Behavioral saboteur designs use a process with a sensitivity list and defined input/output signals. A structural approach uses multiplexers for the internal logic [GGBG03].

Parallel saboteurs have two major drawbacks compared to serial saboteurs: First, they are harder to implement. This is because on the one side, the resolution function of the

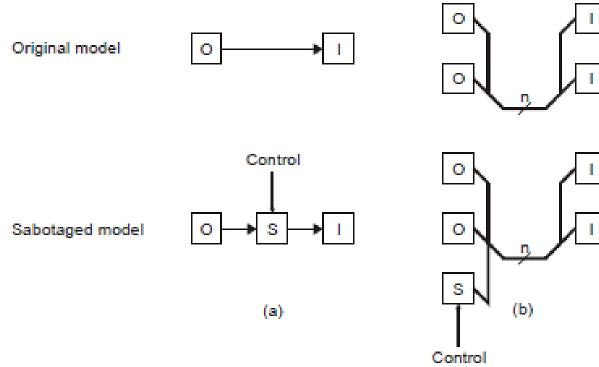


Figure 2.2: Types of Saboteurs. (a) Serial. (b) Parallel [BGGG05]

Saboteur Mode	**Fault Type**	**Description**
Stuck-at-Zero	Permanent	Signal value of '0' until reload
Stuck-at-One	Permanent	Signal value of '1' until reload
Indetermination	Permanent	Undefined signal state until reload
Bridging fault	Permanent	No output propagation until reload
Negation of input	Permanent	Undefined signal state until reload
Bit flip	Transient	Output inverts input for one cycle
Artificial delay	Transient	Input to output propagation delay

Table 2.1: Communication Interface Methods [GKS+11]

signal has to be changed, and on the other side it is required to modify the data type of the signal. Therefore, serial saboteurs are normally preferred.
Similar to the mutants, saboteurs can work in different fault modes. Each of the fault modes have different effects on the signal the saboteur is attached to. An overview of possible fault modes for saboteurs is given in Table 2.1. Figure 2.3 shows what effect these fault modes have on the faulty signal. The first four modes in the table are equal to direct circuit modifications. The bit-flip mode could be forced by short, intensive pulses. Delayed faults can be used to simulate the behavior of operating voltage changes.

Automatic Saboteur Generation

Normally saboteurs have to be placed between a source and a sink in the system. Therefore, system knowledge is required. However, there are approaches described which automatically insert saboteurs into the system [BGGG05], [BGB+08], [GKS+11].
For the automatic placement to work, it is needed to parse the HDL code. This parsing process analyzes the architecture of the model and generates a tree structure. This tree structure is then used to place the saboteurs. For this to work, three steps are required:

1. Declaration of the signals, where required to activate the saboteurs. Additionally, the fault mode needs to be selected.

2. Declaration of the components of the saboteurs.

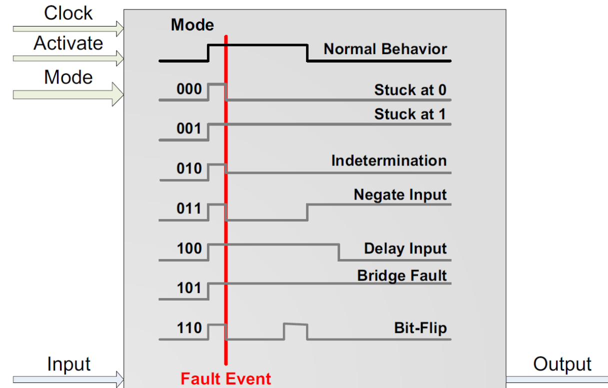


Figure 2.3: Possible Fault Models for Fault Injection Systems [GKS+11]

3. Insertion of the saboteurs between local and formal ports of the used components. This may require new signals which are required to connect the saboteur to the local ports and re-mapping of these signals.

2.2.3 Simulator Commands

A third approach towards fault injections in VHDL models are simulator commands. This fault injection technique uses features provided by the underlying simulator to test faults during simulation. An advantage of this approach is that the existing VHDL code does not need to be altered. The simulator commands are used to change signals and values during simulation time. However, this approach cannot be used in hardware and therefore cannot be used in real hardware setups. Simulator commands allow to perform transient, intermittent and permanent faults [BGGG05].
To generate a transient fault using this technique, the following pseudo-commands need to be performed [GGBG03], [IR86]:

1. **Simulate_Until** *[injection instant]*
2. **Modify Signal** *[name] [faulty value]*
3. **Simulate_For** *[fault duration]*
4. **Restore_Signal** *[name]*
5. **Simulate_For** *[observation time]*

For permanent faults, the same steps as above are required, except that steps 3 and 4 have to be omitted.

2.3 VFIT - VHDL-Based Fault Injection Tool

VFIT stands for VHDL-based Fault Injection Tool and is a fault framework which places saboteurs and mutants during the design phase. The placed saboteurs and mutants are

then later used to inject faults into the system. The tool was developed by the Fault-Tolerance System Research Group (GSTF) of the Technical University of Valencia.
The goal of the tool is to detect possible fault errors in the system as soon as possible. Therefore, three categories for fault injection techniques are described: physical, software implemented and simulation-based [BGGG05]. Simulation based fault injection uses commands of the simulator to change the behavior of the simulation. This behavior change can be the modification of values or timing of signals and variables. Simulator command fault injection can be used to detect faults in a very early design state. Other techniques which were used by VFIT alter the original VHDL code of the model. VFIT implements all fault injection techniques which are illustrated in Figure 2.1 except the *other techniques* branch. These techniques extend the VHDL language by adding new data types and signals. Another method uses modification of VHDL resolution functions. These new data types and signals include the fault injection behavior. These techniques are not included in VFIT, due to their huge complexity. They require the development of ad hoc compilers and simulators as well as the introduction of control algorithms to manage the language extensions [BGB+08].

2.3.1 Fault Injection Techniques

VFIT uses three basic techniques for fault simulation: saboteurs, mutants and simulator commands. Since saboteurs and mutants were already described in Section 2.2, this section will focus on the simulation command technique of VFIT. An overview of the design of VFIT is given in Figure 2.4.
VFIT uses simulation commands, provided by the used simulator at different simulation times. These commands are used to change the value or timing of signals and variables in the model. This technique can be used to perform non-usual fault models, for example delay faults. Additionally, simulation command based faults allow to inject transient, permanent and intermittent faults. However, variables in VHDL cannot be changed permanent due to the nature of VHDL. This simulation technique is the simplest to implement and does not need to change the hardware model. One drawback is that this fault injection technique does not work in real environments.

2.3.2 Design

The main features that were defined for VFIT included:

- Model-Independent
- Build around ModelSim. ModelSim allows to control the simulation using Tcl commands. These can be used to inject faults.
- Automatic fault injection using simulator commands. Additionally, saboteur and mutant faults should be injected. However, this approach requires user interaction for placing them.
- Permanent, transient and intermittent faults.

- Different fault models including stuck-at (permanent), bit-flip (transient) and delayed and pulse (transient) fault models.

The fault injections for VFIT requires three stages and consists of five main elements. The three stages are:

- **Set-up**: Specification of the parameters for the experiment. These parameters include model relevant parameters like workload file and workload duration. Parameters specific to the fault injection as injection technique, number of faults, target duration and others. The last parameters to specify are used to define which analysis to perform. This can either be error syndrome or the validation of a Fault-Tolerant-System.
- **Simulation**: Here the fault simulation is performed. It consists of a golden run to generate the reference values and some faulty runs which are compared to the golden run.
- **Analysis**: At this stage all faulty runs are compared to the golden run. Depending on the analysis type in the first stage, different measures are performed.

The main elements of the fault injection tool are:

- **Tool Configuration**: This module is used to set up the tool and simulator parameters
- **Graphic Interface**: A utility that helps the user to configure the injection campaign. It allows the user to specify the fault points using a tree of the used model.
- **Injection Manager**: Uses the data, created by the configuration tool and the graphic interface to perform the fault injections. First, a golden run is performed to get reference parameters from the model. Then, the faulty runs are performed and the generated data are traced.
- **VHDL Simulator**: Typically the used simulator. In the evaluation design proposed in the paper, ModelSim was used.
- **Result Analyzer**: This tool compares the data created by the golden run with the data created by the faulty traces searching for any mismatches.

Enhanced Fault Models

VFIT introduces new fault models for the saboteur implementation. Traditional fault models for saboteurs were introduced at the beginning of this chapter. The newly introduces fault models are:

- Unidirectional Serial Saboteur - Same structure as the *Serial Simple Saboteur*, but allows new fault models to be injected.
- Bi-Directional Serial Saboteur - Similar to the Serial *Simple Bi-Directional Saboteur*, but as before, allows new fault models. Additionally, the Read/Write control signal is removed.

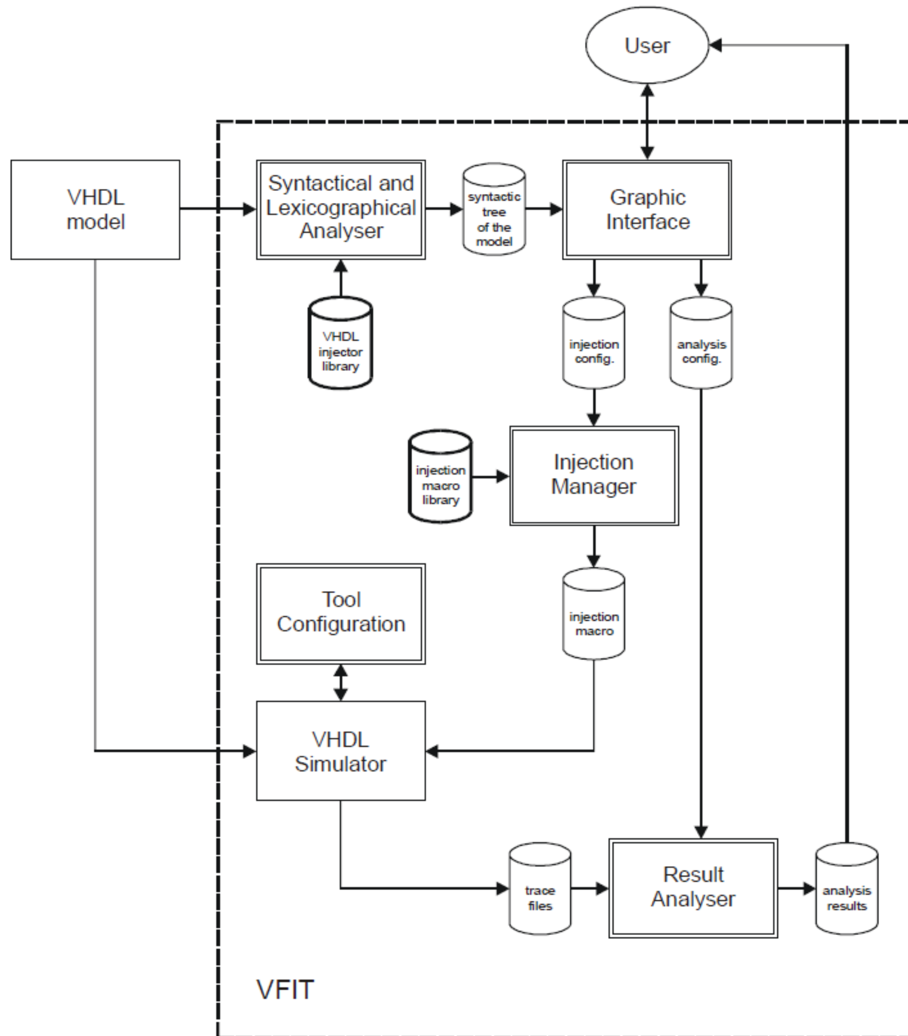


Figure 2.4: VFIT Block Diagram [BGGG05]

- n-Bit Unidirectional Serial Saboteur - Replaces all unidirectional multi-bit models.
- n-Bit Bi-Directional Serial Saboteur - Replaces all bi-directional multi-bit models and removes the Read/Write control signal.

These models were introduced to extend the previously defined models. The new models can all be implemented using behavioral description. This simplifies the usage and the code of a design. Additionally, the n-Bit versions of the saboteurs can be implemented using generic parameters. Another advantage is that the overall number of saboteurs required with these models can be reduced. The new fault models mentioned above that can be used with these new saboteurs are pulse, short and bridging.

2.4 MEFISTO-L: VHDL Based Fault Injection Tools

The first approach towards a VHDL based fault injector was described in the paper for *MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance*. The paper proposed an approach for testing Fault Tolerance Mechanism (FTM). The main features of MEFISTO-L are the embedded fault code analyzer, the observation and injection mechanism, their synchronization and their placement in the target VHDL model [BPC98].
The main goal of this tool is to test the fault tolerance by introducing faults into a system in an early design stage. Most systems are evaluated against their fault tolerance. The fault tolerance is a measurement unit that shows how robust and how well the system reacts to introduced faults. MEFISTO-L was the first step towards the problem of missing tools for efficient fault tolerance coverage. The main goals that should be addressed by this work were [BPC98]:

- Designing and developing means to determine the fault activity sets which are applied in a fault injection campaign.
- Perform efficient fault injection experiments. The goal is to create a minimal set of input patterns that target all deficiencies in the FTM.
- Implement a tool to support this design process, MEFISTO-L.

2.4.1 Framework

The objective of simulated fault injections, is to evaluate the fault tolerance of a system as soon as possible in the design stage. To assure this, the designed framework is integrated into the system design activities. With this approach, the fault tolerance validation can be done early in the development process. Figure 2.5 shows the framework proposed to test the fault tolerance of a system included in the development process. For the test of FTM, two main steps are required: Test pattern generation and test diagnosis.

2.4.2 Design

The main tool which is proposed in this work consists of three blocks. The structure of these blocks can be seen in Figure 2.6. The three blocks are [BPC98]:

- **Parsing Block** - Extracts the data from the VHDL code required for the injection campaign of the targeted model.
- **Injection Block** - Specification of the campaign. Used for the generation of the mutant model.
- **Result Extraction Block** - Uses the traces of the simulation of the mutant model to generate results.

After the campaign is set up, the experimental model is created out of it. The VHDL design is enhanced with saboteur and probe components and a test hierarchy is built. This hierarchy contains of a test bench at the top which controls the saboteurs. Then the

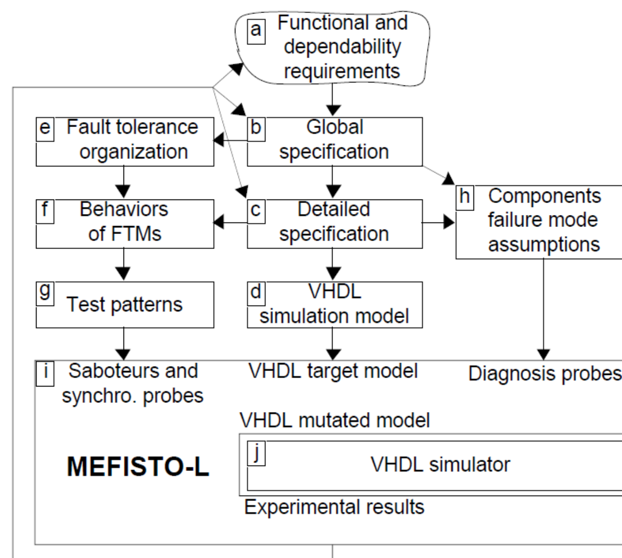


Figure 2.5: Framework of Testing Fault Tolerance with MEFISTO-L [BPC98]

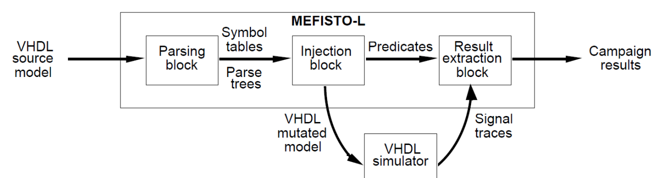


Figure 2.6: Structure of MEFISTO-L [BPC98]

created VHDL code can be used in a simulator to simulate the experiment. When the simulation is finished, the result extraction block evaluates the traces generated by the simulated experiment.
The injection block is the main part of the model. It uses the data generated by the parsing block to automatically generate the mutated model. There are six objectives which are performed by the injection block [BPC98].

Description of the Source Model

Here the source model is analyzed and presented to the user, who can select the target signals from the model hierarchy. Each level of this hierarchy can be described as *Inputs, Outputs and Bidirectionals* or *Described Signals*.

Construction of the Target Signals

The user can select a list of target signals from the hierarchy of the source model. Each of the selected signals is added to the list of targets for the injection.

Saboteur Placement

With this objective, the MEFISTO-L tool automatically generates the saboteur model.

Fault Model to Saboteur Assignment

When all target signals have been selected and the saboteurs have been placed, each saboteur has to be assigned to a specific fault model. MEFISTO-L therefore provides a library of saboteur units.

Probe Specification

Additionally, to the selected saboteurs which are placed in the system, probes are used to specify observation points in the model. These observation points are used to collect data which are required for the evaluation of the faults. One feature of the MEFISTO-L tool is that the information collected by the probes can be used for the injection control and therefore can be used to dynamically control the activation of the saboteurs.

Control Specification of the Saboteurs

At the top level module of the hierarchy, the control of the selected saboteurs is specified. This specification consists of logical expressions on predefined parameters. These parameters can be [BPC98]:

1. Probe values placed anywhere in the model hierarchy.
2. Results of comparisons which use data from the probes as input.
3. A static clock probe, consisting of either the rising or falling edge.

These parameters allow the user of the system to active faults static or dynamic, where each fault can either be permanent or transient.

Definition of Predicates

During this last stage, observation conditions are specified which are used for the generation of the results for the fault injection campaign. These conditions are defined as *Boolean*, using the status of the mutated model.

2.5 MFI - Modular Fault Injector

Similar to the MEFISTO-L fault injector, [GKS+11] describes another approach towards a VHDL based fault injection model. This approach was developed at the Technical University of Graz at the Institute of Technical Informatics. This fault injection approach also focuses on the problems which arise with the increasing complexity of today's hardware models and their fault robustness. The difference of this approach towards previously described designs is that this approach does not use single bit injections, but rather focuses on faults on a wider range. The paper introduces a new fault injection strategy for test pattern injection. As a second step, the generated fault structure is abstracted to a more generic, higher level approach.

2.5.1 Introduction

The approach discussed in this paper targets the increasing complexity of circuits. Another problem that modern circuits face is that circuits get smaller and smaller and get more and more sensitive to external influences like radiation and thermal and electrical degradation. This can lead to faults, which can happen at any time of the program flow. Also they can be transient or even permanent. This can lead to a change of the behavior of the system and lead to security breaches. This can allow an attacker to get access to parts of the system he is not intended to.
Therefore, recent research topics focused on simulation and emulation of faults in a system. The target platform of these systems were often FPGAs, due to their flexibility. A fault injection system which uses this approach, and is also discussed in this paper, is shown in Figure 2.7. Different ways are proposed to perform fault injection. One approach uses the reconfiguration features of the used FPGA. However, since this feature is not supported on all platforms, this heavily limits the choices of the platform. Another way described is to use saboteurs or mutants, as described in Section 2.2.
To support a wide range of possible fault setups, it is necessary to fulfill several design goals. First of all, a standardized test interface is required. Secondly a wide selection of fault models have to be provided. Several fault models were already described in the sections before. In this project, two separate approaches toward introduced faults are inspected. Faults that occur due to radiation or degradation can be simulated by single random faults. However, since the main focus is on security relevant faults, these can happen at multiple locations at once. Therefore, a multi-bit fault model is required.
For an easy setup, this paper proposes a controlling element which is executed on a personal computer. This element uses a simple communication protocol that can be used to set up the fault system.

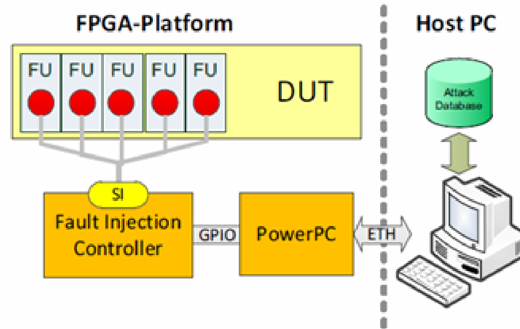


Figure 2.7: Scheme of the Modular Fault Injection System [GKS+11]

The main goal of this paper and the modular fault injection approach were [GKS+11]:

- Fully modular design
- Multi-bit injection
- Online-testing support

2.5.2 Design

For the modular fault injector, several goals were defined for the design. One goal was to support, in contrast to the MEFISTO-L [BPC98], the multi-bit fault injection. This allows fault patterns to change several bits in one configured fault. Additionally, the design supports multi-mode saboteurs to insert multiple different saboteurs in the system. Another goal of the project was its standardized communication interface to allow simple communication with the fault injection unit. This interface uses a General Purpose Input/Output (GPIO) port. Other goals were scalability for automatic placement of saboteurs and an internal storage to store fault configurations. The scalability is required, since extensive fault injection campaigns often require a large amount of saboteur units. Following a description of the components used in the MFI.

