Master's Thesis

Design and Implementation of a Reader/Smartcard Fault Emulation Framework

Daniel Kroisleitner

Institut für Technische Informatik
Technische Universität Graz

Assessor:	Steger Christian Ass. Prof. Dipl. -Ing. Dr. techn.
Advisor:	Steger Christian Ass. Prof. Dipl. -Ing. Dr. techn.
	Druml Norbert Dipl. -Ing. BSc

Graz, im Mai 2013

Kurzfassung

RFID und dessen Nachfolger Technologien erobern mehr und mehr das alltägliche Leben. Ein wichtiger Schlüsselfaktor für den Erfolg neuer Technologien ist das Vertrauen der Anwender. Der Benutzer erwartet sich eine hohe Verfügbarkeit, Sicherheit und die Wahrung der Privatsphäre. Diese Anforderungen müssen schon während der frühen Phasen des Designs eines Systems berücksichtigt werden. Ein wichtiges Werkzeug dafür sind Simulationen. Ein Nachteil von Simulationen ist der hohe Zeitaufwand. Speziell bei komplexen Projekten sind umfassende Simulationen nicht mehr möglich. Um diesen Nachteil zu umgehen wird immer häufiger auf Emulation zurückgegriffen. Diese Arbeit beschäftigt sich mit der Entwicklung eines Frameworks zur Analyse und Verifikation von Software für RFID/NFC Systemen. Das Framework stellt dafür Modelle für Reader, Karte und der kontaktlosen Schnittstelle zwischen diesen beiden Komponenten zur Verfügung. Zusätzlich zur funktionalen Analyse stellt diese Emulationsumgebung einen Mechanismus zur Injektion von Fehlern zur Verfügung. Ebenfalls inkludiert ist ein Modul zur Leistungsabschätzung. Das Framework wird mit Hilfe eines FPGA Entwicklungsboard realisiert (XILINX ML605 - VIRTEX6).

Keywords:
NFC, RFID, Emulation, Wireless Communication, Kryptographie, Power Profiling, Power Aware Computing, Prototyping Platform

Abstract

RFID based technology conquers more and more every days life. An important fact for the success of a technology is that people have confidence in that technology. The user expects high reliability, security against fraud and privacy maintenance. Therefore, reliability and security aspects must be considered during the design of such applications. Various tools have been developed to enable system verification at early design steps. Simulations have become an important factor for design evaluation. Though a common drawback of simulations is the high time effort. Especially for huge projects, simulations are often not feasible. Therefore, emulation based approaches are needed. The goal of this work is the development of an emulation framework focusing fault injection and power emulation. The proposed framework includes models for reader/card and the contactless communication channel for the emulation of RFID based systems. Additional to the functional verification, information about the power consumption of the system can be gained. Another important feature of the proposed framework is the ability to inject faults into the system to emulate fault attacks and/or the occurrence of natural faults (e.g., due to radiation sources in the environment). The whole project is implemented on one FPGA (XILINX ML605 - VIRTEX6).

Keywords:
NFC, RFID, Emulation, Wireless Communication, Cryptography, Power Profiling, Power Aware Computing, Prototyping Platform

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

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)

Acknowledgements

First I would like to thank Ass. Prof. Dipl. -Ing. Dr. techn. Christian Steger and Dr. Josef Haid for enabling this thesis. I would also like to thank Norbert Druml and Manuel Menghin for the assistance and support during the design an implementation of the work.
Finally, I would like to thank my family for the aid over the course of my academic studies.

Graz, March 2013 Daniel Kroisleitner

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Contactless technology conquers more and more every days life. RFID based applications are used in passports, for access control (e.g., access to buildings, public transit), payment, etc. Especially contactless payment systems are of great interest.The expectations to this technology are faster and more comfortable ways to pay, the improvement of existing technologies (e.g., RFID tags instead of traditional product codes) and even higher levels of security. Especially the combination of mobile phones and Near Field Communication (NFC) promises various new applications (Figure 1.1).

Figure 1.1: Use cases of the NFC technology. [31]

An important fact for the success of a technology is that people have confidence in that technology. Therefore it is important that these applications are resistant against attacks

and that their reliability is high (the possibility of faults is low). The user expects high reliability, security against fraud and privacy maintenance. Therefore, reliability and security aspects must be considered during the design of such applications. Simulations and emulation based system evaluation are important tools to meet these goals.

Simulations are a key factor during the developing of a system. These tools cover a broad spectrum, enabling functional analysis, verification, power simulations, etc. Depending on the type of the simulated system (e.g., software, hardware, mechanics) and the level of abstraction, the simulation effort can be enormous.
Regarding embedded systems where the complexity is steadily increasing over the years, traditional simulations become more and more infeasible. Especially system simulation on a very low level of abstraction (e.g., gate-level simulations) can cover only certain aspects of the system behavior due to its high time effort. Considering that power analysis and fault analysis require a low-level view on the system makes it almost impossible to simulate larger software projects. This restriction is unacceptable if the whole system setup (e.g., a communication protocol) should be analyzed.
Hardware accelerated emulation can bridge this gap between accurate results and low time effort. The use of FPGA prototyping platforms promises a good solution for functional, power and fault analysis of embedded systems.

Regarding RFID systems where the setup consists of several components (reader, card, tolling infrastructure, etc.), separate simulation of the individual units are often insufficient. In order to analyze such a system, the interaction between those components must be considered rather than testing these units independent from each other. Providing a framework supporting the emulation of these units under authentic conditions enables a detailed view on the performance and possible vulnerabilities of the system through the whole developing process.

1.2 Goals of this Thesis

The goal of this work is the development of a reader-smartcard emulation framework focusing on fault injection and power emulation. The project is part of META[:SEC:].[1]
Targeting RFID enabled systems, the common setup consists of a reader device and a RFID card/tag. Additionally, the RF channel must be considered if the whole system should be analyzed. Using simulations, the evaluation of such a setup can be challenging and the time effort is high. An emulation based approach is therefore advantageous for the verification of the whole setup.
Additional to the functional evaluation of the system, also the power consumption of

[1] Mobile Energy-efficient Trustworthy Authentication Systems with Elliptic Curve based SECurity. Funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract 829586.

the system is of interest. Especially for passive RFID devices which are powered by the field of the reader, information about the power consumption is from great importance. Correct functionality of such devices is only possible if the power consumption does not exceed the supplied power. Another topic of interest is the behavior of the system under the presence of faults. Faults can be introduced into a system either due to natural sources or intentional by an attacker hoping that the device reveals its secret information. Especially for applications where system failure would lead to high costs (e.g., payment applications) the analysis of the system under the presence of faults is from great importance. Therefore, power and fault emulation are an essential part for system evaluation.

The goal of this thesis is to provide a RFID/NFC emulation framework for software verification. Functional verification should be possible as well as the analysis of the application under the presence of hardware faults. Additional to the functional verification and the fault analysis the power consumption of the system should be estimated. Combining functional verification, fault and power analysis into one framework enables a detailed view on the system during the design of such applications.

According to the requirements the proposed framework supports the following main features.

- Emulation of a reader-smartcard setup concerning an active reader, a passive RFID card/tag and the RF channel.
- Emulation of the power consumption of the system.
- Fault emulation to verify the consequences of faults introduced into the system either by natural sources or during an attack.

1.3 Structure of this Work

Chapter 2 gives an introduction to related topics. Section 2.1 deals with side channel attacks. Especially the concept of Differential Power Analysis is explained which is a powerful method to observe the internal state of a device and can therefore be used to gain secret data stored in a system. Section 2.2 gives an overview about the reasons of faults and the thematic of fault injection where Section 2.3 deals with the simulation/emulation of faults in systems. Section 2.4 gives a short introduction to the topic of power emulation. The chapter concludes with a selection of related projects. Chapter 3 and Chapter 4 deal with the design and implementation of the emulation framework. The results are presented in Chapter 5. The work concludes with Chapter 6 providing an outlook and ideas for future extensions.

2 Related Work and Theory

2.1 Side Channel Attacks

2.1.1 Introduction

Side Channel Attacks are methods to gain restricted information about a system during a noninvasive attack. The term side channel refers to the ability to observe characteristics of a system which carries information. With such an information an attacker might be able to gain secret data. Carrier of such information can be the power consumption, electromagnetic emanation or the timing behavior of the system.

The target of such an attack is for example the retrieval of the secret key of a cryptographic system or to get information about the used algorithm of a system.

2.1.2 Power Analysis

The power consumption of a device can leak information about the current operations being performed or even about data being processed on the device. This fact can be used to attack a cryptographic system by measuring the power consumption during cryptographic computation.
Figure 2.1 shows two possibilities to measure the power consumption of a device [23]. The first possibility is to place a resistor R_{scope} between the V_{ss} pin and the ground line. The current moving through R_{scope} creates a voltage. If the power consumption is high, the higher current creates a higher voltage (higher value of V_{scope}). If the power consumption is low, the lower current creates a lower voltage (lower value of V_{scope}). Therefore, measuring V_{scope} with an oscilloscope gives information about the power consumption of the device under attack (DUA).
The second possibility is to place a resistor R_{scope} between the V_{cc} pin and the power supply. If the power consumption is high, the higher current creates a higher voltage at R_{scope} which leads to a lower voltage V_{scope}. If the power consumption is low, the lower current creates a lower voltage at R_{scope} which leads to a higher voltage V_{scope}. Therefore, a smaller value of V_{scope} indicates higher power consumption and vice versa.

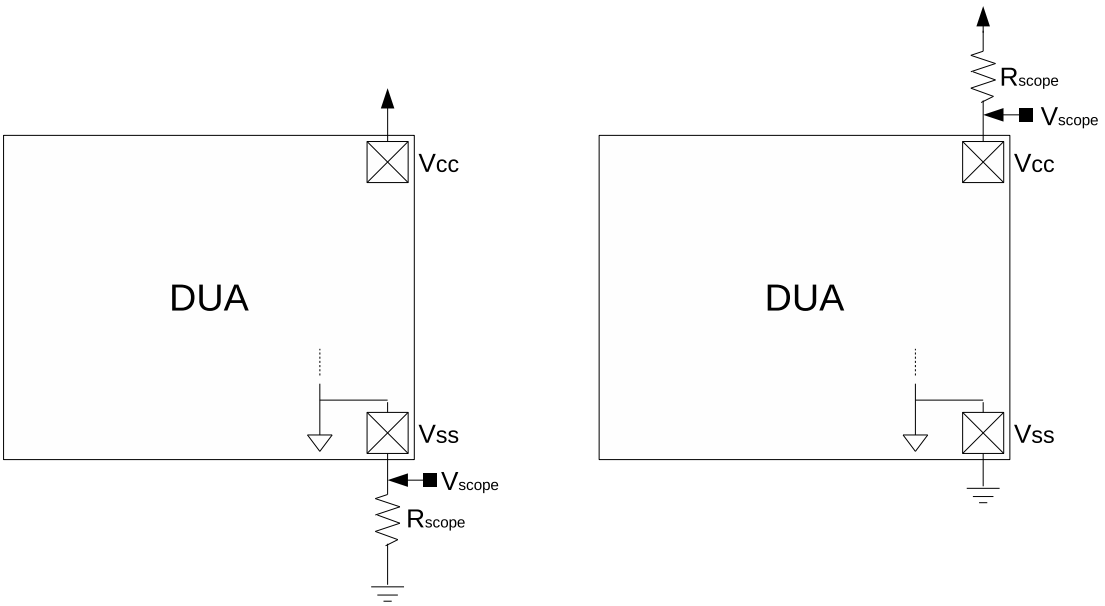


Figure 2.1: Measuring the power consumption of a device using a resistor and an oscilloscope.[23]

Simple Power Analysis

Simple Power Analysis (SPA) is a technique where measurements of the power consumption of a DUA are directly interpreted [25]. Because different operations consume different amount of energy, SPA can be used to identify these operations. Using SPA, it is also possible to get information about data being processed (e.g., the secret key of a cryptographic algorithm).
As shown in [23], the power trace pattern of a 3DES operation can be identified. If an attacker does not know which cryptographic operation is used, such additional information can be helpful. Another example for SPA stated in [23], is the attack on the modular exponentiation of the RSA algorithm. The modular exponentiation is performed using the binary left-to-right algorithm. Because of the different power consumption of the squaring operation and the multiplication and because the multiplication is only performed when a bit of exponent is '1', the secret exponent can be found by analyzing the power trace. The value of the exponent can be clearly identified in the power trace.
Because SPA needs strong data-dependent features in the power trace, SPA is not practicable for situations where cryptographic operations are hidden in noise [23]. To overcome this problem Differential Power Analysis (DPA) can be used.

Differential Power Analysis

DPA is a powerful method to gain secret information from a set of power traces using statistical methods. The idea of this method is that the power consumption is influenced by the processed data and that information about this data can be gained using correlation between different measurements.
Kocher et al. [23] describe an experiment where the basic principle of statistical methods in power analysis is shown. The example uses several power traces captured during an AES-128 operation on a smartcard. The traces were generated using randomly chosen plain text input for every trace. Only power values from a specific time instance are used which means that one power value per trace is required. For the experiment, the moment when the device computes the output of the first S-box was chosen. Depending on the LSB bit of the result, two subsets of power traces were built. For both subsets, the distribution of the power values at the specific time event was computed. The result shows that the distribution of both subsets is significantly different [23]. This shows that the power consumption correlates to the LSB of the S-box computation. The results in [23] also show that these distributions overlap, which means that single measurements would not be sufficient to identify the value of the S-box bit. The experiment makes clear that statistical methods can be used to gain additional information from a set of power measurements.

Using the knowledge from the experiment, DPA can be used to attack an AES-128 cryptographic operation. Goal of the attack is to find the secret key.
The example attack from [23] attacks the $AddRoundKey$ and the $SubBytes$ routines from the first round.[1] Equation (2.1) shows the operation to attack where I_i is the 16-byte intermediate value of the cipher after the $SubBytes$ routine and $I_{i,n}$ the nth byte of I_i. K denotes the first round key and K_n the nth byte of K. X_i is the plaintext and $X_{i,n}$ denotes the nth byte of X_i. The index i indicates a specific trace in a set of power traces. S is the S-box of the AES.

$$I_{i,n} = S[X_{i,n} \oplus K_n] \qquad n \in \{0, \ldots, 15\} \tag{2.1}$$

K_n is the nth byte of the secret key, therefore the intermediate value $I_{i,n}$ is unknown. Because K_n is a 8-bit value, there are 256 possible candidates for the nth byte of the secret key. In order to find K_n, a test is needed which says if a guess of K_n is correct or not. As stated in [23], this task can be solved using DPA.
In order to perform DPA on the given problem, a selection function has to be found [23]. This selection function is used to divide the measured traces into subsets. For the example attack on the AES-128 cipher the LSB of $I_{i,n}$ was used.

[1] For additional information about the Advanced Encryption Standard (AES) see [30].

An attacker now has to measure the power consumption of the cryptographic system during the encryption. This task has to be repeated several times using randomly chosen plain text for every measurement gaining power traces. Now the attacker has to 'guess' one possible value for K_n. Using K_n, the attacker can compute $I_{i,n}$ for every power trace i using the same plain text as used for the measurement of trace i. Depending on the LSB of $I_{i,n}$ (the selection function) the set of power traces can be divided into two subsets. Now, the attacker computes the average value for all time instances in the subset, producing two average traces for each subset. Using these two traces the difference between the subsets can be computed.
This task is repeated for every possible guess of K_n. Because K_n is a 8-bit value there are 256 possibilities. For every guess the attacker computes the difference of the averages. In this example attack 256 difference traces must be computed. Using these difference traces, the right guess can be identified by searching for the traces with the highest peaks [23]. This is motivated by the experiment described above. Instead of evaluating only one specific point in the trace as done in the experiment, DPA considers the whole trace. This is done by computing the average trace for each subset which are then compared. Spikes in the difference trace indicate that the trace is correlated to the selection function because the intermediate value is manipulated by the device [23]. Therefore, higher spikes indicate stronger correlation to the selection function and are therefore an indicator for a right guess.
Because the secret key is a 16-byte key this attack has to be repeated for every K_n to get the whole key. Figure 2.2 shows the tasks for the example attack for two guesses.

This example attack shows the basic principle of DPA. However, in [23] also a more general view on DPA can be found. As stated in [23], the concept of SPA and DPA can also be applied on electromagnetic measurements performing Simple Electromagnetic Attacks (SEMA) or Differential Electromagnetic Attacks (DEMA).
A more complex attack using power analysis is the so called High-Order DPA (HODPA) which can be used to circumvent countermeasures against DPA [21], [28], [33].

2.1.3 Electromagnetic Attacks

Similar to power analysis, information is also leaked due to electromagnetic emanation. Measuring the electromagnetic field of a device instead of the power consumption can be advantageous if a measurement setup to measure the power consumption is not possible. For example passive RFID tags where the analog front-end is integrated on the same IC as the RFID device. In such a case the measurement of the electromagnetic field can be a good alternative. As stated in [1], electromagnetic attacks can even perform better than attacks using power analysis. Additionally, these attacks can be used to circumvent countermeasures against power analysis.
The simplest way to perform an attack is to measure the electromagnetic emanation with one antenna in the vicinity of the device. These measurements can then be interpreted

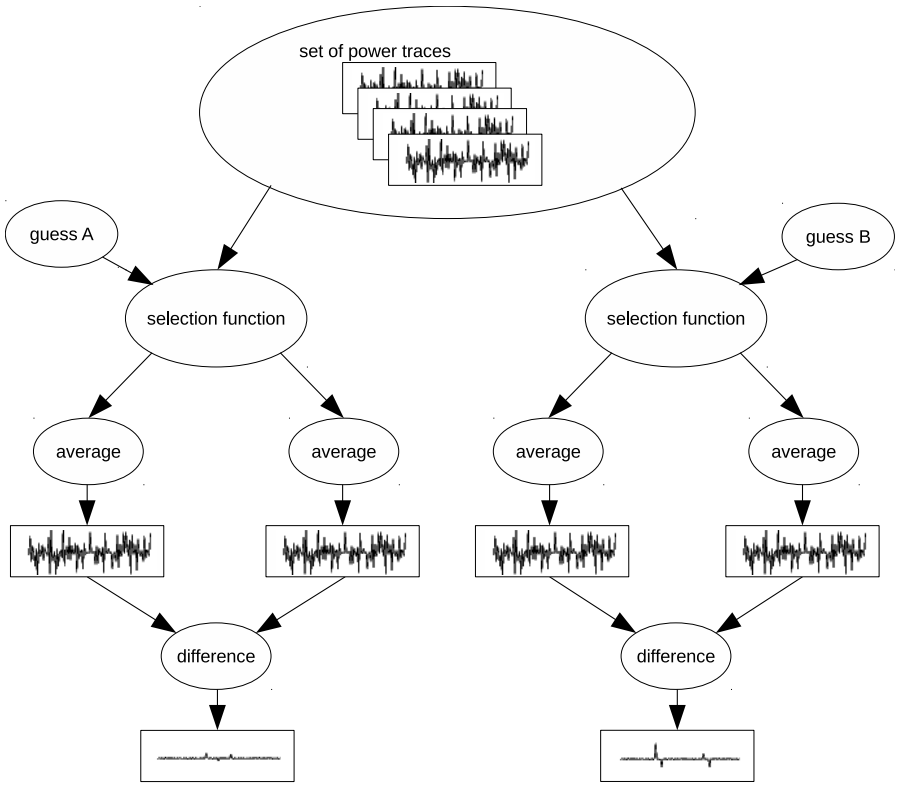


Figure 2.2: DPA process for two guesses. The selection function divides the set of traces into two subsets. For each subset an average trace is computed. The difference trace is used to find the correct key. [23]

applying the mechanisms used for power analysis (see Section 2.1.2). One advantage of electromagnetic attacks in contrast to power analysis is the capability of exploiting local information [16]. Using small probes, an attacker is able to pinpoint specific areas on a chip (CPU, buses, etc.).
Gandolfi et al. [16] propose several attacks on cryptographic systems using simple hand-made solenoids made of a coiled copper wire. The measurements were then amplified and analyzed using DEMA. The experiments show that electromagnetic attacks can yield better results than attacks using power analysis. In the experiments, the DEMA reaches a higher SNR than DPA.
Similar experiments were proposed by Hutter et al. [20], where attacks were performed on passive RFID tags. The experiments were performed on self-made prototypes instead of commercial passive RFID devices. The target of the attacks was the AES algorithm. The experiments performed by Hutter et al. also targeted the challenge of EM attacks in the presence of an external field provided by a RFID reader.

2.1.4 Timing Attacks

Another potential side channel leaking sensitive information is the timing behavior of a device. Timing attacks make use of the data/key dependent runtime of a cryptographic algorithm. Measuring this small time variations during the computation can then be used to gain sensitive data. Such variations are caused for example by processor instructions which run in non-fixed time, conditional statements, etc.
In [24], Kocher presented such timing attacks on implementations of Diffie-Hellman, RSA and DSS.

2.1.5 Countermeasures against Side Channel Attacks

As shown in the sections above side channel attacks are powerful methods to attack a cryptographic system. Therefore, countermeasures have to be found to mitigate the threat of such attacks.
As mentioned in [23] mechanisms such as balancing, blinding and masking can be used to reduce the threat of power analysis. Another idea is to add noise to the measurements using circuits performing computations uncorrelated to the cryptographic operations. As also mentioned in [23] an attacker may be able to circumvent these countermeasures using HODPA and/or performing a higher amount of measurements. As highlighted in [1] electromagnetic attacks can also be used to circumvent countermeasures against power analysis. Therefore, also the electromagnetic emanation of a system must be considered if the side channel leakage of a system should be reduced. This can for example be done by reducing the signal strength and/or shielding [1].
Facing timing attacks, blinding mechanism or random delays added to the computations can be used [24].
In [41], a balancing approach can be found where the problem is addressed on circuit level to suppress the leakage of the power consumption. Even if that approach shows that the effects of a side channel attack can be reduced, the authors also mentioned that perfect security does not exist.

Balancing Mechanism

These mechanisms try to reduce the variations of the power consumption during computations and thus information leakage. This can be done on circuit level using sophisticated designs to make the power consumption of the device less dependent on data and the operations being performed [23].
Examples for such designs can be found in [41], [35] and [39].

Blinding Mechanism

Because many cryptographic systems use operations over a finite field, one can use mathematical relationships to blind the computations. Equation (2.2) shows such a relationship suitable for blinding (where $\phi(.)$ denotes the Euler's phi function).

$$A^{d+k\cdot\phi(P)} mod\, P = A^d mod\, P \tag{2.2}$$

Changing k randomly between computations interferes statistical tests like DPA [23].

Masking Mechanism

Similar to blinding, masking can be used for symmetric algorithms where secret constants and intermediates are split into randomized parts [23]. Computations are then performed on these independent parts. For the final output these partial results are reassembled.

2.2 Fault Injection

2.2.1 Introduction

The term Fault Injection is generally used for intentional faults introduced into a system. There are several reasons why faults are injected into systems.
One reason is to test a system under the influence of faults. In order to test a system under such conditions faults are injected into a running system. The system is then observed. If such a fault does not lead to a failure in the system, the test has successfully passed. If not, the system is vulnerable against such faults. Although this looks very trivial, there is a lot of research targeting such problems. Modern ICs comprise a huge amount of transistors. The challenge is now to find 'good' targets for the test to get valuable results. Another challenge is to choose the right time for injecting a fault. For example changing one bit in the system will not produce any failure if the bit will never be read or if the bit is overwritten before the bit is read again. To find 'good' targets is therefore a challenging task.
Faults injected into a running system can also be used for attacks which are often called fault attacks. If the location of the fault is well chosen an attacker may retrieve secret information from the system. Potential targets of such attacks are for example the program counter or (on-chip) buses.

2.2.2 Reasons for Faults

One important factor for faults in electric devices is radiation. The particles travel through the device and interact with the material of the IC. This interaction can then lead to so called soft errors if the circuit is not permanently damaged (e.g., one bit of the memory is flipped) or can even damage the device. Soft errors are also referred to as single event upsets (SEU). Depending on the amount of affected bits also the terms singlebit upsets (SBU) and multibit upsets (MBU) are used.
As stated in [8], three mechanisms are dominant in the terrestrial environment. The first mechanism introduces faults due to alpha particles (ionized helium). Such alpha particles are emitted by impurities in the packaging material. An early work describing the impact of packaging material on dynamic RAM can be found in [27].
A second source of faults are high-energy cosmic rays. These rays interact with the atmosphere of the earth and produce secondary particles. These secondary particles then travel deeper into the atmosphere and produce tertiary particles and so on. At terrestrial level the particles then interact with the device material which causes faults.
The third source of faults are low-energy cosmic rays where faults are introduced due to the reaction of low-energy neutrons with boron inside the device.
Although these effects can be diminished (e.g., due to purified packaging material or shielding) the source of faults can never be eliminated.[2] Therefore, additional countermeasures have to be taken to deal with this kind of faults.
Where these three mechanisms describe the situation on terrestrial level one should also mention that the occurrence of faults also depends on altitude, latitude and solar activity [32].
Another important factor concerning faults introduced due to radiation is the used technology. As shown in [8] the soft error rate (SER) depends on the technology scaling and even differs for distinct components such as SRAM and DRAM. While the bit SER of DRAMs is decreasing due to technology evolution the bit SER of SRAM has saturated. If the increasing memory density of modern systems is considered these trends lead to constant system SER for DRAMs and an increasing system SER for SRAMs. Similar to memory also logic elements are sensitive to radiation. Faults happen due to single-event transients (SET) introduced into the logic element by radiation. These glitches then lead to an error if this wrong signal actually propagates to the input of a flip-flop/latch during a latching clock signal [8]. The probability that such glitches are latched depends on the propagation delay and the clock frequency. Therefore, the probability of such errors increases with advanced technology because of the reduced propagation delay and higher frequencies.
In order to face these problems, mechanisms are necessary to mitigate the influence of such faults (see Section 2.2.4).

