Michael Zelle, BSc

# Design and Implementation of a Hardware Supported Memory Protection for the Java Card Firewall

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

**Graz University of Technology**

Supervisor

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

Institute for Technical Informatics

Graz, April 2015

# 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 indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

..... .....
date (signature)

# Kurzfassung

Aktuelle Überprüfungen verschiedener Java Card Applets haben ergeben, dass es keinen ausreichenden Sicherheitsschutz gegen Angriffe zur Laufzeit gibt. Dieses Problem betrifft vor allem Java Cards, auf denen mehrere Anwendungen parallel laufen.

Die Java Card Firewall wird nun verwendet, um diese verschiedenen Anwendungen voneinander zu trennen. Dies bedeutet, keine Anwendung darf auf den Speicherbereich einer anderen Zugriff erlangen. Auch der Systemspeicher muss von dieser Firewall geschützt werden. Derzeit ist die Firewall Teil der virtuellen Maschine und rein in Software implementiert.

Es müssen nun Konzepte in eingebetteten Systemen evaluiert werden, welche die Java Card Firewall auf Hardwareebene unterstützen und dadurch sicherer machen. Diese Realisierung kann mittels einer Memory Protection Unit und einer Memory Management Unit durchgeführt werden. Durch Abwägung der Vor- und Nachteile fiel die Entscheidung auf eine Memory Management Unit. Mit dieser wird virtual paging implementiert. Anwendungen verwenden nur noch virtuelle Adressen, um auf den Speicher zuzugreifen. Die Verwaltung dieser Adressen wird vom Betriebssystem übernommen.

Damit diese Verwaltung vom Betriebssystem durchgeführt werden kann, muss eine dementsprechende Anpassung dahingegen erfolgen. Allerdings müssen bei diesen Veränderungen alle Aspekte und Vorgaben betreffend der Java Card Firewall eingehalten werden.

# Schlüsselwörter

Smart Cards, Java Card, Memory Management Unit, Virtual Addresses, Memory Mapping

# Abstract

Recent reviews of different Java Card applets have shown, that there is no adequate security protection against attacks at run-time. This problem mainly affects Java Cards, which are running multiple applications in parallel.

Now the Java Card firewall is used to separate these different applications from each other. This means that no application is able to gain access to a memory area from another one. Also the system memory must be protected by this firewall. At the moment, the firewall is part of the virtual machine and implemented in software.

Then several concepts in embedded systems have to be evaluated, which support the Java Card firewall at the hardware level and make it more secure. This can be realized with a Memory Protection Unit and a Memory Management Unit. After weighing the pros and cons the decision felt for the Memory Management Unit. With this virtual paging will be implemented. All applications must use virtual addresses to access the memory. The management of these addresses will be done by the Operating System.

To make it possible, that this management can be done by the Operating System, it must be adapted. But all changes have to fulfill all aspects and guidelines of the Java Card firewall.



# Keywords

Smart Cards, Java Card, Memory Management Unit, Virtual Addresses, Memory Mapping

# Acknowledgments

This master thesis was carried out during the year 2014/2015 on the Institute for Technical Informatics at Graz University of Technology.

First, I want to thank Christian Steger for the possibility to write my master thesis at the Institute for Technical Informatics.
I also want to express my special gratitude to Reinhard Berlach, who supported me with all of his possiblities over the whole time of this thesis.
Andreas Sinnhofer supported me especially at the end of this master thesis to finish the work.

Last but not least I want to thank my parents and all of my friends, who supported me throughout my student time and I am sure that they will support me in my future.

Graz, April 2015 Michael Zelle, BSc

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This chapter will give a short overview over this master thesis, which was made in cooperation with the Graz University of Technology. First I will explain what the CoCoon Project is. Then I will write about my personal motivation to do this project and why it is necessary to work on it. The next step is a short description of the tasks and the occurring problems with current technologies. In the end there is an overview over the structure of this master thesis. So now it is possible to see what can be expected in the next chapters.

## 1.1 The CoCoon Project

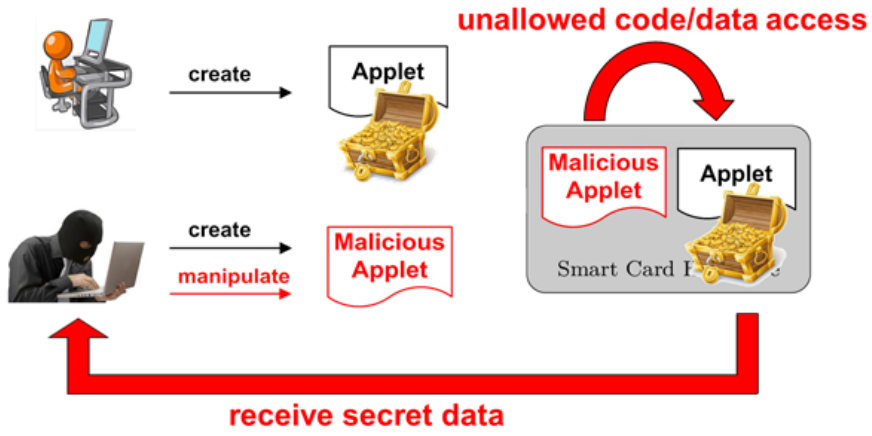


Figure 1.1: CoCoon overview

CoCoon stands for: Codesign for Countermeasures against Malicious Applications on Java Cards. It was a corporate project from the Institute for Technical In-