Fault Injection Controller

The fault injection controller is responsible for activating and setting the mode of the configured saboteurs. It is the main part of the MFI and consists of two interfaces. One of the interfaces is the previously mentioned GPIO interface. This interface is used to set up the fault controller of the MFI by writing test patterns into the memory of the controller. This memory is then used to control the activation and mode of the saboteur units. The other interface is used to connect all saboteurs in the system to the controller. Every saboteur in the unit has its own active signal in the design. An overview of the schematic view of this design is given in Figure 2.8.
The GPIO interface can be used to create automated fault injection campaigns. Since this interface is easy to use, fault campaigns can easily be set up. The proposed evaluation

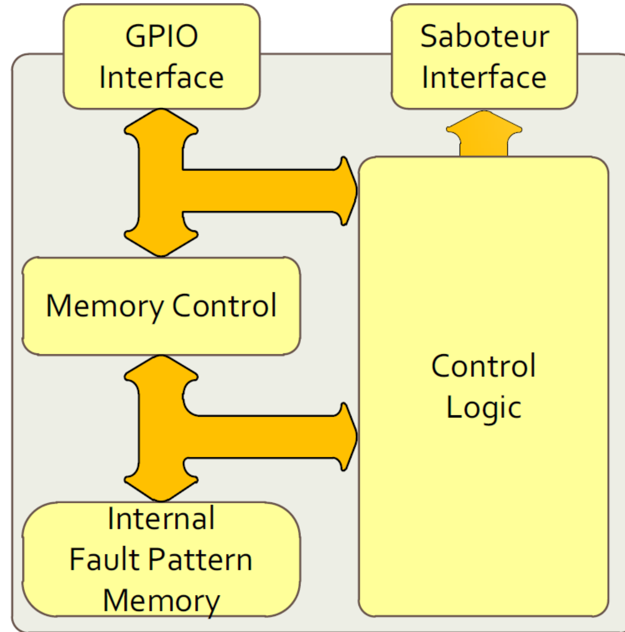


Figure 2.8: Schematic View of the Fault Controller of the MFI [GKS+11]

design of this paper uses a PowerPC which is integrated into the FPGA board to configure the fault campaigns. The advantage of this approach is that the test engineer does not need to know the fault system itself.
The main parts of the fault controller are the interface to the saboteur units, the GPIO interface for the fault campaign setup, a memory controller which writes the fault setups into the internal memory, the memory itself and the control logic which is responsible for the activation of the saboteurs.

Saboteurs

The saboteurs used in this paper were designed as described in Section 2.2. To be more precise, the saboteur units used were unidirectional, serial simple saboteurs. Therefore, the saboteur only affects a single signal and works only in one direction. The available fault models in such a design are shown in Table 2.1.

Fault Pattern Support

Fault patterns are used to map the saboteurs to their fault locations. In most cases, not every fault pattern has to be tested in the fault campaign. The fault patterns have to be defined beforehand and should cover most of the hardware. In the MFI, the fault patterns are created by a random number generator. This approach is used for demonstration purposes only, different to the fault pattern generation described in Section 2.4.

Automatic Saboteurs

As described before, the MFI uses automatic saboteur placement to avoid the time consuming manual placement. The placement works similar to the automatic placement described in Section 2.3. To place the saboteurs in an existing design, the vMAGIC VHDL parser library [vMA] was used. This allows to parse the code and automatically insert saboteur units. These saboteurs are then automatically linked to the fault injection controller so they can be connected to the test patterns.

Attack Scenarios

Additionally, a possible attack scenario was described in the paper. This setup was also used as inspiration for a possible attack scenario in the Java fault injection unit. The steps of the scenario are:

1. Run Golden Model - A golden model is executed. This is used to get the information on the Device Under Test (DUT) and the expected behavior.

2. Store Information - The information, generated by the golden model is stored for later comparison.

3. Reset DUT - Reset DUT to start with plane model.

4. Load Test Pattern - Test patterns for the current campaign are loaded into the memory.

5. Run DUT to Attack - Start execution of the DUT. The execution is halted when a saboteur unit has to be activated.

6. Activate Saboteurs - Depending on the fault patterns the corresponding saboteur units are activated.

7. Run DUT to End - DUT is continued until the execution is finished.

8. Deactivate Saboteurs - All saboteurs are deactivated again.

9. Check Results - The results of the executed DUT are compared to the golden model. If the results are not equal, a fault analysis will have to be performed.

When all test patterns were executed, the MFI stops. Otherwise, the next setup is performed and starts again at step *Reset DUT*.

2.6 Similarities to the Java Card Fault Injection Unit

Before the design and implementation of the Java Card fault injection unit, the designs and approaches described above were studied. Most of the tools explained beforehand were used on VHDL models. The saboteurs, mutants and simulation commands were applied on signals and values. The fault injection tool user defines signals and values in the architecture, where faults should be introduced. Therefore, the VHDL architecture had to be scanned and a syntax tree was created. This syntax tree is then used to specify

the faults. The newly proposed VM fault injection tool of this work uses a different approach. The fault injector is attached to a bus, where high-level fault conditions can be specified. The problem with previous tools was that fault triggers were dependent on other signals or simple counting schemes. With the new approach, more complex fault triggers can be used.
The following techniques explained in the tools described above, were reused in the new fault injection tool:

- **Saboteurs** - The basic approach to introduce faults into the system is equal to the saboteurs described above. The *fault injection unit* described in this thesis is attached between two bus components. This can be, for example, a processor and a memory controller. However, the saboteur approach is only applicable on bus systems.

- **Fault Modes** - Since the design uses saboteurs on a bus to introduce faults, the same fault modes that were described above can be used. This includes *stuck-at-zero*, *stuck-at-one*, *indetermination*, *bridging*, *negation of input* and *bit flip*.

- **Client/Server Approach** - For the setup of the fault campaign, a client/server approach is used. A similar approach is described in the MFI. The injection unit probably runs in an embedded device. It is insufficient to re-program this embedded device every time a new fault pattern needs to be tested. Therefore, a server is implemented which uses common communication ports to set up the client which runs on the embedded device. This allows to perform complex fault campaigns on a remote device without the need to re-program the device.

- **Attack Setup** - The attack setup is similar to the setup of the MFI. First, the setup is performed on the server. Then the attack scenario is executed and at the end the results are compared with reference data. A more detailed attack scenario is provided in Chapter 5.

Some aspects of the design described above are very different. These differences are:

- **Buses Only** - All of the designs described in Chapter 2 allow to place saboteurs on single or multiple signals. This permits a more flexible approach on the saboteur placement. However, these other fault injection designs do not enable to use more complex fault triggers which affect multiple signals. The approach described in this thesis focuses on saboteurs placed on buses and triggers which use bus-specific features to inject faults.

- **Automatic Placement** - MEFISTO-L and the MFI allowed automatic saboteur placement. To setup a fault campaign, the architecture is scanned by a VHDL parser. The user is then presented with the available signals and values where saboteurs can be placed. With the approach described in this thesis, this is currently not supported. The required hardware unit to inject faults is placed on buses and not signals. Therefore, the architecture had to be scanned for bus connections. Depending on the used architecture, the bus interfaces can be defined very differently.

- **Higher Layer Trigger** - The approaches which place saboteurs and mutants in the VHDL code mainly use triggers, depending on other signals. These trigger sequences are very simple and are often not sufficient for more complex requirements. In this paper a trigger is required which is based on memory range information dependent on the actual state of the Java VM. When this memory range is accessed on the bus, the fault unit should be triggered. To the best of our knowledge, this is the first work, where such a complex trigger approach is used.

- **Hardware-Software Co-Design** - For the setup of the fault campaign, a hardware-software co-design approach was implemented in this thesis. Previous designs only allowed to setup fault-campaigns with static trigger data. During the fault campaign, this data was only changed marginally. In the new approach, specific trigger dependent data is dynamically assigned during the execution of a program. The fault injection user specifies *what* he wants to test, but does not require any information on *where* this data is placed in the memory.

Chapter 3

Design of the Fault Injection Unit

For the design of the FIU, several approaches were considered. Each of the designs were valued against defined factors. These factors were namely:

1. Ease of implementation
2. Ease of integration
3. Usability
4. Re-usability and modularity
5. Possibilities of simulation

Every design was valued against these points. A short description of the factors and how they were put into value is shown below.

Ease of Implementation

The design should be easy to implement and not overly complicated. It should be separated into several sub-modules which can be used without each other.

Ease of Integration

The finished modules of the design should be easily integrated into an existing design. All modules themselves should allow the possibility to be used as a standalone variant. If a module should be integrated into an existing design (independent if it is a hardware, software or co-design approach), there should not be major changes required on the existing design.

Usability

After the integration, faults should be easily injected into the running system. The design is supposed to be intuitively usable and ought require a minimum amount of studying to be used.

Re-Usability and Modularity

As already described above, the FIU should be designed in a modular way. This improves the re-usability of the separately implemented modules. A goal of the final design is that every module can be used standalone and can be integrated into an existing design.

Possibilities of Simulation

The faults simulated by the final design of the FIU suppose to be as real as possible. The user of the FIU should be provided with multiple possibilities of faults and the precision of those faults. An optimal solution offers all fault types that were described in Chapter 2. The simulated faults should differ from real faults as little as possible.

3.1 FIU Design Approaches

According to the factors that were described above, several approaches were designed and evaluated. The four designs that were evaluated are two software only approaches, one hardware software co-design approach and a hardware only approach.

3.1.1 Software Only FIU (Byte-Code Layer)

This software only approach introduced a new layer which is placed alongside the original Java VM. Every time a Java byte-code is executed, the control flow of the program is switched to the FIU. The FIU then checks if any faults need to be performed. If a fault should be simulated, the memory that the byte-code accessed, is directly altered (either before or after the byte-code execution).
If the injection unit is implemented this way, the memory would be altered after (or before) the byte-code execution. However, this is not how real faults should behave. To simulate a real fault, the fault should be introduced during the memory access. The memory access should not be introduced before or after the byte-code execution.
Other advantages and disadvantages are listed below. A schematic view of the design is provided in Figure 3.1.

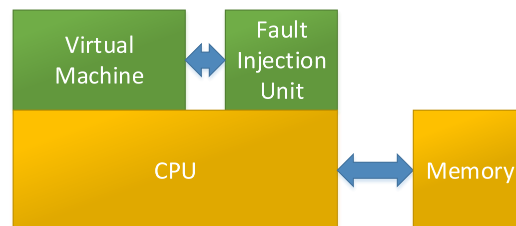


Figure 3.1: Byte-Code Layer Design (SW Only)

Advantages

1. Software only approach
2. Easy to reuse
3. Easy implementation
4. Easy integration into existing software

Disadvantages

1. Faults are simulated before or after the „real“ memory access
2. Does not simulate fault on memory access
3. Software overhead (execution time)
4. Does not simulate a real fault

3.1.2 Software Only FIU (Memory Layer)

The second software only approach that was evaluated, introduced a new memory layer. This memory layer separates the memory accesses, required by the VM from the real memory accesses, performed by the processor. Every time a byte-code is executed and tries to alter the memory, the memory access should be routed through the new layer. This approach would introduce the fault during the byte-code execution, and not before or after as the approach described before. Since every memory access is routed through this memory layer, every time an access is performed, the FIU needs to evaluate if a fault should be introduced or not. This would result in a huge increase of the execution time. Another problem with this approach would be the clear separation between the memory layer and the VM. This approach would be feasible only if the VM or other software is build upon this approach. Otherwise, there would be a big implementation effort (search and replace all memory accesses in the software).
Other advantages and disadvantages are listed below. A schematic view of the design is provided in Figure 3.2.

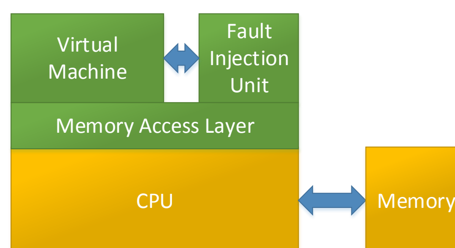


Figure 3.2: Memory Layer (SW Only)

Advantages

1. Software only approach
2. Easy to reuse
3. Memory altered during byte-code execution

Disadvantages

1. Fault is only simulated on memory access, not on the bus
2. Virtual machine or other software has to be based on this approach
3. Memory access through new memory layer, hard to separate
4. Software overhead (execution time)
5. Huge effort to integrate into existing software

3.1.3 Hardware-Software Co-designed FIU

The hardware-software co-design approach works, as the first software-only approach, alongside the existing VM. Additionally, a new hardware module is introduced. This hardware module is responsible for the evaluation of the fault conditions and the fault injection itself. The hardware module should be designed in a way, so that it can be reused in other designs. This can be achieved by enabling a bus interface (for example Advanced Microcontroller Bus Architecture (AMBA)) or other common interface definitions on the hardware unit. With this approach, the hardware unit can be used in every design, where a suitable bus interface is implemented.
The software part is used as a communication and configuration interface. Additionally, it is responsible at configurating the hardware unit with the desired fault conditions. The advantage of this approach is that the configuration can easily be performed by a server which is connected to the device. Especially when the used hardware already provides software drivers for the used communication port, this approach is a good choice. Another advantage is that every module can be reused in other designs. This makes it easy to reuse the FIU and exchange the communication interface.
Other advantages and disadvantages are listed below. A schematic view of the design is provided in Figure 3.3.

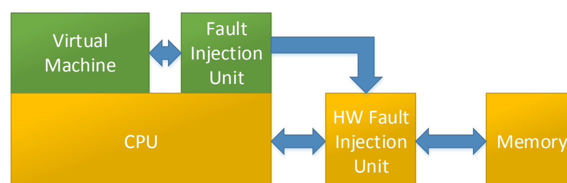


Figure 3.3: Hardware-Software Co-Design

Advantages

1. Small, simple hardware unit for the fault injection
2. Hardware unit can be designed to be easily ported
3. Small software implementation for the communication and setup of the hardware unit
4. Easy integration into existing software

Disadvantages

1. Design and implementation of both hardware and software unit
2. Requires access and changes in the hardware design

3.1.4 Hardware Only FIU

The hardware only approach is similar to the hardware-software co-design approach. The difference is that the possible communication link needs to be established in hardware. A possible predefined fault configuration needs to be stored directly in hardware. The disadvantage to this design is that the communication link may vary from hardware to hardware which makes it hard to port and reuse. Every time a hardware change is performed, the communication link needs to be re-evaluated. Other hardware dependent features need also be ported to the new hardware.
Other advantages and disadvantages are listed below. A schematic view of the design is provided in Figure 3.4.

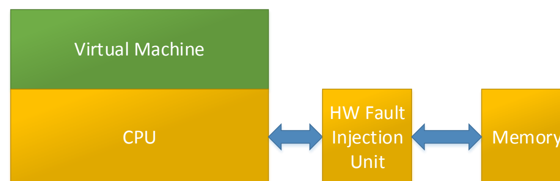


Figure 3.4: Hardware Only

Advantages

1. No software implementation effort
2. No overhead in software
3. Every software can be used (no alteration)

Disadvantages

1. Complex hardware unit
    (a) Communication
    (b) Fault injection
    (c) Configuration
2. Hard to assure re-usability
    (a) Different test environments may require different communication channels
    (b) Different processor may have different memory access approaches

3.1.5 Final Design

The first software solution is far away from simulating a real fault. The second software solution would have required a huge implementation effort. A hardware only solution would have the downside of the porting possibilities. The best overall solution was chosen as the hardware-software co-design approach. The reason was the simplicity and re-usability of the co-design solution.
The co-design solution allows to simulate faults which are very similar to real faults (injection during memory access on the bus). Due to a clean separation of the implemented modules, a high re-usability can be provided. An approach to the final design is displayed in Figure 3.6.
As show in Figure 3.6, the proposed design is split into a client and a server part. The client part consists of the hardware and software module, responsible for the fault injection, while the server part is responsible for the configuration. Basically, the FIU user utilizes the server to specify the fault configuration by defining at which point during the Java applet execution he wants to introduce the faults. The software part of the client implements a basic communication protocol which is used to transmit and receive the information between the server and the client.
Apart from the re-usability of the design, another goal was set for the design. The hardware unit has to work with zero-delay, meaning that every computation and the fault injection itself has to be performed in one cycle. This is needed, so the FIU does not influence existing hardware logic and can be used in a wide range of hardware design. The drawback of this approach is the larger hardware requirement. To achieve this goal, all hardware faults that can occur during execution are performed in serial, while all computations for these faults run in parallel and are illustrated in Figure 3.5.

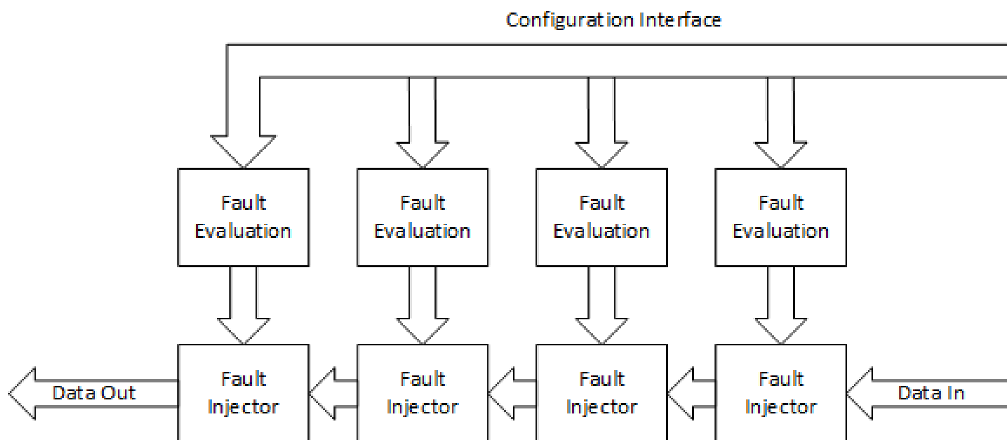


Figure 3.5: Schematic View of the Serial Injection on the Bus and Parallel Computation

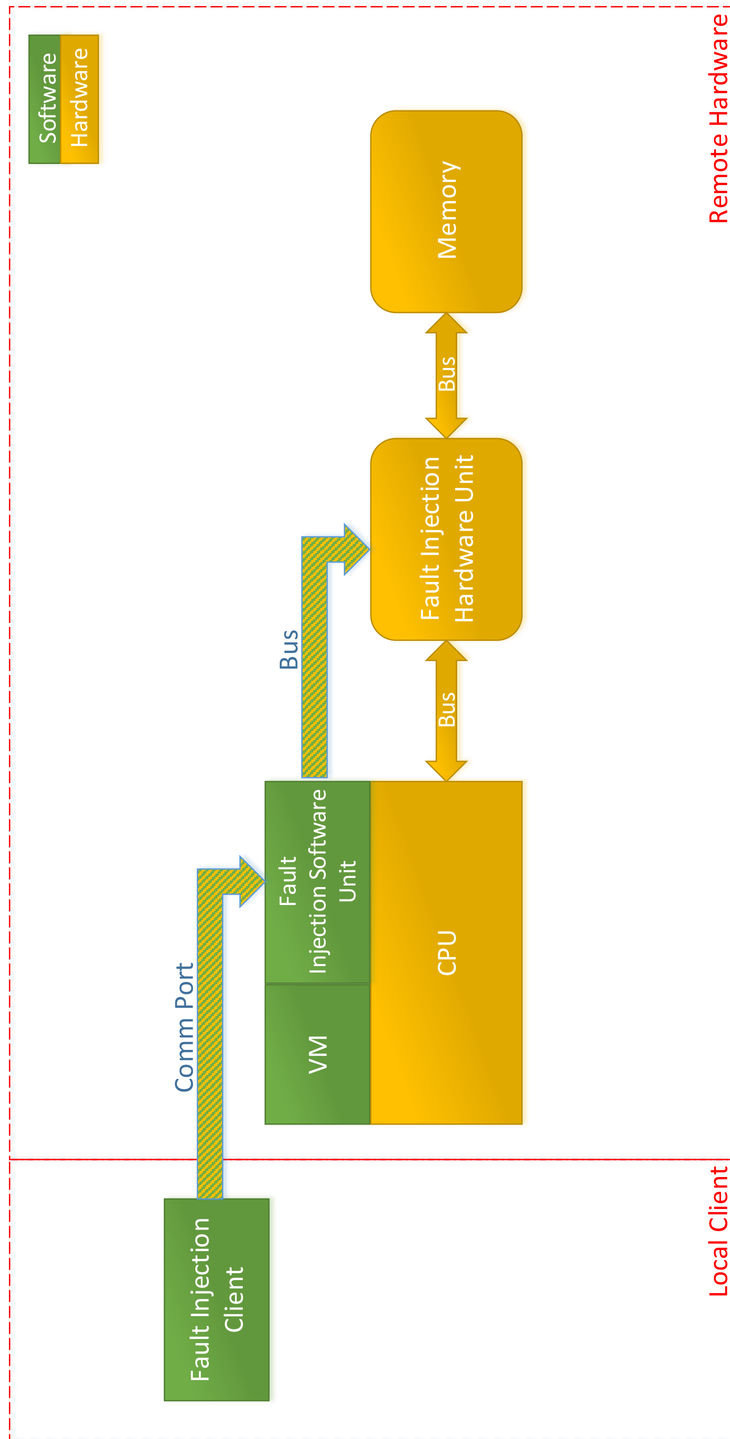


Figure 3.6: Schematic View of the Complete Design

3.2 Fault Injection Hardware Unit

The hardware unit is the main part of the FIU. It is responsible for the fault injection as well as the storing of fault configurations. The fault configurations are stored in hardware so that the faults can be introduced without any delay.
As displayed in Figure 3.6, the hardware unit is placed into the bus between the processor and the memory unit. As described in Chapter 2, this is a saboteur behavior. The defined bus interface has the advantage that the FIU can easily be reused for another DUT.
Two main configurations are provided by the hardware unit: memory area configurations, which define the memory range that the FIU observes, and fault configurations, which define what and when a fault happens. The number of memory configurations is easily configurable in the hardware model so that it can be adapted to the required hardware resources. A more detail view on the design of the configuration registers is given below.
Beside the configuration registers, the hardware unit consist of a bus controller unit, a fault trigger unit and a fault injection unit. Every module is described in more details in the sections below.

Bus Controller Unit

The bus controller in the hardware unit is responsible for converting the signals provided from the bus interface to signals that can be used by the fault trigger and the configuration registers. The bus controller needs to handle all bus relevant features as, for example, burst modes, wait cycles, differentiate between read and write cycles and others.
Following data needs to be provided by the bus controller:

- Read/Write signal
  - According to the bus specification, the bus controller needs to check if a read or write is performed on the bus.
- Number of bytes read
  - Some buses may allow multiple byte reads and writes (e.g. 32-bit bus). Since the FIU works on byte basis, the controller has to provide the system with the number of bytes read or written on the bus.
- Address
  - Not every bus system has a clear addressing scheme. The bus controller is responsible for providing a clear address signal to the system.