Other than natural faults introduced due radiation, faults can be injected into a system

[2] For a detailed view on the fault mechanisms, the physical processes and possible countermeasures see [8].

with the intention to retrieve secret information. In contrast to side channel attacks (see Section 2.1) the attacker takes an action to influence the target of the attack. This can either be invasive, by detaching the package and manipulation of the circuit or noninvasive using radiation or temperature. Another possibility is to manipulate the power supply to introduce faults or to perturb the clock signal. Goal of such attacks is to change the behavior of a system (e.g., to manipulate the execution of a program or to disable specific components).
A major difference between natural faults and intentional faults during a fault attack is their behavior. While radiation caused faults are mostly temporary (e.g., a write on an affected bit in memory overwrites the faulty bit) invasive intrusion can also lead to a permanent fault (e.g., due to manipulation of the circuit). Another difference is that fault attacks can inject more than one fault simultaneously where the probability for multibit faults caused by natural sources is low.

Fault injection can therefore be seen as a tool to test the protection mechanisms added to mitigate the influence of natural faults and as a method to attack cryptographic systems (fault attacks). Additionally, fault injection can be used to analyze the threat of a potential attack (e.g., to oppose the risk of an attack to the costs of the countermeasure).

2.2.3 Fault Injection Methods

Generally, methods for fault injection can be divided into two classes. The first class uses physical effects to introduce faults into a running system. The other class makes use of simulations to analyze the impact of faults. While the first class can be used to attack a system in order to retrieve secrete information, the second class is used to test the vulnerability of the system in presence of faults. This information can then be used to develop robust devices.
This section describes some possible methods to inject faults into a running system while the concept of simulating faults is described in Section 2.3.

As described in [5] the most common methods for fault injection are the following:

- Supply voltage variations
  Varying the supply voltage during runtime of the system may cause a processor to skip an instruction. This can for example be used to omit branches. Especially for contactless smartcards where the card is supplied by an external field, such attacks are from interest.

- External clock variations
  Similar to supply voltage variations, variations of the clock can cause the processor to skip instructions or can lead to data misreads.

- Temperature
  The goal of this method is to vary the temperature until a threshold is reached where the circuit or parts of the circuit does not work correctly anymore. This can for example be used to inject random faults. As also stated in [5], this method can be used to disable read or write operations to memory which can be exploited due to the fact that the temperature threshold for read and write operations are not the same. Temperature attacks can therefore be used to omit counter updates such as used for limited resources (e.g., a cryptographic key which can be used several times until it becomes invalid).

- White light
  Exploiting the photoelectric effect of electric circuits, light can be used to introduce faults into a detached IC.

- Laser beams
  Similar to white light, laser beams can be used for fault injection. The advantage of laser light compared to white light is that a certain part of the circuit can be attacked more precisely.

- X-ray and ion beams
  Other than fault injection with light, these methods can be used without depackaging the IC.

The described methods vary in effectiveness and their costs. Therefore, in order to design robust systems against fault attacks the designer must conclude on which equipment a potential attacker possesses.

2.2.4 Protection Mechanisms against Faults

Depending on the purpose of the protection mechanism different methods exist. While countermeasures against natural faults are often designed for SBUs, this is not sufficient as protection against fault attacks where multibit faults can occur. Therefore, analyzing the threats of a system is of great importance when the system should be robust against a certain kind of faults.

One possible mechanism against faults to protect memory is to add one bit for each data word which stores the parity. The parity bit allows to detect one error in the data word but fails if two errors occur. In general, if one parity bit is used only an even number of errors can be detected. Another possibility is to use error detection and correction EDAC or error correction codes ECC to protect the memory. Other than the parity check, this method also allows to correct a wrong bit. This feature is implemented by adding some bits to the data word to increase the redundancy of the information stored

in the extended data word. Depending on the additional information provided due to ECC one or more erroneous bits can be corrected.
Where these mechanisms are suitable to protect the integrity of the memory, additional methods must be found to protect the combinational/sequential logic. One possibility is to duplicate the circuit and compare their results. If the results differ from each other an error occurred and an alarm can be triggered. If more than two instances of the same circuit are used the result can even be corrected. Also possible is the use of complementary redundancy where the circuits produce complemented outputs. If the result is the same on both outputs, an error occurred. The advantage of such circuits is that it can be used for multibit faults due to the fact that fault injection with complementary effect is complicated [5].
Besides the methods described above, also redundancy in time can be used as protection mechanism. Using this protection scheme the computation is repeated at the same circuit and the results are then compared. Similar to the spatial implementation, errors can be detected or even corrected if the computation is performed more than twice.
A more sophisticated technique is proposed by Moore et al. [29], where self-timed circuits and dual-rail logic are used to improve the resistance against faults and even against side channel attacks.
Another possible countermeasure against attacks is the use of sensors [5]. These sensors can for example be used to detect voltage or frequency variation. Also possible are active shields covering the entire chip. This shield consists of a metal mesh where data is passing. If an attacker tries to modify the circuit also the data in the mesh is corrupted and an error signal is generated.
Similar to the mechanisms implemented in hardware countermeasures can also be implemented in software. Such mechanisms are for example *checksums*, *variable redundancy* or *execution redundancy* [5].

All these mechanisms have in common that size and/or execution penalties must be accepted.

2.3 Fault Injection Platforms and Simulations

2.3.1 Introduction

This section deals with the evaluation of systems under the influence of faults.
One possible method to test systems and error detection/correction mechanisms is to use the fault injection methods described in Section 2.2.3 (e.g., due to the use of radiation or invasive methods). One important drawback of this method is that at least a prototype is needed. Therefore, the information about vulnerabilities of the design is not available at the early design steps. Another disadvantage is the limited capability to monitor the internal states of the system which is important for the understanding of an erroneous

system behavior.
To circumvent these drawbacks one can use simulations or system emulator with integrated fault injectors. Using this method, faults can be injected at arbitrary locations and the effects can be monitored. Simulations can be done using hardware description languages (HDL) such as VHDL, Verilog, SystemC, etc. Another possibility is to implement the system on an FPGA together with dedicated circuits necessary to introduce faults. The advantage of an FPGA based system emulator opposed to simulations is the improved simulation speed and therefore a higher fault injection rate.

2.3.2 Classification of Fault Effects

In the context of simulated/emulated fault injection, faults can be divided into three classes [40].

- Failure
  If the injected fault leads to a wrong behavior of the circuit (e.g., wrong or delayed output) the fault is classified as $failure$.

- Silent fault
  If the injected fault does not lead to a wrong behavior and disappears completely until the testbench has finished, the fault is considered as *silent*. This can be the case if the fault affects a flip-flop/memory cell which is overwritten during execution before the value is used.

- Latent fault
  If the injected fault does not lead to a wrong behavior during the execution of the testbench but remains as a deviation of the system state compared to the state observed during a fault free execution, the fault is classified as *latent*. This class of faults depends on the testbench because if the simulation/emulation would continue, the fault may causes a failure or a latent fault.
  Figure 2.3 shows the two cases where a fault is either classified as $latent\ fault$ or as $failure/silent\ fault$ depending on the duration of the testbench/experiment.

2.3.3 Saboteurs and Mutants

Using a HDL, a common method for fault injection is the usage of saboteurs or mutants. Saboteurs are simple units which are used to modify signals. Therefore, these units are placed between the signal's source and its sink (see Figure 2.4). These units have a signal input and a signal output. Additionally, a saboteur has certain control inputs such as *activate*, *mode* and the *clock* signal. The *activate* signal is used to trigger

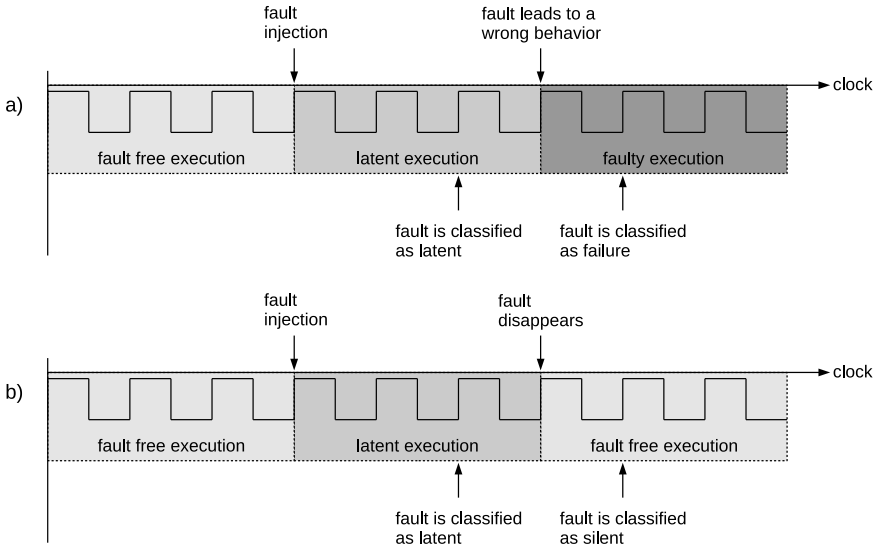


Figure 2.3: Fault classification. a) *latent fault* leads to a *failure* b)*latent fault* leads to a *silent fault*.

the fault. If the saboteur is disabled it behaves like a wire. If the saboteur supports different kinds of faults also a *mode* signal is needed to choose which fault should be triggered. Common fault modes implemented in saboteurs are *stuck-at-zero*, *stuck-at-one*, *indetermination faults*, *negation faults*, *delays*, *bridging faults* or *bit-flips*. Table 2.1 gives an overview about the faults, using the definition from [18].

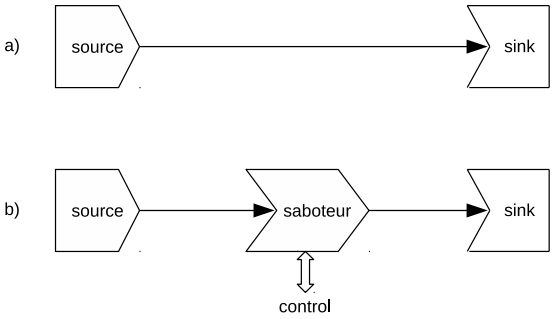


Figure 2.4: Saboteur placement. a) original signal b) instrumented signal using a saboteur.

Other than saboteurs which are used to manipulate signals a mutant is a modified version of a submodule used by the system. The mutant behaves like the original module until the mutant is activated. If the mutant is activated, the module behaves like a faulty version of the original unit.

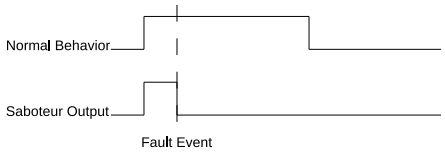
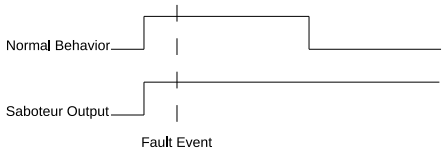
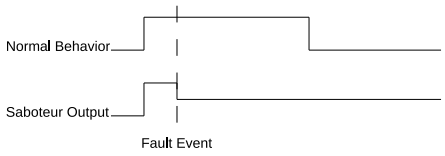
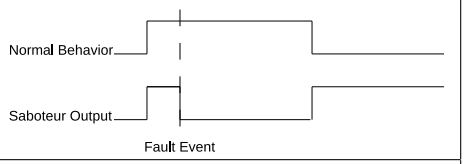
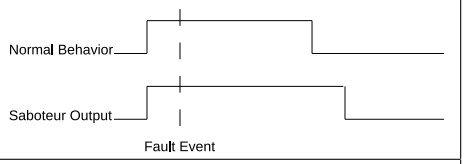
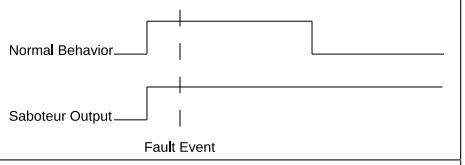
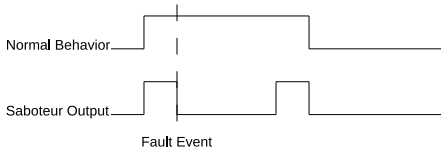
Fault	Description	
Stuck-at-Zero	The output stays permanently at zero. This fault reflects a direct modification of the circuit where the signal is wired to a logic zero.	
Stuck-at-One	The output stays permanently at one. This fault reflects a direct modification of the circuit where the signal is wired to a logic one.	
Indetermination	The signal holds an undefined state (permanent fault). This fault reflects a direct modification of the circuit.	
Negation	The signal is inverted (permanent fault).	
Delay	Input to output delay (transient fault). This fault reflects supply voltage variations.	
Bridging Fault	The input is not propagated anymore (permanent fault).	
Bit-Flip	The input is inverted for one cycle (transient fault). This fault reflects latched SETs (e.g., due to radiation or laser beam).	

Table 2.1: Common fault modes used for saboteurs (the table uses the definition from [18]).

A common requirement of these units is that additional mechanisms are necessary to control the activation and deactivation of the modules.

Both, saboteurs and mutants, are used in simulations and emulation based fault injection.

2.3.4 Simulation based Fault Injection

Simulations execute HDL descriptions of a system. One possible method to use simulations for fault injection is to run the simulation until a specific point, change the value of one or more signals and resume the simulation. After the simulation completed the results can be obtained. This method works without the modification of the system model but depends on the available commands provided by the simulator.
Another possible method for simulation based fault injection is the usage of saboteurs and/or mutants. The system model is extended with these additional modules to allow automated fault injection. Additionally, the fault injection campaign is independent of the used simulator. A common method for such campaigns is to perform a *golden run* where the system is simulated without the presence of faults. These results are then stored and can be compared to the results of faulty runs to monitor the effects of the faults. A common disadvantage of simulation based fault injection is the high time effort especially for huge projects. Therefore, emulation based fault injection has become an important tool to improve the validation process.

2.3.5 Emulation based Fault Injection

The principle of emulation based fault injection is the usage of hardware (usually FPGAs) to speedup the injection campaigns. There are different ways to exploit the advantages of FPGA based fault emulation. One method is to use *global reconfiguration* or *partial reconfiguration* methods [3]. This method injects faults due to reconfiguration of the FPGA. This can either be done by reconfiguring the whole device (*global reconfiguration*) or by changing only parts of the circuit (*partial reconfiguration*). Therefore, one major advantage of this method is that no modification of the original system model is necessary. This means that no additional circuits must be added to the system which is advantageous if the hardware resources of the used FPGA are strongly limited. For example to inject a fault into a combinational subcircuit, which is implemented as LUT on the FPGA, only the description of the corresponding LUT must be changed.
Using *global reconfiguration*, the whole FPGA is reconfigured. Therefore, a mechanism must be provided to restore the system state after the fault is injected to preserve intermediate results (also see [19]). To circumvent this inter-configuration communication,

partial reconfiguration can be used where only a part of the device is reconfigured and therefore does not affect the whole system. The drawback of this method is that the device must possess the capability for *partial reconfiguration*. Therefore, this approach is limited to a certain subset of FPGAs.
Another approach is to modify the system model in order to allow the manipulation of certain units. Similar to simulation based fault injection, the concept of saboteurs and mutants can be applied. As done by the simulation approach, a *golden run* is performed, to get a fault free reference for the analysis. In order to perform the injection campaign a mechanism must be implemented to trigger the fault units and to monitor the results. An important difference between simulations and emulation based fault injection is that the system model and the additional components used for the injection campaign must be available in a synthesizable description.

2.3.6 Optimization of Fault Injection Campaigns

Because the evaluation of a whole system under the influence of faults requires a lot of time, mechanisms are needed to speedup this process and to improve the effectiveness of the fault injection campaigns. As mentioned in Section 2.3.5, hardware accelerated methods can be used to gain better performance compared to simulation based methods. However, speeding up the injection process is often not enough considering the huge amount of possible injection points. Therefore, also the effectiveness of the process must be improved rather than only increasing the speed of the campaigns.
A common method of fault injection tools is to choose the location and time of the faults randomly from the fault-space (two dimensional space spanned by time and location). The problem of this approach is that only a subset of these injected faults has an impact on the system behavior. Therefore, the efficiency of the campaign is often low. For example if a fault is injected into a memory location that is never read, the fault can not affect the system and is therefore useless for evaluation. Similar, if a fault is injected into a location just before a write on that memory, the fault has no effect. Figure 2.5 shows an example fault-space for memory. Fault $F1$ has no effect because the fault is injected in an unused cell. Similar, $F3$ is useless because the cell is not read after the fault has been injected. Also $F2$ has no effect because the cell is written just after the injection. $F4$ and $F5$ are injected in the same cell before a read. Therefore, $F5$ would be sufficient. If the faults are modeled as simple bit-flips, $F5$ may neutralize the effect of $F4$ which is even worse because the effect of the fault disappears. This can lead to a misinterpretation during the analysis phase.
The goal is therefore to identify those locations of the fault-space that influence the system behavior.

A paper targeting the pre-analysis for software based fault injection where proposed by Barbosa et al. [7]. This method uses the assembly code and information obtained by a reference run of the software to find possible locations and time instances for the fault

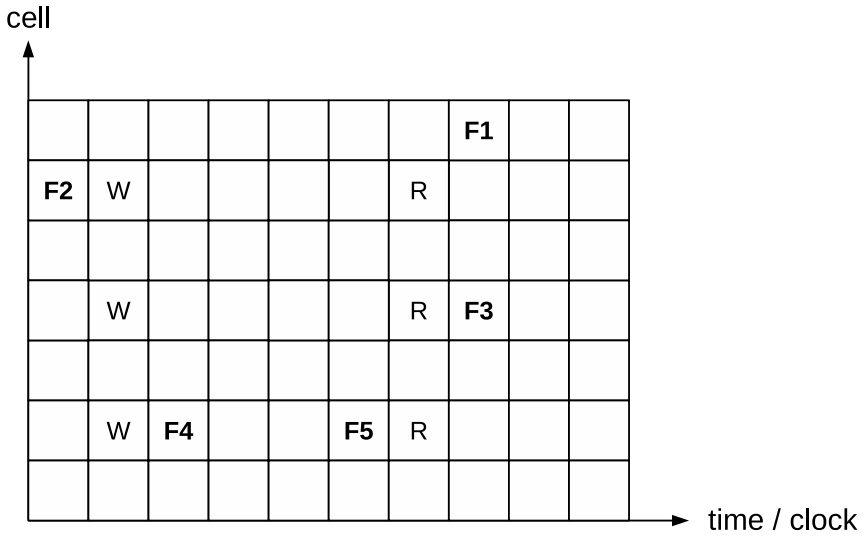


Figure 2.5: Example fault-space of memory. W indicates a write and R a read on the memory cell at a distinct time instance. Injected faults are indicated by F.

injection. The idea is that faults should only be injected into resources immediately before they are read.

Another topic of interest concerning the usage of instrumented system models (e.g., system models with integrated saboteurs/mutants) is how to place these additional units. The most obvious method is to add these units by hand at the desired positions in the model. However, this is only feasible if the amount of the units is low. Therefore, automated mechanisms are needed for a detailed system evaluation. One approach of automatic saboteur and mutant placement can be found in [6], where Baraza et al. propose a method to instrument VHDL models automatically.

2.4 Power Emulation

The power consumption of a system is an important design criteria especially for battery powered devices. Therefore, knowledge about the power consumption is of great importance even during early designs steps of a system.
In order to gain information about the power consumption during early design steps where no prototype is available, simulations become an important tool for power evaluation. However, due to high costs of simulations regarding the time effort, alternative approaches were developed. One of these approaches is power emulation [10].
The idea behind power emulation is that the functions necessary to estimate the power

consumption of a circuit can be expressed as hardware circuits. These additional circuits can then be added to the original system to monitor the power consumption. The concept benefits from the fact that hardware emulation platforms (e.g., FPGAs) can be used to implement the extended system to perform the power evaluation in hardware (hence the name power emulation). The only requirement is that the original system and the estimation functions are available in a synthesizable description.
Coburn et al. [10] demonstrated the concept of power emulation on register-transfer level (RTL). However, the idea can be applied at other levels of abstraction. The whole setup consists of *power models*, a *power strobe generator* and a *power aggregator*. The *power models* are added to every RTL component of the original circuit. These additional units implement a linear regression based model. The inputs to the model are the input and output transitions of the RTL component under consideration. The coefficients of the regression model define the power consumption for a specific input or output transition of the component. The output of the model is the estimated power consumption of the monitored component. Equation (2.3) shows the mathematical representation of the model for N inputs/outputs at the discrete time instance t.

$$\begin{aligned} power[t] = \quad & XOR(x_1[t], x_1[t-1]) \cdot Coeff_1 + \cdots + \\ & XOR(x_N[t], x_N[t-1]) \cdot Coeff_N \end{aligned} \tag{2.3}$$

The *power strobe generator* is used to trigger the *power models* and the *power aggregator* accumulates the estimated power consumption of all components. Additionally, a host PC is used to read out the accumulated power values.
Similar to the approach of RTL power emulation the concept can also be applied to higher abstraction levels. The advantage of higher level power emulation is that the amount of necessary *power models* decreases which also leads to a decreased area overhead introduced by the power emulation circuits. The drawback is that an increased granularity also has an impact on the accuracy of the results. One such high level approach was proposed by Genser et al. [17]. Other than RTL power emulation, distinct units like CPU, memory, etc. are considered.

2.5 Related Projects

This section gives an overview about selected projects related to the emulation framework proposed in this work.

Kasper et al. [22] propose a low budget framework for testing system vulnerabilities of embedded devices such as microcontroller, contactless RFID devices, etc. The project targets the analysis of side channel attacks (e.g., SPA, DPA) and noninvasive fault injection techniques. The framework features a flexible design fulfilled due to partitioning the platform into modules with specific purpose. The platform is divided into the following main components.

- Communication Module
  This module is used to communicate with the DUT in order to perform use cases for the attack/fault injection scenario. The framework provides communication modules for different kinds of interfaces such as contactless interfaces for RFID applications, contactbased protocols (e.g., for communication with a smartcard of the contactbased interface) and arbitrary parallel/serial interfaces (e.g., USB, RS-232, SPI).

- Fault Injection Module
  The fault injection module is used for noninvasive fault attacks on the DUT. This module depends on the used fault scenario. The framework combines several units for different kinds of faults including optical fault injection, electromagnetic fault injection, power fault injection (variations of the voltage supply) and faults due to clock variations. All these units were implemented with low-cost, public available components (e.g., a flash light of a photo-camera for optical faults). In order to control these units an FPGA was proposed.

- Data Acquisition Module
  The acquisition module is used to gather information about the DUT which then can be used for side channel attacks (e.g., SPA, DPA, EMA) or to monitor the results of a fault injection scenario. The framework proposes a PC and/or an oscilloscope to serve for this purpose. Additionally, the PC can also be used to control the fault injection units.

Performing fault/attack scenarios on a DUT has the drawback that at least a prototype must be available. As mentioned in the sections above, simulation or emulation based approaches are necessary to enable the analysis of a system in its early design steps. Therefore, the framework also supports the replacement of the DUT with a module emulating the original system.
The remainder of this section concentrates on simulations and emulation platforms targeting fault injection and fault analysis.

One simulation based fault injection tool is VERIFY (VHDL-based Evaluation of Reliability by Injecting Faults efficiently) [37]. The concept of VERIFY is to change the behavioral description of single components. The fault behavior is directly integrated into the component description by defining the fault frequency and the duration of the fault. Because the behavior of the faults is integrated directly into the model no structural component needs to be changed. Therefore, the entity description which defines the interface of the component stays the same for the faulty and the fault-free model (e.g., there are no control signals necessary compared to approaches using saboteurs/mutants). The project uses a compiler and simulator developed for VERIFY in order to handle extensions in the VHDL language necessary for the fault description.