formatics - Graz University of Technology and NXP Semiconductors Austria GmbH Styria. This project ended in September 2014. In Figure 1.1 is a graphical overview of this project which show, how it works to get secret data from a smart card with a modified applet.
The final goal of this project was to get user centric ownership Java Cards which are compatible with the requirements of future smart card solutions. The problem about that is, that users are able to download untrusted software. It is important, that this software is not able to communicate with other applications (especially critical applications like e.g. payment software). So it must be guaranteed, that the integrity against logical attacks or physical attacks during run-time is given.
Another important reason for making a smart card more secure is because we do not live in a secure environment. A secure environment is, when a smart card can not be stolen or lost. Since that is not possible it can happen that someone is able to gain physical access to a foreign smart card.
In Figure 1.2 the abstraction layers of the Java Card implementation, which were used in the CoCoon project, are shown. The red marked path is the part, where this thesis will focus on. Most of the changes which are made, are on the hardware layer and go up into the software layers of the Java Card Virtual Machine.

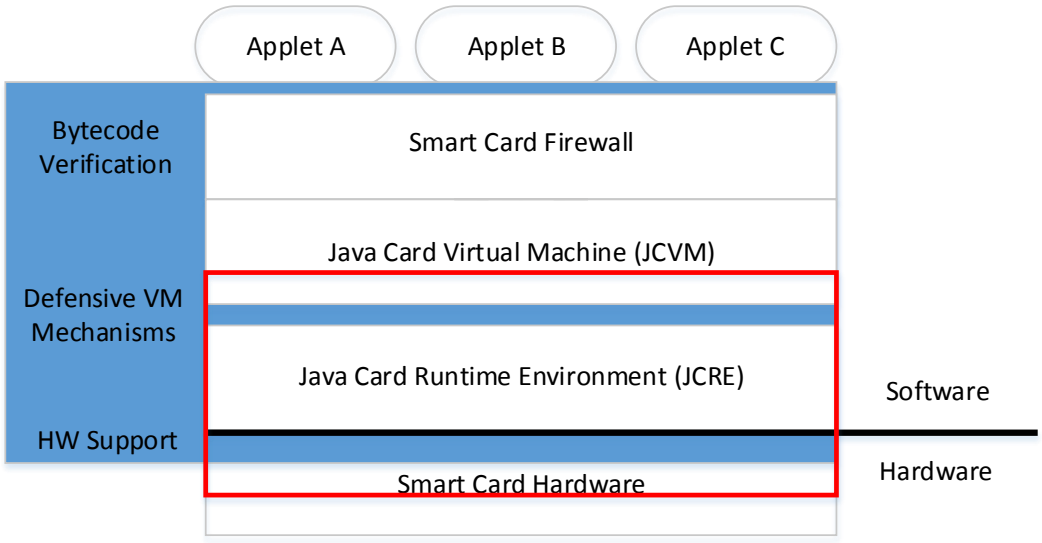


Figure 1.2: Java Card layers

## 1.2 Motivation

In a perfect world, there would be no need for security. But this world is not perfect. So we need security mechanisms to protect private information.
Smart cards are getting more and more important in our everyday life. Almost every person is using one every day. This starts by paying with debit or credit card or using a discount card. Sure, not every card has the same requirements in terms of security in the needed security. The next problem is, that the quantity of smart cards per person is getting higher, and so the stress for the user. It is not possible, to have every single smart card available at every time. Everyone decides which smart cards are most important for daily use. Those smart cards will be taken along against those cards that are not really needed. Therefore more functionality needs to be put on a single card as proposed by the CoCoon Project.
The goal of this master thesis is to develop a hardware device, which supports the Java Card firewall at hardware level. This is needed because current applied static verifications of Java Card applets provide insufficient security protection against fault attacks at run-time. Especially for the multi-application Java Cards it is a big problem. The Java Card firewall now separates different contexts from each other and protects the system space. To support this by hardware, makes it harder to break.

**Tasks:**

- Literature research of existing implementations of embedded memory separation.
- Requirement engineering of the rules of the Java Card Applet Firewall.
- Concept for connecting the Java Card Applet Firewall with a memory protection device.
- Design and implementation of a running prototype.
- Testing of prototype.
- Research of attack type which can be blocked.

## 1.3 Outline

The following provides a short overview of the following chapters and its structure.

In Chapter 2 is the information about the current state of the art. The first part is in Section 2.1, to explain, what a virtual machine is. The next is about Java and Java Card in the Sections 2.2 and 2.3. In 2.4, general information about smart cards and how they can be attacked are given. The last part is in Section 2.5, of

the possibilities of memory seperation.

Chapter 3 is about the design of the hardware. As first in Section 3.1, the system is planned. In 3.2 the memory device is explained in detail. Sections 3.3 and 3.4 explain the work of the virtual addresses and how they are stored in the memory of the protection device. And in 3.5 an overview of the possible usecases is given.

Chapter 4 will show up with all details about the implementation process. It starts in Section 4.1 with the used hardware and software. Section 4.2 shows details of the testing system and 4.3 of the protection device. As last step in 4.4 the test cases are shown. The last Section 4.5 shows how to rebuild the test environment.

In Chapter 5 the results and short analysis of the implementation are followed by the discussion of future developments and conclusions in Chapter 6.