Configuration Registers

Since the FIU hardware unit needs to differentiate between different memory areas and configurations, two types of registers which are accessible to the bus are provided. One is used to store the memory range that the FIU trigger observes, and the other stores the actual fault configuration.
The number of memory areas and configurations can be configured in the hardware model. This allows to scale the FIU to the hardware limitations. Due to the nature of the

design of the FIU (zero-delay fault injection), for each m memory areas, $m * n$ fault configurations are required. Each memory area works separately and therefore requires its own configurations. This can result in large space requirements on the FIU part.
The memory area configuration is very simple and only stores two a-byte values, where a is the configured address width. These two values represent the lower and upper memory range of a memory area.
The fault configuration register is more complex. It stores the required access count when the trigger should be activated, the fault mode as defined in Chapter 2, a mask to select which bits of a byte should be effected by the fault and a status bit which defines if the configuration was already triggered. This is required so that every fault is only triggered once. For a more detailed description on the functionality of the configuration registers, see Chapter 4.

Fault Trigger Unit

The fault trigger unit validates if the fault conditions are met. The fault trigger is provided with the necessary signals from the bus controller and counts the number of accesses on the bus. Each memory area and every fault configuration is provided with its own fault trigger. Therefore, additionally to the count and address signals from the bus controller, the fault trigger requires the memory area and fault configuration stored in the registers. Following data is provided by the fault trigger:

- Trigger signal
- Fault position on the bus
  - As already mentioned at the bus controller, the FIU works on byte basis. When the bus is wider than a byte, the fault trigger is responsible for providing information which byte on the bus needs to be altered by the injected fault.

Saboteur Unit

At the core of the FIU lies the saboteur unit. The saboteur unit uses the signals from the fault trigger and applies the faults on the bus. As with the fault triggers, each memory area and each configuration has its own saboteur unit. This is required to provide a zero-delay fault injection. The saboteur unit was implemented according to the fault models as described in Chapter 2. The following fault models are implemented in the saboteur unit:

- Stuck at one
- Stuck at zero
- In-determination
- Negate input
- Bridging
- Bit flip
- Override

3.3 FIU Software Module

The hardware module works alongside a software module. This software module is responsible for the configuration of the hardware module. The software module can be used as a standalone variant or in cooperation with a server/client architecture.
Like the hardware unit, the software module was designed in a modular way. As before, the goal was to make all the parts as easy to exchange and reuse as possible. Therefore, different modules were defined. These software modules are the FIU controller, a communication module and the data provider.

FIU Controller

The FIU controller contains the logic for configuring the hardware unit. The main software (e.g. VM) needs to give control to the FIU controller before a byte-code is executed. The FIU controller then evaluates if a fault should be introduced. Additionally, the FIU is responsible for the data collection, required for the execution and the communication sequence handling.

Communication Module

For the communication with the environment, which can either be a server or a standalone program, the software part of the FIU provides a communication module. This module defines a communication protocol which can then be used to communicate with the core. The protocol is very simple and contains partially a header and the data. The header only contains the executed command and its length. The communication module also defines a defined interface which needs to be implemented for communication purposes. This defined interface allows to exchange the communication easily.

Data Provider

The third module is the data provider. The FIU controller needs certain information from the VM. This information is provided from the data provider. If the software is exchanged with a different one, the data provider needs to be changed to suite the new one. For example, to use different Java VM in accordance with the FIU controller, the FIU controller could request the current program counter, memory ranges of the operand stack and local variables and other information, required for the execution.

3.4 FIU PC Host

Since the FIU client may run on a different device or a device with no user input support, a simple host was designed to configure the client remotely. Again, the host was designed in order to be used in a variety of environments. The communication link is separated from the rest of the host so it can be exchanged very easily.
The host provides all features to configure the client. The host user has to set up fault configurations, create fault injection points and has to specify which data needs to be inspected. This information can then be used to set up the FIU client. The setup and evaluation of the fault injection happens in four stages:

1. Setup fault configurations
    - The FIU client needs to be configured before run-time. All fault relevant configurations (access count, fault type and mask) need to be configured before the execution of the program.
2. Setup fault injection points
    - Faults should be injected as precise as possible. The host provides possibilities to setup the fault point at byte-code layer. This means that the FIU user can specify the exact byte-code where the fault should occur.
3. Run Java applet under test
    - The host provides the possibility to run and control the flow of the Java applet under test. The FIU user should have the possibility to run and stop the Java applet under test at any given point. This ensures that the user can always evaluate the current state of the Java applet under test and the injected faults.
4. Evaluate the impact of the injected faults
    - To evaluate the fault injection, the host allows the user to inspect memory data on the client. Since the FIU is mainly focused on working in cooperation with a Java Virtual Machine (JVM), the client should provide access to local variables, fields and static members. This allows the user to easily evaluate injected faults.

A diagram of the setup work-flow is provided in Figure 5.2.

Chapter 4

Implementation of the Fault Injection Unit

The design described in the previous chapter was implemented on a predefined environment. The design was permuted on a FPGA development board with a LEON3 processor. The main parts of the implementation are:

- LEON3 IP library from Aeroflex Gaisler [Gaib]
- SimpleRTJ as a Virtual Machine [Com]
- Xilinx GR-XC3S-2000 development board [Gaid]
- Eclipse 4 RCP as a development tool for the host [ECLa]

An overview of the system and the integration of the FIU is shown in Figure 4.1.

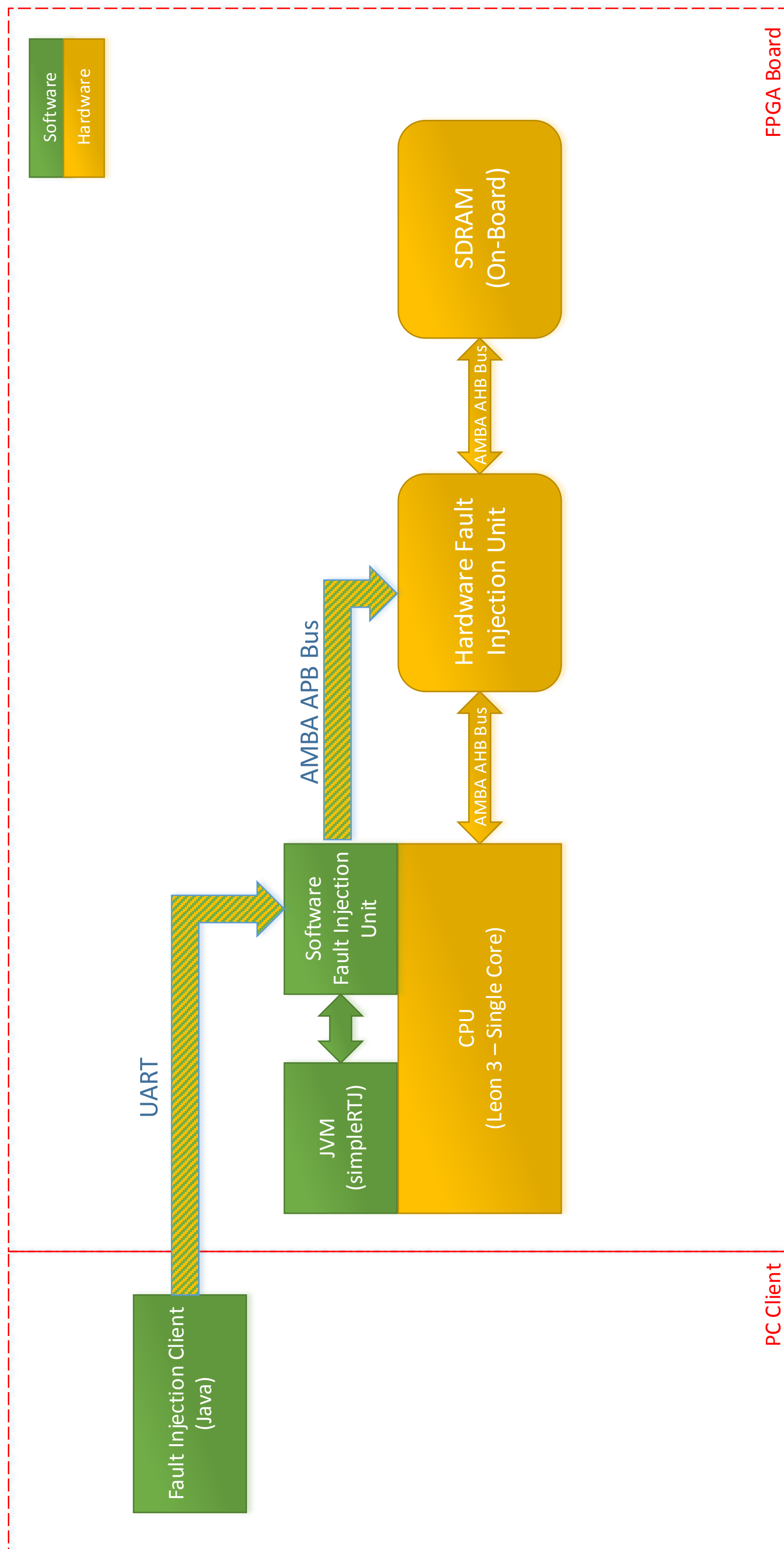


Figure 4.1: Schematic View of the Implementation

4.1 Used Tools

The technical implementation is done using the LEON3 IP library from Aeroflex Gaisler [Gaia] which is available under the GPL license. The advantage of this IP library is that it is highly customizable, open-source, fully-generic and synthesizable for an FPGA or a silicon implementation. The hardware FIU was implemented in the hardware description language VHDL. The FIU was integrated into an existing LEON3 design, provided by Aeroflex Gaisler for the FPGA development board. The FIU was attached to the system using the system AMBA bus. For configuration purposes, the Advanced Peripheral Bus (APB) was used. The integration as a saboteur was placed between the LEON3 and the memory controller using an Advance Hi-Performance Bus (AHB) interface. To test the hardware design, the AMBA test framework from the IP library was used. This assured the basic functionality of the implemented bus controller. The simulation was performed once using Modelsim SE 6.5b [Gra], and once using the Xilinx alternative ISIM [Xil] which was delivered with the development environment. To test the design on the FPGA board, the design was synthesized using Xilinx ISE 14.2 [Xil]. After that, the generated bitfile was used in combination with Xilinx Impact to download the design onto the FPGA.
The FPGA development board used for the evaluation, is the GR-XC3S-2000 manufactured by Pender Electronic Designs [Gaic]. Most of the I/O interfaces were disabled during testing to keep the hardware overhead to a minimum. Components that were used on the board included the Xilinx Spartan-3-2000 FPGA, the Joint Test Action Group (JTAG) port for programming purposes and the two serial RS232 interfaces for communication purposes. One of the serial interface was used for GRSIM to debug programs on the LEON3. The other serial interface was used to communicate directly with the FIU. An overview of the development board is shown in Figure 4.4.
The FIU software module was implemented in bare C using the C99 standard. This standard guarantees a wide range of compatibility, since it is very basic and most compilers support this standard. To test the cross-compatibility to different processors, the software module was compiled with different compilers:

- GNU GCC compiler for x86 instruction set compatible processors [GNU]
- Cygwin GCC compiler for x86 instruction set compatible processors [GNU]
- Scalable Processor ARChitecture (SPARC) Bare C Compiler (BCC) compiler for SPARC instruction set compatible processors provided by Aeroflex Gaisler [Gaib]
- Small Device C Compiler (SDCC) for Intel MCS51 based microprocessors [SDC]

For evaluation purposes, the FIU client was embedded into the Simple Real Time Java (SimpleRTJ) VM [Com]. This VM is a specially optimized for running in embedded environments. It can run standalone on any given processor that provides a C compiler without any needs of an OS. The SimpleRTJ VM supports basic features like thread support, a garbage collector and a class linker. The VM does not support dynamic linking (as in normal Java), but rather uses a class linker which creates a binary with all linked classes. Therefore, it uses static linking much like Java Card.
The client, including the integration into the SimpleRTJ VM, can be build using Makefiles. The Makefiles are designed so that the FIU client can easily be removed from the

final design and also exchange different modules (e.g. for communication).
The FIU host was written in Java using Java Platform Standard Edition (J2SE). Using the JVM, the host can be installed on any Personal Computer (PC) where the JVM is ported. The host uses the Eclipse Rich Client Platform (RCP) 4 [ECLa]. The RCP provides many features which allow implementing nearly any client application. It allows a model-based user interface and a service-oriented programming model. It is also easy to extend existing programs using plug-ins and features. An overview of the features and the architecture of the Eclipse RCP is illustrated in Figure 4.2. The core of an Eclipse RCP 4 project is the application model. The application model describes the structure of the application. It defines all visual and non-visual components of the application. The visual parts are windows, parts, menus and so on. Non-visual parts are key bindings, commands and handlers. The model describes the structure of the application, but not the content of the individual user interface components. The content is provided by the source code. Another non-visual part are model addons. They are globally registered components that can enhance the application with additional functionality. Addons are flexible and can be exchange to alter the behavior of the application without the modification of code. Another library used for the server implementation was RXTX [RXT]. This library provides Java applications the access to required serial communication interfaces. This library was used to establish the connection between the server and client over a serial communication port.

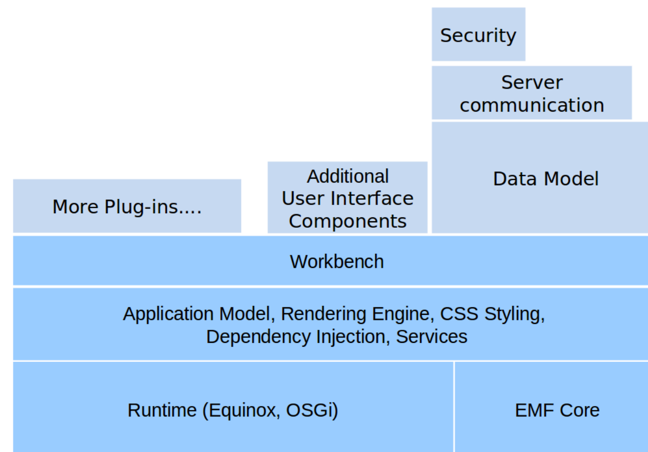


Figure 4.2: Architecture of the Eclipse RCP [ECLa]

Apache Maven [Apa] was used for building the FIU host. Maven is a build automation tool used for Java projects. The advantage of maven over other build tools is its dependency management and build procedure, which makes it easy to build modular projects. This is especially true when the project should be build on different machines. Maven requires mainly the definition of the used modules (structure) to build the project, which requires less setup time than an Ant scripts. For the Eclipse 4 RCP host build, the maven plugin *Tycho* was used [Eclb]. An overview of the software and hardware used during the implementation of the FIU architecture is given in Figure 4.3.

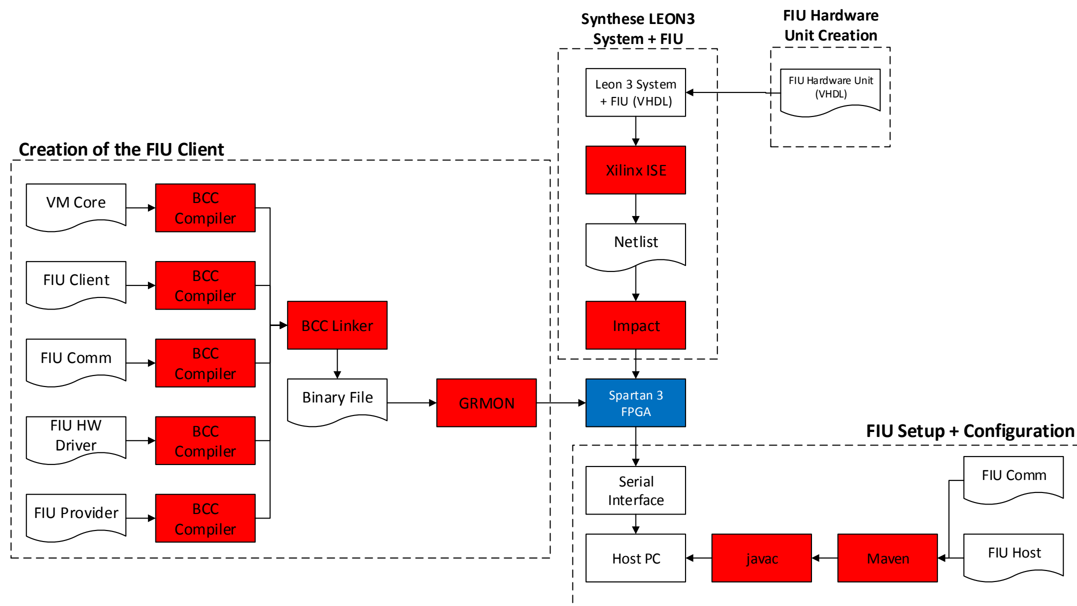


Figure 4.3: Used Tools (Red Rectangles) During the Implementation

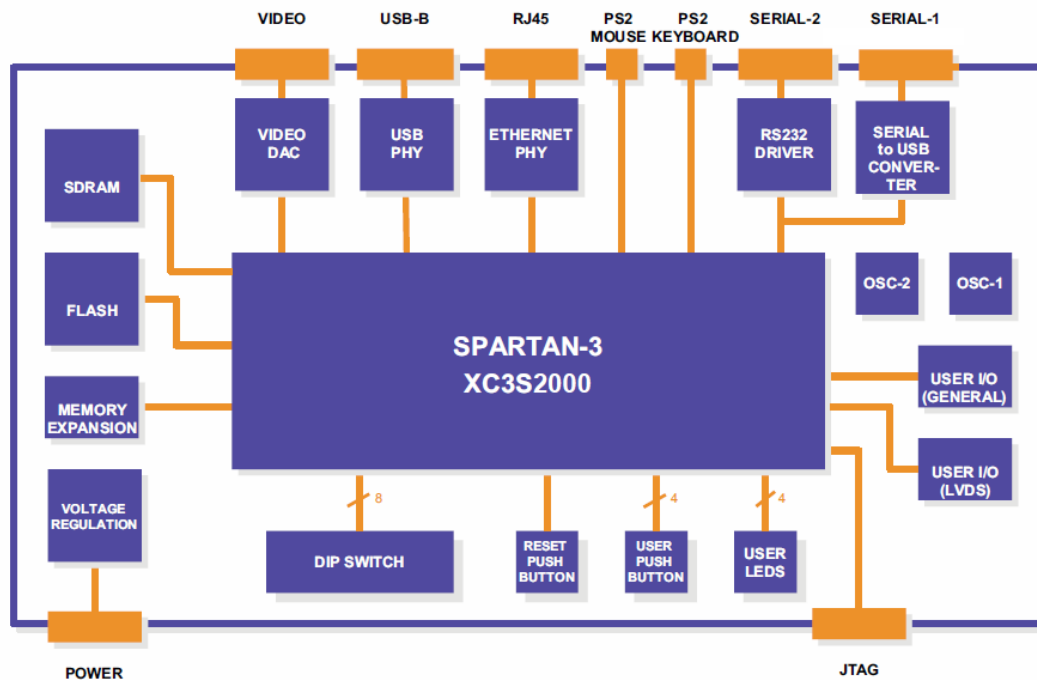


Figure 4.4: Development FPGA Board GR-XC3S-2000 Block Diagram [Gaib]

4.2 GRLIB LEON3 IP Library

For the hardware design, the GRLIB IP Library from Areaflex Gaisler [Gaib] was used. The IP Library is available under the GPL license. The library provides several IPs which can be used to generate a hardware design. The IPs provide a AMBA interface which make them easy to use. All of the cores are synthezisable and therefore usable on our target device. The library also comes with pre-made designs for several development boards. One design is available for the GR-XC3S-2000 board which was used for evaluation.
In the predefined design, the LEON3 processor core is used. The processor is highly customizable through VHDL configuration files and a provided configuration tool. The processor uses the AMBA AHB bus and the AMBA APB bus. The AHB bus is used for high performance devices, such as the memory controller, the Universal Serial Bus (USB) controller and the Ethernet controller. The APB bus is mainly used for configuration purposes, but also for low-performance devices such as the serial controller. All components connected to the LEON3 processor can be accessed using memory mapping. Therefore, Aeroflex Gaisler uses an own bus detection routine which will not be handled here but can be found in the GRLIB IP Library documentation.
The provided design was modified for the needs of the project. Several modules were disabled which were not used (e.g. ethernet and USB controller). For debug purposes the JTAG interface and the Debug Support Unit (DSU) were used. An overview of the design, that was used for evaluation, is shown in Figure 4.5. The DSU supports register modifications, break-point setups and memory reads and writes.

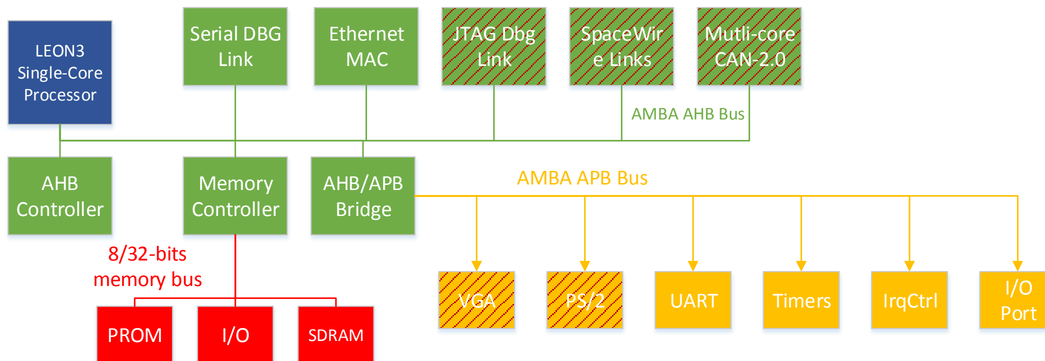


Figure 4.5: LEON3 Design for the GR-XC3S-2000 Development Board [Gaib]

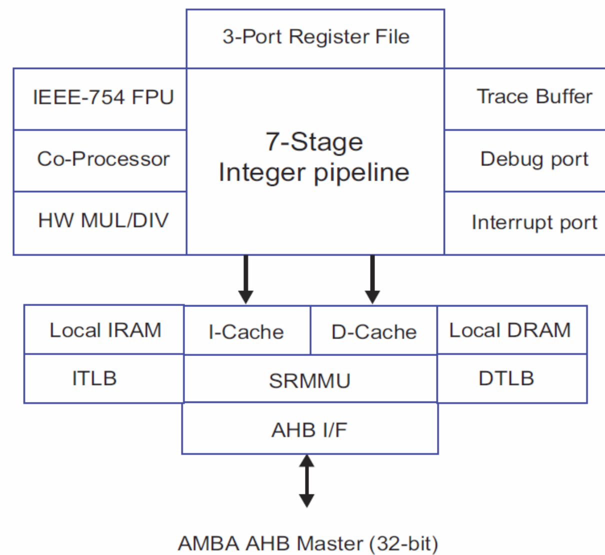


Figure 4.6: LEON3 Core Components [Gai01]

4.2.1 LEON3 Processor

The LEON3 processor is a SPARC v8 instruction set compatible Reduced Instruction Set Computer (RISC) processor. SPARC is an instruction set mainly used in Oracle/Sun products. The processor is a 32-bit architecture using a Harvard architecture with separate instruction and data caches. An overview of the LEON3 core components is given in Figure 4.6. Another feature of the LEON3 is the integer unit with a 7-stage pipeline which allows faster execution. The 7 stages of the pipeline are [Gai01]:

- Fetch - The next instruction is fetched. This can happen either from the cache or the memory.
- Decode - Instruction is decoded and addresses are generated.
- Register access - Operands are read from the register file or from internal data bypasses.
- Execute - Instruction is executed.
- Memory - Data cache is read or written.
- Exception - Traps and interrupts are resolved.
- Write-back - The result is written back to the register file.