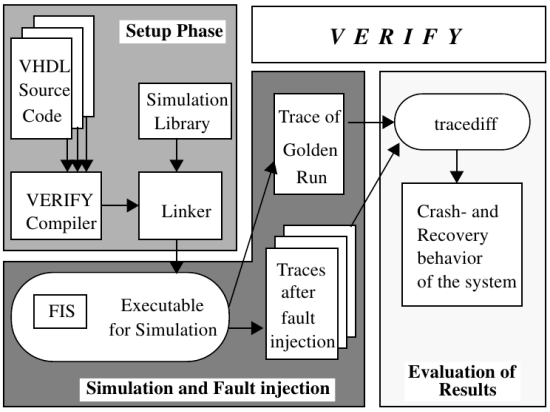


Figure 2.6: VERIFY - Overview. The fault behavior is directly integrated into the component description. The project uses a compiler and simulator developed for VERIFY to handle extensions in the VHDL language for fault description [37].

Another simulation based fault injection tool using VHDL models is MEFISTO-L [9]. Other than VERIFY a saboteur based approach is used. Additionally to the saboteurs which are used to inject the faults, so called probes are used to monitor the system and to control the fault injection process (e.g., to trigger the fault injection depending on specific system state). The tool supports the extraction of potential target signals from the VHDL model, automatic placement of the saboteurs and configuration of the probes. To produce a mutated system model from the initial model MEFISTO-L uses its own code analyzer. Other than VERIFY, MEFISTO-L does not rely on a specific VHDL simulator.
Where the above described projects make use of simulations, FADES (**F**PGA-based framework for the **A**nalysis of the **D**ependability of **E**mbedded **S**ystems) [2] uses an emulation based approach. The fault injection is done due to *partial reconfiguration* of the FPGA circuits. The whole setup consists of a FPGA capable for *partial reconfiguration* and a host PC to manage the injection process and to observe the behavior of the system. FADES allows to analyze a bitstream file generated during synthesis to define injection points (sequential and combinational logic) and to generate a list of possible observation points. FADES also allows to choose the desired fault model such as stuck-at, delay, bridging, etc. One major advantage of FADES is that the tool works on the final implementation of the model and is therefore independent of the used HDL. This feature also allows to use third party IP cores where no source code is available.
Another FPGA based fault injector was proposed by Grinschgl et al. [18]. Other than FADES, a saboteur based approach is used. The project targets a modular design for multibit fault injection to support fault attack emulation. The setup consists of saboteurs inserted into the DUT, a fault injection controller to control the saboteurs and a PowerPC as interface to a PC (see Figure 2.7). The saboteurs are designed as such that various fault modes can be configured. The control of all saboteurs is handled by

the fault injection controller. This controller is responsible to configure the mode of each saboteur and to activate and deactivate these units. In order to fulfill this task, the fault injection controller possesses an internal memory which holds the fault pattern (defines the attack/test scenario). This fault pattern memory allows to automate the fault injection campaign. The general architecture of the controller is pictured in Figure 2.8. The fault injection controller itself is controlled by a PowerPC. The PowerPC is used to load the fault pattern into the internal memory of the fault injection controller and as interface to a host PC. A special feature of the tool is that the communication between the fault injection controller and the PowerPC is realized by a general purpose input/output (GPIO) interface which can be controlled by software. Therefore, the PowerPC can easily be replaced by another control unit.

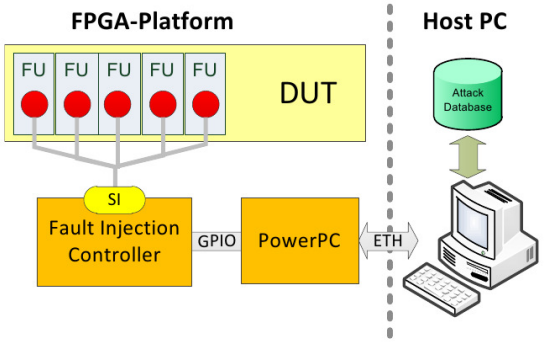


Figure 2.7: MFI - System Overview. The setup consists of saboteurs inserted into the DUT, a fault injection controller to control the saboteurs and a PowerPC as interface to a PC [18].

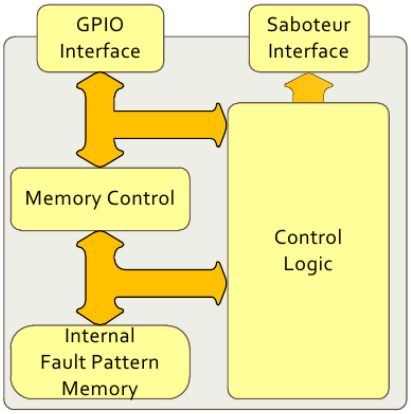


Figure 2.8: MFI - Controller. The controller is responsible to configure the mode of each saboteur and to activate and deactivate these units. The fault injection controller possesses an internal memory which holds the fault pattern [18].

In [11], Daveau et al. present a fault injection emulation platform for industrial requirements enabling the verification of exhaustive fault campaigns.
The platform makes use of several FPGA prototyping platforms controlled by a host system (PC) enabling a huge degree of parallelism. Figure 2.9 shows the architecture of the emulator. The host system is connected to the hardware platforms in order to define the fault scenarios. Each of the platforms includes a controller executing the fault campaign and several instances of the DUT. The HW controller also serves as monitor for the results of the campaigns. The DUT instances are composed of the actual hardware under test and a testbench (e.g., a processor and a RAM with a test program).
In order to inject faults into the target hardware, flip flops, latches and memory are instrumented with a mechanism allowing different fault models (stuck-at-zero, stuck-at-one, SEUs).
The particularity of the platform is the possibility to perform campaigns in parallel at a huge degree. The performance of the emulator was demonstrated using LEON2 processors as DUT attaining a fault injection rate of 400K faults/hour/domain, where domain denotes a FPGA prototyping platform.

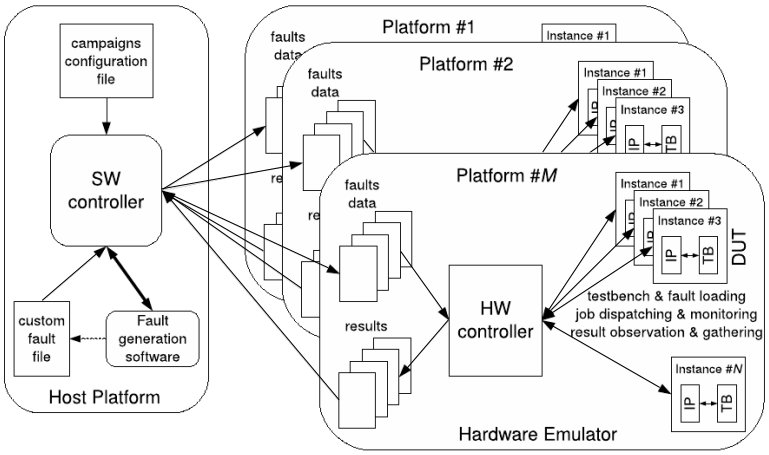


Figure 2.9: Industrial Fault Injection Platform - Overview. The platform makes use of several FPGA prototyping platforms controlled by a host system (PC) enabling a huge degree of parallelism [11].

3 Design

3.1 General Considerations

The goal of this work is the development of a reader-smartcard emulation framework focusing on fault injection and power emulation. Building an emulator for such a RFID setup implies several considerations concerning the principle structure, accuracy and authenticity of the emulator.
The following questions have to be considered:

1. How is the emulator divided into reader, card and the emulated contactless data channel?

2. Which hardware is used to represent the internal configuration of reader and card?

3. How can the whole setup be controlled?

4. How can data be gathered and transferred to a host system for further computations?

An important goal of this thesis is the combination of reader, card and the contactless channel into one emulation framework. The general structure of the emulator depends on the available hardware (prototyping platform). The first consideration that has to be made is therefore the used target hardware. Two distinct approaches were considered.
One possible approach uses two FPGA prototyping boards where one board is used to emulate the reader and the other to emulate the card. The channel could be implemented via cable to connect the two prototyping boards for data exchange. Additionally modules can be implemented on both boards to model the behavior of a contactless channel. This approach gives also the ability to use a contactless interface instead of the contact based connection. The advantage of a contactless interface is that the channel itself doesn't not have to be emulated. A disadvantage of the contactless interface is the reduced controllability of the channel.
The second approach uses only one prototyping platform combining the emulation of reader, card and channel into one FPGA. The main advantage of this approach is that the whole application resides in one prototyping board which enhances the usability of the framework. Another advantage is the better controllability of the whole setup

obtained by the fact that the setup uses the same clock domain. Therefore, no synchronization has to be done which is not the case if the reader and the card model resides on different hardware. A drawback is that the used FPGA platform has to be capable to hold the whole setup which reduces the set of capable FPGA prototyping platforms. Figure 3.1 and Figure 3.2 picture the two different approaches.
Another consideration that has to be made is how the hardware of the reader/card is implemented. Because original reader/card hardware is not available due to disclosure policies a substitute has to be found.
In order to use the hardware setup for emulation purpose additional modules for controlling have to be implemented. The purpose of these additional units is to trigger the emulation process, to configure the whole setup and to download the reader/card software. Another purpose of these modules is to gather information during emulation and to allow the information transfer from emulator to a host system for further computations.

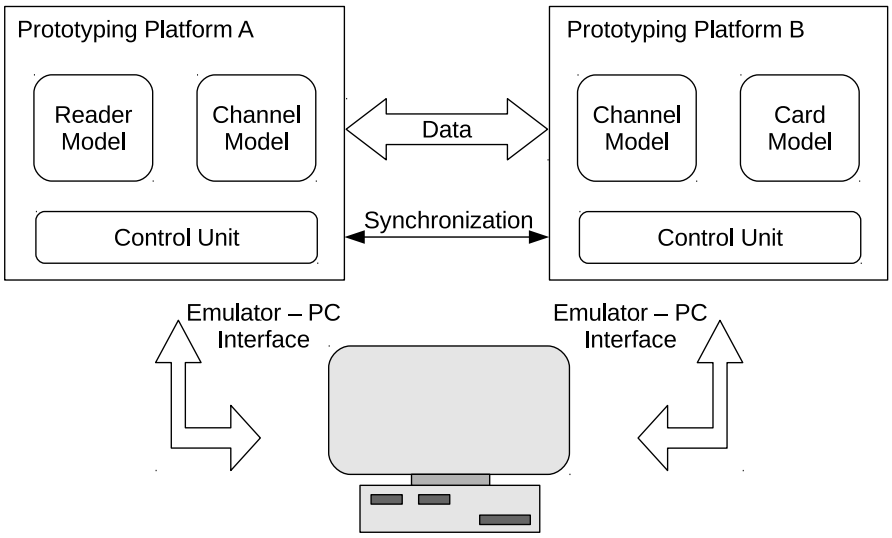


Figure 3.1: Architecture Choice - Two Board Approach. Two developing platforms are used to implement the reader and card model. Synchronization has to be done in order to allow instantaneous fault injection.

Taking into account the considerations mentioned above the emulator is structured as follows. The whole setup is implemented on one FPGA developing platform exploiting the advantages of a synchronized system with common time basis. The hardware related design is based on the open source IP library from Aeroflex Gaisler ([14], [15]). This library provides an implementation of the SPARC V8 architecture (LEON3 processor) and an implementation of the AMBA[1] bus. Additional to the setup of the developing platform, JAVA applications are used to configure the emulator and to transfer data from and to the system.

[1] Advanced Microcontroller Bus Architecture

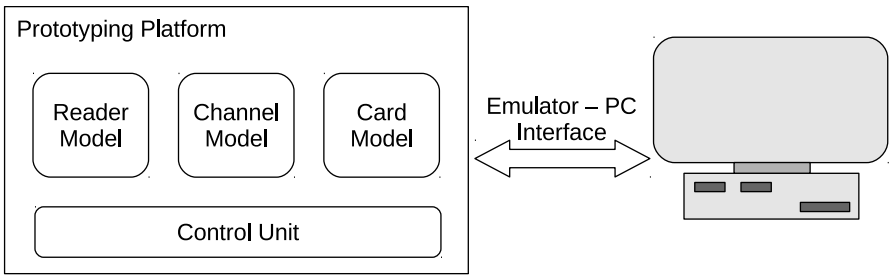


Figure 3.2: Architecture Choice - Single Board Approach. Reader and card model is implemented on a single developing platform.

The remainder of this chapter deals with a detailed view on the design of the reader-card emulation framework.

3.2 Use Cases

In order to identify the requirements of the reader-card emulator the following use cases were considered.

(A) A whole RFID system setup shall be verified concerning the power consumption for a given software implementation.

(B) A whole RFID system setup shall be verified concerning fault tolerance.

(C) The influence of faults on the power transfer between reader and card shall be verified.

(D) Performance data for specific units in the system shall be gathered (e.g., switching activity of distinct signals).

Flow:

- The software for reader and card is developed
- The emulator is configured (initialize reader and card registers, define fault scenario, etc.)
- The software for reader and card is uploaded to specific areas in the RAM of the emulator

- The emulation is started
- During emulation the system gives a response about the state of the processors
- During emulation the emulator gathers information about the power consumption of the system
- The emulator signals when the program has finished
- The gathered power and debug information is downloaded from the emulator for further computations

3.3 Requirements

The following requirements have to be fulfilled in order to reach the design goals.

- The emulator shall model the reader, card and the behavior of the contactless communication channel.
- The emulator shall estimate the power consumption of the reader and card hardware components.
- The emulator shall be capable to model the behavior of hardware faults at arbitrary time instances and at arbitrary locations in the modeled system.
- The emulator shall provide a mechanism to collect cycle accurate information about the system state.
- The emulator shall be built in a modular manner.
- The emulator shall be easy expandable.
- The emulator shall implement an interface to a PC for configuration and data exchange.

3.4 Overview

Figure 3.3 shows an overview of the whole emulator configuration. The setup consists of a PC (host system) responsible for configuration and post-computations and the actual emulation framework.

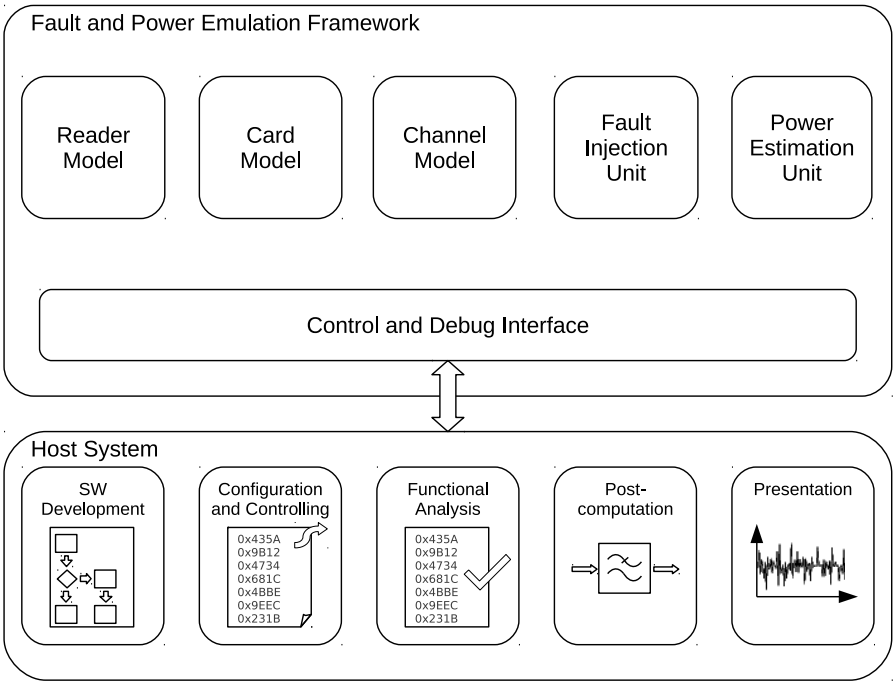


Figure 3.3: Emulation Framework - Overview. The setup consists of a host system responsible for configuration and post-computations and the actual emulation framework.

The main tasks of the host system are as follows:

- Software Development
  The software for reader and card is developed following the normal design flow. Additionally, functionality needed for the emulation process is integrated into the implementation (e.g., start and stop commands for power tracing).

- Configuration
  The emulator is programmed with the software for reader and card. This is done by specifying the location of the program in the RAM module of the emulator. Additionally, the program counter and the stack address is specified for each processor on the emulator.

- Controlling
  Each processor can be started and stopped individually. During runtime the emulator gives a response about the processor state. The emulator signals if a processor has finished execution. After execution the gathered data can be downloaded from the emulator.

- Functional Analysis
  The gathered information loaded from the emulator is verified (e.g., outcome of a cryptographic algorithm).

- Post-computation
  The gathered data is processed (e.g., filtering, downsampling of the collected power information).

- Presentation
  The processed data can be plotted.

The emulation framework itself holds the model of reader and card. These models consist of at least one processor, an RAM module and the bus. Additional units like a cryptographic co-processor etc. can also be added. In order to perform the estimation of the power consumption a Power Estimation Unit is included. This unit estimates the power consumption for each processor during the runtime of the emulation and generates cycle accurate power values. This information is provided to the host system which processes the data for further usage. The Fault Injection Unit is used to control the injection of faults at arbitrary locations into the target modules. This module makes use of saboteurs and trigger units. Trigger units are used to listen for a specific event (e.g., an activation signal or a specific value from a counter/timer). This information is then provided to the Fault Injection Unit. The Fault Injection Unit activates the saboteurs depending on the information from the trigger modules and its configuration. The saboteurs inject the faults if activated. In order to control the whole setup the Control and Debug Interface is used. This component consists of at least the interface to the PC and a bus for data exchange with the modules mentioned before. Optional, this component includes a processor to perform the configuration by software (e.g., to configure the Fault Injection Unit by a program executed by this additional processor). Also included into the emulator is a module to provide the exchange of data between the reader model and the card model. Together with a module modeling the behavior of the contactless channel the link between reader and card is emulated.
The whole emulation framework is designed as such that no interaction between emulator and host system is necessary during the emulation process. The emulator is configured and started by the host system. The emulation is performed without operation of the host. When the emulation has finished the gathered information can be downloaded from the platform.

Figure 3.4 shows the emulation framework from a different kind of view. The whole setup can be divided into a software related part and the hardware part. The hardware includes the components necessary to model reader/card and the contactless communication channel as well as modules for fault emulation and power estimation. The design and implementation of these components are subject of this thesis.
The framework can be used to verify software components such as communication protocols or high-level applications running on an operating system. The implementation of protocols and applications is not part of this thesis.

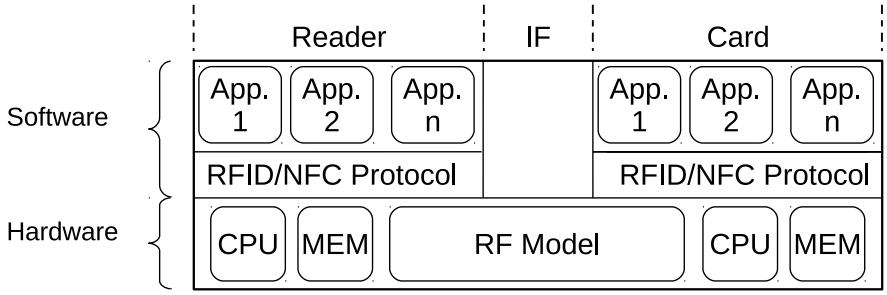


Figure 3.4: Layer of the emulation framework.

3.5 Components and Interfaces

This section deals with the requirements of the main components and its interfaces. The section is divided into two subsections dealing with the components on the emulation framework and the host system.

3.5.1 Emulation Framework

The emulation framework is divided into the following main components:

- Processor
  Executes the reader/card software.

- Bus
  System bus of the reader/card model.

- RAM
  Memory of reader/card.

- Channel Model
  Emulates the behavior of a contactless communication channel.

- Fault Injection Unit
  Enables the injection of faults in arbitrary hardware modules.

- Power Estimation Unit
  Estimates the power consumption of reader and card.

- Debug Interface
  Enables controlling and configuration of the platform via PC.

Processor

The processor executes the reader/card software. Because original hardware is not available due to disclosure policies a substitute has to be found. The emulator uses therefore an implementation of the SPARC V8 architecture (see Chapter 4).
An important requirement for this component is that the processor provides an interface to configure all relevant registers of the processor through the Debug Interface. Another requirement is that the processor must implement a mechanism to start and stop execution. Additionally, an interface to the reader/card bus is necessary to provide access to RAM, a co-processor, etc.
The processor is instantiated twice, once for the reader and once for the card model.

Bus

The bus connects all components necessary to emulate the reader and card. The main purpose of this component is to provide an interface between the processor and the RAM module. Additionally, the bus architecture provides an easy way to add new hardware components if necessary (e.g., a cryptographic co-processor). The emulator makes use of the AMBA implementation from Aeroflex Gaisler (see Chapter 4).
The bus module is instantiated twice, one for the reader the other for the card model.

RAM

Reader and card model have its own RAM module. An important requirement is that the module can be accessed through the debug interface. This interface is necessary to load the reader/card program into the RAM. Consequently, this module provides two interfaces. One interface connects the RAM to the reader/card bus. The other interface

supports the initialization of the RAM through the debug interface.
The RAM module is instantiated twice, one for the reader the other for the card model.

Channel Model

In order to emulate the contactless communication channel two aspects have to be considered. First, the data transfer between reader and card has to be possible. Second, the physical characteristics have to be modeled.
The data transfer between reader and card is implemented as a bidirectional FIFO. Therefore, the implementation of the FIFO has to provide an interface to the bus system of the reader and an interface to the bus system of the card.
The emulation of the physical behavior of the contactless communication channel is more complex. Task of the Channel Model is to emulate the power transfer from the reader device to the card. Because in a passive RFID system the card is supplied by the field of the reader, the applied model has to consider the field provided by the reader and the power consumption of the card. Consequently the Channel Model has to provide an interface to the reader bus which is used to configure the field strength defined by the reader device. Another interface is necessary to consider the current power consumption of the card. With this information the inductive coupling between reader and card can be modeled. Figure 3.5 shows the principle architecture of the module.
The available supply voltage of the card is modeled with a dedicated hardware module. According to [42], an equivalent circuit of the supply network of a contactless smartcard, as pictured in Figure 3.6, is used. $v_i(t)$ denotes the rectified voltage provided by the antenna of the card (the energy is provided by the reader device). $v(t)$ represents the voltage available for the card circuit. The capacitor C serves as buffer for the supply voltage to suppress voltage drops where the Zener diode is used to regulate the output voltage. The capacitor is charging/discharging depending on the energy provided by the reader and the current power consumption of the card circuit. As stated in [42] the error of the estimation is less than 2 %.

In order to implement this behavior in hardware, the Channel Model makes use of the Supply Voltage Estimation Unit (SVE) adapted from [12]. The SVE unit implements the relation given by Equation 3.1. $Q_C[t]$ is the charge of capacitor C at time instance t and Δt represents the reciprocal value of the clock frequency. V_z represents the breakdown voltage of the reverse biased Zener diode.

$$v[t+1] = \frac{Q_C[t] + \frac{v_i[t]-v[t]}{R_i} \cdot \Delta t - i[t] \cdot \Delta t}{C} \qquad if \quad v[t] < V_z \tag{3.1}$$

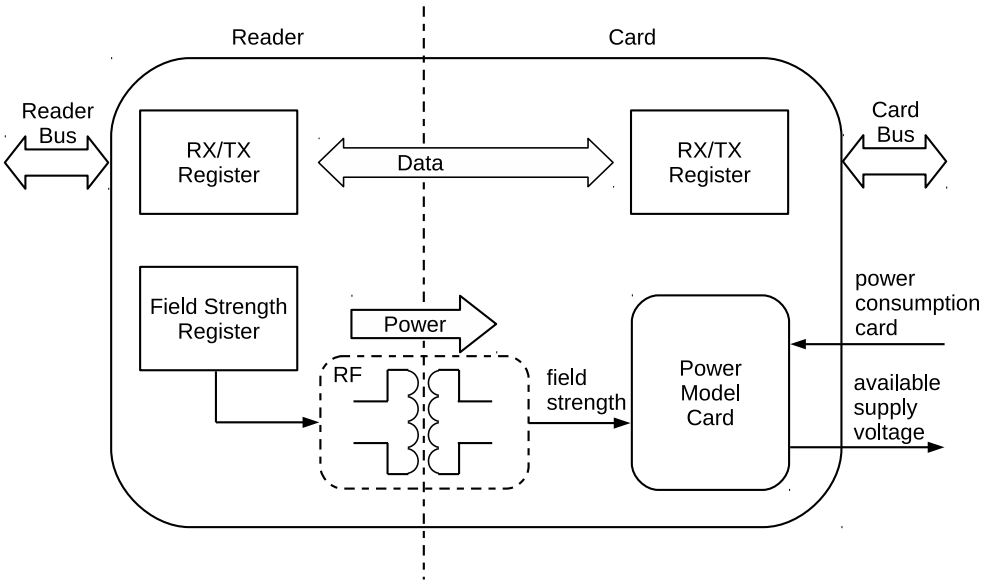


Figure 3.5: Channel Model. The model provides an interface to the reader bus for field strength configuration. Additionally, the current power consumption of the card is considered. With this information the available supply voltage for the card is modeled. Additionally, the model implements a bidirectional communication channel for data exchange.