# Chapter 2

# State of the Art

## 2.1 Virtual Machine

In computing, a Virtual Machine (VM) emulates a special part of a computer system. This type of emulation can be classified into two parts, dependent from the range of the virtualisation:

- System virtual machines
- Process virtual machines

A system virtual machine, can be used to simulate a complete computer. It is also able to run Operating Systems, which were designed for real computers like explained in [24]. This type is also called full virtualisation. It is based on the definition from Robert Gold and Gerald Popek: "A virtual Machine is an efficient, identical and isolated duplicate of a real processor" [23]. Good known software examples of this type of emulation are: Microsoft's "VirtualPC", VMWare's "Workstation" or Oracle's "VirtualBox".
The process VM, also called application VM or Managed Runtime Einvironment (MRE) runs an application within an Operating System. It only supports one single process. That means when the process is started it is created, and is destroyed, when the process is finished. The applet now runs in this process, which is independed from the hardware or the OS. This is really important for developing applications, which has to be independent from the used platform. Good known examples of this virtualisation type are Oracle's Java Virtual Machine or the Common Language Runtime from Microsoft's .NET Framework.

## 2.2 Java

### 2.2.1 General information

Java Card (JC) is a variant of the widely used programming language Java. Java has three components:

1. The Java programming language
2. The Java Development Kit
3. The Java Runtime Environment

Java is a simple object oriented programming language with the goal of being secure, dynamic, portable and architecture independent. It was developed from the company Sun Microsystems which was acquired by Oracle Corporation in 2010. The main goal of java is to provide the principle: "Write once, run everywhere" [7]. The program is written by the developer in the so-called "source code".
This code is not executable. For this it must be translated by the compiler into the byte code. The compiler is part of the Java Development Kit (JDK). Here every tool for programing and testing is provided. The biggest competitor to Java on the market is .NET from Microsoft.
The next layer is the Java Runtime Environment (JRE). This is a complete software platform, in which programs can be executed without dependencies of the Operating System (OS) of the device. Part of this JRE is the Java Virtual Machine (JVM). Because the generated byte code is mostly not tested on real hardware, a virtual machine is used. So it is possible to get independent from the platform. The JRE is provided for the following OS:

- Linux
- Windows
- Sloaris
- OS X
- other manufacturer with certified JRE

In Figure 2.1is the example for showing that the applications in Java are independent from the used hardware. Different applications run in the Virtual Machine which is embedded in the Runtime Environment. The applications are independent from the hardware, but the software from the lower layers must be working with this hardware. For example, a computer runs the Java Virtual Machine while a smart card runs a Java Card Virtual Machine.

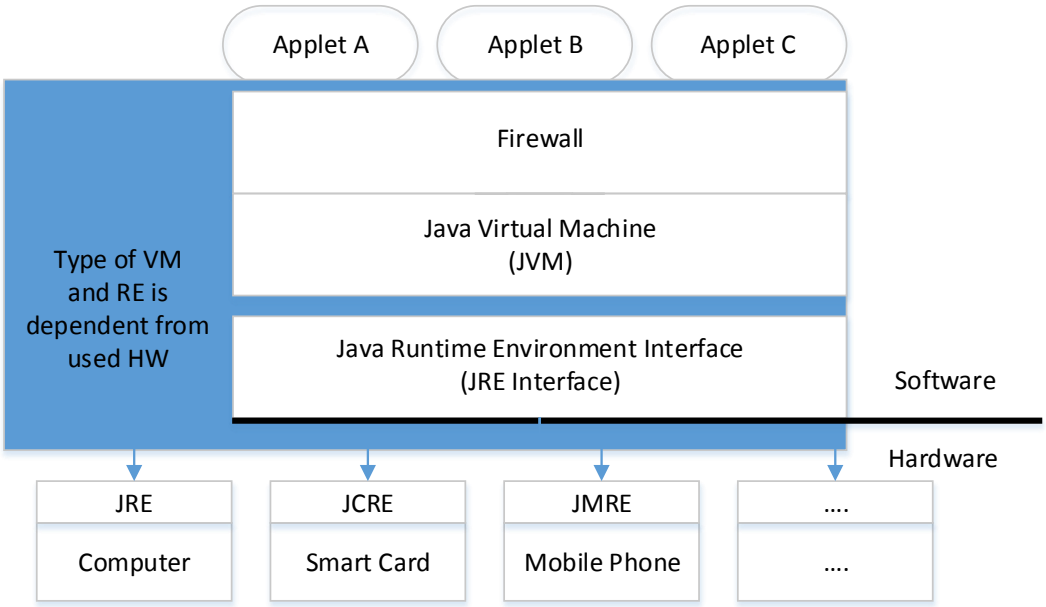


Figure 2.1: Independence of hardware with Java

### 2.2.2 Different Java Runtime Environment Platforms

In addition to the different Operating Systems, there are also different platforms. A graphical overview of these platforms is illustrated in Figure 2.2.

- **Java Platform Enterprise Edition (Java EE)**
  Java EE is for computer- and web-based applications. It is also one of the biggest platforms for the middleware market. The defined specifications are used to provide interoperability.

- **Java Platform Standard Edition (Java SE)**
  Java SE is the base for Java EE and Java ME. It is for general use with computers. Here a wide range of general purpose APIs are defined. The most important ones are the APIs for the Java Class Library which includes the specification for the Java Virtual Machine or the Java Language. The Java Development Kit is one of the best known implementations of the Java SE.