For the FIU to work properly, the data cache was disabled. The hardware unit is placed on the bus between the processor and the memory controller. When the cache is enabled, the unit would not receive all memory accesses. This would result in an inconsistent behavior of the FIU.

4.3 SimpleRTJ

SimpleRTJ [Com] is a small Java VM implementation especially designed for embedded environments. This VM is highly customizable and most of the features can be enabled or disabled through configuration files. It was used for the evaluation implementation where the FIU software unit was integrated. SimpleRTJ implements an old version of the Java byte-code standard. It supports implementations on Java source level 1.1 and native code execution.
The VM can be compiled for most 8/16/32 bit embedded systems, as long as there is a C compiler available. The advantage of this VM is that is does not require any underlying OS. Everything, including memory management, is performed by the VM. Other features of this VM are:

- Thread Support - Implements time sliced pre-emptive multi-threading.
- Garbage Collection - Automatically heap clean-up of objects that are no longer needed.
- Memory Allocation - A simple memory management unit is implemented.
- Heap Management - Free spaces on the heap are managed.
- Class Linker - The executed binary is linked before execution. This allows faster execution since dynamic linking is not required.

4.3.1 Structure of SimpleRTJ

Memory

SimpleRTJ has built in support for managing the heap memory required by the application. The VM needs a provided memory location for the heap memory during the start-up phase. This allows dynamic heap sizes which can be easily changed for different environments. This allows the VM to be run without any OS. An overview of the memory structure is shown in Figure 4.7.

When the VM is initialized, the heap memory is split into several sections which can be seen in Figure 4.7. Each one of these sections contain various data elements for the Java application. The Java application heap section (used for storing arrays and object instances) is allocated from the heap area that lays between the object references and the method frames sections [Com].
During the execution of the Java application, the heap section grows upwards when arrays or objects are created. SimpleRTJ creates all class instances with the same size. This size is equal to the largest class instance in the current application. This makes the memory management easier and faster. When an array is created, the memory manager allocates as much memory as necessary for storing all array elements.
SimpleRTJ supports enabling and disabling the garbage collector. When the garbage collector is enabled and no free more memory is available, then the garbage collector is invoked and frees all objects that are no longer referenced. Additionally, the garbage collector may perform heap compaction if the available memory falls under a certain level.

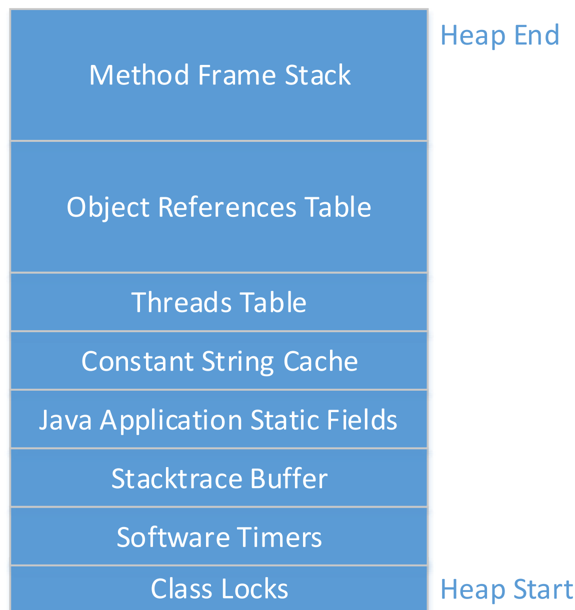


Figure 4.7: Overview of the Memory Structure of SimpleRTJ [Com]

Method Frame Structure

Every time a method is called in the Java application, a method frame is created. This frame stores all method relevant data, such as the stack pointer, local variables and the program counter of the current method. The method frame section grows dynamically downward as new methods are called during the execution process. To ease the memory management and the garbage collection process, every frame is allocated the size of the largest frame in the program. This means that the space requirement of the largest possible frame structure (in the case of SimpleRTJ, this is the frame with the most local variables) is used for every frame. A method frame with three local variables requires the same amount of memory as a method frame with only one local variable.

4.4 FIU Hardware Unit

The FIU hardware unit is located in the lowest layer of the design. It is implemented in a hardware description language, namely VHDL. The unit is built from several sub-modules, which all themselves can run separately. This ensures that the design can easily be adapted for other bus systems. The design used for evaluation implements an AMBA AHB bus as interface between the processor and the memory controller. Therefore, the implemented controller was implemented using the AHB specification. For configuration purposes the APB bus was used. A schematic view of the hardware modules is illustrated

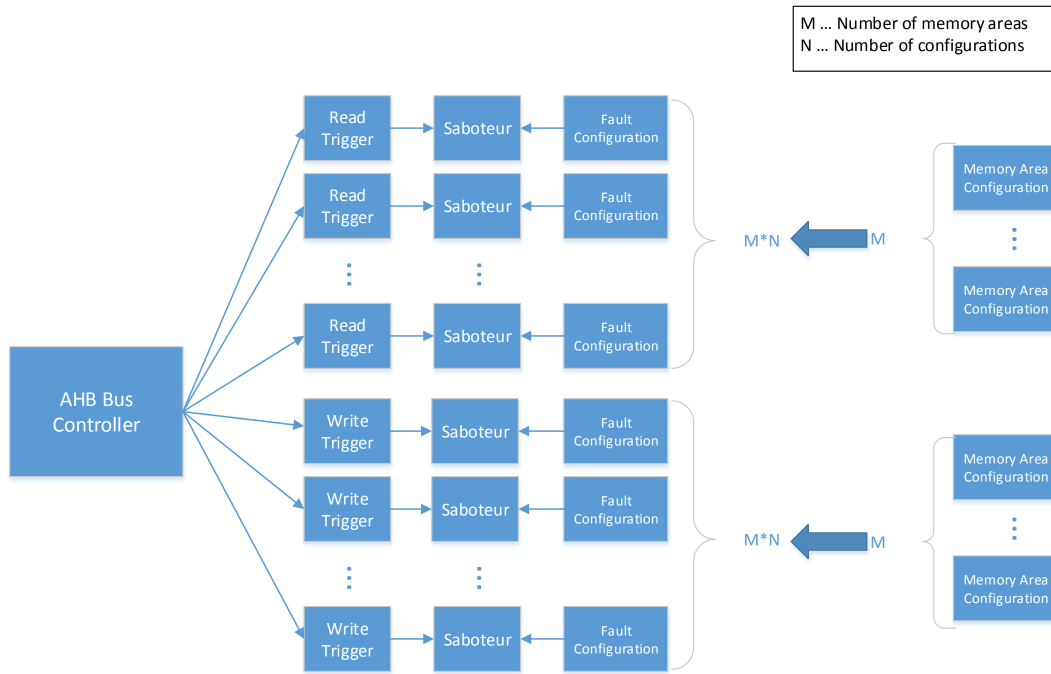


Figure 4.8: Schematic View of the Modules in the FIU Hardware Unit

in Figure 4.8.

4.4.1 Memory Area Register

One type of registers are the memory area registers. They are implemented in a standalone module so that they can be reused in the assembly for another bus interface. The register can be configured to be synchronous or asynchronous by a configuration flag. The memory area register stores the upper and lower bound of the memory area, watched by the fault trigger. The width of the stored addresses can be configured and were chosen 32-bit in the evaluation project (since the LEON3 is a 32-bit processor with 32-bit addresses). Each direction, read and write, got their own configuration registers.

4.4.2 Configuration Register

Another type of registers are the configuration registers. They are implemented in a standalone module so they can be reused in the assembly for another bus interface. The register can be configured to be synchronous or asynchronous by a configuration flag. It is used to store the main information about when and what fault should be introduced into the system. This includes the access count for the trigger, the fault mode and mask for the saboteur unit and the flag if this specific configuration has already been triggered. Each memory area has its own configurations so that every memory area can be handled independently.

4.4.3 AHB Bus Controller Unit

The bus controller is responsible for counting the number of memory accesses on the bus. Therefore, the controller has to implement all bus relevant features. Additionally, to accomplish the zero-delay goal, these evaluations have to be performed in a single cycle so that the bus communication is not altered. This also enhances the portability into other bus systems since the original system is not slowed down.
The bus controller was implemented for the AHB bus, where simple bus transfer consists of an address and a data phase. In the most simple scenario, each of these cycles requires one clock cycle. This simple transfer is displayed in Figure 4.9. After an address was put on the bus, the data is either read or written in the following cycle. The master and client can specify if they are ready over the *HREADY* signal. When the read or write requires more cycles to be performed, the *HREADY* signal remains low.

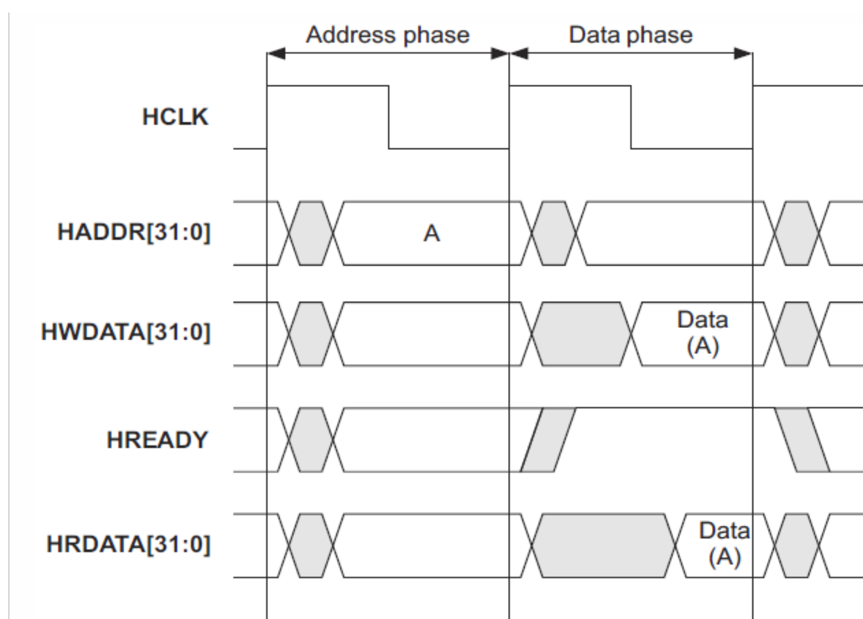


Figure 4.9: Simple AHB Transfer [ARM]

The AHB bus is a high-performance, multi-master bus, intended for devices which require high transfer rates. To accomplish this, the bus specifications describe several features which also need to be implemented by the bus controller [ARM]:

- Burst transfers - Higher transfer rates are achieved by consecutive read/write cycles without an address phase.
- Split transactions - When an operation lasts longer than expected, the master can choose to split the transaction and finish it later. In the meantime the bus can be used for other operations.
- Single cycle bus master handover - Master can switch the bus access within one cycle.

- Single clock edge operation - Operations are only performed on a clock edge, not both.
- Non-tristate implementation - The bus can perform either a read or write access. There are no other states such as an idle state.
- Wider data bus configurations (64/128 bits) - The bus can be configured to be wider than 32-bits. The AHB allows bus widths up to 128-bit.

For the bus controller to work correctly in all given environments, all these features need to be considered, implemented and tested. To accomplish this, the controller has an internal state machine implemented which switch through the address, data (separated for read and write cycles), error and split transaction state. This state machine is illustrated in Figure 4.10. The states are:

- Idle - The controller remains in this state until a read or write is performed on the bus. As per AMBA specification, this is detected when an address change occurs.
- Read - This state indicates that a read is performed on the bus. From this state, the bus can switch back to the idle state (when there is no new transfer immediately after the current one) or to the read or write state (when there is a new transfer immediately after the current one). Additionally, the controller detects error responses from the client and split transactions. When an error response is received (*HRESP* signal), the controller switches to the error state. A split transaction is detected when the *HRESP* signal contains a split response and leads to a switch to the split state. A split transaction can only occur during a read.
- Write - This state indicates that a write is performed on the bus. From this state the bus can switch back to the idle state (when there is no new transfer immediately after the current one) or to the read or write state (when there is a new transfer immediately after the current one). Additionally, error responses, as described in the read state, are supported.
- Error - When in the error state, the client reported an error for the current transaction. In this state, the error response which is sent by the client is ignored because it need to be handled by the master. After that the controller switches back to the idle state.
- Split - When a split transaction occurs, the bus controller waits until the client notifies the master that he is ready. After that the controller switches back to the read state.

Additionally, to the state machine described above, the bus controller needs to output the number of bytes which are read or written from or to the bus. As described in Chapter 3, the FIU works on byte basis. Since the AHB bus can be up to 128-bit wide, and therefore can read up to 16 bytes simultaneously, the controller needs to detect the data size requested by the master. This is done by analyzing the *size* signal of the bus.

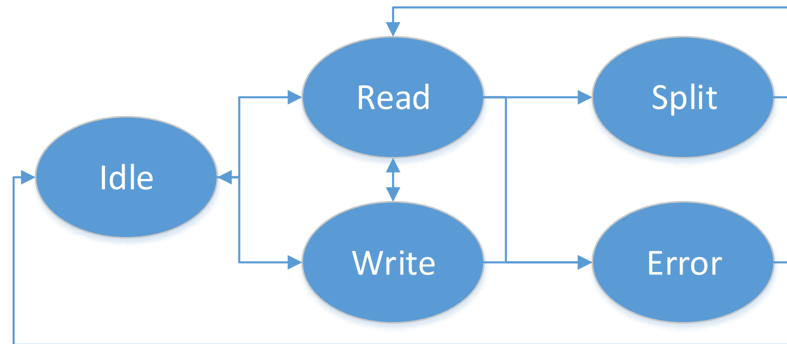


Figure 4.10: Internal State Machine of the Bus Controller

4.4.4 Fault Trigger Unit

The fault trigger is responsible for the evaluation of the fault conditions. It gets a signal from the bus controller whether or not a read or write is active. Additionally, the trigger is provided with the number of bytes read from the bus. From the configuration register, the trigger gets the memory area range and the number of accesses required for the trigger to be active. A flowchart of the trigger is given in Figure 4.11.

Each time a read or write is performed, the trigger first evaluates if the address on the bus is inside the configured memory area. Additionally, the trigger checks if the current configuration has already been triggered. This is achieved by a flag, stored in the configuration. Every fault configuration is only triggered once. After that, a new fault needs to be configured. When the address is inside the memory area and the configuration was not triggered, then the current access count is increased. This access count is stored in the configuration.
The trigger was implemented in a way, that it only counts and triggers in the range of the memory area. For example, when the bus tries to read 4 bytes from an address that is only 3 bytes in the memory range, than the counter is only increased by 3 bytes. This prevents any fault injections outside of the memory range and helps the FIU user to have full control of the fault unit.
When all computations have been performed, the trigger unit checks if the trigger should be active during the current read or write cycle. Therefore, the current access count is compared to the access count stored in the configuration. When this condition is met, three signals are generated on the output:

- Trigger signal - Signal to indicate that a fault injection needs to be performed. This is provided to the saboteur unit.
- Configuration trigger signal - Required to define that the configuration has been triggered. This avoids multiple triggers of one configuration.
- Data position signal - This signal is required to specify the position on the bus where

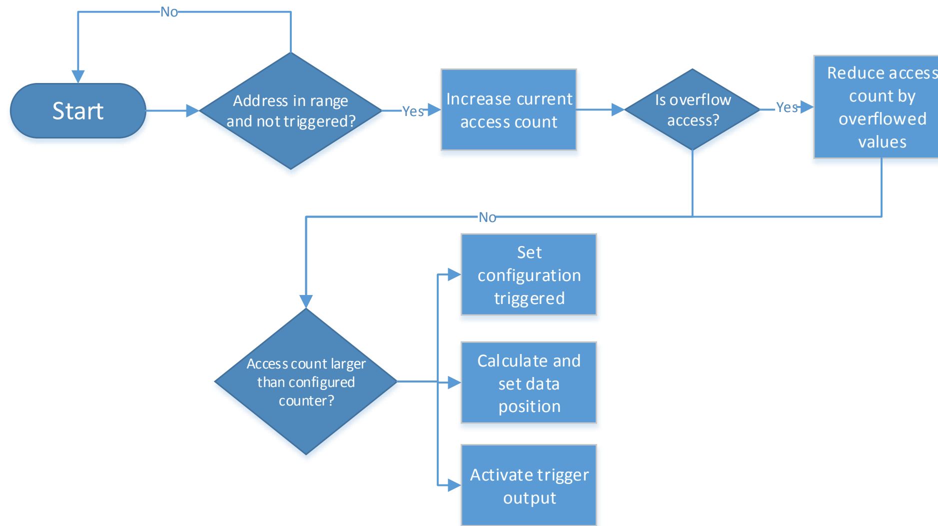


Figure 4.11: Flow Chart of the Internal Trigger Logic

the fault needs to be active. This is required by the saboteur unit to perform the fault on the correct position. The position is calculated by subtracting the current access count (without the currently active read/write) from the configured one. When a read or write is performed, that is smaller than the bus width (e.g. 1 byte read from 4 byte bus), than an additional offset is added. This offset is bus dependent and is required to specify where the read or written byte is on the bus. The behavior of the calculation is described in more detail in Section 4.4.8.

All these computations are performed in a single clock cycle, which helps to achieve the zero-delay FIU.

4.4.5 Saboteur Unit

The saboteur unit is the main part of the hardware fault injector. It is responsible for changing the values according to the fault that was configured and the signals provided by the fault trigger. Each cycle, the saboteur unit checks if a fault should be injected by a signal, provided by the trigger unit. The fault gets then placed at the position that was calculated in the trigger unit as described above. Depending on the configured fault and the fault mask, the data on the bus is manipulated. Following fault modes are implemented in the saboteur unit:

- Stuck at one - All bits set in the mask will be one during the bus transaction where the fault is active.
- Stuck at zero - All bits set in the mask will be zero during the bus transaction where the fault is active.
- Indetermination - All bits set in the mask will be set to an unknown/random value during the bus transaction where the fault is active.
- Negate input - All bits set in the mask will be negated (zero becomes one, one becomes zero) during the bus transaction where the fault is active.
- Bridging - For each bus transaction, the value of the previous bus access is stored in the trigger unit. When the bridging fault mode is active, the trigger will set the bus data to the previous value, when the fault is active.
- Bit flip - All bits set in the mask will be flipped during the bus transaction, where the fault is active. This is the same as with negating the input, but only lasts one cycle.
- Override - The value on the bus is replaced with the mask during the bus transaction, where the fault is active.

4.4.6 FIU Assembly

All FIU components need to be assembled correctly to work as intended. In the evaluation design, the APB bus was used to configure the registers and the AHB bus as saboteur interface between the processor and the memory. Therefore, the required signals for the top module are the reset and clock signal, the APB and AHB bus signals to the memory and processor.

One goal of project was that the faults and the evaluations required for introducing the faults need to be performed in a single cycle. This is required so that the original hardware design is affected as less as possible. To achieve this, each memory area and each configuration has its own trigger and saboteur unit. Additionally, each area works in read and write direction. Therefore, every memory area requires two times the number of trigger and saboteur units.

As an example, the total number of hardware components required for an FIU configuration with 3 memory areas and 2 configurations is calculated. First of all, 3 memory registers for the memory areas are required. For each of these memory areas, 2 configurations are generated. Each configuration requires a configuration register, a trigger and

a saboteur. Therefore, for 1 configuration, 3 components are required. This results in a total of $3 * 2 = 6$ components required for one memory area. The example uses 3 memory areas, therefore $6 * 3 = 18$ components are required for all memory areas. Now the bi-directional flow needs to be added to the calculation. This doubles the number of components required for one memory area. This concludes the total number of components required for the configurations to $18 * 2 = 36$. At last, the two memory area registers are added to the number of components. This leads to the final result of $36 + 3 = 39$ required components for this example. For this approach to work correctly, the saboteur units need to be connected in serial on the bus. One disadvantage of this approach is that the more configurations, and therefore saboteurs are needed, the longer the critical path becomes. Also the design can increase in size quickly which is described in Chapter 5. A schematic view of this complete design which gives an overview of the serial approach of the saboteur units is given in Figure 4.4.6.