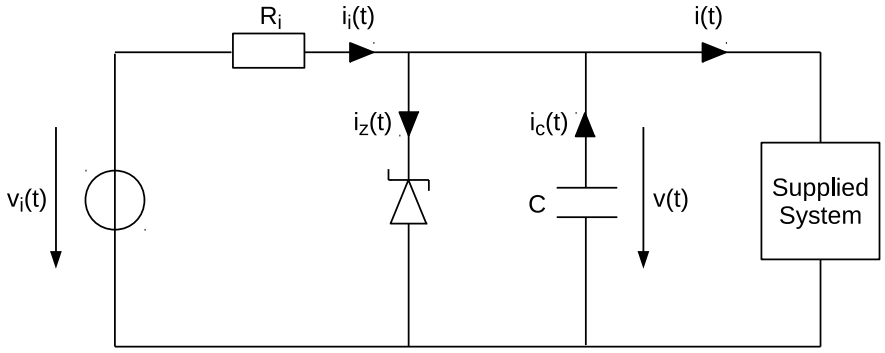


Figure 3.6: Equivalent circuit of the supply network of a contactless smartcard adapted from [42]. $v_i(t)$ denotes the rectified voltage provided by the antenna. $v(t)$ represents the voltage available for the card circuit.

Fault Injection Unit

The Fault Injection Unit is used to emulate the presence of faults. Faults can be injected at arbitrary locations in the target hardware at arbitrary time instances. The unit is partitioned into three parts and is adapted from [18]. The trigger units are used to observe the current system state. These modules are connected to certain signals of the target hardware. The results of the trigger units are provided to the fault injection controller. This unit processes the information obtained from the trigger units. The fault injection controller has also an interface for configuration. Depending on the information of the trigger units and the configuration the controller activates the saboteurs. The saboteur units are placed between a signal source and its sink. At this location the saboteurs are able to change the signal state (e.g., bit-flip, stuck-at-zero, stuck-at-one) observed by the sink.
The configuration of the controller has to be done either by the reader or by the card through software. Another possibility is that the fault injection controller is programmed through the Debug Interface. The interface of the fault injection controller can either be connected to the reader bus, the card bus or the Debug Interface. This feature is provided by the modular design of the emulation framework.
Figure 3.7 shows the principle structure of the Fault Injection Unit for N trigger units and M saboteurs. The signals denoted as STx ($x \in \{1, \ldots, N\}$) are signals used for the trigger units. With this information a predefined fault scenario can be executed depending on the system state. The signals denoted as SFx ($x \in \{1, \ldots, M\}$) are signals where faults should occur.

Power Estimation Unit

The Power Estimation Unit is used to estimate the power consumption of the reader and the card processor. This module estimates the power consumption in real time using a regression based model (see Section 2.4).
The design of the estimation unit makes use of the approach from [17]. The estimation process follows the relations (3.2) and (3.3). The estimated power consumption $\hat{P}$ can be partitioned into a *static* component including leakage and a *dynamic* component due to switching activity of the CMOS circuits. Equation (3.3) shows the implemented linear regression model. The vector $\mathbf{c}$ represents the coefficients determined during a characterization process. The vector $\mathbf{x}$ combines the model parameters. These parameters are derived from the state of the individual hardware components (e.g., CPU modes, memory access) which means that no data dependent information is used. The gained power information is provided to the Debug Unit.

$$\hat{P} = \hat{P}_{Static} + \hat{P}_{Dynamic} \tag{3.2}$$

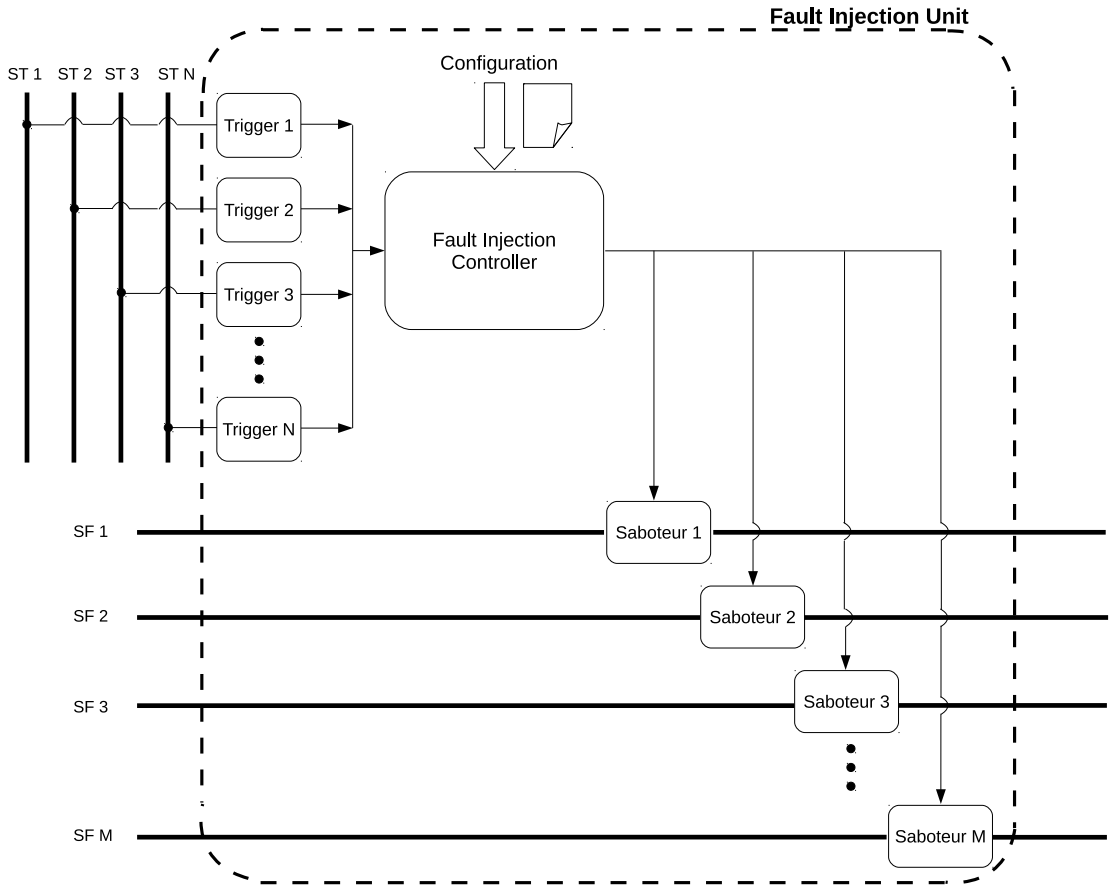


Figure 3.7: Architecture of the Fault Injection Unit for N trigger units and M saboteurs. Signals denoted as STx ($x \in \{1, \ldots, N\}$) are used for the trigger units. Signals denoted as SFx ($x \in \{1, \ldots, M\}$) are signals where faults should occur. The fault injection campaign is defined by the controller depending on its configuration.

$$\hat{P}[\mathbf{x}[t], \mathbf{c}] = c_0 + \sum_{i=1}^{n} c_i \cdot x_i \qquad \mathbf{x} = [x_1, \cdots, x_n], \mathbf{c} = [c_0, \cdots, c_n] \tag{3.3}$$

The estimated power consumption is related to the used state signals $\mathbf{x}$ and the power model. Therefore, the accuracy depends on the characterization process of the reader/card processor.
The Power Estimation Unit is instantiated twice, one for the processor of the reader the other for the processor of the card.

Debug Interface

The main tasks of the Debug Interface are to provide an interface to configure and control the emulation and to collect power information provided by the Power Estimation Units. Additionally, the Debug Interface can collect information about the system state. This information is then provided to the host system. The link between the Debug Interface and the host system is provided by UART. The Debug Interface resides between the UART interface provided by the FPGA prototyping platform and the implementation of the reader-card emulator. The module is implemented as bus architecture. The nodes of the bus are those components which need a direct link to the host system together with the interface controlling the data transfer from and to the host system. All nodes are configured as slave except the interface controlling the data transfer to the host system. The Debug Interface uses the AMBA bus implementation from Aeroflex Gaisler (see Chapter 4).
In order to collect data produced during the emulation process two distinct modules are implemented. The first one is the Power Performance Debug Unit (PPDU) which is adapted from [26]. This module was designed to send performance statistics of a system (e.g., power consumption) during run-time to a host PC over a 100Mbit/s Ethernet interface. Because of the limited bandwidth and protocol overhead the transfer of cycle accurate information is not possible. Therefore, the PPDU provides a mechanism to aggregate the collected performance data which is then sent as one packet to the host PC. Figure 3.8 shows the principle architecture of the PPDU. The inputs to the PPDU are the performance values of the observed system. For each performance value an aggregation counter is instantiated. These counters accumulate the information until a new Ethernet frame is ready to send. The accumulated performance information is built into packets. These packet are then inserted into the Ethernet frame.
The PPDU implements an AMBA interface to start and stop the tracing process. This allows the configuration by software. The instructions necessary to start and to stop the PPDU are directly inserted into the target code. Additionally, a generic interface is provided to define the performance signals to observe.
The module is available as synthesizable VHDL model.

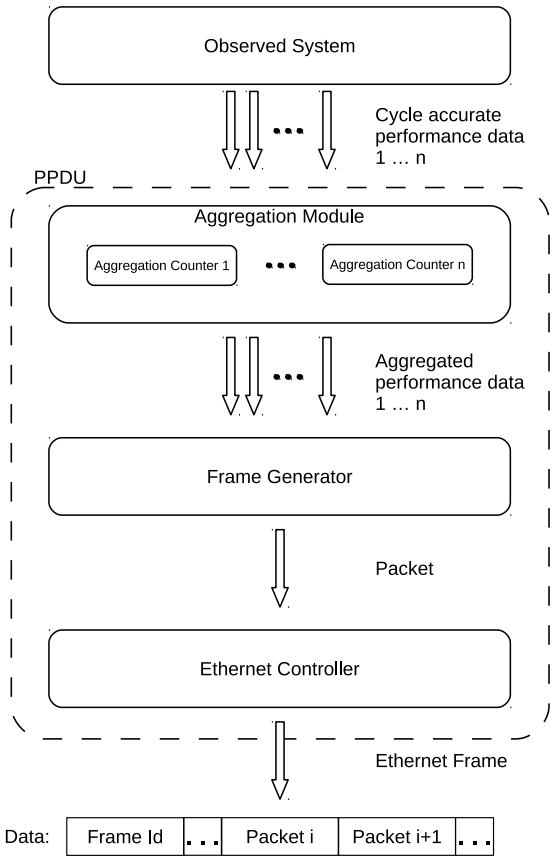


Figure 3.8: Principle architecture of the PPDU.

Because cycle accurate information is not handled by the PPDU a second module is implemented to fulfill this requirement. The idea of the Cycle Accurate Debug Unit (CADU) is to store the debug information into memory during emulation. After the emulation process has finished the information can be downloaded from the platform.
In order to keep the design of the CADU as simple as possible the bit size of the logged information per cycle is fixed. Therefore, no overhead due to additional information needed for decoding of the data is introduced. Similar to the design of the PPDU, the CADU provides an AMBA interface to start and to stop the tracing process. This functionality is provided by the CADU controller. During the logging, the CADU writes a fixed amount of debug information to the memory. After the CADU is stopped the number of logged information is also written to the memory. If the logged information exceeds the amount of free memory the CADU is stopped and the number of logged information is also written to the memory. With this information the logged information can be downloaded by the host system.
Figure 3.9 shows the principle architecture of the CADU.

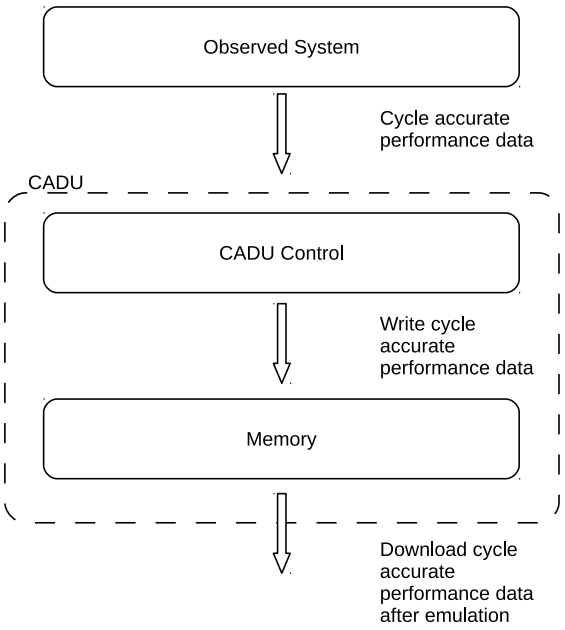


Figure 3.9: Principle architecture of the CADU. During the logging process the debug information is written to the memory. The information is then provided to the host system.

The architecture of the Debug Interface is shown in Figure 3.10. The main component of the module is the AMBA bus with the AMBA-UART bridge as master. All components with a need for a direct interface to the host system are nodes of this bus. The remaining components are the CADU and the PPDU. The CADU is connected to the bus to provide a direct link to the host system. The PPDU has its own interface to the host and is not connected to the remainder of the Debug Interface.

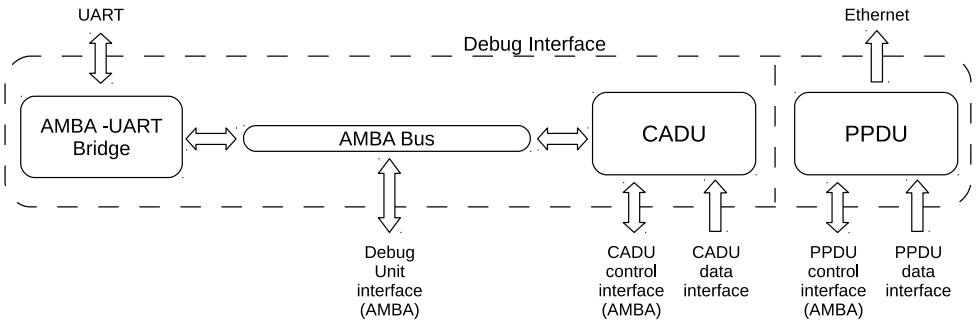


Figure 3.10: Principle architecture of the Debug Interface.

System Dependencies

Table 3.1 gives an overview about the dependencies between the main components of the emulation framework. The fault injection unit is omitted because the interfaces of the trigger, saboteurs and the controller depend on the implemented use case.

	Processor Reader	Processor Card	Bus Reader	Bus Card	RAM Reader	RAM Card	PE Reader	PE Card	RF	Debug IF
Processor Reader			x				x			x
Processor Card				x				x		x
Bus Reader	x				x				x	
Bus Card		x				x				
RAM Reader			x							x
RAM Card				x						x
PE Reader	x									x
PE Card		x							x	x
RF			x					x		
Debug Interface	x	x			x	x	x	x		

Table 3.1: Dependencies between main components of the emulation framework. The fault injection unit is omitted.

3.5.2 Host System

This section deals with the software design of the host system. Two JAVA applications are used to communicate with the emulation setup. The first one is needed to control and configure the whole setup and to download emulation results. The second is adapted from [26] to capture the power traces during an emulation process.

Fault Injection Interface

The Fault Injection Interface application is used to control the whole setup and to download the results after emulation (e.g., read results of the CADU). To enhance the usability a graphical user interface is provided. With this application the emulation

can be started and stopped. Additionally, access to the debug bus of the emulator is provided. The main functionalities are as follows.

- **Open** and **close** serial connection.
- **Write** arbitrary data to the debug bus.
- **Read** data from the debug bus.
- **Start** and **stop** emulation.
- Download software for reader and card to RAM (**program**).

Figure 3.11 shows a simplified class diagram of the application. The design uses four classes. The class $FaultInjectorIf$ provides the graphical interface for the user. All commands are forwarded to the *Control* class which abstracts the underlying functionality. Depending on the command, the request is either handed directly to $SerialInterface$ or to $DebugSupportUnit$ if the command involves the CPUs. The class $DebugSupportUnit$ holds all information necessary to control the emulation (e.g., initialize CPU registers, start and stop CPU). The $SerialInterface$ class handles the communication with the platform.

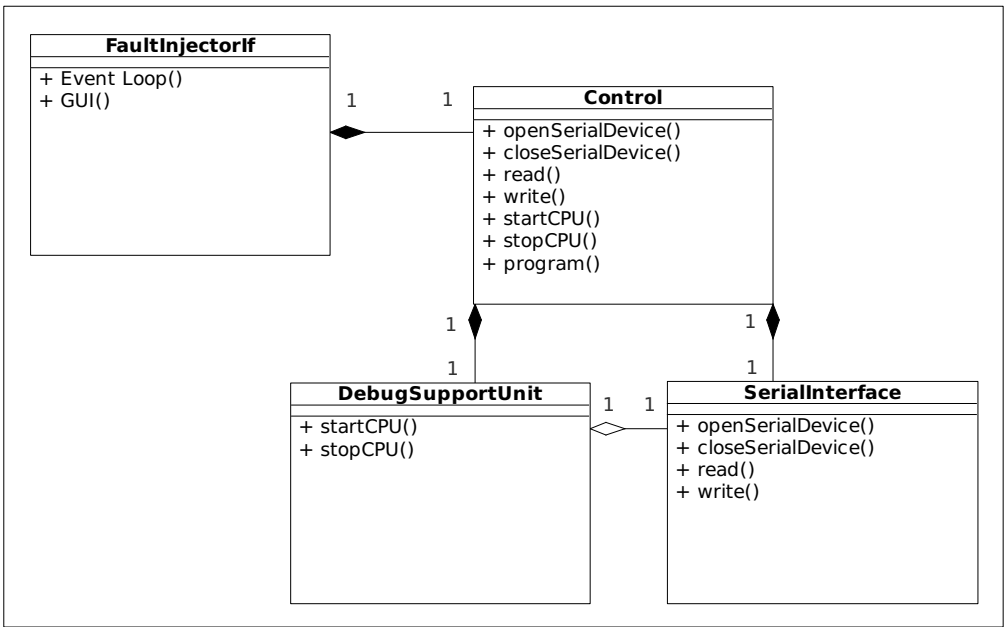


Figure 3.11: Fault Injection Interface - Class Diagram.

Ethernet Dump

The Ethernet Dump application is adapted from [26] to capture the information provided by the PPDU during emulation. The application provides a graphical user interface to control the capture process. When the capture process is started the application collects all Ethernet frames sent by the PPDU. After the capture process, the PPDU packages are extracted from the Ethernet frames. The PPDU packages are then parsed depending on a predefined package format. The package format defines the content of a package, which must reflect the hardware configuration (number of traced signals, signal bit width, etc.).
After all information is extracted the data can be filtered and plotted. Additionally, the information can be exported for further computations.

3.6 System Architecture

The architecture of the emulator is based on the AMBA bus (see Chapter 4). Depending on the required bandwidth, the Advanced High-performance Bus (AHB) or Advanced Peripheral Bus (APB) is used.
Because of the modular design of the emulation framework, different architectures are possible. Figure 3.12 and 3.13 picture two different possibilities. The modular design is enabled by the bus-based design. The advantage of the modular design is that the framework can be adapted for specific use cases. Additionally, modules can be omitted or added in a simple manner.
The first configuration (Figure 3.12) shows the architecture for a system where the reader model also functions as controller for the fault injection unit and the power and debug trace units (PPDU, CADU). This architecture has the advantage that no additional unit is needed to configure the system. The configuration and control instructions can be directly included into the software of the reader model.
The second configuration (Figure 3.13) shows the architecture for a system where the fault injection unit and the power and debug trace units are controlled through the debug bus. This can either be done by the host system which has full control over the debug bus or alternative by an optional processor which is connected to the debug bus. Beside the architectures shown, other configurations are possible. The actual architecture depends on the use case. Additionally, modules can be omitted to spare the space on the FPGA or the system can be extended by new modules.
It should be noted that the reader and card system are separated from each other. Therefore, no unwanted interferences between the systems are introduced. Also the debug bus is separated from reader and card which allows to monitor the system without manipulation of the systems.

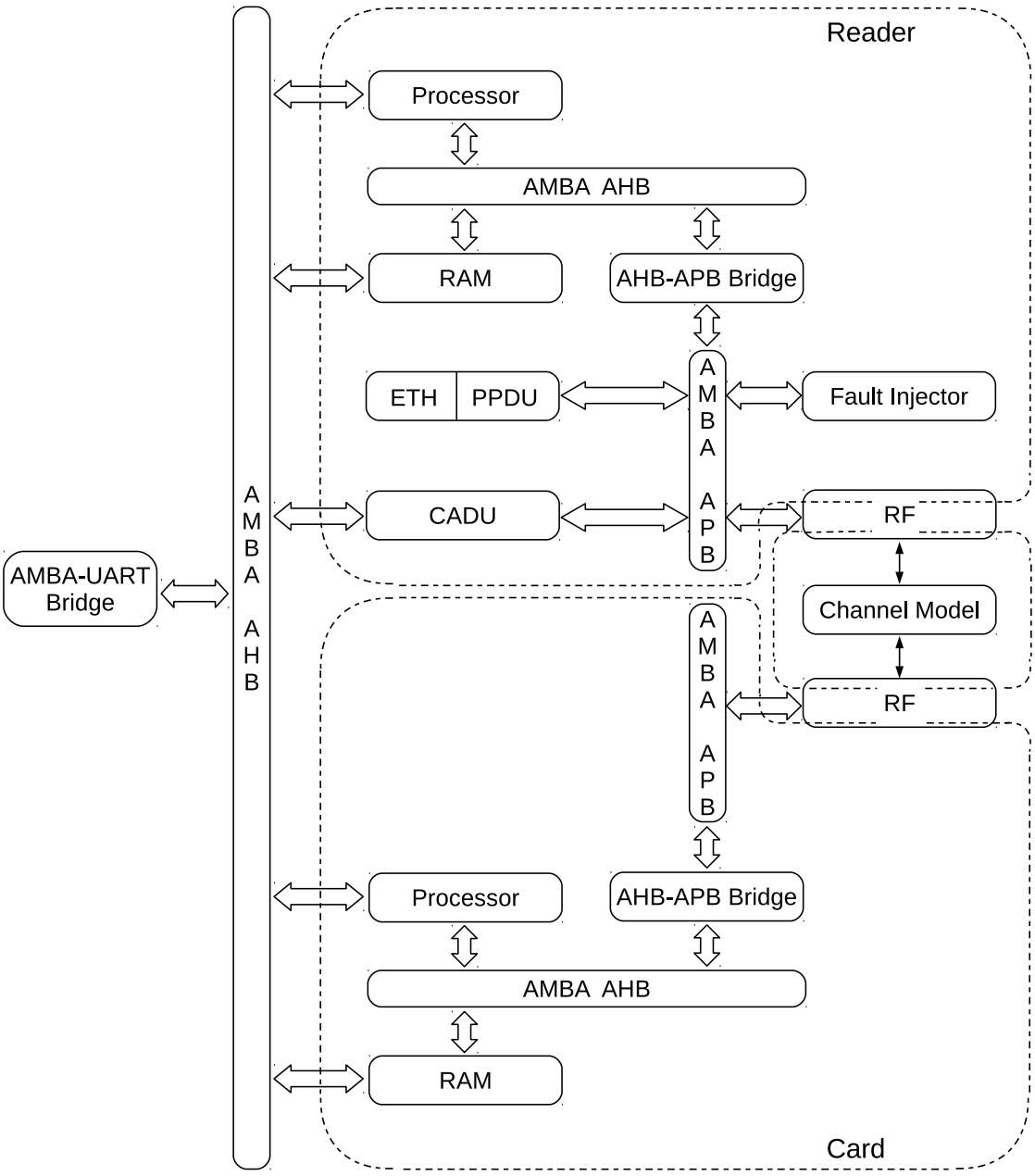


Figure 3.12: Architecture of the emulator hardware.