- **Java Platform Micro Edition (Java ME)**
  This platform is used for mobile phones, industrial controls, PDA's or set-top boxes. Java ME implements configurations and the profiles. It is a subset of Java SE. It can be used to grant access to internal functions.

- **Java Platform Java Card**
  It is a reduced subset of Java, which is used to run Java applets on chipcards.

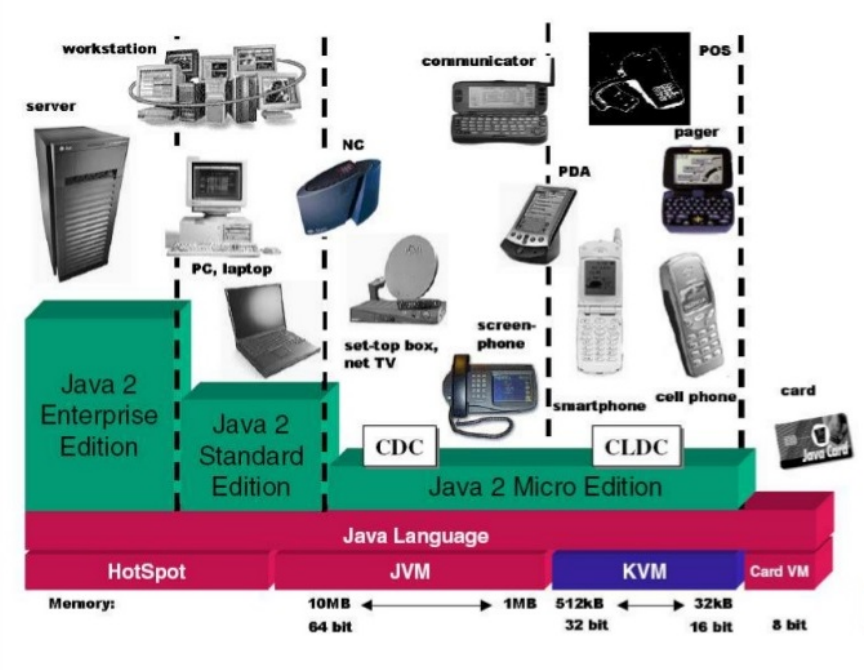


Figure 2.2: Different Java platforms [12]

## 2.3 Java Card

### 2.3.1 Components

The Java Card technology consists of two parts. The Java Card Platform Specification and the Java Card Development Kit are published by Oracle Corporation [2]. In version 2.2.2 of this specification three documents are included. This is to provide cross-platform and cross-vendor interoperability:

- The Java Card Virtual Machine Specification
  In this area everything for the Java Card technology is specified. This includes the behavior, the features and the services, which must be supported. Also included is a VM instruction set and the used file format for installing applets and libraries on any device which runs on Java Card technology.

- The Java Card Runtime Environment Specification
  Here the necessary behavior of the Java Runtime Environment, together with any implementation of Java Card technology, is defined.

- Application Programming Interface for Java Card Platform
  This part is only complementing to the Virtual Machine and Runtime Environment. The Application Programming Interface (API) must be compatible with the industry standards.

The Java Card Development Kit is for developers. This developing suite provides tools for two things: the first is for creating implementations of JC. The second - and most important one - is intended for developing applets based on the API specifications. This Development Kit also includes a fully working reference implementation and forms the basis to be used by developers:

- C - Java Card Runtime Environment
  This is a complete reference implementation of the Java Card Runtime Environment (JCRE) in C programming language.

- Off - card
  To provide a complete development chain, some platform components are off-card. This is for example a off - card installation application and a off -card converter. The converter produces a Composite Application Platform (CAP)-File and the installer initiates the transfer into the JCRE.

- Additional design and testing tools
  This is for prototyping and testing of applications.

### 2.3.2 Benfits

By using Java Card technology, some benefits can be taken directly from the Java technology. It is possible to use an object-oriented programming language with common development tools. Furthermore, also unique were developed for JC:

- *Interoperable*
  Adapted from Java technology ("Write once, run everywhere" [7]), means that any applet, which was developed by Java Card technology, runs on any smart card using JC. With running this, it is independent from the card hardware or vendor.

- *Security*
  The security mechanisms of the Java programming language are also available in Java Card. So it is possible, that the latest security algorithms, which are available at the moment, can be used by the developers.

- *Multi-Process*
  It is possible that multiple applets can run on one smart card.
- *Flexibility*
  New applications can be installed securely. Even after it has been issued. This makes it easier to respond dynamically on threads or requirements.
- *Compatible with Existing Standards*
  The APIs have international guidelines, which refers to given standards like EMV or ISO7816. Also some industry-specific standards are available.

Together with these benefits and also because of the fact that there are a lot of good open source development tools, it is clear, that Java is one of the most widely used programming languages around the world.

### 2.3.3 Differences between Java and Java Card

Java Card applications are designed to be executed on very lightweight hardware, therefore there are some differences compared to the full Java specification covering the following areas. These differences are:

- Programming Language
- Bytecode
- Libraries
- Specific Features
- Development

#### Programming Language

Some Java language features are not supported by Java Card. Due to the limited memory capabilities of smart cards, Java Card only provides support for basic types and does not support types like char, float, double, long or multi-dimensional arrays. Further limitations apply to the run-time system, which does not support among other features object cloning, multi-threading or garbage collection.
However, every language construct which is available in Java Card has the same behavior as in Java. So it is possible, that a Java Card program can be compiled by a Java compiler.