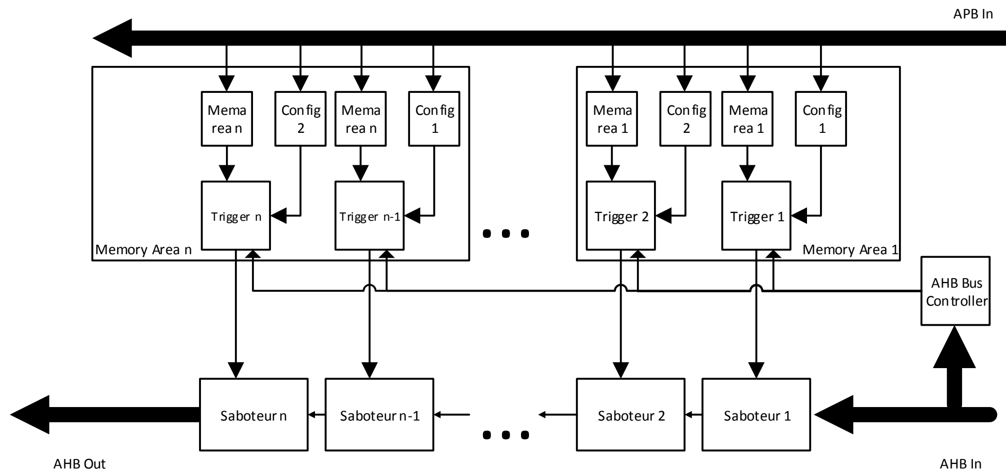


Figure 4.12: Schematic View of the Serial Implementation of the Hardware Design

APB Register Access

To configure the memory areas and the configurations for each memory area, an APB bus interface was implemented. The hardware user can access the registers using this APB bus interface. To address the registers, the least 9 bits of the address strobe of the bus are used. The 9 bits are assigned as:

- Bit 0 to 1 - Ignored. Since the bus has a width of 32-bit, only every fourth address can be accessed. The addresses go from 0 to 4 to 8 and so on.

- Bit 2 - Defines which register should be accessed. For the fault configuration, one register is used for the access count and the fault type, and the other one for the fault mask. Regarding the memory areas, one register is used for the upper memory range and the other is used for the lower memory range. If the bit is set to 0, the first register is accessed. Otherwise the second register is accessed.
- Bit 3 to 6 - Used to address all the registers. For the first configuration register, these 4 bits are set to 0x0, for the second register to 0x1 and so on. This is the same for the configuration as well as the memory area registers.
- Bit 7 - Only used for read and write registers. With this bit the FIU can differ if the data on the bus is destined for a read or a write configuration. For memory area configurations this bit is ignored. If the bit is set to 0, read configurations are accessed. Otherwise write configurations are accessed.
- Bit 8 - Separates between configuration and memory area registers. A 0 defines that a configuration is accessed. Otherwise a memory area configuration is accessed.

For the configuration register, two, from the APB bus accessible, registers are required. The first register contains the access count and the fault type. This is illustrated in Figure 4.13. The other register contains the fault mask.

31 – 10	10 – 7	7 – 0
Unused	Fault Type	Access Count

7:0 Access Count – Defines how much bus accesses are required for the fault to be injected.
10:7 Fault Type – Defines the fault type: 0: Stuck at one
1: Stuck at zero
2: Indetermination
3: Negate input
4: Bridging
5: Override
6: Bit Flip
7: Normal

Figure 4.13: APB Configuration Register

With the description above, the following memory ranges are reserved and accessible by the APB bus:

- Read Fault Configuration Registers: 0x000 - 0x07A
- Write Fault Configuration Registers: 0x080 - 0x0FF
- Memory Area Registers: 0x100 - 0x1FF

This means, when the FIU is mapped to address 0x80000000 in the system, the first read register is accessible at address 0x80000000. The second read register is accessible at address 0x80000004 and so on. The first write register would be mapped to address

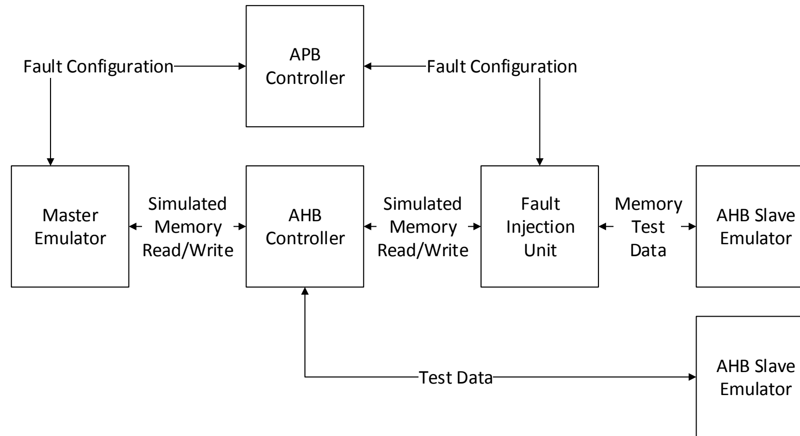


Figure 4.14: Schematic View of the Hardware Test Setup

0x80000080, the second write register to 0x80000084 and so on.
The memory ranges that are available, scale with the configuration of the registers. The more configurations are set in the configuration, the wider is the range where registers are accessible. For example, two configurations would result in four read registers. Therefore, the addresses from 0x000 to 0x01C are accessible. For a more detailed view see Section 4.4.8.

4.4.7 Hardware Testing

To test the hardware unit before using it in the FPGA board, it was tested in a testbench using the AMBA test framework, delivered with the GRLIB IP library. The test setup is illustrated in Figure 4.14. The AMBA test framework consists of an emulated slave and an emulated master. The master can either be set up from file or directly from the testbench. The master is set up to configure all faults in the FIU hardware unit. After that, some registers are read from the AHB slave (which is placed between the FIU and the emulated master). Since the AMBA test framework fulfills the basic standards of the AMBA specification, simple test cases could be tested against the design.

4.4.8 Working Example

This section will give an overview of a working example for the hardware unit and will also describe how the hardware unit is behaving. For this example it is assumed that the hardware unit was configured with 2 memory areas and 2 fault configurations per memory area. This results in a total of 8 configurations. As for the memory mapping, the example considers that the APB configuration bus of the hardware unit is mapped to address *0x80000000*. The result would be a memory mapping as given in Table 4.1. The addresses scale with the number of configured configurations. The more configurations are set up, the more addresses are available to configure. The values from Table 4.1 are

Conf. Type	Area	Conf. Number	Address	Register	Values
Read Register	1	1	*0x80000000*	0	Fault type, access count
			0x80000004	1	Fault mask
		2	*0x80000008*	0	Fault type, access count
			0x8000000C	1	Fault mask
	2	1	*0x80000010*	0	Fault type, access count
			0x80000014	1	Fault mask
		2	*0x80000018*	0	Fault type, access count
			0x8000001C	1	Fault mask
Write Register	1	1	*0x80000080*	0	Fault type, access count
			0x80000084	1	Fault mask
		2	*0x80000088*	0	Fault type, access count
			0x8000008C	1	Fault mask
	2	1	*0x80000090*	0	Fault type, access count
			0x80000094	1	Fault mask
		2	*0x80000098*	0	Fault type, access count
			0x8000009C	1	Fault mask
Memory register	1	-	*0x80000100*	0	Memory area upper address
			0x80000104	1	Memory area lower address
	2	-	*0x80000108*	0	Memory area upper address
			0x8000010C	1	Memory area lower address

Table 4.1: APB Memory Ranges
Memory addresses for the working examples on the APB bus with an offset of $0x80000000$.

interpreted as follows: The *first column* defines which type of register is accessed. This can either be a read configuration (for faults in read direction), a write configuration (for faults in write direction) or a memory area configuration (to set up memory ranges). The *second column* defines for which memory area the configuration is used, meaning a value of 1 defines that the configuration is used in accordance of memory area 1. This is only relevant for read and write configurations. The *third column*, the configuration number, defines which configuration in this memory area is accessed. The *address column* defines the accessible address. The *register column* defines which register of the configuration is accessed. Each configuration has two registers which need to be written to. For memory areas, this is the lower- and upper memory range. For fault configuration registers, this is on the one hand the access count and fault type, and on the other the fault mask.
The FIU acts as a saboteur and therefore needs to be placed on the bus between a master and the slave (this can be seen in Figure 4.14). During this example the slave is assumed to be mapped to address *0x40000000* on the bus. The steps required to configure and setup a fault are described below.

1. At the start, the hardware unit needs to be configured. Therefore, the registers described in Table 4.1 need to be written. For the FIU to work as expected, at least one memory area and one configuration needs to be configured. To configure the memory area, the value *0x40001000* is written to address *0x80000100*, and the

value *0x40000000* to address *0x80000104*. Now the FIU observes the memory from address *0x40000000* to *0x40001000*.

2. After the memory area configuration is written, the fault needs to be configured. In this example a write fault will be set up. Therefore, address *0x80000080* and *0x80000084* need to be written. The register at address *0x80000080* will be written according to Figure 4.13. For an access count of 4 and a negate input fault type, this value is set to *0x304*. The mask on address *0x80000084* is set to *0xFF* so that every bit is affected by the fault.

3. After the FIU configuration, a bus write into the observed memory area must be performed. To trigger the FIU, four bytes must be written into the observed memory area. In this example, a four byte variable is written with the value 0x30000.

4. When the value is put on the bus, the bus controller of the hardware module will detect a 4-byte write access on the AHB slave. The controller will set the write-access output signal to high and also sets the detected access count to 4.

5. The trigger will receive the activate and access count signal from the controller. The current access count is zero, and the configured trigger access count is 4. The 4 bytes from the current write command will be added to the current access count and will result in a total of 4. After that, the trigger evaluates if the fault condition which is met (current access count is equal to configured access count). Therefore, the triggered flag of the configuration is set to one and the trigger output flag for the saboteur unit is also set to one. The data position on the bus, where the fault is introduced, is set to the 4th byte. The trigger calculates the data position dependent on the current access count.

6. The saboteur unit for the configured memory area receives the trigger signal and the data position. Additionally, it gets the fault type from the configuration. With these values, the saboteur unit can trigger the fault. The original value of *0x30000* is changed to *0xFF030000* since the 4th byte is inverted.

7. Finally, the AHB slave receives the modified value. When the AHB slave is ready and received the value, the FIU stops the fault injection. Every other access onto the slave is performed without a fault.

4.5 FIU Software Module

The FIU software module of the evaluation implementation is the middle layer between the hardware unit and the host. One main goal is to implement a reusable approach. Therefore, the design uses the C99 standard which is supported by most compilers. The implementation also does not use any operating system dependent functions, such as dynamic memory allocation. This allows running the software unit in a minimal environment without any operating system. The drawback of this is that every required memory buffer has to be defined at program start.
There are 4 main parts of the module: The communication interface module, the client module, the data provider module and the hardware driver. The communication module defines a simple communication protocol, used to communicate with the host. The commands used by the protocol range from configuration commands to run time commands to halt the execution of the VM. A more detailed description of the communication protocol is given in Section 4.5.2. The client module is the core of the system. It is responsible for the setup of the communication and the hardware unit and is also responsible for the data exchange between the FIU and the VM. Another module is the data provider which is dependent on the used VM and is required to provide all the data to the VM. This is for example the current Java program counter as well as the current state and internal data of the VM. A more detailed description can be found in Section 4.5.3. The last module is the hardware driver. This driver was especially made for the evaluation example and can be used for the APB bus to configure the hardware unit. All the modules described above are designed in a modular way so that they can easily be exchanged for different requirements. An overview of the modules and how they interact with each other is given in Figure 4.15.
For the software module to work correctly, the SimpleRTJ VM needs to give control to the client at two points: once for the initialization at the start-up of the VM, and then after each byte code instruction is executed. A flow chart of the initialization and execution routine is given in Figure 4.16(b).

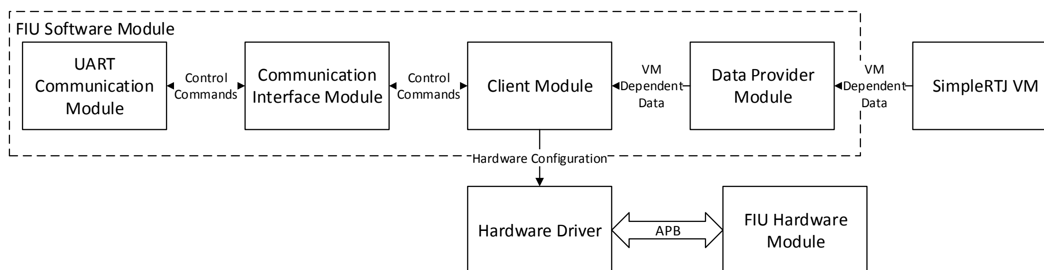


Figure 4.15: Schematic View of the Modules in the FIU Software Module

4.5.1 Client Module

The client module of the FIU software unit is responsible of controlling the communication routine with the host using the communication interface module and the data collection using the data provider module. The client module defines the communication order and when a fault is triggered.
To fully integrate the FIU into an existing VM, two modules need to be implemented and adapted to the needs of the new VM. The first module is the data provider. The data provider provides data, required by the FIU. This can be the current program counter position, the memory area of the local variables or other VM dependent data. The second module is already implemented and needs to be integrated into the VM implementation. It is a predefined interface which activates the FIU software unit. This is required for the software modules to perform their work. For the FIU to work correctly, the program flow has to give control to the FIU at four places:

1. **At the VM initialization:** Here, the FIU needs to perform its own setup routine. This resets the software configuration of the FIU and initializes the communication with the host. Additionally, the program flow stops at this point until a *Run* command has been received. Here, the host can configure the faults using the commands defined in the communication interface.

2. **After each byte-code execution:** At this point the client evaluates if a fault needs to be introduced into the system or not. After each byte-code the client checks all configured break-points and evaluates if one is set for the current program counter. When a break-point is reached, the fault is configured in the hardware unit of the FIU system. Additionally, the program execution can be halted at any Java byte-code when the specified flag is set. This program halt is used when the host wants to inspect the value of a Java variable or restarts the program execution.

3. **When an exception has occurred:** The host has to be notified about exceptions that occur during program execution. This allows the host to react and eventually restart or replace the current program.

4. **At the end of the program:** When the program has ended, the host is notified. The VM does not remain in a command dispatch routine but rather stops.

These are the only changes required in the VM. They are required so that the FIU can run and eventually stop the VM from execution.

The communication between the client and the host is done in three stages:

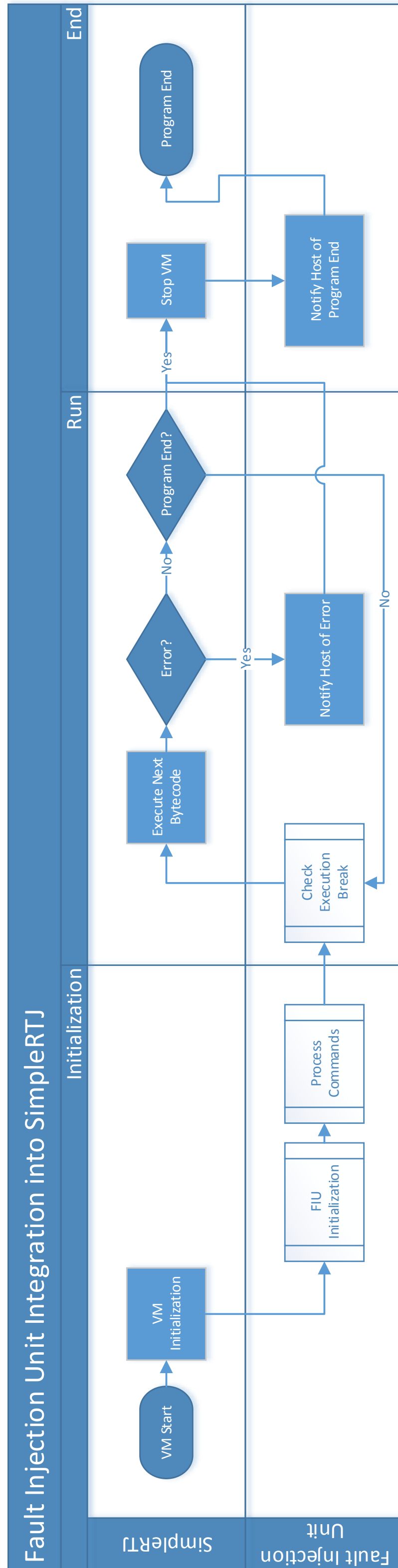
1. **During the initialization of the FIU:** Here the communication is initialized. Different communication ports require different initializations. For example, a socket implementation could instantiate a new socket and connect to the server. An example of a Unix socket implementation is given in Appendix A.1.

2. **After the VM has started but has not executed any byte-code:** At this position of the program flow, the VM is halted by the FIU until a *Run* or equivalent command is received. This VM halt helps to setup the machine, the break-points,

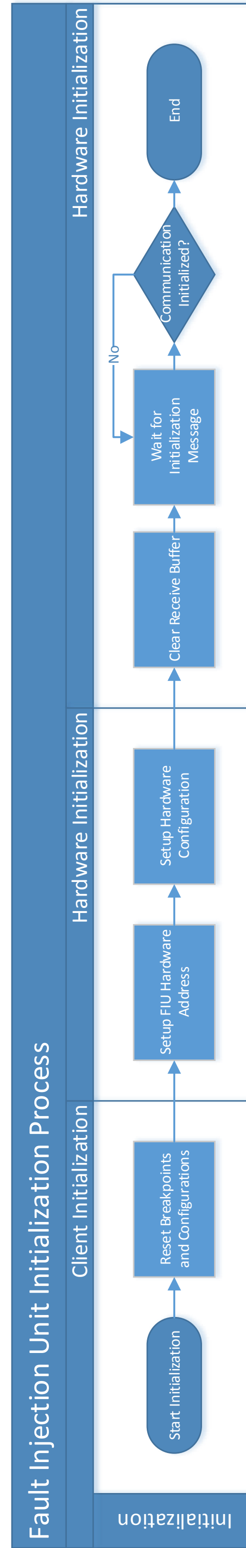
fault injection points and other features. All these initializations have to be done through the communication interface with the appropriate commands.

3. **When the byte-code is executed:** As described above, before each byte-code execution the VM requires the control flow of the program to check if any break-points are reached. The break-points can be set up to halt the execution. When this happens, the communication interface listens to new commands that may be received. All commands that are available during initialization are available here. The FIU host can therefore reset the execution at any given point of the program flow.

In the enumerations above, the term „break-points“ was used. Break-points in the context of the FIU are not handled the same as normal break-points in languages like C or Java. FIU breakpoints store additional information if either a fault should be injected or not. Fault injection at a break-point is optional. Another use of break-points can be the halting of the execution of the VM. This may be used to restart the program execution or inspect variables. Both of these features, execution halt and fault injection, can be used simultaneously. When a fault should be introduced during a break-point, the break-point also stores the memory area where the fault is introduced, the fault configuration and the direction of the fault (read or write).
Memory and fault configurations are generally separated into software and hardware configurations. When the FIU host configures the client and sets up the memory areas and fault configurations, all these configurations are stored in software. The client has its own memory area to store this information. The actual hardware configuration happens later, when a break-point is reached and a fault configuration should be performed. This approach has the advantage that the software can store much more configurations as the hardware (dependent on the available memory). As described in Section 4.4, the hardware unit can be very limited in storing memory areas and fault configurations. All the configurations stored in hardware, are performed simultaneously. Most of the time these configurations are enough. But it may happen that during the whole program execution, multiple faults at multiple locations may be used. To make this process as smooth as possible, the host can configure all the faults in software. Therefore, they do not have to be configured later in the program flow. After every byte-code execution, new faults can be configured in hardware. So for every byte-code in the program flow, the maximum of simultaneous configurable faults is only limited by the available hardware registers of the hardware fault injection. This behavior can also be observed in Figure 4.16(a). The sub-process of the initialization routine is displayed in Figure 4.16(b). Figure 4.17(a) shows how commands are dispatched during the initialization and after the byte-code execution. At last, Figure 4.17(b) shows the break-point evaluation after each byte-code.

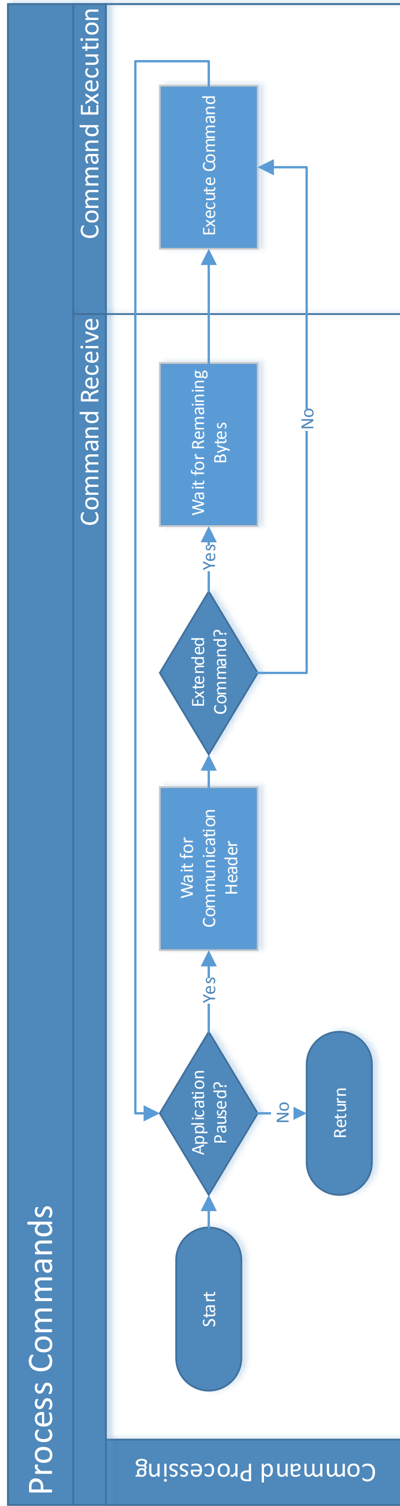


(a) Flowchart of the FIU Integration into SimpleRTJ

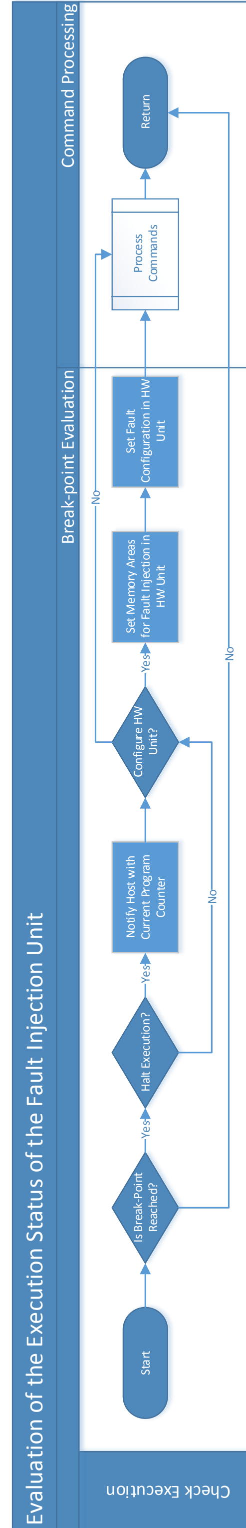


(b) Flowchart of the FIU Initialization Routine

Figure 4.16: Overview of the FIU Integration into SimpleRTJ and the Initialization Routine



(a) Flowchart of the Command Dispatch



(b) Flowchart of the Execution Evaluation after Byte-Code Execution

Figure 4.17: Overview of the FIU Command Dispatch Routine and the Execution Evaluation

Method Name	Return Type	Description
init_communication()	Boolean	Used to initialize the communication. Every communication dependent data should be initialized here. For example, the hardware unit can be set up in this routine. The return value defines if the initialization succeeded or not.
close_communication()	None	Closes the communication. This is called at the end of the VM life-cycle.
transceive()	Integer	Performs a transmit and receive. Returns the received bytes.
transmit()	Integer	Transmits the given bytes to the host. Returns the number of transmitted bytes.
receive()	Integer	Receives the number of specified bytes. Returns the number of received bytes.
is_data_available()	Boolean	Returns if data is available at the communication port or not.