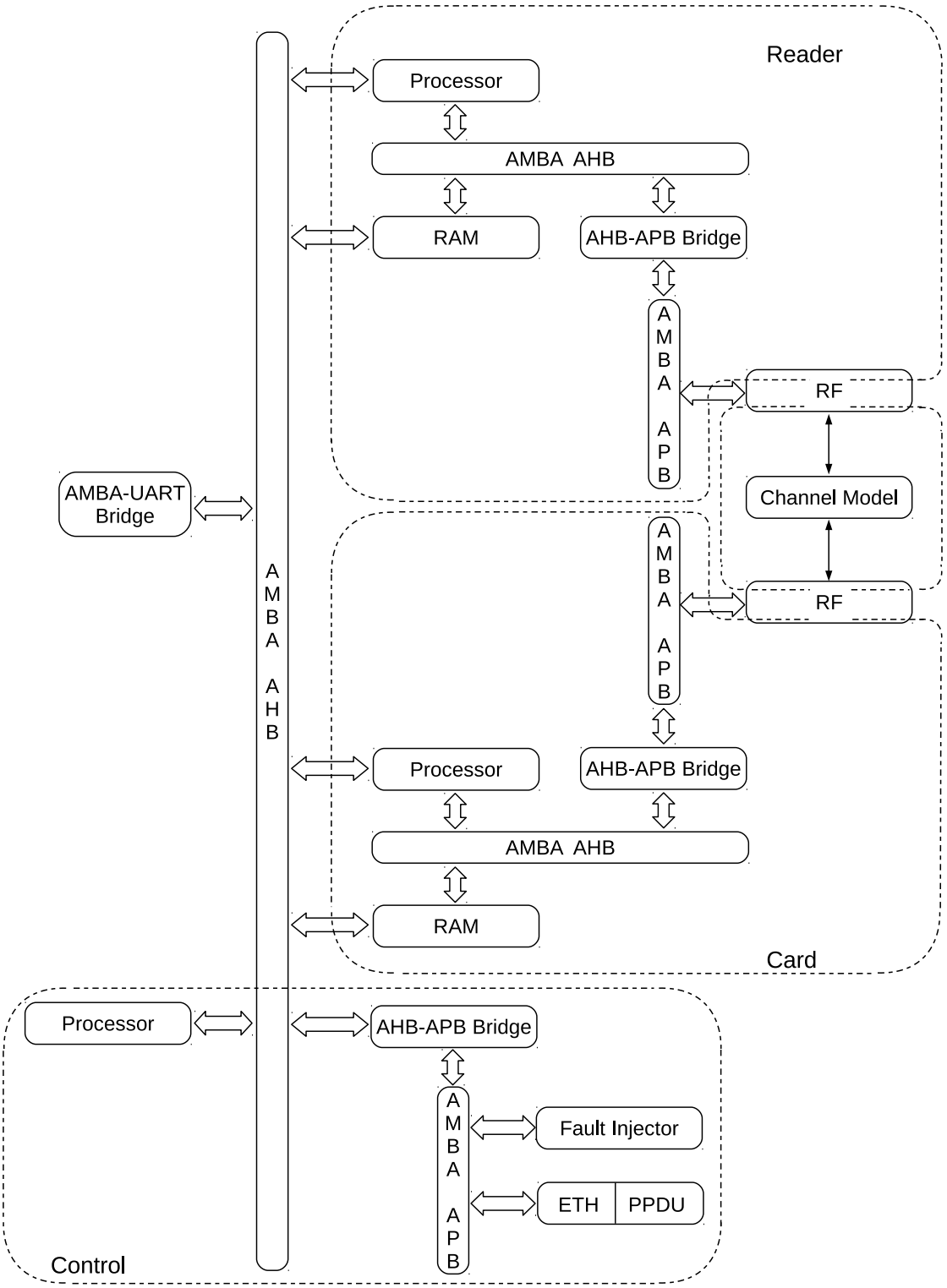


Figure 3.13: Optional architecture of the emulator hardware.

4 Implementation

This chapter deals with implementation specific topics of the thesis.
First the target platform and the used IP library are described. The considerations from Chapter 3 are refined and implementation details about the distinct modules are provided. The chapter concludes with software related tasks necessary for controlling and data acquisition.

4.1 Tools

Software for the LEON3 processor was compiled using the BCC - Bare-C Cross-Compiler [13] which includes several packages (e.g., GNU GCC C/C++ compiler v3.4.4, v4.4.2). The hardware of the emulator was synthesized using XILINX ISE 13.4. Hardware simulations were performed with ModelSim 6.6g. Additionally, MATLAB R2012a was used for post-computations and the Eclipse JAVA developing environment was used to develop the software interface of the emulator.
Synthesis and simulation of the hardware was performed on a 6-core 3.2 GHz AMD Phenom-II with 16GB of RAM.

4.2 Xilinx ML605 Prototyping Platform

This section gives an overview about the used prototyping platform ML605, developed by Xilinx [44]. The platform provides various peripherals together with a Virtex-6 XC6VLX240T-1FFG1156 FPGA. The configuration of the FPGA is performed using the JTAG USB port. A high-level block diagram of the ML605 is shown in Figure 4.1. The following features of the platform are used for the implementation of the emulator.

Virtex-6 XC6VLX240T-1FFG1156

The Virtex-6 FPGA family is built on 40 nm technology and is divided into three subfamilies [45].

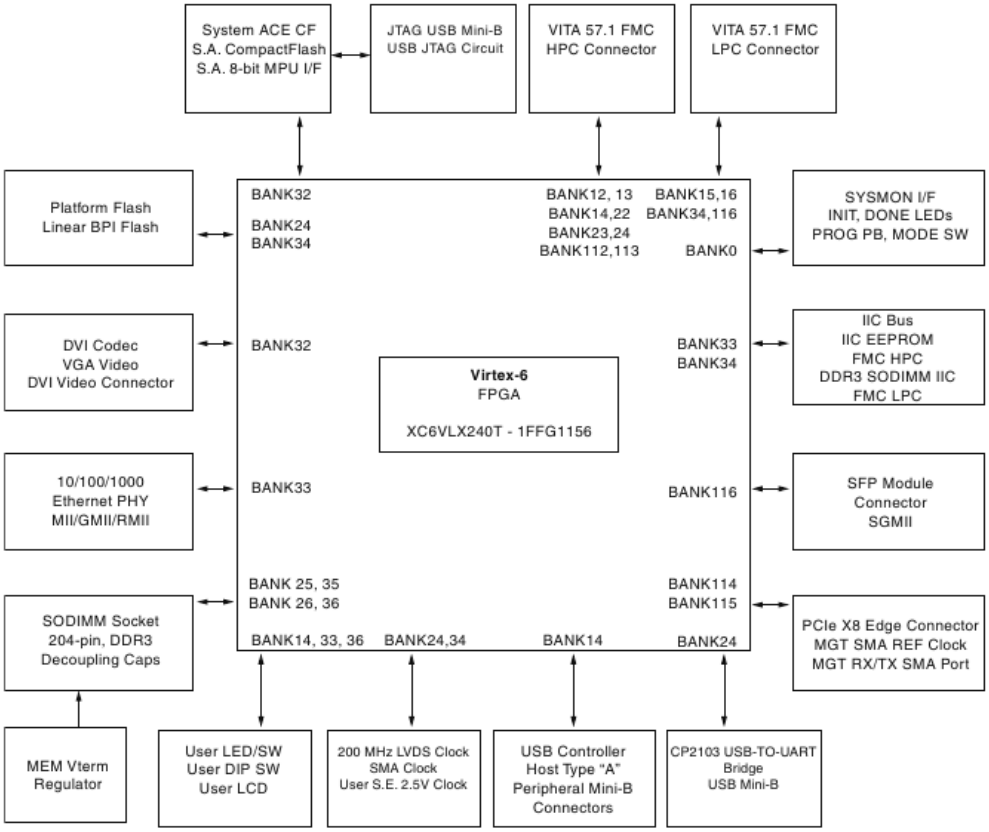


Figure 4.1: ML605 Developing Platform - Block Diagram [44].

- **Virtex-6 LXT supporting high-performance logic with advanced serial connectivity**

- Virtex-6 SXT supporting highest signal processing capability with advanced serial connectivity

- Virtex-6 HXT supporting highest bandwidth serial connectivity

The used version includes 37680 slices which are composed of four LUTs, eight flip-flops as well as multiplexers and arithmetic carry logic. Two slices form a configurable logic block (CLB). Additionally, the Virtex-6 XC6VLX240T includes 14976 kbit of block RAM which results in approximately 1.785 Mbyte RAM. The block RAM is arranged in 416 dual-port blocks of 36kbits.

512 MB Memory

The ML605 developing board includes 512 MB of DDR3 SODIMM volatile memory available for user applications.

10/100/1000 Tri-Speed Ethernet PHY

The developing board includes an onboard PHY device for Ethernet communication at 10, 100 and 1000 Mbit/s. The interface between FPGA and PHY device can be configured through jumpers as GMII/MII, SGMII and RGMII.

USB-to-UART Bridge

The ML605 board includes an USB-to-UART bridge CP2103-GM from Silicon Labs [38] for communication with a PC over an USB cable. The interface between the FPGA and USB-to-UART bridge are established by four signal pins: Transmit (TX), Receive (RX), Request to Send (RTS) and Clear to Send (CTS).

4.3 Advanced Microcontroller Bus Architecture AMBA

4.3.1 Advanced High-performance Bus

This section gives an introduction to the Advanced High-performance Bus (AHB) defined by the AMBA 2.0 specification [4], developed by ARM Limited.
The AHB is a multi master high-performance bus which features high-bandwidth operations. It is used as system backbone to connect processors, memory etc. An AHB includes the following components:

- AHB master
  Only one master at a time is allowed to use the bus actively. The AHB allows a maximum of 16 masters.

- AHB slave
  Slaves respond to a master request. They must provide status information about the ongoing operation.

- AHB arbiter
  The arbiter is responsible to ensure that only one master uses the bus at a time.

- AHB decoder
  The decoder is used to translate the address into a select signal for each slave.

The AHB supports the following features:

- Burst transfers
- Split transactions
- Single-cycle bus master handover
- Single-clock edge operation
- Non-tristate implementation
- Wider data bus configurations

Because the AMBA AHB features a non-tristate implementation the bus interconnection is built upon a central multiplexer scheme. All bus masters drive address and control information for the transfer they wish to perform. The arbiter is then responsible to decide which master is forwarded to all slaves. Similar a central decoder is necessary to route the response of a selected slave back to the masters.
If a master wants to perform a transfer, a request is generated to the arbiter. The arbiter then chooses from all transfer requests which master is allowed to perform the bus transfer depending on a priority scheme. Details on the priority scheme are not defined by the AMBA specification.
A transfer consists of exactly one address cycle and at least one data cycle. The simplest operation is a non-burst transfer which lasts two cycles after the master is granted bus access. After the first rising clock edge the master drives the address and the control information. At the second rising clock edge the slave samples the information provided by the master and at the third rising clock edge the master samples the slave response. The result is a two cycle, zero wait state transfer. If the slave is not able to perform the master request immediately the slave has the ability to insert wait states. This is done by a *ready* signal which is hold to zero as long as the master has to wait. Figure 4.2 shows the zero wait state transfer where Figure 4.3 shows a write transfer with wait states. Additionally the slave gives a feed back about the ongoing transfer using a *response* signal. The possible values for the *response* signal are OKAY, ERROR, RETRY and SPLIT. If the transfer has successfully completed the slave drives the *ready* signal high and replies with the OKAY response. The ERROR response indicates a problem during the transfer where the RETRY and SPLIT response is used to indicate that the transfer should be repeated and provides therefore a mechanism to release the bus for other

masters. The RETRY response indicates that the master should retry the transfer until it completes. Only masters with higher priority will gain access to the bus. If the SPLIT response is used the arbiter will change the priority scheme. Any other master requesting a transfer will get bus access. In order to complete a SPLIT transfer the slave must inform the arbiter that the transfer can be finished. The AHB provides therefore a mechanism to release the bus if a slave is not able to handle the transfer immediately to prevent high latencies.

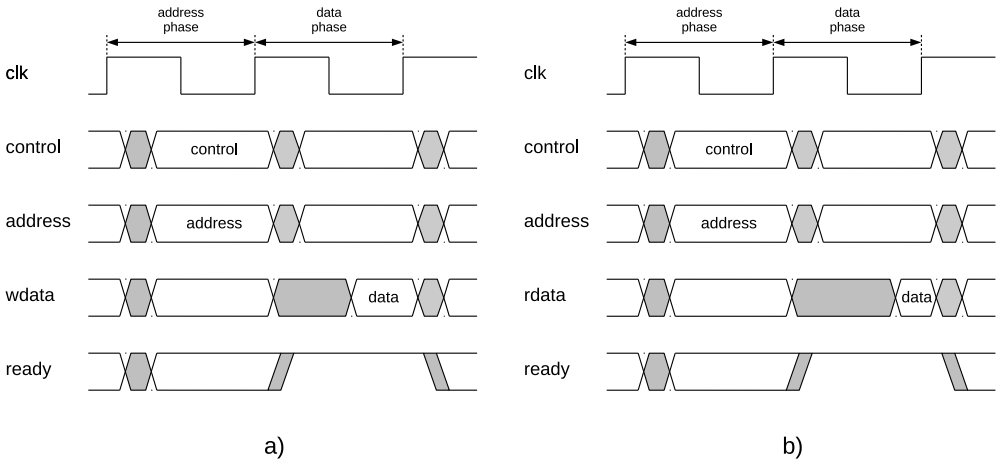


Figure 4.2: Zero wait state AHB transfer: a) write transfer (write data to slave) b) read transfer (read data from slave) [4].

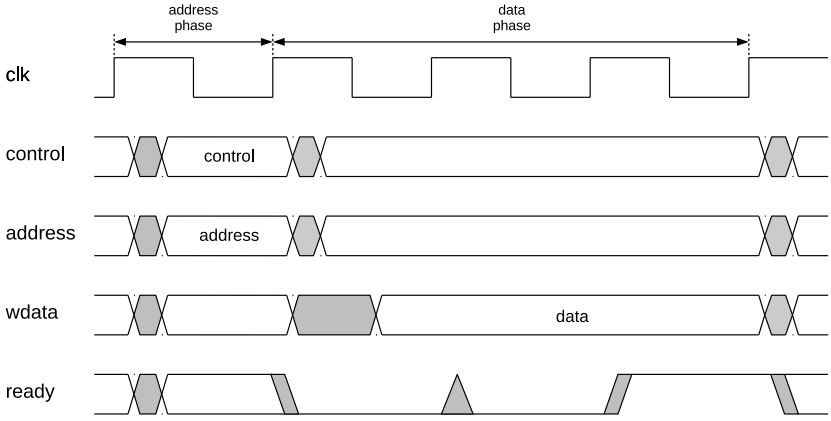


Figure 4.3: AHB write transfer (write data to slave). The slave drives the *ready* signal low indicating that the write operation can not be performed immediately [4].

4.3.2 Advanced Peripheral Bus APB

This section gives an introduction to the Advanced Peripheral Bus (APB) defined by the AMBA 2.0 specification [4], developed by ARM Limited.
The APB bus is typically used to connect peripherals with low-bandwidth requirements to a high-performance system backbone bus (e.g., AHB). The APB is therefore optimized for minimal power consumption and reduced interface complexity. There are two different kinds of nodes. The APB Slave which represents the peripheral modules with an APB interface and the APB Bridge. The APB Bridge functions as bus master and is also slave to a high-performance bus. The purpose of this unit is to convert a transfer on a high-performance bus to an APB transfer. Only one APB Bridge is allowed per APB.

The protocol operates on three states.

- IDLE
  No transfer is required (default state).

- SETUP
  If a transfer is performed the state changes to SETUP. This state lasts only one clock cycle and sets the appropriate *select* signal for the slave.

- ENABLE
  After SETUP the state changes to ENABLE and the *enable* signal is asserted. The *address*, *write* and *select* signal do not change during the transition from SETUP to ENABLE. The ENABLE state lasts also only one cycle. After ENABLE, the state changes to IDLE if no further transfer is pending or to SETUP for another transfer.

Figure 4.4 shows the basic write and read transfer. In order to optimize the power consumption of the protocol the *address* and *write* signal do not change until the next transfer.

4.4 Gaisler IP Library

This section gives an overview about the open source IP library from Aeroflex Gaisler ([14], [15]). The library includes various modules and provides reference designs for different FPGA development platforms (including ML605). The whole library is based on the AMBA AHB/APB.
Following, IP cores used for the emulator are discussed.

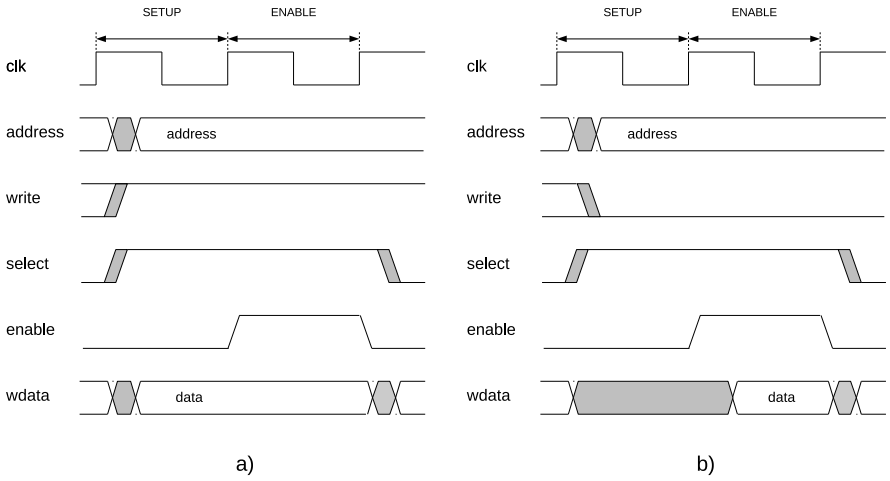


Figure 4.4: AMBA APB transfer: a) write transfer b) read transfer [4].

4.4.1 LEON3 - High-performance SPARC V8 32-bit Processor

Figure 4.5 shows a block diagram of the LEON3 processor. The processor implements a 7-stage pipeline with Harvard architecture, hardware multiplier and divider, a floating-point unit, a memory manage unit and provides an interface for a co-processor. The cache is divided into data-cache and instruction-cache and provides an AMBA AHB master interface. Additionally, a debug interface and multiprocessor support are provided [14]. The processor implements the SPARC V8 architecture.
The module is provided as VHDL and can be configured through generics (e.g., enable/disable FPU, MMU).

4.4.2 DSU3 - LEON3 Hardware Debug Support Unit

In order to debug the implemented hardware a Debug Support Unit is provided. This unit has a direct interface to the LEON3 processor's register. With the DSU3 the execution on the processor can be started and stopped (the processor is in debug mode). When the processor is halted the registers can be accessed through the DSU3.
The DSU3 implements an AMBA AHB slave interface and can be accessed by an AHB master residing on the same bus. The DSU3 can be used together with an interface for external communication (e.g., UART, ETHERNET, JTAG) to enable debugging from an external host. The DSU3 is able to control up to 16 processors.

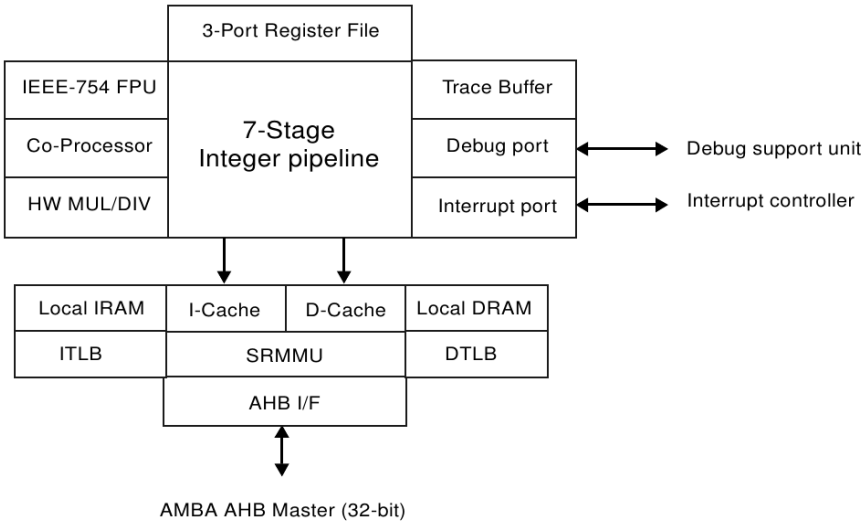


Figure 4.5: Block diagram of the LEON3 processor [14]. Distinct features can be disabled through generics.

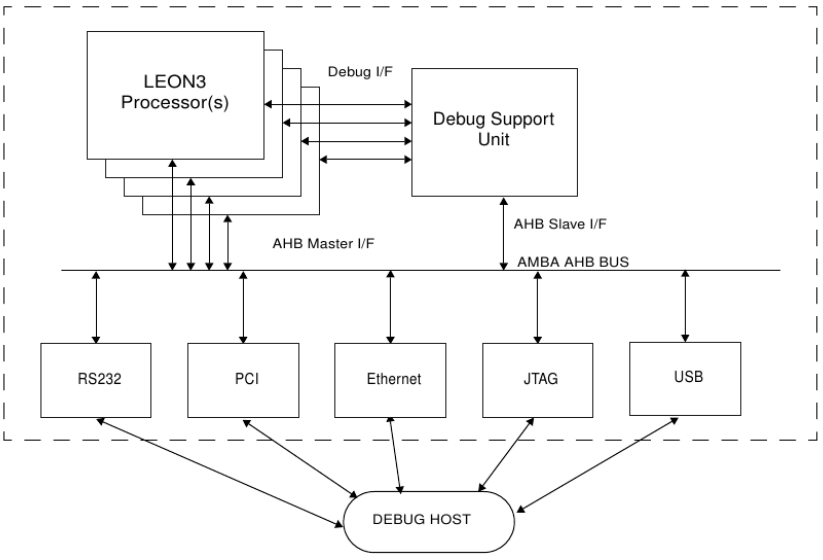


Figure 4.6: Overview of the Debug Support Unit [14].

4.4.3 AHBUART- AMBA AHB Serial Debug Interface

The AHBUART enables access to AHB from an external host through UART and can be used to communicate with the DSU3. With this unit any address on the AHB can be accessed. The module implements an AHB master interface for data exchange and an APB interface for configuration (e.g., set baud rate). Additionally, if the baud rate is not set by software, the baud rate will be discovered automatically [14].
The AHBUART implements the protocol shown in Table 4.1. The protocol starts with an eight bit header including the command (2 bit) and the data length (6 bit) defining the number of words (4 byte). The write command consists of the header, a four byte AHB address and 1 to 2^6 words of data (defined in the header). The write access does not return a response. The read command consists of the header and a four byte AHB address. The read command returns the number of data words defined in the header of the read command (1 to 2^6 words).

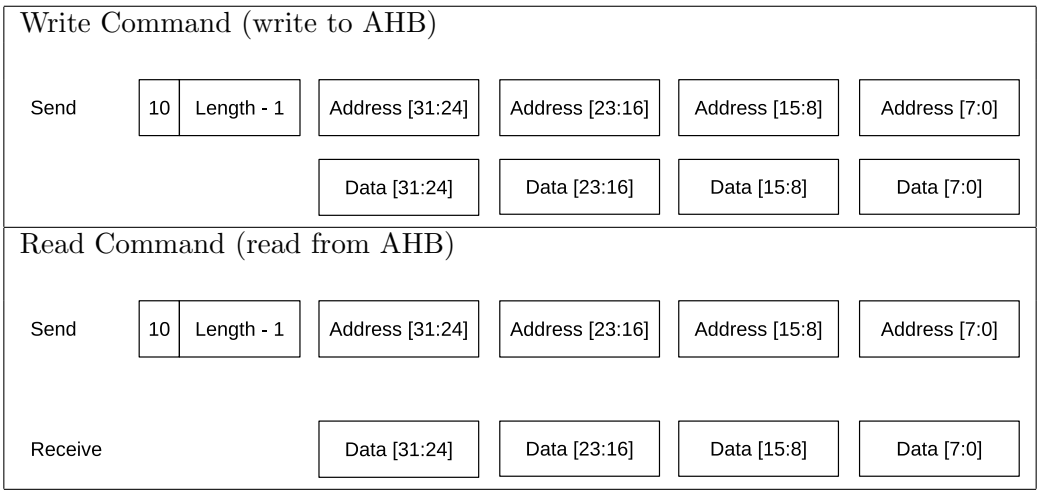


Table 4.1: AHB-UART protocol. *Length* defines the number of data words (4 byte)

4.4.4 AMBA AHB/APB

The Gaisler IP core library is based on the AMBA AHB/APB according to the AMBA 2.0 standard (see Section 4.3). Therefore, mostly all modules include an AHB or APB interface. The AHB is implemented through the AHBCTRL [14] module which supports up to 16 AHB masters and 16 AHB slaves. The APB is implemented through the APBCTRL [14] module which supports up to 16 APB slaves and acts as an AHB to APB bridge.
Additionally, the AMBA implementation supports a plug&play mechanism for AHB and APB [15]. The plug&play information is implemented as read-only table mapped to a fixed address space on the AHB (0xFFFFF000 default). This mechanism provides

information about the nodes and their configuration on the bus. This information can be read by software in order to detect the available hardware components.

4.4.5 AHBRAM - Single-port/Dual-port RAM with AHB interface

AHBRAM utilizes the block RAM of the used FPGA and implements an AHB slave interface with 32 bit data width. The size of the RAM can be defined through VHDL generics. The AHBRAM module is available as single-port and dual-port version if the target hardware supports dual-port block RAM. If dual-port is used, read and write collisions between the ports are not handled and must be prevented by the user. The write access has one wait state, where the read accesses are zero-wait state.