#### Bytecode

The generated bytecode is encoded differently. Again, as smart cards are very limited in terms of available memory, the generated byte code is optimized for size. So a Java Card applet normally use less bytecode than a Java applet, even when compiling the same source code. This is necessary because smart cards have lower resources than a computer. Therefore a technique, which limits the package size to 64KiB is used. Application code have to be divided to fit in this size limit.

#### Libraries

This is one of the parts with the most differences from Java. For example, the Java Security Manager class is not supported. Security policies are implemented in the Java Virtual Machine. Even some features which are not supported in Java are supported in the Java Class Library, like fast RAM variables.

#### Specific Features

Java Card Virtual Machine and Java Card Runtime Environment supports some features, which are specific to the Java Card platform:

- In Java Cards, objects are stored to the persistent memory because the Random-Access Memory (RAM) is only used for temporary objects. JCRE and bytecode have been adapted for that.
- Smart cards are externally powered and therefore rely on persistent memory. The JCRE inludes limited transaction mechanisms.
- The Java Card firewall which isolates applets in different contexts from each other.

#### Development

Because of the mentioned differences, even the development process changes. Other coding techniques are used to get better performance or memory usage. The earlier a code has been debugged in a real Java smart card, the better the results will be.

## 2.4 Smart Cards

### 2.4.1 General smart card Information

Smart cards are pocket sized cards with an integrated circuit. In 1969, the german scientists Jürgen Dethloff and Helmut Gröttrup patented their first smart card [4]. It was made in the memory card concepte. This is a simple card with a memory, which can be read or written. Later it was also possible, to prevent access by others

| | Name | Description |
|---|---|---|
| **C1** | VCC | Power Supply for the smart card provided by the reader. No battery is needed. |
| **C2** | RST | Reset Signal. Can be used to reset the smart card during communication. |
| **C3** | CLK | Clock signal for the smart card. |
| **C4** | RFU | Reserved for future use. |
| **C5** | GND | Reference Voltage (Ground) |
| **C6** | VPP | This is an input for voltages higher than VCC. It is often used as programming voltage to programm the persistant memory. |
| **C7** | I/O | This is a serial I/O Port which operates in half-duplex mode. |
| **C8** | RFU | Reserved for future use. |

Table 2.1: Contacts for smart card pinout

with a Personal Identification Number (PIN) or a passphrase. Even though, this is a "simple" type of smart card, it is still in use nowadays.
The most common used system nowadays is the processor chip card. Examples are EC-Cards, credit cards or debit cards. The name of this smart card type comes also from the used microprocessor. So there is no possibility to access the memory directly. The advantage of this architecture is, that an Operating System can be installed (for example a Java Card Virtual Machine (JCVM) on a Java Card). This gives the possibility to run applets on this system and protect it.

In Figure 2.3 the difference of those two systems can be seen. The only thing both architectures have in common is the input/output (I/O) Part of the card. It depends on the card type. It is split-up into three types:

### Contact smart cards

This type of smart cards use a contact area as shown in Figure 2.4. This is called the smart card pinout. It has an area of approximately one square centimeter. They can look different, but they are always split into eight subareas. They are explained in Table 2.1.

### Contactless smart cards

The communication with another device is without physical contact. The communication channel is established using Radio-frequency identification (RFID) technology. The only thing which is needed by the smart card for working is an antenna as illustrated in Figure 2.5. Like the contact smart card the supply voltage is provided by the reader through RF-induction. The communication process is defined in ISO/IEC 14443-4 [17].