Table 4.2: Communication Interface Methods

4.5.2 Communication Interface Module

The communication interface module is separated into two parts. One part is written once and does not change, depending on the communication port that is used for communication (Universal Asynchronous Receiver Transmitter (UART), Ethernet, ...). This unchangeable part defines the communication protocol and handles the communication. The hardware dependent communication interface is used by the already implemented communication unit. For every new communication port a new interface implementation is required. The interface that needs to be implemented, dependent on the communication port that is used, is shown in Table 4.2. An example of an implementation of the communication interface, using Unix sockets is given in Appendix A.1. The communication interface does not necessarily require a client/host implementation. A local implementation is also possible where the commands are directly coded into the implementation. This allows to set up standalone implementations without the use of a host interface.

Communication Protocol

For data exchange with the server, a simple communication protocol was implemented. Since the FIU software unit does not use dynamic memory allocation, a fixed receive and transmit buffer is initialized at the start time of the program. The communication is bi-directional, meaning that both the host and the client can issue a command. After each transmitted command, a response is required. This response is required so the host and client know, when a command has finished. Therefore, each command consists of a request and response part.
The structure of the commands are simple. Each command consists of a header and then the data. The data contains the length of the command in bytes, and a command code

to identify the command. With those two parameters, the server and client know how much bytes they can expect from the communication. Every command requires a header byte. The data is optional. Command and response messages do not need attached data. A diagram of the structure of the commands is given in Figure 4.18. An overview of the most important commands with a detailed description can be found below.

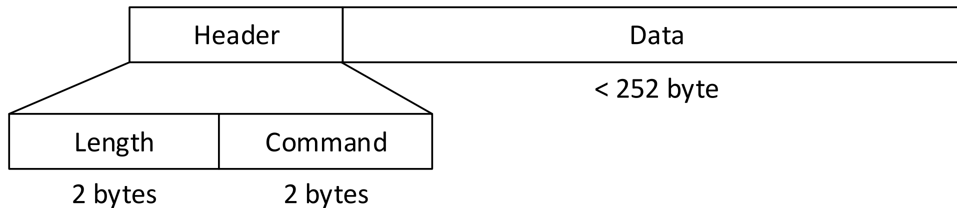


Figure 4.18: Command Structure of the FIU Communication Protocol

- **Synch** - Initiated by the host. Used to synchronize the host to the client. Provides all data, required for the host to calculate fault positions and data to check if the used client is valid.
- **Memstat** - Initiated by the host. Returns the memory status of the client. For example, how much heap space is left and how much strings are used.
- **Inspect** - Initiated by the host. Inspects a Java variable and returns the value. The host needs to specify which variable needs to be inspected and what type of variable the host wants to inspect. This can either be a local variable, a field, a static variable, an array or an object.
- **Set Break** - Initiated by the host. Sets a break-point at a specified location in the code. The break-point contains the code position, a flag which indicates if whether or not a fault should be indicated and what fault should be triggered. Additionally, a flag needs to be provided if the VM should be halted. This break point functionality is required when the host wants to inspect a variable.
- **Set Memarea** - Initiated by the host. Sets a software memory area in the client. This memory area can then be referenced by a break-point.
- **Set FIU Configuration** - Initiated by the host. Equal to the *Set Memarea* command, but uses fault configurations. Can be referenced by break-points.
- **Run** - Initiated by the host. Starts or continues the execution of the VM. For this command to work, the software module has to be in the command dispatch routine.
- **Abort** - Initiated by the host. Cancels the current execution of the VM. After this command is executed, the VM is halted.

- **Restart** - Initiated by the host. Restarts the execution of the current program in the VM.
- **Break** - Initiated by the host. Breaks the execution of the current program in the VM. The program can be resumed using the *Run* command.
- **Set Param** - Initiated by the host. Sets start parameters of the program, executed in the VM. These parameters are the starting arguments for the Java program.
- **Download** - Initiated by the host. Replaces the current program in the memory of the VM with the program transferred during this command. In general, this command is called multiple times since the largest data chunk that can be transmitted is limited to 252 bytes.
- **PC Position** - Initiated by the client. Notifies the host that the client stopped or halted at a specific program counter position.
- **Program End** - Initiated by the client. Notifies the host that the program execution has ended and the VM has halted.

4.5.3 Data Provider Module

The FIU data provider is the third module of the software implementation. The data provider is VM dependent, therefore every VM has to implement the data provider interface. This abstraction is used, so the client can work independent of the used VM. The data provider merely provides a simple interface which needs to be implemented. An example implementation for the SimpleRTJ VM is given in Appendix A.4. The explanation of the required interface functions is given below. Most of these data provider functions are Java frame dependent. This Java dependency means that they work on information like memory region locations which change for every executed method.

- **Fill Synch** - Provides all the values for the synchronization command described in Section 4.5.2. Since these values are dependent on the implementation details of the currently used VM, this data collection method is outsourced.
- **Fill Memory Stats** - Provides all the values for the memory stats command, described in Section 4.5.2. Since these values are dependent on the VM, this data collection method is outsourced.
- **Set Program Memory** - Sets the program memory at a specific offset. This function is used when a program is downloaded to the VM. The VM has to exchange the byte-code of the current memory with the provided one.
- **Is Program Counter Reached** - Evaluates whether or not a specific program counter is reached in the program execution.
- **Get Current Operand Stack Memory Range** - Returns the upper and lower memory bounds of the operand stack of the current frame in the memory. This is required to dynamically set the memory areas for the fault injection.

- **Get Current Local Variable Memory Range** - Returns the upper and lower memory bounds of the local variables of the current frame in the memory. This is required to dynamically set the memory areas for the fault injection.
- **Get Current Program Counter Memory Range** - Returns the upper and lower memory bounds of the program counter of the current frame in the memory. This is required to dynamically set the memory areas for the fault injection.
- **Get Current Byte-code Area Memory Range** - Returns the upper and lower memory bounds of the byte-code area in the memory. This is required to dynamically set the memory areas for the fault injection.
- **Inspect Local Variable** - Used for the inspect command. Inspects a local variable of the current frame.
- **Inspect Field** - Used for the inspect command. Inspects a field of the object where the current frame is executed.
- **Inspect Variable** - Used for the inspect command. Inspects an arbitrary variable in the memory. Can be an object or a native value.
- **Inspect Current Object** - Used for the inspect command. Inspects a value of the object where the current frame is executed.
- **Inspect Array** - Used for the inspect command. Inspects an array in the program.
- **Inspect Object** - Used for the inspect command. Inspects a value of an arbitrary object where the current frame is executed.

4.6 FIU PC Host

The last part of the FIU is the host implementation. The host is used to remotely set up the fault injection unit. Often, the used VM is not executed on the same machine as the host and therefore needs remote configuration. This is especially true when the VM runs on an embedded device or a remote FPGA as in our reference implementation.
One of the most important features of the host is, to provide an intuitive and easy user interface to make fault injections as easy as possible. It is not sufficient, when every time a new test environment is set up or small changes are made, all memory offsets have to be analyzed. Injecting a fault has to be as easy as writing the code itself. Therefore, in the reference implementation an editor-like user interface was created.
Figure 5.2 in Chapter 5 shows an example workflow of a fault injection setup. This is the basic workflow with all the required steps to perform a fault injection using the proposed FIU host. When the host is started, the first thing to do is to open a project. The project consists of a created debug file. In this reference implementation, this debug file is created by the class linker of the SimpleRTJ VM. This file contains several debug information, such as detailed information of the program counter. Additionally, is shows all methods, fields and other information. After the project was opened, a connection adapter has to be selected and configured. There can be multiple adapters defined, for example a connection

adapter for Unix sockets and one for serial connections. An adapter configuration for a serial interface port is shown in Figure 4.19. It shows how specific parameters of the communication may be configured. More details are listed in Section 4.6.1.
The next step is to connect to the client, using the configured adapter. When a connection was established, a synchronization command is exchanged as described in Section 4.5.2. After that, all fault relevant configurations need to be made. This includes the memory area, fault and inspection configuration. When this step is done, the configuration has to be downloaded to the client. Optionally, the current binary can be downloaded if necessary.
This is the end of the configuration setup. The next step is to control the program flow of the Java program, stored in the VM. At the beginning, the program has to be started. When a break-point is reached, the inspection utility can be used to evaluate if certain variables contain expected values. When this condition is satisfied, the normal program flow ends. Otherwise the program can be reconfigured and restarted.

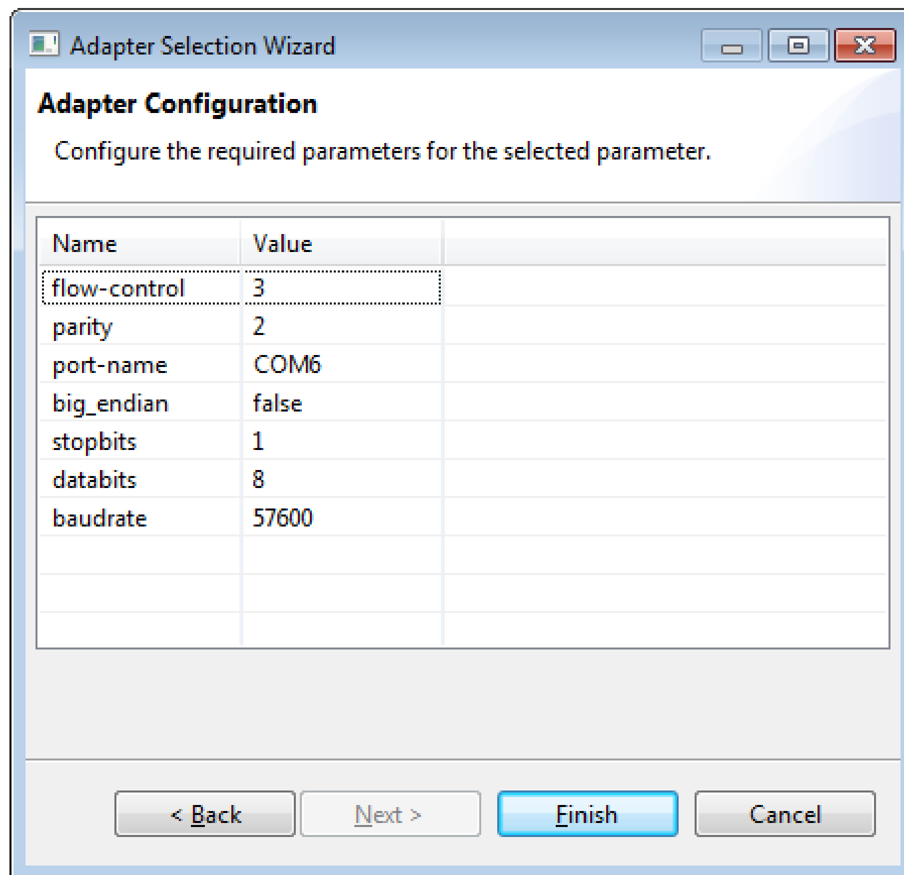


Figure 4.19: Dialog for the FIU Adapter Configuration

4.6.1 User Interface

For the implementation of the user interface, an application model for Eclipse RCP 4 was generated. This application model is used to describe not only the General User Interface

(GUI), but also the structure of the application in Extensible Markup Language (XML). It defines all visual and non-visual elements of the application. Visual parts are, for example, menus, toolbars, windows and so on. Non-visual parts are handlers, commands and key-bindings.
The main purpose of the application model is to describe the structure of the application rather than the content of the individual user interface components. This means that the application model describes what parts are available, but not what these parts contain. In the reference example, this may be the part containing the editor with the source code. The existence of this part is described in the application model, but the data is provided from the source code.
Another feature of the Eclipse 4 RCP application model are model addons. These are globally registered objects which enhance the application with additional functionality. These addons can easily be exchanged without modifying the existing code. In the evaluation design, this was used to convert the responses received from the client into general objects understood by the application.
An image of the user interface is shown in Figure 4.20. The image shows the main screen of the FIU host. It shows an open project with several information displayed.

- **Class Information** - Shows information on the debug project. It shows all classes statically linked to the project. Additionally, it shows all methods, fields and the line number information if debug information is present.

- **Client Information** - This information is available after the host has successfully connected to the client. This part shows all information provided by the client such as the available heap space, the name of the program stored in the memory and others.

- **Communication Log** - Displays all the information about the communication between the host and the client. It shows the name of the command and the transmitted bytes in hexadecimal values.

- **Fault Configuration** - Here, all software fault configurations are listed that were configured, using the GUI. It shows details such as the fault type and the access count.

- **Fault Injection Point** - This part shows all fault injection points, configured in the host. Is shows the program count, pass count and other information.

- **Inspection Configuration** - Shows all inspection configurations.

- **Source Window** - Displays the sources of the current project. This is also used to set up the fault configurations. The host automatically detects the cursor position and calculates the program counter at this location. Additionally, the source window supports search functionality and syntax highlighting for Java programs.

- **Byte-Code Window** - The byte-code window shows the byte-code instructions at the current cursor point of the source window. This can be used to set up the fault in more detail and set it at a specific byte-code instruction, rather than a source-line.

There are also two menus visible on the top. The main menu is used for managing the project, setting up the communication interface, configuration of the memory areas and fault injection points as well as inspection configurations and connecting to the client. The toolbar in the second row is used to control the program flow of the program executed in the VM. These elements are only enabled, when connected to the VM and a valid program is loaded. At the bottom, the current connection status is shown as well as the current program counter that is selected in the source window.

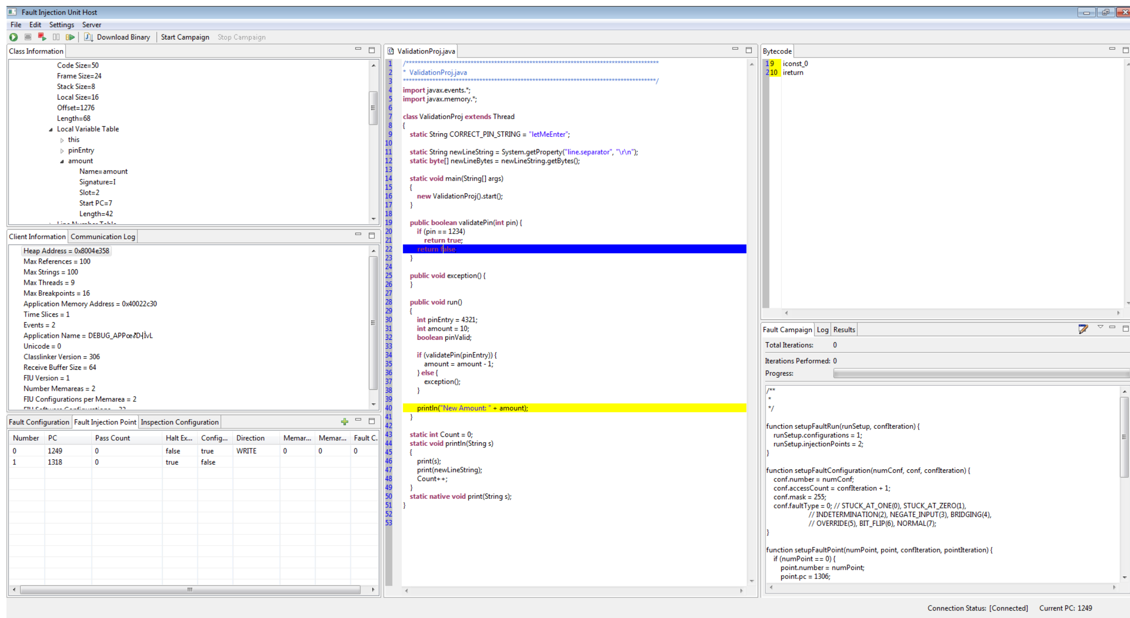


Figure 4.20: User Interface of the FIU Host

4.6.2 Communication Interface

As for the client interface, the host uses an expandable and exchangeable method to integrate new communication ports. Therefore, an existing abstract class, the *AbstractStreamServer* class, has to be implemented and the missing methods have to be added. An example for a Unix socket implementation is given in Appendix A.2. The following methods have to be implemented by a new communication interface to be active:

- **Get Name** - Returns a representative name for the communication interface. This is mainly used to display the interface to the user in a readable and understandable way (e.g. „Serial Server“ for a serial communication port).
- **Get Description** - Similar to *Get Name*, this is only used for displaying purposes. This method should return a description about the communication interface (e.g. „Server which uses serial port to communicate with the fault injection unit client“ for a serial communication port).
- **Connect** - Performs the connection to the client. After calling this connect method, the host should be connected to the client and is able to exchange data. For example,

an initialization message could be exchanged to validate a connection.

- **Get Required Properties** - Returns all connection properties that may be set and changed by the user. This is used to provide a general configuration routine for all communication interfaces. For a socket communication interface, this may be the port, for a serial communication interface the baud rate and number of data bits.
- **Set Property** - This sets a connection property for the communication implementation. This is used to provide a general configuration routine for all communication interfaces.
- **Initialize Server** - This method is called after the configuration has been performed for the communication interface. There, the communication interface may perform some pre-connect setups.
- **Disconnect** - Disconnects the host from the client. After this method is called, the connection should be closed and ready for a new connection.

4.6.3 Fault Campaign Support

Another feature of the FIU host is the support of a so called fault campaign. Fault campaigns allow the user to perform multiple fault scenarios in one session, where the fault setup can be changed before each fault scenario. This is done by providing a fault script, which is based on the ECMAScript standard. The skeleton of such a fault campaign setup is shown in Appendix A.5. To set up a fault campaign, this skeleton can be used and expanded.
When a new fault campaign is started, the user defines the number of configurations which should be iterated and, additionally, the number of fault points to iterate. The total number of fault campaigns is $numberOfConfIterations * numberOfPointIterations$.

Chapter 5

Results

The proposed fault injection unit was split into a hardware and a software unit. Both of these modules were integrated into an existing system. To prove the functional correctness and effectiveness of the proposed design, it was evaluated on a LEON3 SPARC V8 processor by Gaisler Research [Gaib].
The setup consisted of an FPGA development board with a Spartan-3 XC3S2000. Simulation results were acquired using the ModelSim software, provided by Mentor Graphics [Gra]. The results of the implemented tool were evaluated for the resulting size of the VHDL design, the overall increase of the design compared to the original one, the simulation speed and the fault injection performance using real hardware (FPGA board). At last, an overall attack scenario was described, how it works and how the FIU helps to realize it.

5.1 Platform Setup

The used LEON3 processor is configured as a single-core implementation with the default setup for this development board (GR-XC3S-2000). Several unused components were disabled, such as the USB controller, to reduce the overall size of the hardware. The hardware unit is placed between the LEON3 processor and the used memory controller and can be configured using the *xconfig* tool provided by the Gaisler IP library. With this tool, the number of memory areas and configurations can be configured. For this evaluation 2 memory areas with 2 configurations are used. This setup is configured with the tool *xconfig* as shown in Figure 5.1. The FIU configuration on the APB bus is mapped to address *0x80000A00* and therefore results in the memory mapping as displayed in Table 5.1.
The software unit was used in combination with the SimpleRTJ VM and uses the serial port of the development board to communicate with the server. The server was running on a Windows 7 environment with the Java Development Kit (JDK) 7u51 and Java Runtime Environment (JRE) 7u51. Additionally, to the serial port, the board was connected via Ethernet to use the debug unit of the LEON3. The server was used to set up the fault campaign and test the fault injection unit.

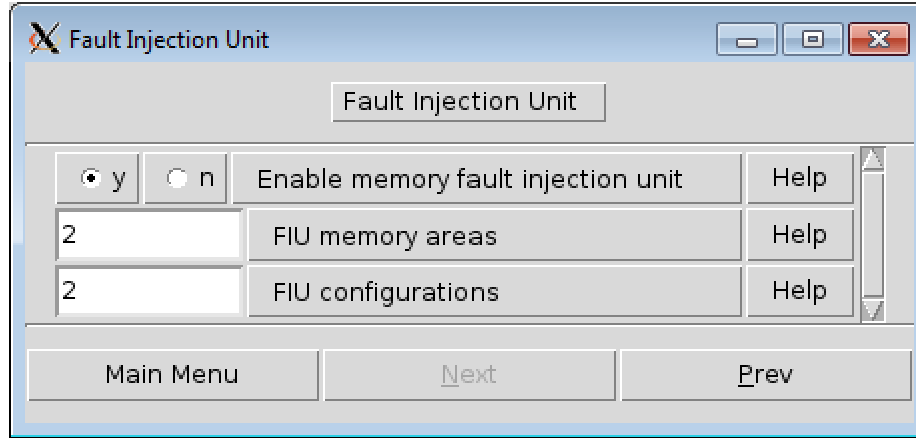


Figure 5.1: FIU Setup within the xconfig Tool

Conf. Type	Area	Conf. Number	Address	Register	Values
Read Register	1	1	*0x80000A00*	0	Fault type, access count
			0x80000A04	1	Fault mask
		2	*0x80000A08*	0	Fault type, access count
			0x80000A0C	1	Fault mask
	2	1	*0x80000A10*	0	Fault type, access count
			0x80000A14	1	Fault mask
		2	*0x80000A18*	0	Fault type, access count
			0x80000A1C	1	Fault mask
Write Register	1	1	*0x80000A80*	0	Fault type, access count
			0x80000A84	1	Fault mask
		2	*0x80000A88*	0	Fault type, access count
			0x80000A8C	1	Fault mask
	2	1	*0x80000A90*	0	Fault type, access count
			0x80000A94	1	Fault mask
		2	*0x80000A98*	0	Fault type, access count
			0x80000A9C	1	Fault mask
Memory register	1	-	*0x80000B00*	0	Memory area upper address
			0x80000B04	1	Memory area lower address
	2	-	*0x80000B08*	0	Memory area upper address
			0x80000B0C	1	Memory area lower address

Table 5.1: APB Memory Ranges
Memory addresses for the evaluation setup on the APB bus with an offset of $0x80000A00$.