4.4.6 AHB to DDR3 Wrapper

The Gaisler IP core reference design for the ML605 developing platform utilizes the DDR3 RAM of the developing board. In order to access the external RAM the XILINX Memory Interface Generator (MIG) tool [43] is used which enables the generation of a memory interface for different XILINX FPGAs and memory.
The Gaisler ML605 reference design provides a tool chain to generate the memory interface to the external RAM via MIG. Additionally, an AHB wrapper module is provided to connect the memory interface to an AHB. The purpose of this module is to translate the AHB access to the native interface of the RAM.
Figure 4.7 and Figure 4.8 show a simplified state diagram of a read and write access. The module implements two state machines. The first state machine works on the system clock domain and handles the AHB requests. These requests are translated and provided to the second state machine. The second state machine works on the memory clock domain and handles the communication with the memory interface.

4.5 Emulation Framework

This section refines the considerations from Section 3.5.1 and gives a detailed view on the implementation of the emulation framework.

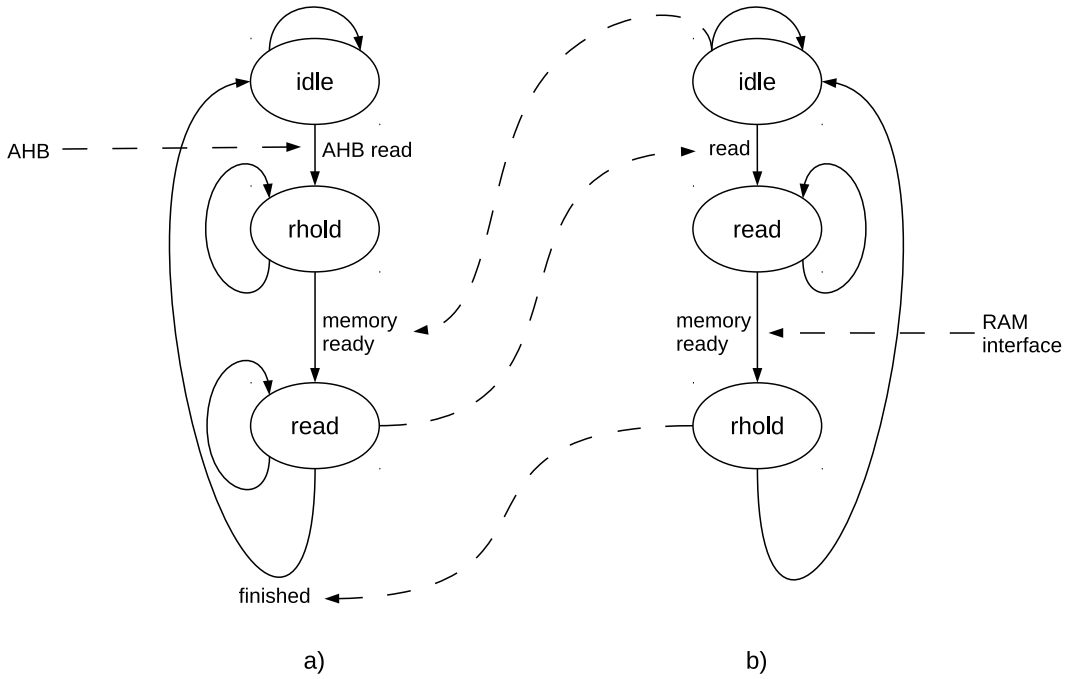


Figure 4.7: AHB DDR3 memory read access. The solid lines represent state transitions. The dashed line represents dependencies between the state machines and the involved components. a) state machine on AHB clock domain b) state machine on memory clock domain.

4.5.1 Processor

As mentioned in Section 3.5.1 the SPARC V8 LEON3 is used to implement the reader and card processor. The reader processor is connected to the system bus of the reader where the card processor is connected to the card bus. The processor is the only AHB master in the reader/card system.
The configuration and controlling of reader and card is done via the DSU3 (see 4.4.2). Both processors are connected to one DSU.

4.5.2 Bus

The system bus of reader and card are implemented as AHB/APB using the design from Gaisler (see 4.4.4). The AHB is used for modules with high-bandwidth requirements where the APB provides access to modules with lower bandwidth. The default configuration of the AHB includes the processor as AHB master, the RAM as AHB slave and an AHB/APB bridge with its AHB slave interface. Additional modules can be easily

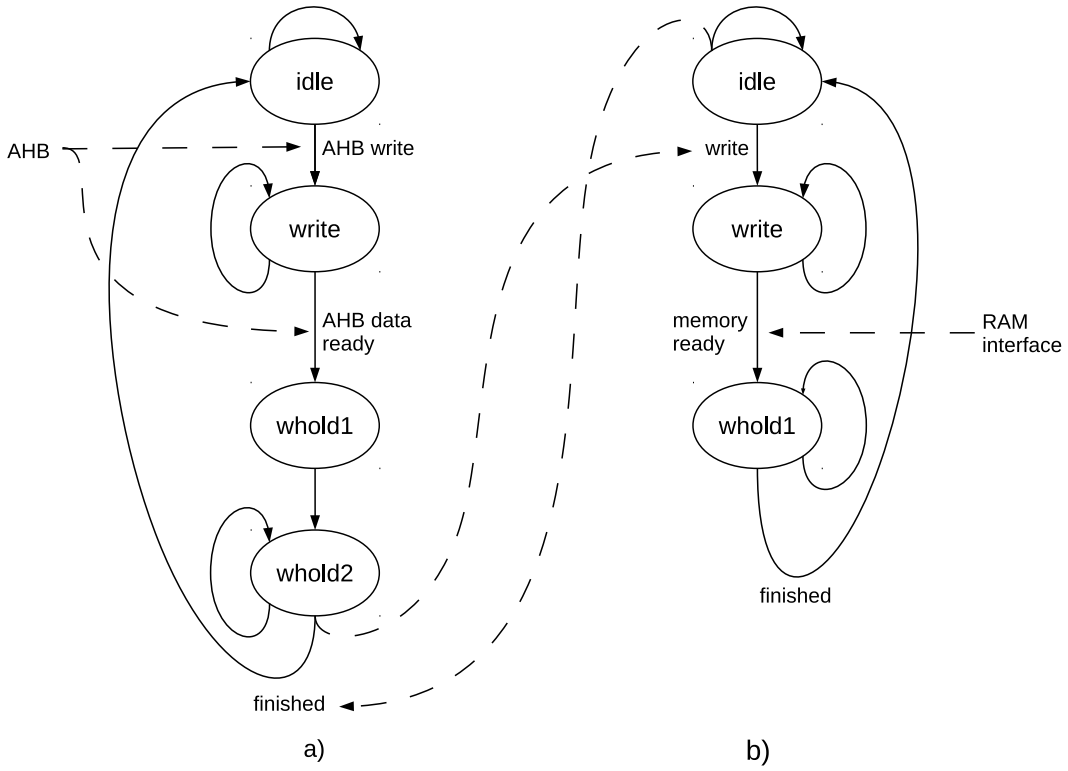


Figure 4.8: AHB DDR3 memory write access. The solid lines represent state transitions. The dashed line represents dependencies between the state machines and the involved components. a) state machine on AHB clock domain b) state machine on memory clock domain.

integrated by connecting them to the AHB or APB depending on their requirements.

4.5.3 RAM

The RAM modules hold the reader and card software applications. An important requirement of the used RAM module is that access from the host system must be possible in order to initialize the emulator. The initialization should be done via the debug interface. Therefore, the module must provide two interfaces. One for the normal operation on the system bus of reader and card and a second interface for configuration via the debug bus.
The ML605 developing platform provides two possibilities to implement RAM. The first possibility is to utilize the block RAM of the FPGA using the AHBRAM module provided by Gaisler (see 4.4.5). The second possibility is to exploit the external DDR3 RAM provided by the developing board. In order to exploit the full capabilities of the

ML605 developing platform a modular approach was taken.
The design supports the usage of the block RAM as well as the usage of the external RAM. As mentioned in Section 4.4.5 and Section 4.4.6 the Gaisler IP core library supports modules to connect block RAM and DDR3 RAM directly to the AHB. This simplifies the design of the reader/card RAM. However, because of the requirement of a dual AHB interface, additional considerations have to be taken. The requirement is fulfilled using an AHB interface multiplexer. The purpose of this module is to allow the usage of one AHB slave interface on two distinct AHB. The general principle of the approach is shown in Figure 4.9. The multiplexer resides between an AHB slave and two distinct AHB. The *control* signal defines which bus has access to the slave. In order to prevent a violation of the bus protocol an additional mechanism has to be provided. Whenever a bus has access to the slave the other bus interface is wired to an unit providing a *dummy response*. This *dummy response* ensures that any master accesses results in a response regardless if it is a response from the slave or from the *slave dummy*. Figure 4.10 shows the architecture of the multiplexer. The *control* signal is provided by the Debug Interface. Whenever the processors are in debug mode (the processor is halted) the RAM is connected to the debug interface of the multiplexer. This allows the initialization of the RAM module. When the processors are not in debug mode the RAM is connected to the system bus of reader/card providing access to the reader/card software application.

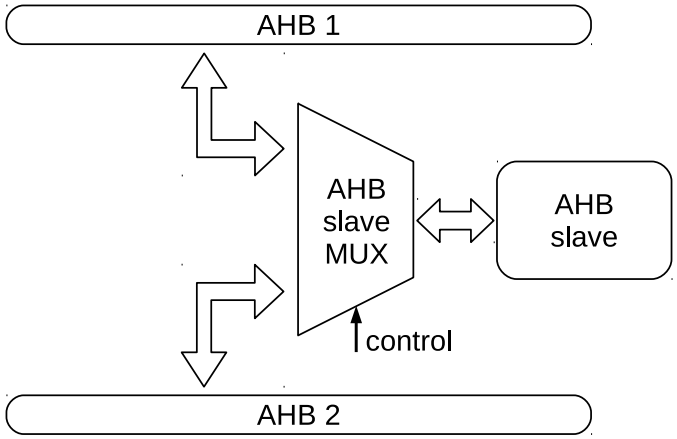


Figure 4.9: Principle architecture of the AHB Multiplexer. The module allows to access an AHB slave from two different AHB. The *control* signal is used to select the AHB interface to use.

Another possibility would have been to use the dual-port capabilities of the block RAM and the DDR3 RAM. However, in order to reduce dependencies from the used developing platform the multiplexer approach was taken. Another reason for the multiplexer approach was that the AHB wrapper for the DDR3 RAM (Section 4.4.6) does not support

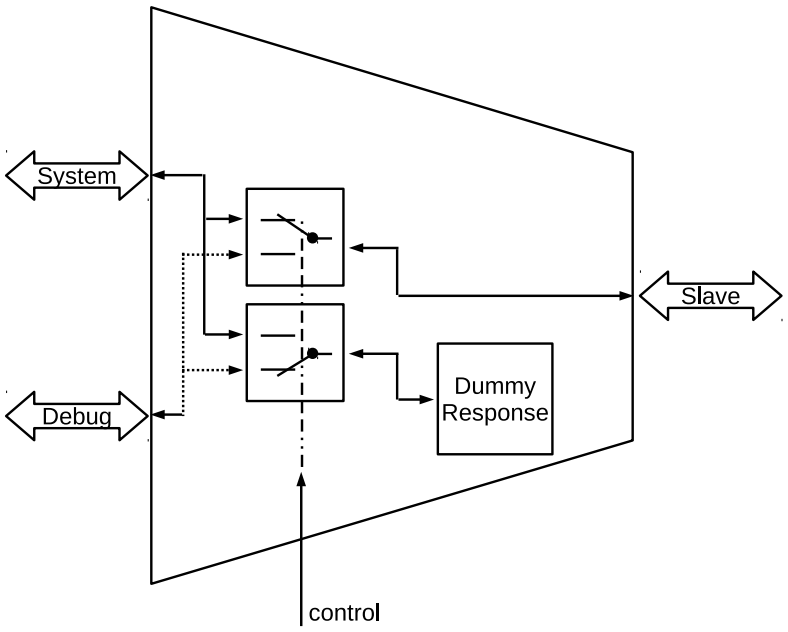


Figure 4.10: Internal structure of the AHB Multiplexer. The multiplexer allows slave access either from the system bus or the debug bus. Whenever a bus has access to the slave, the other bus interface is wired to an unit providing a *dummy response*.

a dual AHB interface.

4.5.4 Channel Model

The Channel Model is divided into two modules. One implementing the data transfer between reader and card. The other, modeling the power transfer from reader to the passively powered card.

Data Transfer

The data transfer between reader and card is implemented as bidirectional FIFO and serves as a bridge between the system bus of the reader and the system bus of the card. The connection to the system bus is implemented via an APB interface. The module allows the exchange of information between the reader and the card model. The FIFO length can be defined via a VHDL generic. The FIFO width is fixed to 32 bit. Additionally, a control register is provided to read the current state of the data transfer (e.g., are packets available, is FIFO full). Table 4.2 shows the structure of the APB

control register and its purpose.
The FIFOs are also equipped with saboteurs to enable the injection of faults into the data transfer. The interface of the module provides also debug signals such *read port*1 and *read port*2 which can be used to trigger on these events for effective fault injection.

Address Offset	Bit	Name	Length (bit)	Description
0x00	0 to 31	TX/RX Register	32	A write transfer to this register places the data into the TX FIFO. A read from this register returns the next data word from the RX FIFO.
0x04	0 to 7	RX free slots	8	Number of free slots in RX FIFO.
0x04	8 to 15	TX free slots	8	Number of free slots in TX FIFO.
0x04	16	RXF	1	RX FIFO full.
0x04	17	RXE	1	RX FIFO empty.
0x04	18	TXF	1	TX FIFO full.
0x04	19	TXE	1	TX FIFO empty.
0x04	29	READY	1	Can be set by the communication partner via SR bit. The purpose of this flag is to provide a synchronization mechanism between reader and card.
0x08	0	RESET	1	Resets the FIFO.
0x08	1	SR	1	Set ready flag for communication partner.

Table 4.2: Channel Model - FIFO control register.

Power Transfer

The power transfer is modeled using the SVE unit proposed by [12]. The SVE unit provides an APB interface to define the voltage provided by the field of the reader. Additionally, the module makes use of the results provided by the power estimation unit of the smartcard model. The result of the module represents the available supply voltage of the card.

The voltage provided from the reader can be configured by software through the APB interface. The resulting supply voltage information can be traced using the PPDU or CADU.

4.5.5 Fault Injection Unit

The Fault Injection Unit is adapted from [18]. The controller provides an APB interface for configuration by software which can be connected either to the system bus of the reader, the system bus of the card or to the debug bus. The actual placement of the module depends on the use case. Additionally, saboteur and trigger units have to be integrated into the system depending on the fault scenario.

4.5.6 Power Estimation Unit

The power estimation unit is adapted from [17] and is used to estimate the power consumption of the reader and card processor (LEON3). The module is continuously working and performs the computations on the linear regression model depending on the states of the processor components (e.g., pipeline, divider, multiplier). The results are provided to the PPDU of the Debug Interface.
In order to suppress the output during the halt of the processor the result of the estimation unit is gated providing a zero vector to the PPDU if the processor is inactive.

4.5.7 Debug Interface

As mentioned in Section 3.5.1 the Debug Interface consists of the debug bus, a AHB-UART bridge, the PPDU adapted from [26] and the CADU.
The debug bus is implemented as AHB using the AMBA AHB module from Gaisler (see Section 4.4.4). Together with the AHB-UART bridge, communication between the host system and the debug bus is possible. The AHB-UART bridge is implemented using the AHBUART module from Gaisler (see Section 4.4.3) and serves as the bus master. The PPDU is adapted from [26] with its Ethernet interface and does not rely on the remainder of the Debug Interface.
The CADU consists of a control unit to start/stop the tracing and the actual memory for the gathered information. Following, a detailed view on the implementation of the CADU is provided.

CADU - Control

The CADU is controlled via software through an APB interface. Table 4.3 shows the structure of the APB control register and its purpose. The control signals are handed over to the CADU memory. Additionally, a 16 bit interface is provided for the information to be logged.

Address Offset	Bit	Name	Length (bit)	Description
0x00	0	RESET	1	Resets the CADU. Previous stored data is lost.
0x00	1	ENABLE	1	Enable CADU. Every clock cycle a 16 bit value is stored in the memory until the CADU is stopped (ENABLE = '0') or the memory runs out of free space.
0x00	2	FULL	1	Indicates if CADU memory is full (FULL = '1'). No information is stored until a reset.

Table 4.3: CADU Control Register.

CADU - Memory

The CADU memory is implemented using the external 512MB DDR3 RAM provided by the developing board. Because of the modular design of the emulation framework (the DDR3 RAM should also be usable as RAM for reader/card) the *AHB to DDR3* wrapper from Gaisler (see Section 4.4.6) was instrumented to provide both functionalities. With this approach the external memory can be configured by software, either to function as ordinary RAM or as memory for the debug information.
The state machine of the *AHB to DDR3* wrapper was extended by two additional states *CADU* and *finish CADU* (see Figure 4.11). During operation as RAM the states are handled as normal providing read and write access from the AHB interface. If the CADU operation is activated, the state machine changes to state *CADU* as soon as all pending operations are finished. During the *CADU* state the information provided by the CADU controller is stored. The *CADU* state is hold until the CADU is stopped or the memory runs out of free space. During the operation as CADU memory, no write or read access through AHB is possible. After the CADU has stopped, the AHB interface is again unlocked and the stored data can be downloaded through the AHB interface.
Additional to the state machine changes, the interface of the *AHB to DDR3* wrapper

was extended. This interface provides access from the CADU control unit.
Figure 4.12 shows the architecture of the module. The DDR3 RAM can be used as ordinary AHB memory as long as the CADU is inactive. The CADU operation can be started via software through the APB interface of the CADU Controller. During this operation the debug signals (*CADU data*) are forwarded to the memory and stored. The original AHB interface of the memory is used to download the gathered information.
Using the external 512MB DDR3 RAM, the CADU is able to trace 8.95 seconds of debug information at system frequency of 30 MHz (16 bit/cycle). Figure 4.13 shows the memory organization of the logged data.

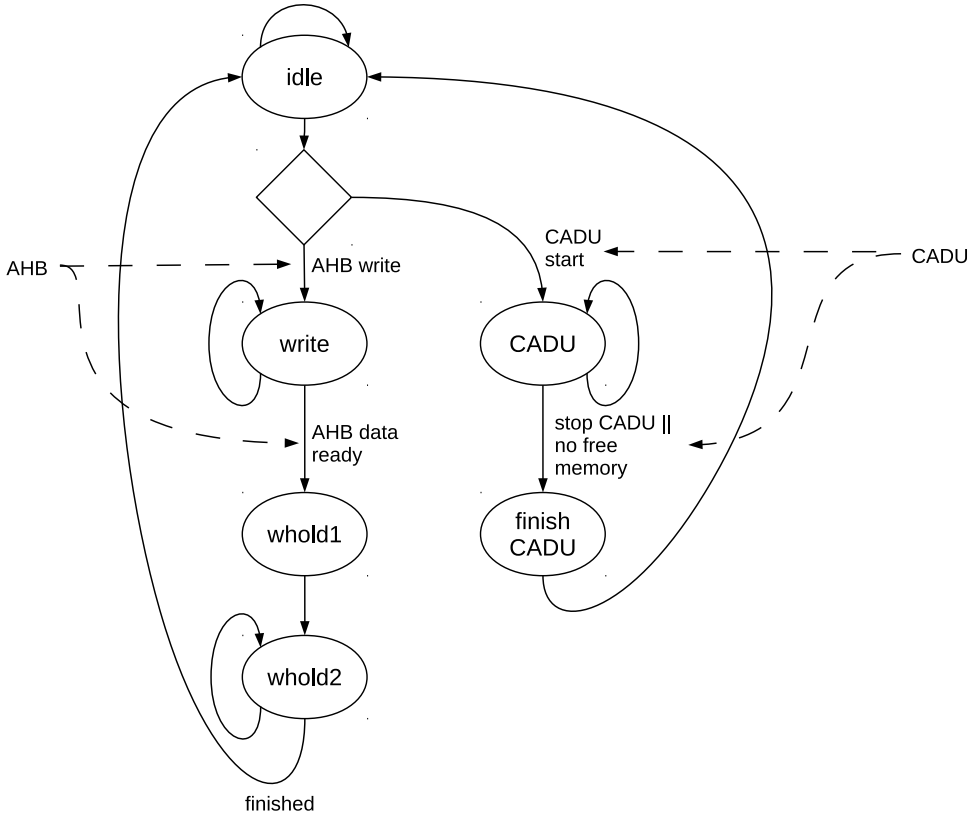


Figure 4.11: Instrumented AHB to DDR3 Wrapper - Write States (compare to Figure 4.8). The DDR3 RAM can be used as ordinary AHB memory as long as the CADU is inactive. During the operation as CADU memory, no write or read access through AHB is possible. After the CADU has stopped, the AHB interface is again unlocked and the stored data can be downloaded.

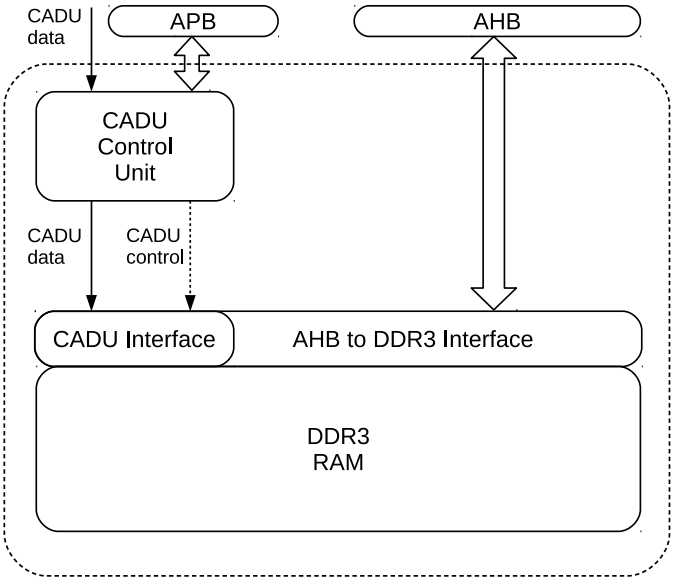


Figure 4.12: CADU architecture. The DDR3 RAM can be used as ordinary AHB memory or as memory for the CADU.

4.6 Host System

This sections refines the considerations from Section 3.5.2 and gives a detailed view on the implementation of the host system.
The emulation is controlled by the host system via the Fault Injection Interface JAVA application. The connection between the developing board and the PC is implemented via UART. The main purpose of the application is to control the processors including initialization of the registers. The second task of the application is to upload the reader/card application. These requirements are fulfilled by providing access to the Debug Interface (see Section 3.6). Using the AHBUART module provided by Gaisler, the host system can communicate with the nodes on the AHB of the Debug Interface including the RAM of reader and card, and the DSU3.
The processors are controlled by writing to the corresponding registers of the DSU3. Using this mechanism the registers of each processor in the system can be accessed and initialized. The DSU3 also provides a register to start/stop the execution on each processor. If the processors are stopped the application has access to the debug interface of the RAM multiplexer (Section 4.5.3). This allows the host system to upload the reader/card application. Figure 4.14 shows the relation between the application on the host system and the hardware modules on the emulator. Using the plug&play information provided by the Gaisler implementation of the AMBA AHB/APB (see Section 4.4.4) the application has a detailed knowledge about the hardware on the debug bus. After the application is connected, the plug&play information is read from the bus in-

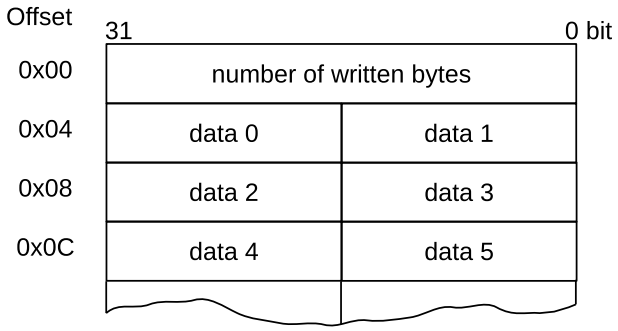


Figure 4.13: CADU memory organization.

cluding the address space of each module. This information allows the application to initialize its internal representation of the hardware (e.g., where are the DSU3 registers, how much processors are implemented on the platform). When all information is loaded the application stops all processors on the emulator and waits for user interaction.

User Interaction

After the application is connected to the emulator the following operations are provided to the user.

- Read/Write Debug Information
  The user has read/write access to the whole address space of the debug bus. This function allows to set the hardware registers manually for debug purpose.

- Program
  The user can initialize the RAM modules by defining the start address of the RAM and a binary file.

- Initialize Processor
  The user can initialize the registers of the processor by defining the program counter and the stack address.

- Start/Stop Processor

- Export Binary Dump
  The user can export RAM content by defining the start address and the length in byte. The data is loaded from the emulator and written to a file.