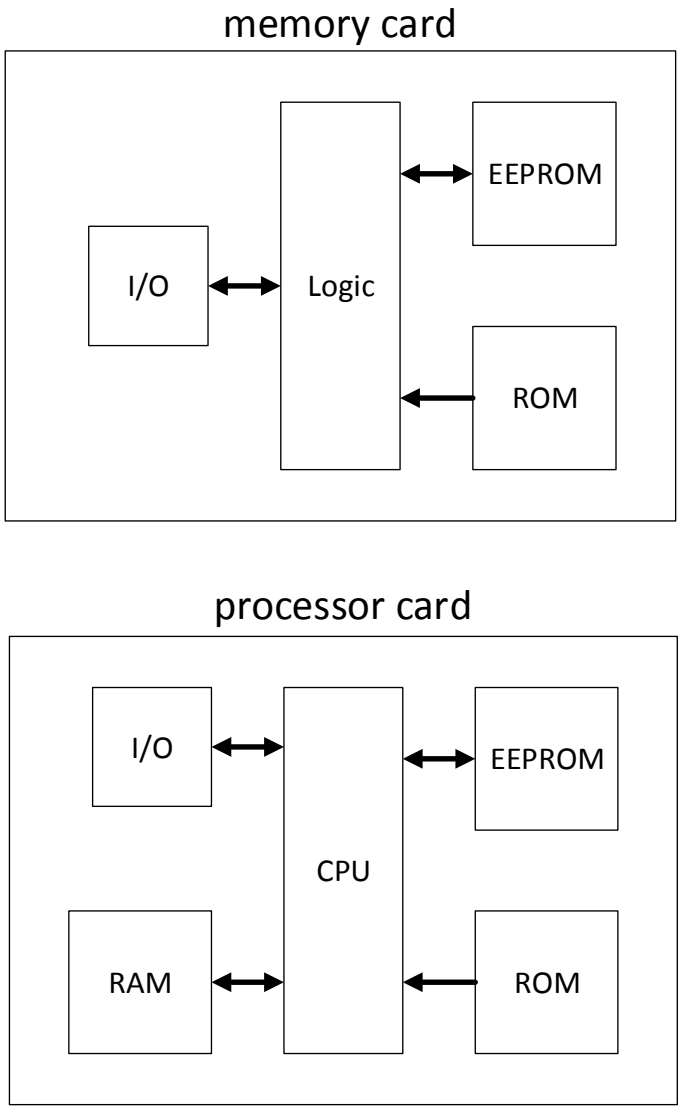


Figure 2.3: Block diagramm of different smart card types

### Dual interface smart cards

This type of cards have communication interfaces for contactless and contact communication on a single card. It is controlled by one chip and is currently the most utilized type.

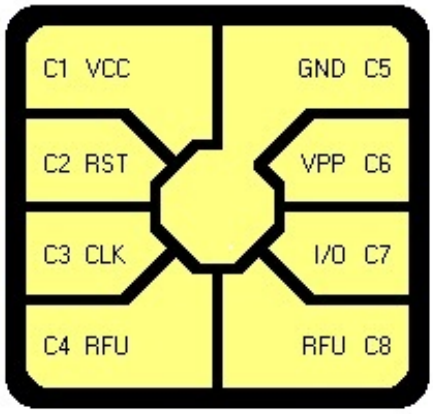


Figure 2.4: Smart card contacts [8]

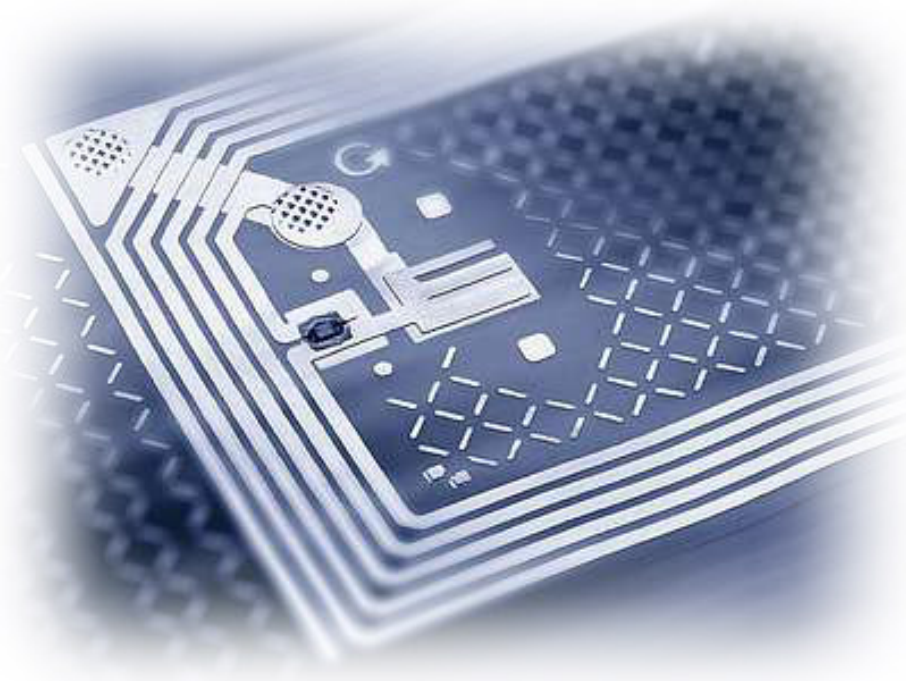

Figure 2.5: RFID card with chip and antenna [6]

### Hybrid smart cards

On each smart card, multiple chips are available and antenna and pinout are again the used I/O interfaces. Every chip is separately connected to every contact inter-

face.

#### Multi component smart cards

This is a very special type of smart cards and is normally used only in specific solutions. An example is a smart card which uses an integrated fingerprint sensor to authenticate the owner.

### 2.4.2 Different types of smart card attacks

Smart cards are a common used tool in daily life. So these cards are normally not stored in a secure area. So it can happen easily, that an attacker gets it into his hands. That is the reason why smart cards need a lot of security procedures to avoid in all circumstances, that critical and sensible data can be accessed. A good example is a debit card. If the attacker gets the PIN of the card, he is able to access the bank account. That is the reason why a big amount of money is invested in finding working attacks, so possible countermeasures can be developed. In this section the different types of attacks on smart cards are shown. In this thesis we relate to the subdivisions made by Marc Wittenberg [26], because this are the most common types which are used for definitions.