Module	Look-Up-Tables	Slices
AHB Bus Controller	98	51
Access Trigger	227	123
Saboteur	160	97
Configuration Register	5	3
Memory Area Register	1	1

Table 5.2: Overview of the Space Requirements on a Spartan-3 FPGA for the FIU Units

Memory Areas	Configurations	Look-Up-Tables	Overhead [%]	Slices	Overhead [%]
0	0	14067	0	7748	0.0
1	2	16111	14.5	8757	13.0
1	4	18041	28.3	9659	24.7
2	2	18065	28.4	9846	27.1
2	4	23104	64.2	12218	57.7
3	2	20649	46.8	10975	41.7
3	4	26098	85.5	13088	68.9

Table 5.3: Space Requirements of Different Hardware Setups with the Overhead to the Original Design

5.2 VHDL Synthesis Results

The FIU hardware unit was used on a Spartan-3 FPGA. Therefore, the synthesization was done by the Xilinx ISE design suite. The synthesis results of each submodule of the hardware unit are illustrated in Table 5.2. The synthesis results including all hardware components using different hardware setups with a varying number of memory areas and configurations for the hardware unit are illustrated in Table 5.3. This size estimations include the complete design, including the LEON3 and all other required components. It can be seen that the size and the overhead to the original system hardly depend on the number of memory areas and configurations required by the system. This can result in an overhead from 13% to 70%. Depending on the number of configurations, the increase of space requirements when an additional memory area is added, is varying. When the system is configured with 2 configurations, the size increase is smaller than for a system with 4 configurations.
The overall increase in space requirements can be associated due to the serial structure of the hardware unit. For each new configuration, $2 * m$ new access triggers, saboteurs and configuration registers are added to the system, where m represents the number of memory areas. The same goes for every new memory area that is added. For each memory area, $2 * n * m$ new components required, where m represents the number of memory areas and n the number of configurations.

5.3 Simulation Results

Before the system was evaluated on hardware, the correct behavior was evaluated by simulation using the ModelSim software package. The simulation evaluation was performed in three steps: The hardware unit with an emulated bus master and slave, the complete system with the LEON3, using a simple test program written in C and the complete implementation with SimpleRTJ, using a simple test scenario in a Java program. The results of these simulations are given in Table 5.4. All simulations were performed using 2 memory areas with 2 configurations.

Basic FIU Hardware Verification

This chapter shows a simple simulation scenario where the basic functionality of the FIU hardware unit is tested. It includes the hardware unit, an emulated master (which would represent the processor) and an emulated slave (representing the memory). However, neither the software module of the FIU nor the host are included in this test scenario. The emulated master and slave are part of the Gaisler IP library [Gaib] and can be configured which addresses are read/written. This simulation setup evaluated the proper use of the APB configuration registers and the AHB master/slave interface. The simulation consisted of the configuration of a memory area, two fault configurations and a simple fault injection, while reading from the emulated slave which represents the memory. This fault injection consisted of four consecutive 8-bit reads, where after the fourth read, the fault is triggered. The fault configuration used an access count of 4 and a *Stuck-At-Zero* fault model. The fault mask was set to *0xFF*. An example of the configuration setup using the emulated master is listed in Listing 5.1. The example configures two registers on address *0x10000000* and *0x10000004*. The four reads performed on the bus to trigger the fault are shown in Listing 5.2. The simulation consisted of VHDL code only, so no software program was used to inject the faults. The time requirements of this simulation using the ModelSim simulator is shown in Table 5.4.

```
1 ahbwrite(x"10000000", x"00000104", "10", "10", '0', 1, false, ahbtbctrl);
  wait for clk_period;
3 ahbtbmidle(true, ahbtbctrl);
  wait for clk_period;
5 ahbwrite(x"10000004", x"000000FF", "10", "10", '0', 1, false, ahbtbctrl);
  wait for clk_period;
7 ahbtbmidle(true, ahbtbctrl);
  wait for clk_period;
```

Listing 5.1: Configuration of the fault registers using the methods provided by the Gaisler IP library

```
  ahbread(x"20000040", x"AAAAAAAA", "00", 2, false, ahbtbctrl);
2 ahbread(x"20000040", x"BBBBBBBB", "00", 2, false, ahbtbctrl);
  ahbread(x"20000040", x"CCCCCCCC", "00", 2, false, ahbtbctrl);
4 ahbread(x"20000040", x"DDDDDDDD", "00", 2, false, ahbtbctrl);
```

Listing 5.2: Four consecutive reads from the AHB slave using the methods provided by the Gaisler IP library

Attack Scenario using a C Program

The attack scenario using a C program uses the complete system as illustrated in Figure 4.5. All components are connected to the LEON3 using the AHB and APB bus system. Neither the memory nor the processor is emulated. The LEON3 is set up without a caching system and without a Memory Management Unit (MMU). This enhances the simulation speed and is not used for these small examples. This simulation was performed to see how the FIU reacts in a real environment whith real, simulated, components. This allowed to find all remaining problems with the implementation. The used test program written in C is listed in Appendix A.6. At the beginning, the used variables are defined. After that, the memory areas and fault configurations are set up and stored in the hardware. The setup of the hardware unit was done using the implemented driver for the FIU integration into the LEON3 system. After that, the values are read two times to evaluate that the FIU works with two consecutive setups. Once a trigger was triggered, it does not fire again. Only after a new configuration is written to the registers, the FIU injects a new fault. The time requirements of this simulation is shown in Table 5.4.

Simulation with SimpleRTJ

The last simulation scenario uses the same hardware setup as the scenario using the C program. The difference with this scenario is that it does not use a simple C program to test the functionality. The complete SimpleRTJ VM is included in the test setup and executed on the LEON3 in a simulated environment. Additionally, the FIU is integrated to configure the faults. The executed program is illustrated in Appendix A.7 and will be described later in Section 5.5 of the results chapter. This simulation is the nearest approach to a real fault injection using real hardware. The only difference is that the communication to the host is missing (not feasible/possible). However, the host communication was simulated using a buffer with the commands in the C program. The FIU reacts as if there was a real communication with an external system, bus does not really need a real connection. This was done by simply using a byte-buffer with the corresponding commands. The simulated command sequence is:

- **Reset Breakpoints** - At the start of the program, all breakpoints are reset. In a simulated environment, this is not necessary because the hardware is in a clear state after the start-up. This test program was also used with real hardware where it is required and therefore included.
- **Set Memory Area 1** - The first memory area was set up using the range of the local variables in the VM.
- **Set Memory Area 2** - The second memory area was set up using the range of the local variables in the VM.
- **Set Fault Configuration** - A fault setup is configured, using *Stuck-At-Zero*, an access count of 1 and the mask *0xFF*.
- **Set Break-Point** - Set the first break-point which is used to set up the fault configuration in the hardware. At this point the FIU is ready to inject faults.

Simulation Type	Simulated Time	Simulation Time
Emulated Master/Slave	142ns	524ms
C Program	10ms 363us 146ns	3min 7sec
SimpleRTJ Program	444ms 348us 971ns	1hours 50min 8sec

Table 5.4: Results of the Different Simulation Scenarios using ModelSim

Fault Test Type	Simulated Time	Simulation Time	Hardware Time
C Program	10ms 363us 146ns	3min 7sec	<1sec
SimpleRTJ Program	444ms 348us 971ns	1hours 50min 8sec	<1sec
Fault Campaign (Average)	$\approx 500ms$	$\approx 2hours$	$\approx 6.5sec$

Table 5.5: Results of the Time Measurements for Different Fault Setups using ModelSim and Real Hardware

- **Set Break-Point** - Sets a second break-point. This is used to evaluate if the injected fault triggered at the correct point. This break-point is then used to read out the modified variable.
- **Run** - Runs the program until both breakpoints have been reached.

The time results of this simulation are shown in Table 5.4.

5.4 Fault Injection Performance

Table 5.4 shows the simulation time for different use cases. The emulated master/slave approach can only be used in a simulated environment but is the only one feasible regarding the simulation time. The second approach can be used in hardware and as a simulated setup. The 3 minutes 7 seconds required for the simulation are still feasible, however, it does not include the complete system, including an executed Java program. The last setup which simulates the complete system, including a VM and a executed Java program, has a very long simulation time of nearly 2 hours. The time requirements are too high to efficiently test VM systems regarding faults. Table 5.5 shows the results using real hardware for the last two fault injection approaches. Additionally, a third approach which uses a fault campaign to test several fault injections consecutively is added to the table. For the fault campaigns, an average of 100 fault injection setups were used to get the size estimations. It can be seen that the speedup of the hardware execution is massive. With this setup, numerous faults can be set up and tested in an efficient way. The last test setup which uses a fault campaign requires a stable communication with the FPGA board. The communication and configuration using the serial interface is very slow and requires about 5 seconds of the overall test time. A test time decrease could be achieved by accelerating the speed of the communication link.

5.5 Attack Scenario

This section focuses on describing an attack scenario which uses the FIU components to introduce faults into the system. For this scenario, the hardware module, the FIU client and host are required. The Java program which is executed in the VM is listed in Appendix A.7. The scenario shown in the source listing is a simplified pin verification program. During this attack scenario, a fault will be introduced during a writer operation performed on the operand stack. When the method for the pin verification is called on line 16 of the source code, the return value is put onto the operand stack and a fault is introduced to the system. The basic steps to set up the fault injection from the FIU host are illustrated in Figure 5.2 and are straightforward. The main steps to setup and prepare the fault injection are:

- **Analyze the Java Code** - At the beginning, the source code of the program has to be analyzed. In this attack scenario, a simple pin verification is performed. Depending on the return value of the *validatePin* method, the amount stored in the application is decreased or not. In line 31 the *validatePin* method is called with the pin 4321. When the method is analyzed, it becomes visible that the pin 1234 is expected as the input. Therefore, in a normal scenario, the method will always return *false*. However, we want to return *true* at line 19 of the source. The next step is to analyze the byte-code at that line, to see what is executed.

- **Analyze the Byte Code** - After the code position, where the fault should be injected has been found, the byte-code is analyzed. The example shows the byte-code for the return statement. From Figure 4.20 it can be seen that the executed line consists of two byte-codes: *iconst_0* and *ireturn*. The interesting byte-code is *iconst_0*, which places a constant value of 0 onto the operand stack. The 0 represents the value of the *false*. To successfully introduce a fault and change the boolean return value of the method from *false* to *true*, a 1 instead of a 0 has to be put onto the operand stack. At this point the user knows, where he wants to inject a fault (*iconst_0* byte-code) and what the fault should do (replace the 0 with a 1). To collect all necessary data and perform a successful fault, an analyzation of the C code of the executed byte-code is required.

- **Analyze the C Code** - This step is not always required, depending on the user knowledge of the used VM. It is necessary to see how many memory accesses on the operand stack are performed during the execution of the byte-code. The byte-code executed for *iconst_0* in the SimpleRTJ VM is shown in Listing 5.3. This byte-code is very simple and only performs one write onto the operand stack. Each entry on the operand stack consists of a 32-bit value, and therefore 4 8-bit values are written at this point.

- **Setup Fault Routine** - Now, all required data is collected. To set up the fault in the FIU host, the user has to add a fault configuration and a fault injection point.

  - Memory Area - First, the user has to configure the used memory areas. Only one memory area is required in this fault scenario and has to be set to the operand stack.

  - Fault Configuration - For the fault configuration, the fault number, fault type, access count and mask are required. The fault number is an arbitrary value from 0 to 32, which represents a software fault configuration in the FIU client. The fault type can either be *Stuck-at-One* or *Negate Input*, since both of these fault types result in a 1 stored on the operand stack instead of the 0. Since only one access is performed during the byte-code execution, the access count is set to 1. The mask can be set to any uneven value (this always leads to the first bit to be 1 in the mask). In this example it is set to *0xFF*.
  - Fault Injection Point - For the fault injection point, the injection number, program count, pass count, the memory area and fault direction have to be set up. The injection number represents a software injection point in the FIU client (in most cases an ascending index number). The program count is known from the source code analysis and can be read using the GUI of the host. In this case it is 1249. The pass count can be set to 0, since the fault should be injected when the injection point is reached for the first time. The memory area is set to the memory area of the operand stack and the fault direction has to be set to *WRITE*.

- **Setup Evaluation** - Concerning the fault attack, it has to be evaluated if it was successful or not. Therefore, at some point of the program, the execution needs to be halted and a value has to be retrieved. This value can then be compared to a reference value. In this attack scenario, the amount variable is used. In a normal program execution, the *amount* variable should be 10 at the end of the program execution. In the attack scenario, this value should be changed to 9. The user can set an additional break-point at line 40 to halt the program execution. When this point is reached, the user can use an inspection configuration to read the variable. In the attack scenario, the *amount* variable is a local variable in slot 2. This information can be retrieved by looking into the „Class Information“ part of the GUI.

- **Run Program** - After everything is set up, the program can be executed. After each executed byte-code, the FIU client evaluates if an injection- or break-point is reached. So when the program counter reaches the byte-code of the return statement (program counter 1249), the FIU software module evaluates the fault configuration and configures the hardware unit. Therefore, the memory area and the fault configuration are stored in the hardware unit. After that, the C code for the byte-code is executed. Now the VM accesses the memory area of the operand stack and the hardware unit triggers a fault. Due to the fault setup, the value 0 is changed to 255. When the VM now reads this value from the operand stack, it is interpreted as a Boolean *true* instead of a *false*.

- **Evaluate** - When the second break-point is reached, the execution is halted and the host is notified. Now the user can use the inspection configuration to retrieve the value of the *amount* variable. When 9 is returned, the attack was successful. Otherwise the attack was unsuccessful and the user has to analyze the fault parameters.

```c
int16 _iconst(int16 val) {
  vm_sp->i = val;
  vm_sp++;
  return ACTION_NONE;
}
```

Listing 5.3: Executed C Code for the *iconst* Byte-Code

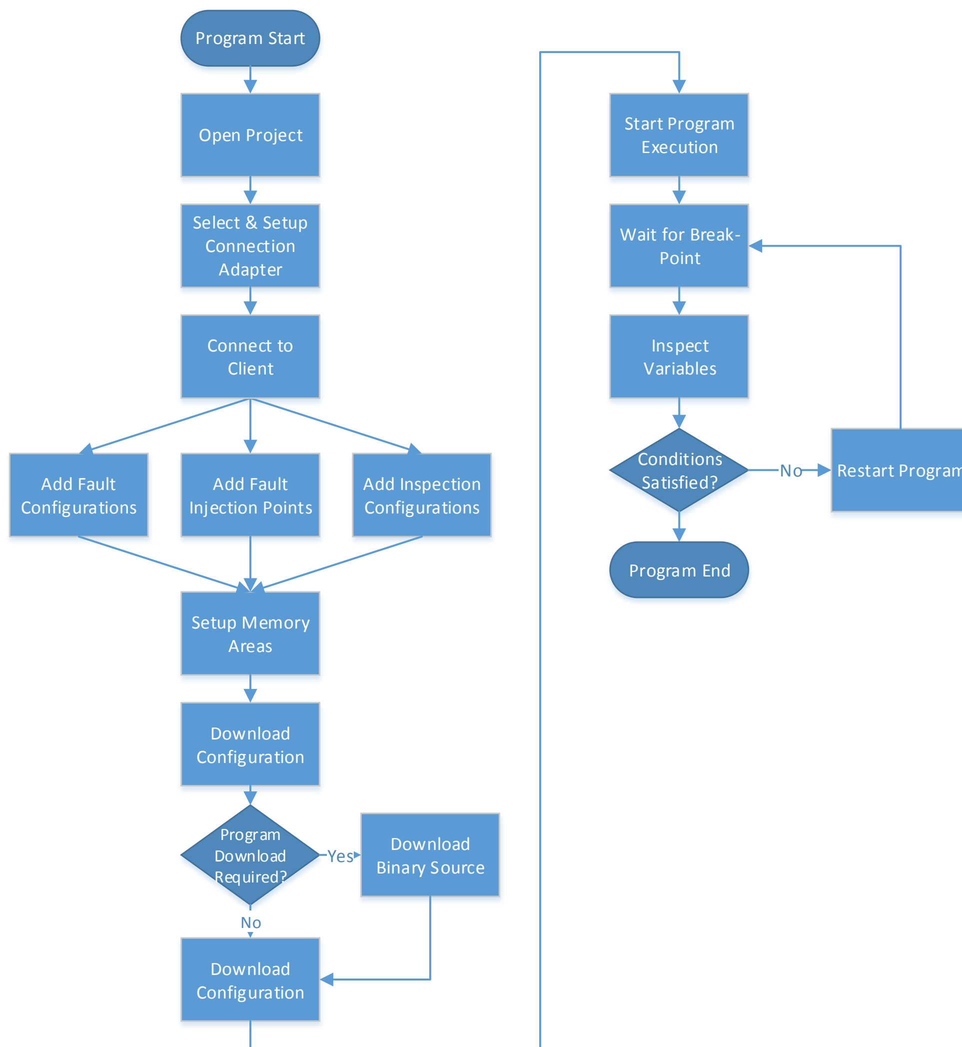


Figure 5.2: Example Workflow of a Fault Setup in the FIU Host

5.6 Drawbacks

During the implementation and evaluation of the FIU components, a problem occurred in combination with the LEON3 processor. Since the hardware unit is placed between the processor and the memory controller, the caching system and the MMU of the LEON3 were disabled. This was done to avoid cache accesses and to count all memory accesses. For write accesses performed on the memory, everything works as expected: When a 8-bit value is written by the processor, this 8-bit value is put onto the bus and stored into the memory. The same holds for 16- and 32-bit values. The FIU hardware unit can count all memory accesses according to the implemented bus standard.
When a read access is performed by the LEON3, the behavior of the bus access is not as imagined. For reasons unknown, every time a read (*ld*, *ldub*, etc.) is performed in the processor, the processor executed 8 consecutive 32-bit reads from the memory controller. This behavior occurs independent of the size of the read value (8, 16, 32 bit). When the processor executes a *ld* instruction from address *0x400ffee4*, all values from the address range *0x400ffee0* to *0x400ffefc* are read from the bus and therefore the memory controller. This leads to problems with the setup of the FIU hardware unit, since every one of these accesses is counted. Therefore, when this whole memory area is in range of the configuration of the hardware unit, the count is set to $8 * 4 = 32$. This leads to the issue that precise faults cannot be performed for read accesses. To counter these problems, fault campaigns can be used to evaluate different access counts.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Within this thesis, the design and implementation of a Java fault injection system was illustrated. A LEON3 system has been implemented, including the fault injection hardware module which injects the faults into the system and which is controlled by the software unit and the host PC.
At the beginning of this thesis, different designs towards a Java fault injection system were evaluated. The final conclusion was that a hardware/software co-design approach is the best solution due to its flexibility, portability and realism of fault simulations. This design was then implemented, consisting of three main parts: the hardware unit, the software unit and the host.
The hardware unit was designed to be used within a bus system. The system is intended to be used in a broad range of applications and different environments, and therefore a standardized interface has to be used. Depending on the used bus system, a bus controller needs to be implemented to get a working fault injection system.
The reference implementation was done on a LEON3 development board with a mounted Spartan-3 FPGA. For the hardware unit, the system AMBA bus was used: the APB for the configuration and the AHB as fault interface. The hardware unit is intended to inject faults on the bus between the LEON3 processor and the memory controller. Therefore, the hardware fault injection unit was placed between these two communication parts. With this setup, the hardware unit can be configured to inject faults when certain memory addresses are accessed. The software unit was integrated within the SimpleRTJ VM implementation. Therefore, several data providing functions had to be implemented which are used by the FIU software unit. Additionally, the VM had to be changed to be controllable by the software unit. The communication between the client and the host was implemented using the serial UART interface of the FPGA board.
Additionally, to the hardware and software unit, a host was implemented. Since the system is mostly intended to be used in embedded devices, an easy-to-use configuration interface had to be added. The host interface is intended to help the user to set up a fault scenario. This is done by providing useful features which allows browsing the source code and find the exact location of a certain byte-code where a fault should be injected. Simple mechanism to control the control flow of the executed Java program are also part of the

host. The host is implemented in Java, and uses the EclipseRCP technology to generate a user-friendly user interface.
To set up a fault scenario using the tool implemented in this thesis, several steps need to be performed. At first, the exact location, where the fault should be introduced in the system needs to be found. The host provides the user features to help with this task. When the location (program count) has been found, a new fault setup needs to be added to the system. This includes the fault type and an access count. This fault configuration is bound to a specific location in the executed Java program. When the VM running on the embedded device reaches this location in the code, the hardware unit is configured and observes the memory accesses to the memory from that point on. When the fault conditions are met (access count), the fault is triggered. When the Java program has been completed or is halted, the user can use an inspection feature provided by the host to evaluate if the fault was a success or not.
At the end of the thesis, the hardware overhead and performance of the fault injection framework was evaluated. The hardware overhead of the fault injection system hardly depends on the number of configuration registers and memory areas used by the hardware unit. Another factor increasing the hardware size was that the hardware unit was designed to work with zero-delay, so the original system is not altered in any way (regarding the clock cycles). The hardware overhead compared to the overall reference system was measured to be between 13% to 70%. When the system was compared, regarding the simulation time and the time required to execute on hardware, a speedup of approximately 6000 times was observed. Due to the high complexity of the complete system, simulation is not a sufficient medium to inject faults. At last, an attack scenario was shown which described how the complete system interacts with each other to inject a fault into the system.