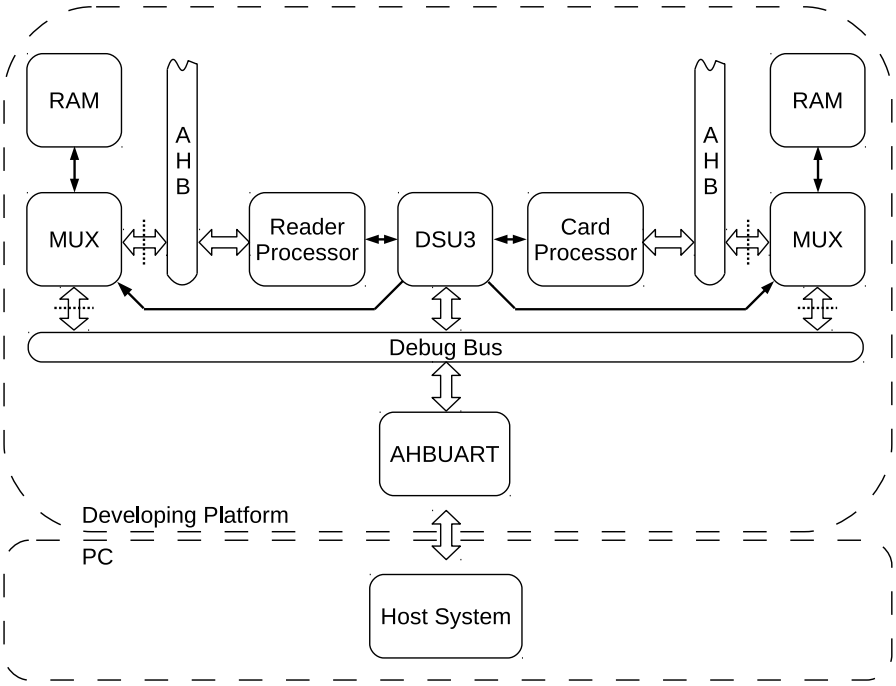


Figure 4.14: Relation between the Fault Injection Interface application and emulator. The host system has read/write access to the whole address space of the debug bus.

- Read CADU Data
  Similar to the export of the RAM content, the application supports the automatic export of CADU data. The application reads the number of collected bytes (see Memory Organization Figure 4.13) and exports the gathered information to a file for further computations.

Software Architecture

As mentioned in Section 3.5.2 the application is implemented with four classes.

- $FaultInjectorIf$
  This class provides the graphical user interface and the event loop. All user interaction is forwarded to the *Control* class. The $FaultInjectorIf$ verifies all user input and gives response if the provided input values (e.g., addresses, data length) are out of bounds.
  Additionally, the $FaultInjectorIf$ class implements a monitor thread, which reads continuously the state of the processors (e.g., running, stopped, error).

- *Control*
  The *Control* class receives the verified user interaction from the $FaultInjectorIf$ class and abstracts the underlying functionality. Depending on the command, the request is either handed to the *DebugSupportUnit* or directly provided to the $SerialInterface$ class. If interaction with the DSU3 of the emulator is necessary (e.g., start/stop CPU), the request is handed to the *DebugSupportUnit*. If the command involves only read/write commands (e.g., program RAM, read/write arbitrary data) the request is forwarded to the $SerialInterface$.

- $SerialInterface$
  The $SerialInterface$ class abstracts the underlying PHY and provides commands such as write, read, open, close on the physical interface. The class uses the RXTX [36] open source library for serial ports in JAVA.

- *DebugSupportUnit*
  This class receives DSU3 related commands from the *Control* class.
  The *DebugSupportUnit* holds an internal structure of the DSU3 registers and translates the commands into read/write requests which are forwarded to the $SerialInterface$ class.
  The *DebugSupportUnit* performs all necessary task, hiding complexity from the upper classes (e.g., performing cache flushes, correct initialization of the CPU registers).

Additional to the Fault Injection Interface, the Ethernet Dump application from [26] (see Section 3.5.2) is used. The Ethernet Dump application uses the Ethernet interface and has a direct connection to the PPDU.

5 Results

5.1 Emulator - Characteristics and Performance

The system clock of the emulator and the modeled systems (reader/card) is 30 MHz. The clock of the external DDR3 RAM is 400 MHz. Table 5.1 shows the address space of the emulator and Table 5.2 provides an overview on the utilization of the VIRTEX-6 FPGA.
One should note that the architecture depends on the use case of the emulation. Therefore, address space and utilization vary depending on the actual configuration. The provided data is captured form the default configuration also used for the experiments in Section 5.3.

Component	System address	Debug bus
Reader RAM	0x40000000 - 0x40080000	0x40000000 - 0x40080000
Reader APB	0x80000000	-
Card RAM	0x40000000 - 0x40080000	0x60000000 - 0x60080000
Card APB	0x80000000	-
DDR3 RAM	-	0xA0000000 - 0xC0000000

Table 5.1: Address space - Emulation Framework.

Component	Slices	LUTs
Reader model	3829	7574
Card model	3834	7576
Channel model	699	1068
Control (DSU3, Debug bus, etc.)	1110	1979
(Control with DDR3 RAM)	4748	7927
Fault controller	526	839
Power estimation	255	515

Table 5.2: Device utilization - Virtex-6. Trigger and saboteurs are omitted.

5.2 Software Interface of the Emulation Framework

This section shows the results of the software interface provided to control and configure the emulation setup. Figures 5.1 to 5.4 show the graphical user interface of the application.

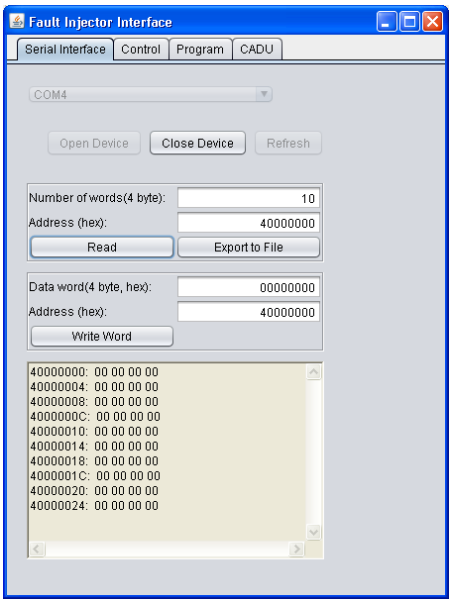


Figure 5.1: Serial Interface. Allows the user to choose a serial port to connect to the platform. The interface also provides a direct read/write access to the debug bus of the emulator. Additionally, a function is provided to export the content of the RAM to a file.

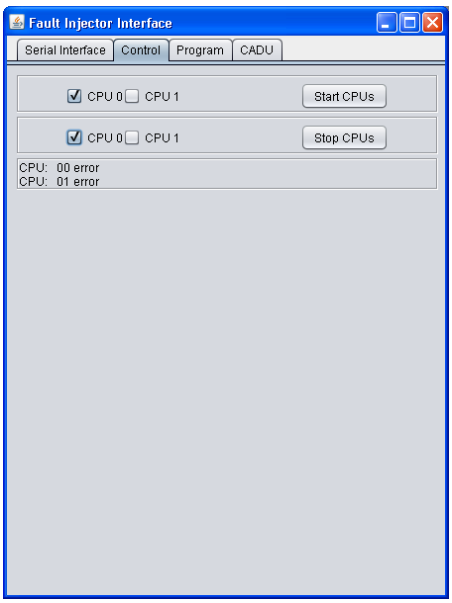


Figure 5.2: Control. Allows the user to start/stop the processors. The application automatically detects all processors on the emulator and provides a control interface for each. Additionally, the state of each processor is shown.

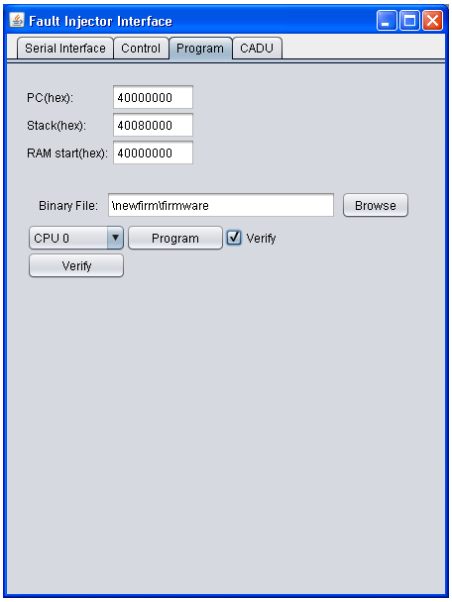


Figure 5.3: Program. Allows the user to define the program to download. The user has to define the program counter and the stack address to use in the system. Additionally, the user has to define the address on the debug bus where the RAM is mapped to (RAM start).

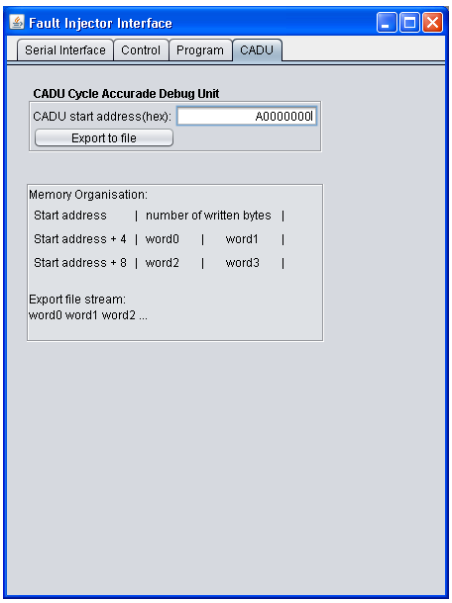


Figure 5.4: CADU. Allows the user to export the content of the CADU to a file.

5.3 Experiments

This section demonstrates some use cases showing the capabilities of the emulation framework.

5.3.1 Power Supply - Smartcard

To demonstrate the capability of emulating a full RFID reader-card system, the following use case is considered.
A Diffie-Hellman key exchange protocol should be analyzed regarding its power consumption. Figure 5.5 shows the principle of the Diffie-Hellman protocol. The reader chooses suitable values for p (prime number) and g (primitive root mod p) and computes A (a is a secret kept by the reader). These values are transmitted to the card. The card computes B with its own secret b and the shared secret K. The card responses with B. Using B, the reader can compute the shared secret K. For suitable values p and g, the reader and the card compute the same shared secret K which can be used as key for a symmetric cypher. Eavesdropping the channel does not provide enough information to reconstruct the shared secret.

Figure 5.6 shows the result of the emulation performing the computations necessary for the protocol. All computations are done in software. $\hat{P}_R(t)$ represents the estimated

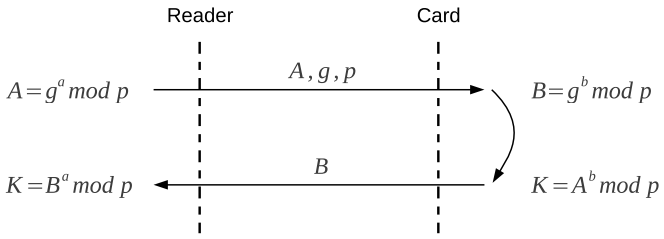


Figure 5.5: Diffie-Hellman Key Exchange - Protocol.

power consumption of the reader, $\hat{P}_{SC}(t)$ represents the estimated power consumption of the smartcard. Additionally, the supply voltage $\hat{v}(t)$ of the card is emulated depending on $v_i(t)$. $v_i(t)$ is the voltage provided by the reader through the contactless interface. V_Z denotes the Zener voltage and V_T the minimal supply voltage threshold. $\hat{P}_Z(t)$ is the estimated power consumed by the Zener diode (see Figure 3.6). The Zener Diode serves as a simple voltage regulator for the supply voltage of the smartcard. Therefore, the lower $\hat{P}_Z(t)$, the more efficient the reader/card system.
The results pictured in Figure 5.6 are gained with a constant voltage $v_i(t)$ provided by the field of the reader. This configuration leads to an inefficient use of the available energy. The surplus energy received by the card is transformed into heat at the Zener diode, acting as shunt to protect the card circuit.

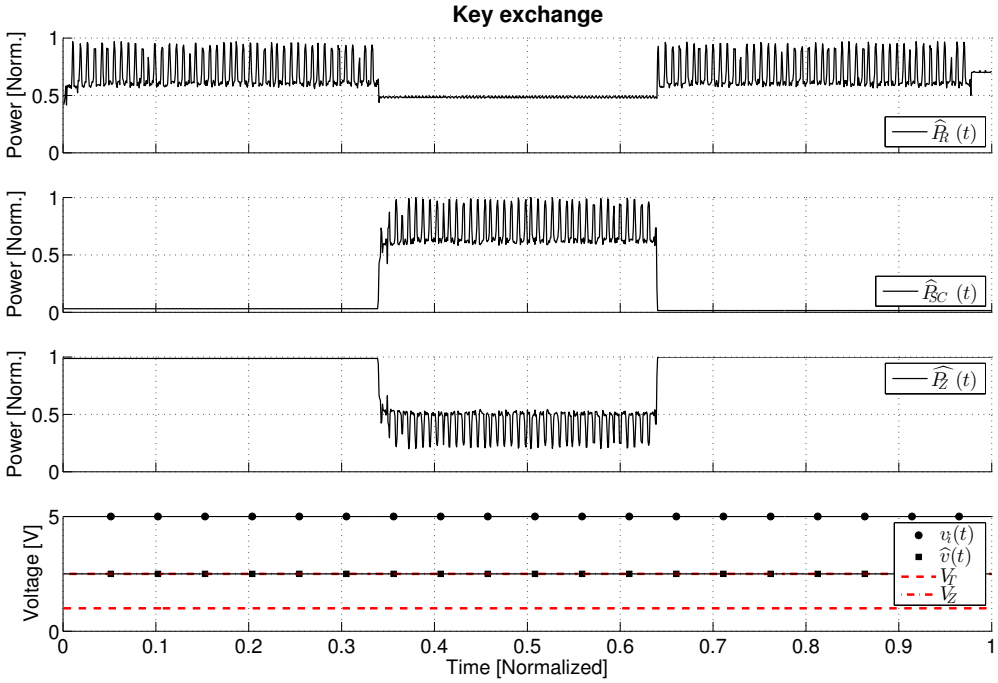


Figure 5.6: Diffie-Hellman Key Exchange.

In order to use the available energy more efficient, the reader can adapt the emitted field depending on the needs of the card. Knowing that the card has a low power consumption when it is idle, implies that the reader can reduce the field strength during this time. Figure 5.7 shows the results of the same use case, adapting the field strength according to the needs of the card. The reader supplies the card with less energy during idle time and increases the field if computational effort of the card is required. The results show that $\hat{P}_Z(t)$ can be reduced to zero, without endangering the integrity of the cards computations ($\hat{v}(t)$ always above V_T). One should note that this experiment shows the ideal case. In a real setup also the distance variations between the reader and card has to be considered. Therefore, $\hat{P}_Z(t)$ can only be minimized.

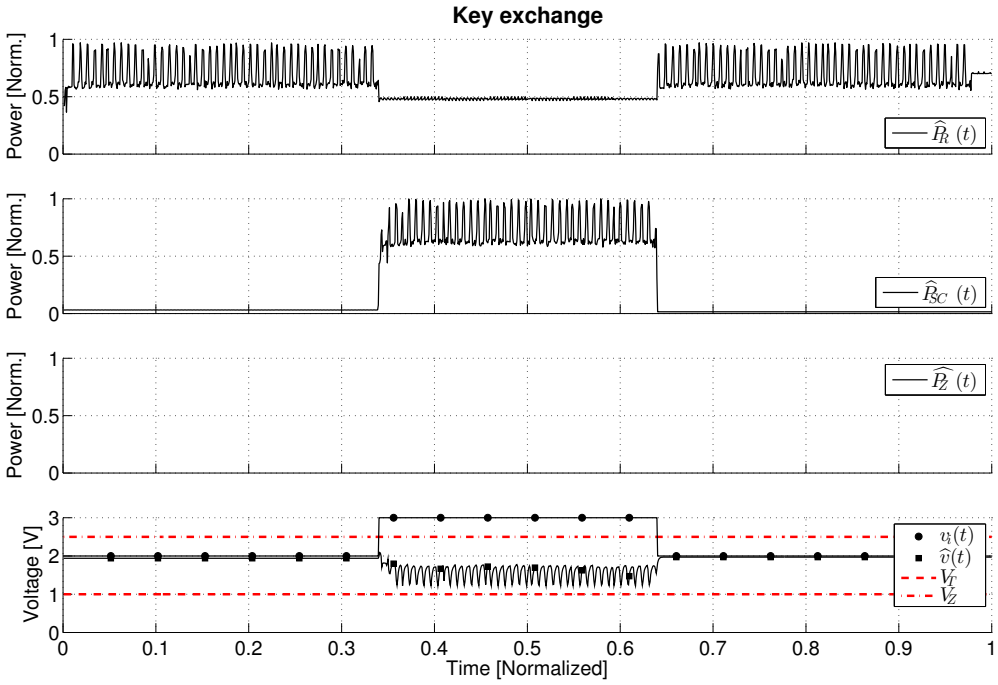


Figure 5.7: Optimized Diffie-Hellman Key Exchange.

5.3.2 Power Supply - Fault Injection

The experiment above shows that adapting the field strength according to the actual power consumption of the card can improve the efficiency of the whole system. This implies that the reader has a mechanism to control the emitted field strength.
The following experiment aims to emulate a scenario where this mechanism is compromised. Even due to an attacker manipulating the reader system or due to an internal hardware failure.
Figure 5.8 shows the simplified internal configuration of a supposed mechanism to regu-

late the field strength. The reader device includes a register which defines the strength of the emitted field together with the RX/TX buffer etc. The card harvests the energy provided by the reader in order to supply its computational circuits.
The goal of this experiment is to show the effects of a simple bit fault in the register defining the field strength.

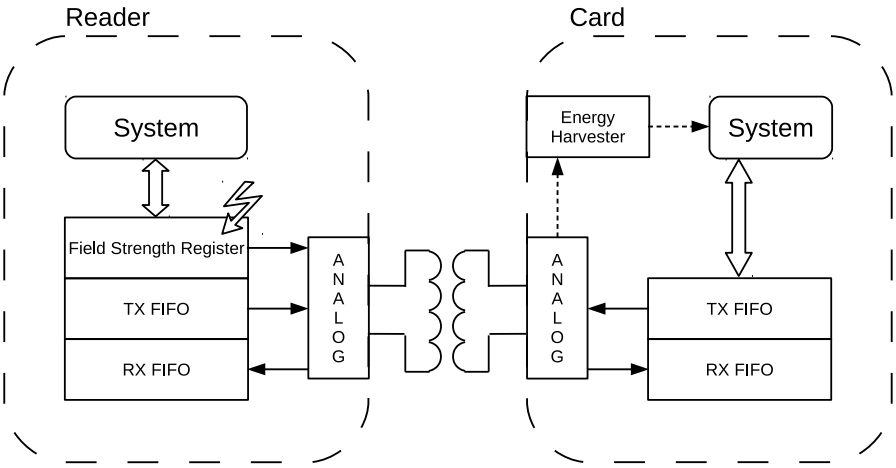


Figure 5.8: Field Strength Scaling - Architecture.

Figure 5.9 shows the result of the experiment. As in the experiment described above, reader and card perform a Diffie-Hellman protocol. The fault injection unit of the emulator is configured to inject a single bit fault into the field strength register of the reader. The effect of this single bit fault can be clearly identified in the resulting emulation output. Where the supply voltage $\hat{v}(t)$ was always above the threshold voltage V_T in the fault-free scenario, the single bit fault causes a significant drop of the supply voltage on the card. The supply voltage $\hat{v}(t)$ drops below the minimal required voltage V_T endangering the operational stability of the smartcard if no precaution is taken to prevent computations during undersupply. Even if the card provides a mechanism to protect the computational circuit during the undersupply (e.g., by scaling down the system frequency), the protocol is affected by the fault (e.g., due to small timing variations).

5.3.3 Multiplier - Fault Injection

Rahimi et al. [34] show that variations in the operating conditions such as temperature or supply voltage of an integrated system can lead to an erroneous behavior (e.g., due to variation of the critical path). The experiments were performed on a LEON3, examining

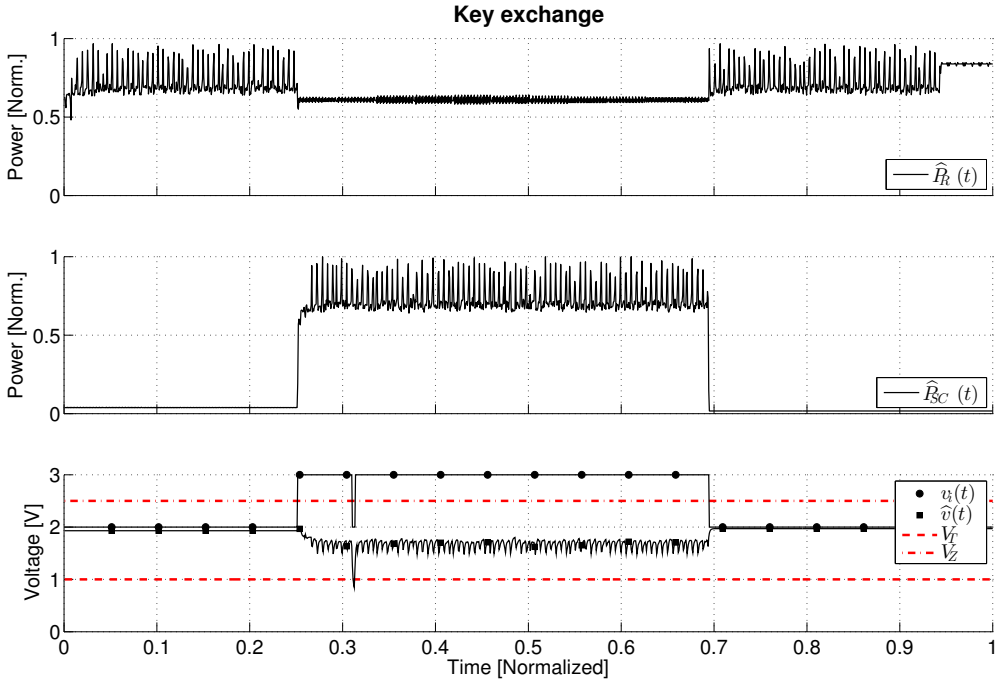


Figure 5.9: Diffie-Hellman Key Exchange. A single bit fault in the hardware of the reader causes a significant drop of the supply voltage on the card.

the integer unit of the processor and were aimed to identify vulnerabilities of the instruction set at specific corner cases (e.g., low/high temperature). For example, the multiply instruction fails with a probability of 4.2% for certain conditions[1] or even higher if the conditions are changed. This result makes clear that fault emulation and analysis are important factors during the design of a system. Especially if security relevant data is processed, erroneous behavior of one component can lead to vulnerabilities affecting the whole system. The following use case shows the capabilities of the emulation framework regarding fault analysis and is motivated by the work of [34].
The setup of the use case is pictured in Figure 5.10. A simple protocol between a smartcard and a reader device is examined. The protocol works as follows. The smartcard should transmit a secret to the reader device. This secret can either be a secret message read from the memory or a key for a symmetric cypher which has to be generated. This secret is then encrypted with an asymmetric cryptographic algorithm. In the test scenario a cypher based on the modular exponentiation algorithm is used which encrypts the secret with the public key of the reader. The encrypted secret is then transmitted together with the result of a cyclic redundancy check (CRC).
The reader decrypts the message using its secret key and performs a CRC on the decrypted data. If the result of the CRC equals the CRC received from the smartcard an

[1] The experiment was performed with a supply voltage of 1.1V at a temperature of 125°C and a cycle time of 0.82ns

error-free transfer can be assumed. If no error is detected, the message from the card can be processed or if the protocol is used for key-exchange, the following data transfer can be done using a symmetric cypher. In this use case all computations are done by software meaning that no additional hardware units are used (e.g., a cryptographic co-processor).
The goal of the experiment is to examine the behavior of the protocol under the presence of faults. As target for the faults, the multiplier unit of the LEON3 integer unit on the smartcard was chosen. Because of the fact that the exponentiation algorithm is the only component which uses the multiplier, the introduced faults affect only the result of the encryption. The number of the multiplications depend on the input data of the algorithm.

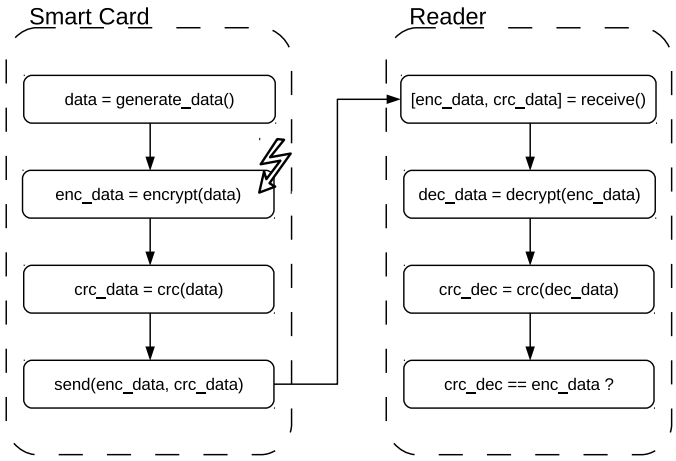


Figure 5.10: Asymmetric Security Protocol - Faults during multiplication lead to erroneous behavior.