As shown in Figure 2.7, attacks on smart cards can be distinguished between

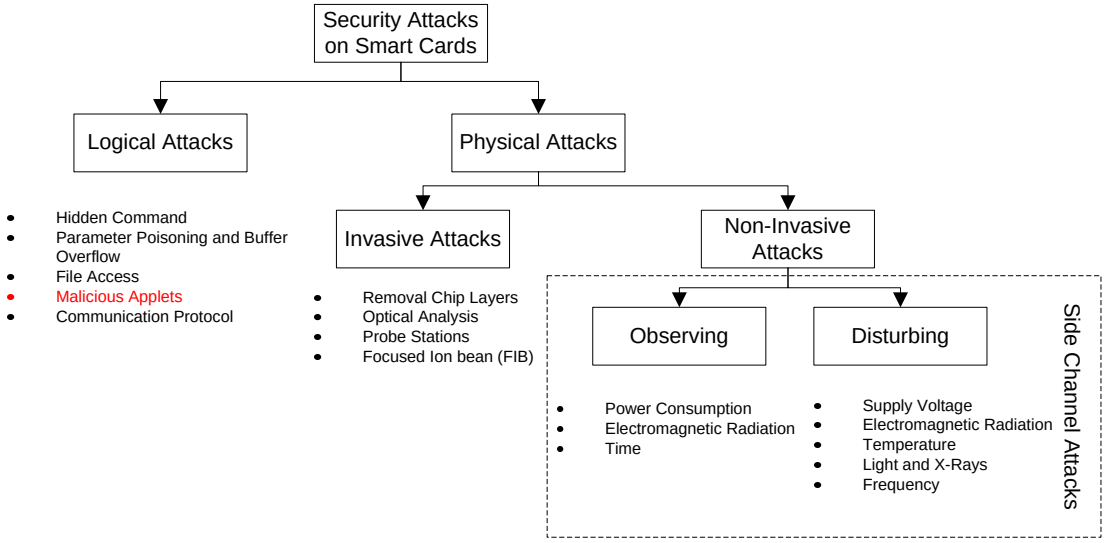


Figure 2.6: Types of smart card attacks [19]

**physical attacks**, **logical attacks** and **side channel attacks**. There are several forms of attacks and countermeasures against it:

| CLA | INS | P1 | P2 | Lc | Data | Le |
|---|---|---|---|---|---|---|
| Header | | | | Trailer | | |

Table 2.2: Command structure for smart cards

### Physical attacks

Physical attacks are splitting again into two categories: *invasive* and *non-invasive* attacks.

Invasive Attacks

Invasive means active attacks were physical changes are made. The controller has to be removed from the smart card. This kind of attack can only be realized with a great effort and special equipped laboratories.

The removal process of the controller starts with heating the card until the controller can be removed. Sometimes it is also possible to remove it with a normal knife. After that, the chip needs to be cleaned from the epoxy. This happens with warmed concentrated nitric acid (<98%). The last step is to clean in an ultrasonic bath with acetone. After this procedure the contact areas of the chip can be connected with an analysis or manipulation environment. These kind of attacks are often done by the card manufacturer itself. The gained information of such tests are used to find potential weaknesses.

Non-Invasive Attacks

This part of physical attacks can also be considered as side channel attacks. Due to this classification this section will be explained more detailed in "Side channel attacks".

### Logical attacks

Logical attacks are the most used technique for attacking a smart card. This has a simple reason. This type of attack does not need a lot of equipment. Only a computer and a working smart card reader are needed. Via this reader the whole communication is done. The next thing is that smart cards communication works with commands. This command structure is shown in Table 2.2. Every smart card uses only a specific number of supported commands. By not disabling of not supported commands, attackers have the possibility to use them for their purpose. The second chance for attackers are bugs in the software implementation. The goal of this master thesis is to avoid some of this logical attacks to raise the security level of smart cards.

Countermeasures for logical attacks

Logical attacks are dependent on bugs in the implementation. The higher the complexity of an implemented code, the higher the risk of bugs. But there are some "simple" strategies to avoid it. The most important ones are:

- The most important thing to avoid bugs is testing. Beside positive tests, which lead to expected results, it also is inherently important to provide error test cases to see how software reacts in error cases.
- Building of small functional blocks which are easier to understand.
- Keep the code as simple as possible.
- Using of standardized interfaces or re-use of proven software.
- Using of Java Card Operating System or .NET Micro Framework to get the advantages of object-oriented programming languages which makes security features easier to use.

### Side channel attacks

Side channel attacks are non-invasive (passive) physical attacks. The goal is to get or manipulate data without making a physical change to the smart card. This is possible because the integrated circuits of switching semiconductors are sensitive to basic physical phenomena. There are two different types of side channel attacks: observing and disturbing. The phenomenons that can be used for observing are:

- **Power Consumption**
  The power consumption directly depends on the processes which are running at the moment on a chip. So knowing the power consumption makes it possible to get information about the processed information, because different commands need different amount of power. Now it is possible to relate to the command sequences.
- **Time**
  The amount of time, which is needed by a processor to complete a task, can be related to to the process parameters.
- **Electromagnetic Radiation**
  Appears every time a transistor is switching. Like the power consumption this can be related to the current processes.

Disturbing of some parameters can be used to modify them like changing bits or make it easier to perform the observing. The possible parameters are:

- **Electromagnetic Radiation**
  With a strong pulse it is possible to induce signals into the chip to change the behavior.
- **Power supply**
  This changes the behavior of the circuits because they are designed to run at a defined voltage. Glitches can appear which makes it possible to change commands.

| | Logical | Physical | Side Channel |
|---|---|---|---|
| Equipment | PC | PC, Probe Station, SEM FIB, Microscope Chemistry Lab, etc. | PC, Oscilloscope Function Generator |
| Cost | $1-10K | $100K-1M | $10-100K |
| Sucess Rate | Low | High | Medium |
| Development Time | Weeks | Months | Months |
| Execution Time | Minutes | Days | Hours |

Table 2.3: Attack statistic [26]

- **Temperature**
  Also changing of the behavior of the circuits because the devices have a limited temperature range to work within normal parameters.