6.2 Future Work

1. **Thread Support** - The current implementation of the FIU does not support threads. Since this system is mainly intended to be used with Java Card systems, this feature was not that relevant for the implementation. However, it may be supported in the future.

2. **Other Bus System Support** - Currently, only the AMBA AHB system is supported by the system. To reuse the system in an easy way, other bus systems could be supported. Therefore, a new bus controller needs to be implemented.

3. **Other VM Support** - Currently, the only supported VM is SimpleRTJ. This VM is not a Java Card VM, but rather a subset of the JVM standard. Other VM support would require additional data providers.

4. **Enhanced Communication Interface** - The currently selected communication interface is the serial port. Since this is not a very fast communication interface, the main time during a fault injection is required to transfer the data to the embedded device. This is especially true for small programs which have a short execution time. Additional communication interfaces (e.g. Ethernet) could enhance the fault performance.

5. **Extended Campaign Scripting Interface** - The scripting interface for the fault campaigns is currently very simple. Only the basic functionality is provided. An extended scripting interface could enhance the possibilities of the fault campaigns.

6. **Automatic Placement** - Other VHDL base fault injection frameworks support automatic placement of saboteur units. However, these frameworks focus on single-line saboteur placement. The algorithm is quite easy, searching the VHDL code for signals where saboteurs can be placed. Using a similar approach for the FIU proposed in this thesis would be rather complex, since bus systems and connections need to be searched in the code.

Appendix A

Code Examples

A.1 Client Socket Communication

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <netdb.h>

#include <conf.h>
#include <fiu_comm.h>
#include <fiu_comm_socket_conf.h>

int sockfd;
int portno = FIU_COMM_PORT;
struct sockaddr_in serv_addr;
struct hostent *server;

bool init_communication() {
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0) {
    printf("ERROR opening socket\n");
    return 0;
  }

  server = gethostbyname("localhost");
  if (server == NULL) {
    fprintf(stderr, "ERROR, no such host\n");
    return 0;
  }
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  bcopy((char *) server->h_addr, (char *) &serv_addr.sin_addr.s_addr,
      server->h_length);
  serv_addr.sin_port = htons(portno);
```

```
    if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) <
      0) {
      printf("ERROR connecting\n");
      return 0;
    }

    return 1;
}

void close_communication() {
    close(sockfd);
}

uint32 transceive(uint8* req, uint32 req_length, uint8* resp,
      uint32 resp_length) {
    transmit(req, req_length);
    return receive(resp, resp_length);
}

uint32 receive(uint8* resp, uint32 req_length) {
    return read(sockfd, resp, sizeof(uint8) * req_length);
}

uint32 transmit(uint8* resp, uint32 req_length) {
    return write(sockfd, resp, sizeof(uint8) * req_length);
}

bool is_data_available() {
    int count;
    ioctl(sockfd, FIONREAD, &count);

    return count > 0;
}
```

Listing A.1: Example Implementation of an FIU Client Communication Interface for Unix Sockets

A.2 Server Socket Communication

```
public class FIUSocketServer extends AbstractStreamServer {

    private final static String PROPERTY_KEY_PORT = "port";

    private ServerSocket serverSocket;
    private Socket clientSocket;

    private int port;

    public FIUSocketServer() {
        super();
    }

    @Override
    public String getName() throws IOException {
```

```
16     return "Socket Server";
   }
18
   @Override
20 public String getDescription() throws IOException {
     return "Server which uses TCP/IP sockets to communicate with the fault
    injection unit client";
22 }

24 @Override
   protected void connectToClient() throws IOException {
26   serverSocket = new ServerSocket(port);

28   log.info("Waiting for client to connect...");
     clientSocket = serverSocket.accept();
30   log.info("Server initialized. Socket connected.");

32   setInStream(clientSocket.getInputStream());
     setOutStream(clientSocket.getOutputStream());
34 }

36 @Override
   public Map<String, Object> getRequiredProperties() {
38   Map<String, Object> props = new HashMap<String, Object>();
     props.put(PROPERTY_KEY_PORT, 6666);
40   return props;
   }
42
   @Override
44 public void setProperty(String name, String value) {
     if (name.equals(PROPERTY_KEY_PORT))
46     port = Integer.valueOf(value);
   }
48
   @Override
50 protected void initializeServer() throws IOException {
   }
52
   @Override
54 protected void disconnectFromClient() throws IOException {
     super.disconnectFromClient();
56
     if (clientSocket != null)
58     clientSocket.close();

60   if (serverSocket != null)
       serverSocket.close();
62 }

64 }
```

Listing A.2: Example Implementation of an FIU Host Communication Interface for Sockets

A.3 FIU Hardware Register Usage

A.4 Fault Injection Unit SimpleRTJ Provider

```
uint8* APP_NAME = (uint8*) "DEBUG_APP        ";

void fiuFillSynchResponse(fiu_resp_synch_t* resp) {
  resp->heap = (uint32) ABS_LOC(config->heap_start);
  resp->refs = config->references;
  resp->strings = config->strings;
  resp->threads = config->threads;
  resp->app_start = (uint32) ((uint32*) config->app_start);
  resp->time_slice = config->time_slice;
  resp->timers = config->timers;
  resp->events = config->events;
  resp->unicode = ENABLE_UNICODE;
  resp->cl_ver = SW_VER;

  copyMem(resp->app_name, APP_NAME, 9);
}
void fiuFillMemstat(fiu_resp_memstat_t* resp) {
  resp->heap = (uint32) ABS_LOC(config->heap_start);
  resp->refs = config->references;
  resp->strings = config->strings;
  resp->threads = config->threads;
}
void fiuSetProgramMemory(uint32 offset, uint32 length, uint8* data) {
  copyMem(config->app_start + offset, data, length);
}
bool isPCReached(uint32 brkPC) {
  if (CONVERT(REL_LOC(vm_pc)) == brkPC)
    return true;
  return false;
}
void getCurrentOSMemoryRange(fiu_memrange_t* range) {
  frame_t* curr_frame = thr_active-> curr_frame;

  uint32 sp_size = frame_size - curr_frame->method->locals - FRAME_HDR_SIZE;
  range->addrlo = (uint32) (curr_frame + curr_frame->method->locals +
    FRAME_HDR_SIZE);
  range->addrhi = range->addrlo + sp_size;
}
void getCurrentLVMemoryRange(fiu_memrange_t* range) {
  uint32 locals = thr_active->curr_frame->method->locals;

  range->addrlo = (uint32) &(thr_active->curr_frame->locals[0]);
  range->addrhi = ((uint32) &(thr_active->curr_frame->locals[locals])) +
    sizeof(value_t);
}
void getCurrentPCMemoryRange(fiu_memrange_t* range) {
  range->addrlo = ((uint32) thr_active->curr_frame) + sizeof(frame_t*) +
    sizeof(value_t*);
  range->addrhi = ((uint32) thr_active->curr_frame) + sizeof(frame_t*) +
    sizeof(value_t*) + sizeof(uint8*);
}
```

```
void getCurrentBAMemoryRange(fiu_memrange_t* range) {
  range->addrlo = (uint32) config->app_start;
  range->addrhi = (uint32) (config->app_start + FIU_BA_SIZE);
}
uint32 inspectLocalVariable(uint32 object, uint16 slot) {
  return thr_active->curr_frame->locals[slot].val;
}
uint32 inspectStaticVariable(uint32 object, uint16 slot) {
  return static_start[slot].val;
}
uint32 getFIUPC() {
  return (uint32) REL_LOC(vm_pc);
}
```

Listing A.3: Example Implementation of an FIU Data Provider for SimpleRTJ

A.5 Fault Campaign Sekelton

```
function setupFaultRun(runSetup, confIteration) {
  runSetup.configurations = 1;
  runSetup.injectionPoints = 2;
}

function setupFaultConfiguration(numConf, conf, confIteration) {
  conf.number = numConf;
  conf.accessCount = confIteration + 1;
  conf.mask = 255;
  conf.faultType = 0;
}

function setupFaultPoint(numPoint, point, confIteration, pointIteration) {
  if (numPoint == 0) {
    point.number = numPoint;
    point.pc = 1306;
    point.passCount = 0;
    point.memareaConfNum = 0;
    point.haltExecution = false;
    point.configureFIU = true;
    point.direction = "WRITE";
    point.memarea = 0;
    point.faultConfiguration = 0;
  } else if (numPoint == 1) {
    point.number = numPoint;
    point.pc = 1318;
    point.passCount = 0;
    point.memareaConfNum = 0;
    point.haltExecution = true;
    point.configureFIU = false;
  }
}

function setupInspectionConfiguration(inspection, confIteration,
    pointIteration) {
  inspection.inspectionType = "LOCAL";
```

```
  inspection.object = 0;
  inspection.slot = 2;
}

function assertInspectionValue(value, confIteration, pointIteration) {
  return value == 10;
}
```

Listing A.4: Skeleton Script for a Fault Campaign

A.6 C Program Testing Fault Injection

```
int main() {
  printf("Starting FIU test\n");

  uint8 value1 = 0;
  uint8 value2 = 0;
  uint8 value2_1 = 0;
  uint32 value3 = 0;
  uint32 value4 = 0;
  uint32 value5 = 0;

  fiu_configuration_t fiu_conf = { FIU_HW_BASE_ADDRESS, FIU_HW_MEMAREAS,
   FIU_HW_CONF_PER_MEMAREA };
  fiu_hw_init(&fiu_conf);

  fiu_memarea_t memarea1 = { ((uint32) & value1) + sizeof(uint8), (uint32) &
    value5 };
  fiu_memarea_t memarea2 = { ((uint32) & value1) + sizeof(uint8), (uint32) &
    value5 };

  // Setup memory areas
  fiu_store_memarea_configuration(&memarea1, 0);
  fiu_store_memarea_configuration(&memarea2, 1);

  // Setup fault configurations
  fiu_config_t conf1;
  fiu_config_t conf2;
  fiu_config_t conf3;
  fiu_config_t conf4;

  fiu_set_accesscount(13, &conf1);
  fiu_set_faulttype(FAULT_TYPE_NEGATE_INPUT, &conf1);
  fiu_set_mask(255, &conf1);

  fiu_set_accesscount(14, &conf2);
  fiu_set_faulttype(FAULT_TYPE_NEGATE_INPUT, &conf2);
  fiu_set_mask(255, &conf2);

  fiu_set_accesscount(15, &conf3);
  fiu_set_faulttype(FAULT_TYPE_NEGATE_INPUT, &conf3);
  fiu_set_mask(255, &conf3);

  fiu_set_accesscount(16, &conf4);
```

```
    fiu_set_faulttype(FAULT_TYPE_NEGATE_INPUT, &conf4);
    fiu_set_mask(255, &conf4);

    value1 = 123;
    value2 = 152;
    value2_1 = 12;

    uint8 *value1p = &value1;
    uint8 *value2p = &value2;
    uint8 *value2_1p = &value2_1;

    // Storing configurations
    fiu_store_configuration(&conf1, 0, 0, 1);
    fiu_store_configuration(&conf2, 0, 1, 1);
    fiu_store_configuration(&conf3, 1, 0, 1);
    fiu_store_configuration(&conf4, 1, 1, 1);

    // Reading values
    printf("Value1: %d; Expected: 123\n", *value1p);
    printf("Value2: %d; Expected: 152\n", *value2p);
    printf("Value2_1: %d; Expected: 12\n", *value2_1p);

    // Storing configurations
    fiu_store_configuration(&conf1, 0, 0, 1);
    fiu_store_configuration(&conf2, 0, 1, 1);
    fiu_store_configuration(&conf3, 1, 0, 1);
    fiu_store_configuration(&conf4, 1, 1, 1);

    // Reading values second time
    printf("Value1: %d; Expected: 123\n", *value1p);
    printf("Value2: %d; Expected: 152\n", *value2p);
    printf("Value2_1: %d; Expected: 12\n", *value2_1p);

    return 0;
}
```

Listing A.5: Test Program used to evaluate the FIU using a C Program

A.7 Java Program for Fault Testing

```
import javax.events.*;
import javax.memory.*;

class ValidationProj extends Thread
{
    static String CORRECT_PIN_STRING = "letMeEnter";

      static String newLineString = System.getProperty("line.separator", "\r\n
      ");
      static byte[] newLineBytes = newLineString.getBytes();

      static void main(String[] args)
      {
          new ValidationProj().start();
```

```
14    }

16  public boolean validatePin(int pin) {
      if (pin == 1234)
18      return true;
      return false;
20  }

22  public void exception() {
    }
24
      public void run()
26    {
      int pinEntry = 4321;
28    int amount = 10;
      boolean pinValid;
30
      if (validatePin(pinEntry)) {
32      amount = amount - 1;
      } else {
34      exception();
      }
36
      println("New Amount: " + amount);
38    }

40    static int Count = 0;
      static void println(String s)
42    {
          print(s);
44        print(newLineString);
          Count++;
46    }
      static native void print(String s);
48 }
```

Listing A.6: Test Program written in Java used to evaluate the FIU Integration into SimpleRTJ

Glossary

communication interface is used as a term to describe the implementation for a specific communication port. In this context it is mostly referenced to in software description to separate between the physical port and a software interface implementation for this port. 38, 68, 69, 72, 78, 79

communication port references a physical communication port. This can either be a Ethernet adapter or a serial port. 38, 68, 72, 78, 79

fault campaign consists of several fault pattern. They are used to test a hardware system with a large amount of faults. In general a fault campaign should evaluate a system of its fault tolerance.. 30, 33, 85

fault condition is a point where a fault is introduced into the system. A fault condition could be specified by time or other conditions, as, for example, a memory access at a specified memory area. 12

memory area is a pre-defined range in a memory. The memory area defines the start address and end address of a region in the memory. 43

zero-delay is used to describe operations which happen in one cycle. This term is only used during the hardware description and means that the no additional clock cycles are required for the operation. 13, 41, 44, 57, 60

Acronyms

AHB Advance Hi-Performance Bus. 49, 52, 55, 57, 58, 61, 64, 66, 83, 84, 90, 91

AMBA Advanced Microcontroller Bus Architecture. 37, 55, 58, 63, 64, 90, 91

APB Advanced Peripheral Bus. 49, 52, 55, 61–64, 67, 80, 83, 84, 90

BCC Bare C Compiler. 49

DSU Debug Support Unit. 52

DUT Device Under Test. 32, 43

EEPROM Electrically Erasable Programmable Read-Only Memory. 15

FIU Fault Injection Unit. 7, 15, 35–38, 41, 43–47, 49, 50, 53–55, 58–75, 77, 79, 80, 82–84, 86–92

FPGA field-programmable gate array. 13, 29, 30, 47, 49, 63, 75, 80, 82, 85, 90

FTM Fault Tolerance Mechanism. 26

GPIO General Purpose Input/Output. 30

GUI General User Interface. 76, 77, 87

ICC integrated circuit card. 10

J2SE Java Platform Standard Edition. 50

JCRE Java Card Runtime Environment. 10

JDK Java Development Kit. 80

JRE Java Runtime Environment. 80

JTAG Joint Test Action Group. 49, 52

JVM Java Virtual Machine. 46, 50, 91

MFI Modular Fault Injector. 7, 30–33

MMU Memory Management Unit. 84, 89

OS Operating System. 1, 2, 49, 54

PC Personal Computer. 50

RCP Rich Client Platform. 50

RISC Reduced Instruction Set Computer. 53

RTL Register Transfer Level. 1, 12, 13, 18

SDCC Small Device C Compiler. 49

SE secure element. 10

SimpleRTJ Simple Real Time Java. 49, 54, 55, 75, 83, 84, 86, 90, 91

SPARC Scalable Processor ARChitecture. 49, 53

SW Software. 40

UART Universal Asynchronous Receiver Transmitter. 72, 90

USB Universal Serial Bus. 52

VHDL Very High Speed Integrated Circuit Hardware Description Language. 17, 19, 22, 23, 26, 27, 29, 31–33, 49, 52, 55, 80, 83, 92

VM virtual machine. 12, 13, 15, 16, 32, 33, 36, 37, 45, 49, 54, 67–69, 72–76, 78, 80, 84–87, 90, 91

XML Extensible Markup Language. 76

Bibliography

[Apa] Apache. Project homepage of apache maven. http://maven.apache.org/. Accessed: 2014-01-09.

[ARM] ARM. AMBA Bus Specifiation. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php. Accessed: 2014-01-08.

[BDH11] G. Barbu, G. Duc, and P. Hoogvorst. Java Card Operand Stack:Fault Attacks, Combined Attacks and Countermeasures. In E. Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 297–313. Springer Berlin Heidelberg, 2011.

[BECN+06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BGB+08] J.-C. Baraza, J. Gracia, S. Blanc, D. Gil, and P.-J. Gil. Enhancement of Fault Injection Techniques Based on the Modification of VHDL Code. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(6):693–706, 2008.

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

[BGGG05] J.-C. Baraza, J. Gracia, D. Gil, and P. Gil. Improvement of fault injection techniques based on VHDL code modification. In *High-Level Design Validation and Test Workshop, 2005. Tenth IEEE International*, pages 19–26, 2005.

[BICL11] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In E. Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 283–296. Springer Berlin Heidelberg, 2011.

[BPC98] J. Boue, P. Petillon, and Y. Crouzet. MEFISTO-L: a VHDL-based fault injection tool for the experimental assessment of fault tolerance. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 168–173, 1998.

[BTG10] G. Barbu, H. Thiebeauld, and V. Guerin. Attacks on java card 3.0 combining fault and logical attacks. In *Proceedings of the 9th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Application*, CARDIS'10, pages 148–163, Berlin, Heidelberg, 2010. Springer-Verlag.

[CI03] H. C. Chien-In. Behavioral test generation/fault simulation. *Potentials, IEEE*, 22(1):27–32, 2003.

[Com] RTJ Computing. Homepage of simplertj. http://www.rtjcom.com/. Accessed: 2014-01-08.

[ECLa] Project homepage of eclipse rcp. http://www.eclipse.org/home/categories/rcp.php. Accessed: 2014-01-09.

[Eclb] Eclipse. Eclipse Typcho Project. http://eclipse.org/tycho/. Accessed: 2014-01-09.

[Gaia] Aeroflex Gaisler. GRLIB IP Core Users Manual. Version 1.3.0 - B4133, July 2013.

[Gaib] Aeroflex Gaisler. Homepage of aeroflex gaisler. http://www.gaisler.com/. Accessed: 2014-01-08.

[Gaic] Aeroflex Gaisler. Homepage of pender electronics. http://www.pender.ch/. Accessed: 2014-01-09.

[Gaid] Aeroflex Gaisler. LEON3 GR-XC3S-1500 Template Design. October 2006.

[Gai01] Gaisler Research. *The LEON Processor Users Manua*, 2001.

[GGBG03] D. Gil, J. Gracia, J.C Baraza, and P.J Gil. Study, comparison and application of different vhdl-based fault injection techniques for the experimental validation of a fault-tolerant system. *Microelectronics Journal*, 34(1):41 – 51, 2003. ¡ce:title¿Special Section on Defect and Dault Tolerance in {VSLI} Systems (DFT)¡/ce:title¿.

[GKS^{+}11] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, and J. Haid. Modular Fault Injector for Multiple Fault Dependability and Security Evaluations. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 550–557, 2011.

[GNU] GNU. Homepage of gnu gcc. http://gcc.gnu.org/. Accessed: 2014-01-08.

[Gnu13] Gnu. *Using the GNU Compiler Collection*, 2013.

[Gra] Mentor Graphics. Modelsim se 6.5. http://www.model.com/. Accessed: 2014-01-09.

[IR86] R. K. Iyer and D. J. Rossetti. A Measurement-based Model for Workload Dependence of CPU Errors. *IEEE Trans. Comput.*, 35(6):511–519, June 1986.

[ITI] ITI. Homepage of the Institute of Technical Informatics. https://www.iti.tugraz.at. Accessed: 2014-01-10.

[Koc96] P.-C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[LBH+13] M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger. A Defensive Java Card Virtual Machine to Thwart Fault Attacks by Microarchitectural Support. In *International Conference on Risks and Security of Internet and Systems (CRiSIS)*. in press, 2013.

[LBL+13] M. Lackner, R. Berlach, J. Loinig, R. Weiss, and C. Steger. Towards the Hardware Accelerated Defensive Virtual Machine Type and Bound Protection. In S. Mangard, editor, *Smart Card Research and Advanced Applications*, volume 7771 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2013.

[MMLC11] J.-B. Machemie, C. Mazin, J. Lanet, and J. Cartigny. SmartCM a smart card fault injection simulator. In *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on*, pages 1–6, 2011.

[MP08] W. Mostowski and E. Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In G. Grimaud and F.-X. Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008.

[Ora11a] Oracle. *Java Card Platform - Runtime Environment Edition*, 2011.

[Ora11b] Oracle. *Java Card Platform - Virtual Machine Specification, Classic Edition*, 2011.

[Pey99] Patrice Peyret. *Java Card Technology for Smart Cards - Architecture and Programmer's Guide*. Pearson Education, 1999.

[RXT] RXTX. Homepage of rxtx project. http://rxtx.qbang.org/. Accessed: 2014-01-20.

[SDC] Project homepage of sdcc. http://sdcc.sourceforge.net/. Accessed: 2014-01-09.

[STB97] V. Sieh, O. Tschache, and F. Balbach. VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 32–36, 1997.

[Ste05] Craig Steiner. *The 8051/8052 Microcontroller: Architecture, Assembly Language, And Hardware Interfacing*. Universal Publishers, 2005.

[Ver06] O. Vertanen. Java type confusion and fault attacks. In L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography*, volume 4236 of *Lecture Notes in Computer Science*, pages 237–251. Springer Berlin Heidelberg, 2006.

[VF10] E. Vetillard and A. Ferrari. Combined Attacks and Countermeasures. In D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 133–147. Springer Berlin Heidelberg, 2010.

[vMA] vMAGIC. vmagic vhdl source parser. http://sourceforge.net/projects/vmagic/. Accessed: 2014-02-09.

[Xil] Xilinx. Xilinx ise 14.2. http://www.xilinx.com/. Accessed: 2014-01-09.