In order to perform the fault injection process the integer unit of the LEON3 was extended. A simplified view on the architecture of the integer unit is given in Figure 5.11. The main components of the unit are the pipeline with its cache interface, a divider and the multiplier. The multiplier was extended with a trigger and a saboteur enabling the manipulation of the multiplication result. These units can be controlled and configured through the fault injection controller. The trigger unit observes the control signals of the multiplier. Whenever a multiplication is performed, the controller is informed. The fault injection controller then decides depending on its configuration if a fault should be introduced. If a fault should be introduced the saboteur is activated which alters the multiplication result. The experiment was performed twice with different configurations. One experiment consists of 100000 runs of the protocol described above using a secrete message of 128 byte. During one run, exactly one bit in a multiplication result was altered. The position of the faulty bit and the multiplication to alter were chosen randomly. This means that during one run, one randomly chosen multiplication is modified. First, the effect of a stuck-at-one fault was examined (during one multiplication, one bit

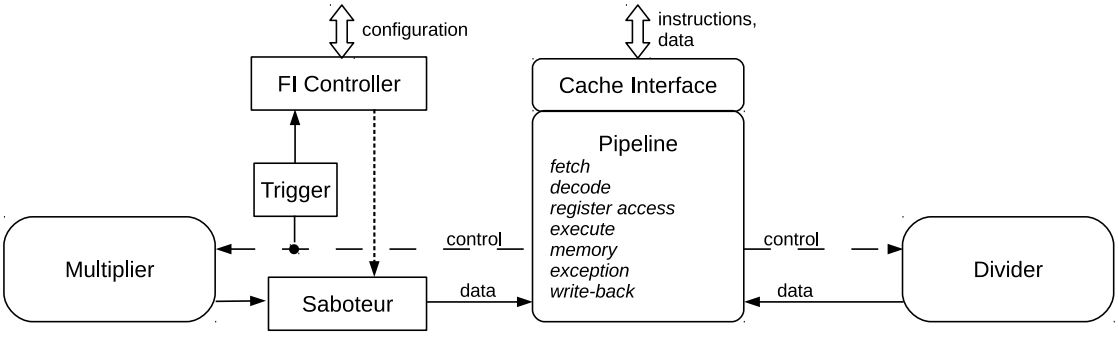


Figure 5.11: Simplified architecture of the LEON3 integer unit (with integrated trigger and saboteur).

of the output of the multiplier was set to '1'). Second, the effect of a stuck-at-zero fault was examined (during one multiplication, one bit of the output of the multiplier was set to '0'). The results of the experiment are summarized in Table 5.3.
The stuck-at-one faults resulted in 83535 erroneous computations. This means that this kind of fault had effect on 83.54% of the 100000 runs. The remaining runs were computed without error. This can be explained by the simple fact that an error is only introduced if the original bit was zero.
The whole setup was repeated with a stuck-at-zero configuration of the fault injection unit. Other than the stuck-at-one experiment, this setup resulted in only 15427 (15.43%) erroneous computations. The results strongly depend on the data being processed during the faulty computation.
The experiment makes clear that the used fault model has a significant influence on the results when a system is analyzed. Another result of the experiment is the performance of the CRC. The computations were verified using a 16 bit and a 32 bit CRC. The CRC32 had a detection rate of 100% were the CRC16 algorithm lead to 0.02% wrong results.
One should note that the implemented fault injection mechanism provides a high efficient solution for fault analysis. The trigger-saboteur approach enables the analysis of every injected fault. This is possible due to the fact that a fault is only injected if it has an impact on the system behavior. In contrast to traditional approaches where faults are injected randomly, every fault leads potentially to a deviation from normal behavior of the system. One should also note that once the multiplier is extended with trigger and saboteurs, arbitrary algorithm can be tested on their fault behavior without additional effort.

Fault	Errors	Errors (%)	CRC16	CRC32
stuck-at-one	83535	83.54	83517 (99.98%)	83535 (100%)
stuck-at-zero	15427	15.43	15427 (100%)	15427 (100%)

Table 5.3: Asymmetric Security Protocol - Results.

6 Conclusion

The goal of the thesis was to develop a RFID/NFC emulation framework for software verification. One important requirement was that the framework combines models for reader and card enabling the emulation of the whole system setup. Another requirement was that a model of the contactless channel should be included. The emulation of faulty hardware components as well as the estimation of the power consumption of the system should be possible.
The framework is designed in a high modular manner providing a concept for reader/card emulation rather than a fixed system solution. The framework is easy scalable and does not rely on a specific hardware/developing platform. Therefore, the framework can be easily adapted to fit the requirements of the preferred hardware solution. The framework allows functional analysis and verification, the emulation of the power consumption and the analysis of vulnerabilities due to hardware faults or attacks.

The whole project is implemented on one FPGA prototyping platform and is built upon the open source IP library from Gaisler. The framework combines a model for the reader device, a card model and a model of the contactless communication channel. Additionally, a mechanism is provided for efficient fault injection and the models are equipped with power estimation units. In order to control the emulator, a software interface is provided. This interface enables to start/stop the emulation, configuration of the models and a simple access to the produced emulation data.

The project makes the following contributions:

- The framework provides models for reader/card and the contactless communication channel.
- The setup is structured into a host system and the actual emulation framework implemented on a prototyping platform. The host system is used to configure and control the emulator. The framework itself is designed to work without host interaction during the emulation process.
- The emulator enables functional analysis and verification.
- The emulator supports modules for real time power emulation.

- The emulator includes a mechanism for efficient fault injection and fault analysis.

Future Work

The emulator can be easily extended exploiting the bus structure of the framework. The following extensions are conceivable:

- Adding support for data dependent power analysis.
  Currently the power estimation only considers control signals to model the power consumption. In order to provide a system feasible for side channel attacks such as DPA, also data dependent power information is necessary.
  This requirement can be fulfilled by extending the regression based power model with a data dependent term. Together with the CADU, cycle accurate power information can be gathered feasible for DPA.

- Integration of a real wireless interface. The substitution of the channel model with a real contactless interface is feasible in order to improve the realism of the emulation. However, this substitution would also lead to a system, which is less controllable (assuming the channel as black box). Additionally, repeatability of experiments is difficult due to changing conditions (e.g., environment).

- Integration of various cryptographic hardware.

7 Appendix

7.1 Manual

7.1.1 Software Interface

In order to use the *Fault Injector Interface* application install the RXTX library [36] and the driver for the USB-to-UART Bridge device of the ML605 prototyping platform (see [44])

Connect to prototyping platform
After the application has started the control window for the serial interface appears (Figure 7.1). Select the port to connect and press the *Open Device* button. Use the *Refresh* button to update the list of available devices.

Access to the Debug Interface
When the application is connected to the prototyping platform the user has access to the Debug Interface of the emulation framework (Figure 7.2). Define a start address, the amount of words (word = 4 bytes) and press the *Read* button to read data from the Debug Bus. Use the *Export to File* button to export the data into a file. Data can be written to the Debug Bus by defining a data word and the address (*Write Word* button). The application can be disconnected by using the *Close Device* button.

Download software and initialize CPUs
Switch to the *Program* window in order to download software (Figure 7.3). Define the start address (*PC*), stack address (*Stack*) and the binary file (*Binary File*) with the program. Additionally, the address of the RAM on the Debug Bus has to be set (*RAM start*) which defines the address mapping. This task has to be done for each processor in the system.

Example:

Setup: The system setup consist of a reader and a card model. The RAM of both models reside from address 0x40000000 to 0x40080000. The RAM of the reader model is mapped to the address space from 0x40000000 to 0x40080000 of the Debug Bus. The

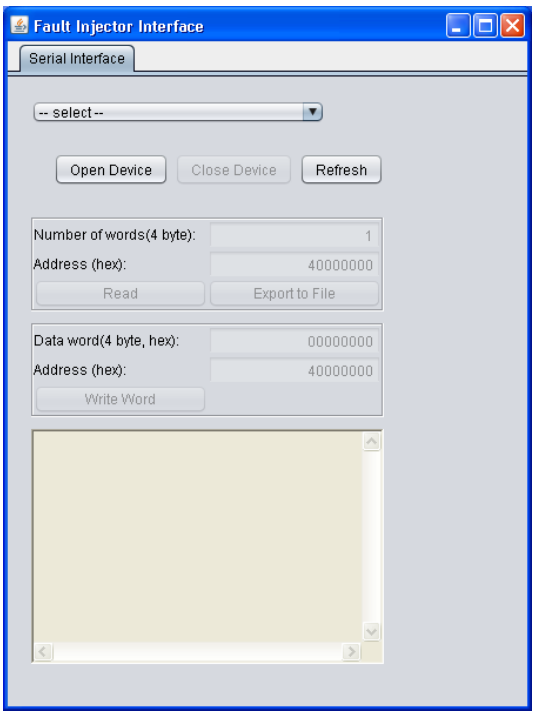


Figure 7.1: Connect to platform.

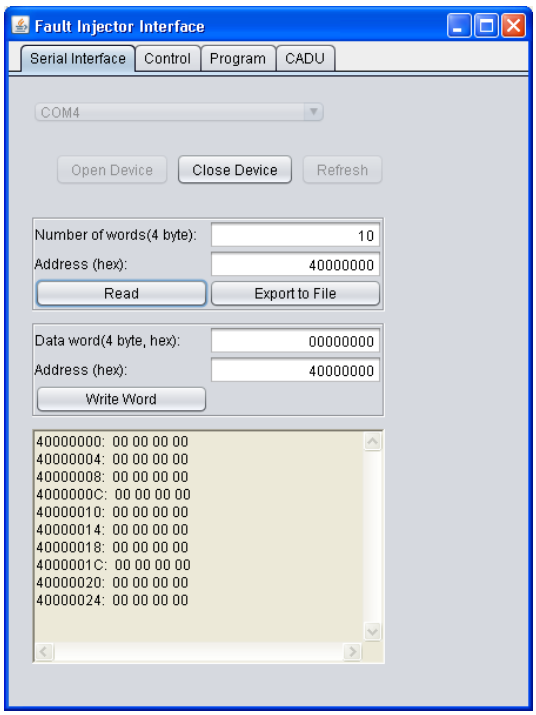


Figure 7.2: Access Debug Interface.

RAM of the card model is mapped to the address space from 0x60000000 to 0x60080000 of the Debug Bus.

Initialization: According to the setup, the system is initialized as pictured in Table 7.1.

Control CPUs
Switch to the *Control* window in order to start/stop the CPUs on the platform (Figure 7.4). The CPUs can be controlled independent from each other. Additionally, the status of each CPU is shown.

CADU Interface
The information collected by the CADU can be downloaded using the *CADU* control window (Figure 7.5). The user has to define the start address of the CADU memory. The information can be exported to a file using the *Export to file* button.

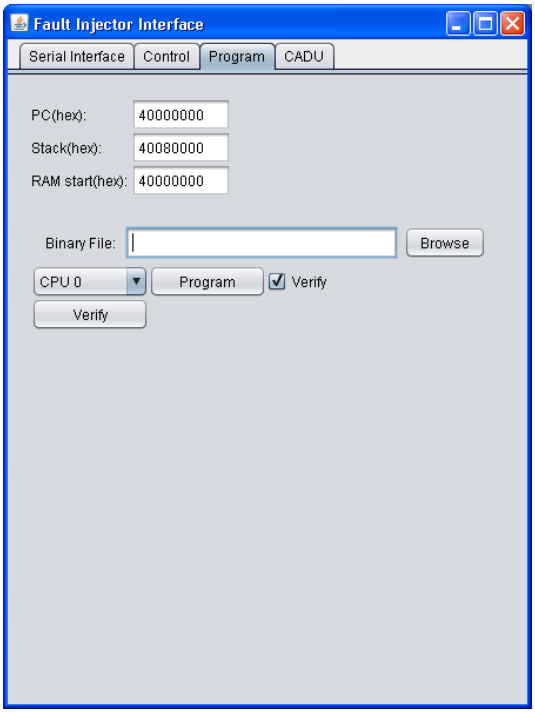


Figure 7.3: Initialize CPUs.

	Reader	Card
PC	0x40000000	0x40000000
Stack	0x40080000	0x40080000
RAM start	0x40000000	0x60000000

Table 7.1: Example configuration.

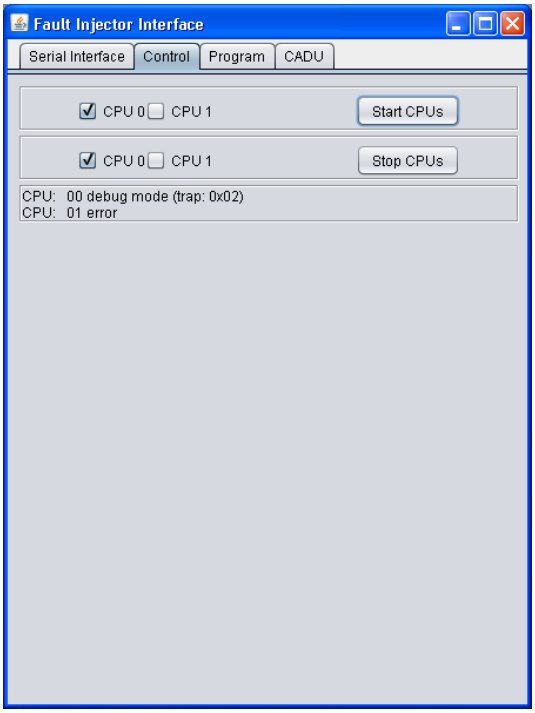


Figure 7.4: Control CPUs.

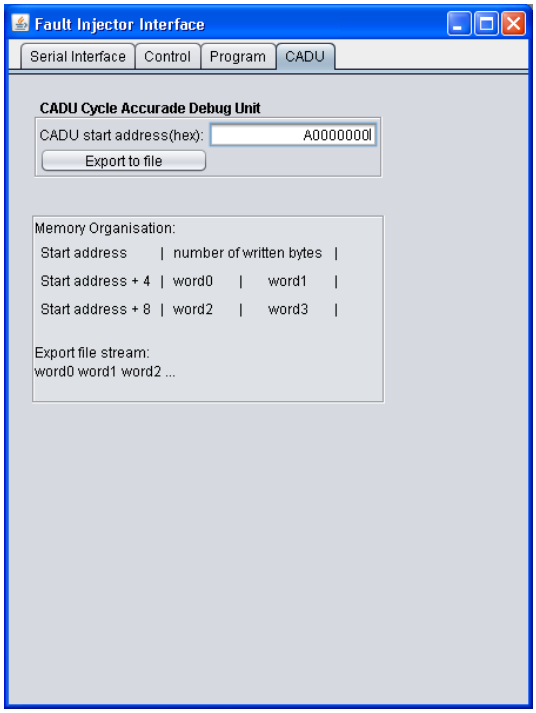


Figure 7.5: CADU Interface.

7.1.2 Emulation Framework

The emulation framework is directly integrated into the open IP library from Aeroflex Gaisler [14]. Figure 7.6 shows the hierarchy of the framework.

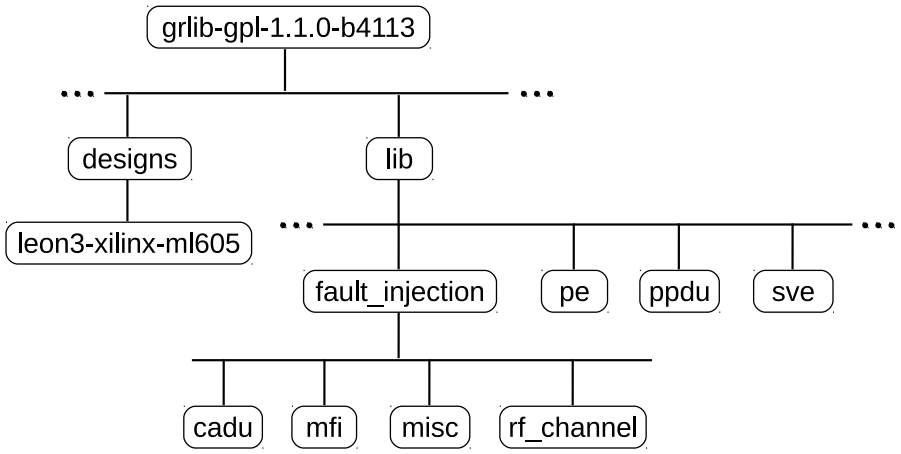


Figure 7.6: Hierarchy of the framework.

Hierarchy:

- **leon3-xilinx-ml605**
  Includes the top module of the VHDL project (*leon3mp.vhd*).

- **cadu**
  Contains the control unit for the Cycle Accurate Debug Unit (CADU).

- **mfi**
  Contains all modules of the Modular Fault Injector (MFI) such as fault controller, trigger and saboteur units.

- **misc**
  Contains the AHB slave multiplexer module. This module can be used to connect an AHB slave to two different AHB.

- **rf_channel**
  Contains all modules necessary to enable the data exchange between a reader model and a card model.

- **pe**
  Contains all modules of the Power Estimation Unit (PE).

- **ppdu**
  Contains the Power Performance Debug Unit (PPDU) project.

- **sve**
  Contains all modules for the Supply Voltage Estimation Unit (SVE).

The whole framework is build upon the AMBA AHB/APB implementation from Aeroflex Gaisler. Therefore, all provided modules implement an AHB or APB interface. The following example shows the instantiation of an AHB RAM module.

```
ahbram0 : ahbram
       generic map (hindex => 3,
                    haddr  => CFG_AHBRADDR,
                    tech   => CFG_MEMTECH,
                    kbytes => CFG_AHBRSZ)
       port map    (rstn,
                    clkm,
                    ahbsi,
                    ahbso(3));
```

ahbsi and *ahbso* are vectors of signals defined by the Aeroflex Gaisler IP library. These signals are connected to all other modules on the bus and the AHB arbiter. The *hindex* parameter defines the index of the module on the bus.
The emulation framework defines three buses. One for the reader model, one for the card model and a third one for the debug interface. The corresponding signal pairs are *ahbsi_1b*/*ahbso_1b*, *ahbsi_2b*/*ahbso_2b* and *ahbsi*/*ahbso*. Using these signals, new modules can be easily added to the reader model, the card model or to the debug bus.

Bibliography

[1] Dakshi Agrawal, Bruce Archambeault, Josyula Rao, and Pankaj Rohatgi. The EM Side - Channel(s). *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 29–45, 2003.

[2] David De Andres, Juan Carlos Ruiz, Daniel Gil, and Pedro Gil. Fades: A fault emulation tool for fast dependability assessment. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 221–228, Dec. 2006.

[3] L. Antoni, R. Leveugle, and B. Feher. Using run-time reconfiguration for fault injection applications. *Instrumentation and Measurement, IEEE Transactions on*, 52(5):1468–1473, Oct. 2003.

[4] ARM Limited. *AMBA Specification v2.0*, 1999.

[5] 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, Feb. 2006.

[6] J.C. Baraza, J. Gracia, D. Gil, and P.J. 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, Nov. - Dec. 2005.

[7] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Assembly-level pre-injection analysis for improving fault injection efficiency. In *Proceedings of the 5th European conference on Dependable Computing*, EDCC'05, pages 246–262, Berlin, Heidelberg, 2005. Springer-Verlag.

[8] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305– 316, Sept. 2005.

[9] 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, June 1998.

[10] J. Coburn, S. Ravi, and A. Raghunathan. Hardware accelerated power estimation. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 528 – 529 Vol. 1, Mar. 2005.

[11] J.-M. Daveau, A. Blampey, G. Gasiot, J. Bulone, and P. Roche. An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip. In *Reliability Physics Symposium, 2009 IEEE International*, pages 212–220, 2009.

[12] N. Druml, C. Steger, R. Weiss, A. Genser, and J. Haid. Estimation based power and supply voltage management for future rf-powered multi-core smart cards. In *Design, Automation Test in Europe Conference Exhibition, 2012*, pages 358 –363, Mar. 2012.

[13] Aeroflex Gaisler. BCC - Bare-C Cross-Compiler User's Manual. Version 1.0.41, July 2012.

[14] Aeroflex Gaisler. Grlib ip core user's manual. Version 1.1.0 - B4113, January 2012.

[15] Aeroflex Gaisler. Grlib ip library user's manual. Version 1.1.0 - B4113, January 2012.

[16] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results Cryptographic Hardware and Embedded Systems - CHES 2001. volume 2162 of *Lecture Notes in Computer Science*, chapter 21, pages 251–261. Springer Berlin / Heidelberg, Berlin, Heidelberg, Sept. 2001.

[17] A. Genser, C. Bachmann, J. Haid, C. Steger, and R. Weiss. An emulation-based real-time power profiling unit for embedded software. In *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS '09. International Symposium on*, pages 67–73, July 2009.

[18] 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, Sept. 2011.

[19] Brad L. Hutchings, Michael J. Wirthlin, and I Overview. Implementation approaches for reconfigurable logic applications. In *In International Workshop on Field-Programmable Logic and Applications*, pages 419–428. Springer, 1995.

[20] Michael Hutter, Stefan Mangard, and Martin Feldhofer. Power and em attacks on passive 13.56 MHz RFID devices. In *Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, CHES '07, pages 320–333, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] Marc Joye, Pascal Paillier, and Berry Schoenmakers. On second-order differential power analysis. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop*, pages 293–308. Springer, 2005.

[22] Timo Kasper, David Oswald, and Christof Paar. Transactions on computational science X. chapter A versatile framework for implementation attacks on cryptographic RFIDs and embedded devices, pages 100–130. Springer-Verlag, Berlin, Heidelberg, 2010.

[23] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, Apr. 2011.

[24] Paul 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.

[25] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *CRYPTO' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397, Berlin, Heidelberg, Dec. 1999. Springer Berlin Heidelberg.

[26] Michael Lackner. Design and implementation of a multi-core power and performance emulation platform. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2010.

[27] Timothy C. May and Murray H. Woods. A new physical mechanism for soft errors in dynamic memories. In *Reliability Physics Symposium, 1978. 16th Annual*, pages 33–40, Apr. 1978.

[28] Thomas S. Messerges. Using second-order power analysis to attack dpa resistant software. In *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '00, pages 238–251, London, UK, UK, 2000. Springer-Verlag.

[29] S. Moore, R. Anderson, P. Cunningham, R. Mullins, and G. Taylor. Improving smart card security using self-timed circuits. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 211 – 218, Apr. 2002.

[30] National Institute of Standards and Technology (NIST). *FIPS Publication 197: Advanced Encryption Standard (AES)*, Nov. 2001.

[31] NFC Forum. *Essentials for Successful NFC Mobile Ecosystems*, Oct. 2008.

[32] E. Normand and T.J. Baker. Altitude and latitude variations in avionics SEU and atmospheric neutron flux. *Nuclear Science, IEEE Transactions on*, 40(6):1484–1490, Dec. 1993.

[33] E. Prouff, M. Rivain, and R. Bevan. Statistical analysis of second order differential power analysis. *Computers, IEEE Transactions on*, 58(6):799–811, June 2009.

[34] A. Rahimi, L. Benini, and R.K. Gupta. Analysis of instruction-level vulnerability to dynamic voltage and temperature variations. In *Design, Automation Test in Europe Conference Exhibition, 2012*, pages 1102–1105, 2012.

[35] S. Rammohan, V. Sundaresan, and R. Vemuri. Reduced complementary dynamic and differential logic: A CMOS Logic Style for DPA-Resistant Secure IC Design. In *VLSI Design, 2008. VLSID 2008. 21st International Conference on*, pages 699–705, Jan. 2008.

[36] RXTX. http://rxtx.qbang.org. Visited Apr. 2013.

[37] 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, June 1997.

[38] SILICON LABS. *Single-Chip USB to UART Bridge CP2103*, 2010. (1.0).

[39] Vijay Sundaresan, Srividhya Rammohan, and Ranga Vemuri. Power invariant secure IC design methodology using reduced complementary dynamic and differential logic. In *Very Large Scale Integration, 2007. VLSI - SoC 2007. IFIP International Conference on*, pages 1–6, Oct. 2007.

[40] M.G. Valderas, M.P. Garcia, R.F. Cardenal, C. Lopez Ongil, and L. Entrena. Advanced simulation and emulation techniques for fault injection. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pages 3339–3344, June 2007.

[41] I. Verbauwhede, K. Tiri, D. Hwang, and P. Schaumonr. Circuits and design techniques for secure ics resistant to side-channel attacks. In *Integrated Circuit Design and Technology, 2006. ICICDT '06. 2006 IEEE International Conference on*, pages 1–4, 2006.

[42] M. Wendt, M. Grumer, C. Steger, R. Weiss, U. Neffe, and A. Muehlberger. System level power profile analysis and optimization for smart cards and mobile devices. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1884–1888, New York, NY, USA, 2008. ACM.

[43] XILINX. *Memory Interface Solutions User Guide UG086*, Sept. 2010. (v3.6).

[44] XILINX. *ML605 Hardware User Guide UG534*, Oct. 2012. (v1.8).

[45] XILINX. *Virtex-6 Family Overview DS150*, Jan. 2012. (v2.4).