- **Frequency**
  Microprocessors are designed to work at a specific clock frequency. If the frequency gets higher, it is possible that errors occur. So the time for this operations raise and analysis of the function easier. On the other side with a lower clock frequency it is gets easier to observe the controller-BUS. Especially modern smart cards with high frequencies are really hard to be observed.

- **Rowhammer**
  A new method from Mark Seaborn, Matthew Dempsky und Thomas Dullien from March 2015. A side effect in dynamic RAMs is, that the memory cells leak their charges to the memory cells arround. So it is possible to change the content of memory cells, which are not directly addressed.

Countermeasures for logical attacks
To avoid such manipulation, smart cards get equipped with sensors for voltage, frequency and temperature. Those are so called "Watchdogs". If limits of these sensors get exceeded, it is possible to reset the card or make it unusable. But this also makes the card less robust because false alarms can occur.

#### Statistic

In Table 2.3 a statistical overview of the presented techniques is shown. Physical attacks have the highest success rate, but a lot of equipment, money and time is needed. On the other hand, Logical attacks are the cheapest ones where only a computer is required. But the success rate is lower compared to the others.

### 2.4.3 Possible attacks

There are a lot of known attacks to Java Cards. The background for this is, that attackers try to read the whole memory of the card to gain information. But it is

also useful because it allows reverse engineering. With this in mind it is possible to understand the implementation of some parts of the JCVM. A second but also very important detail is, that attackers get the ability to read private keys. With this, it is possible to read the communication (this is always possible) and decrypt the data from the crypto algorithm. This applies to the whole card communication.
Iguchi-Cartigny and Lanet [16] present such an attack. They use a mutable applet that makes it possible to break the security box of the JCVM. The prerequisites of their attack are:

- post issuance is allowed,
- the attacker has the credentials,
- the card must not include a BCV on-card.

Starting from here, they load their Trojan applet on the card and can modify it in a way, so that they are able to read out the memory of the card. They use a weakness in the invokestatic bytecode to gain access to the memory.
Hogenboom and Mostowski [15] also present an attack which allows them to read out the whole memory of a Java Card. This attack is based on type confusion. They exploit a very common bug in the transaction mechanism of the JCRE. This attack is not detectable by any BCV since this is a complete correct Java Card applet.
An attack very similar to the one of Hogenboom is presented by Mostowski [21].

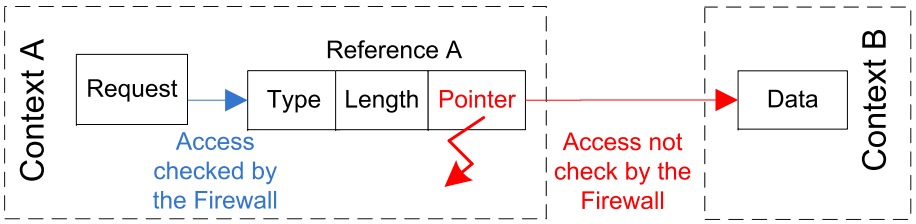


Figure 2.7: Simple illustration of the principle of the attacks [19]

Both of these attacks gain access to the memory due manipulating the pointer address to the data-field of the array. This principle is illustrated in Figure 2.7.

All of these attacks gain access to the memory due to the fact that they can break through the Java layer of the JCVM. The attack shows that the Java Cards need a protection mechanism to secure more than only the objects of a context. The cards also need a mechanism to protect the memory itself.

## 2.5 Memory Separation in Embedded Systems

### 2.5.1 Overview

With our new technologies, embedded systems get more and more powerful. Another advantage is that they are easier to handle and therefore they can be better included in our everyday life. Furthermore, that this embedded systems are getting more and more multifunctional.
The attack schematic has changed in the last ten years. Back in that days, software attacks, like shown in Section 2.4, were targeting powerful devices like computers and laptops. But in the last years, this has changed rapidly. This leads to the requirements of good security and protection techniques. Memory separation is only one of these. But while it is tried to increase security, it has to be ensured, that the system costs and power management do not increase too much. And the performance of the system has to be ensured. Especially in time critical embedded systems.
Because of the needs in separation, for secure data and code segments on mobile devices, the company Advanced RISC Machines (ARM) started working on a new technology. Although the name is rather unknown outside the involved community, most of the microprocessors worldwide are produced by this company. The TrustZone technology [3] for most of ARM systems is the result of an intensive research and development phase. It is a System-on-Chip (SoC) technique, which enables separation between non-trusted and trusted execution. The second important thing was that the context switch kept being fast. This worked because TrustZone added an additional address bit to the system. Afterwards the SoC adaption was needed so that all interfaces, devices etc. took care of that bit.
An alternative way for protecting is system virtualization [14]. But there was one big problem in this technology. Embedded systems had only a limited hardware virtualization support, so this led to a performance overhead [13]. But with scheduling of tasks and regulate processes which do not have to run at any time, this technology started to make its way.
Nowadays there are more different technologies for memory separation, because embedded systems have higher performance than few years ago. To solve this issue, it is possible to use a software and hardware solution. An example for software solution is the Java Card firewall (2.5.2). Cortex-M series are good examples for a hardware solution. From Cortex-M3 and higher versions there is an optionally Memory Protection Unit available. In this master thesis, the used processor is a Cortex-M1. This is the reason, why an implementation of a security unit like in Section 2.5.3 is necessary.

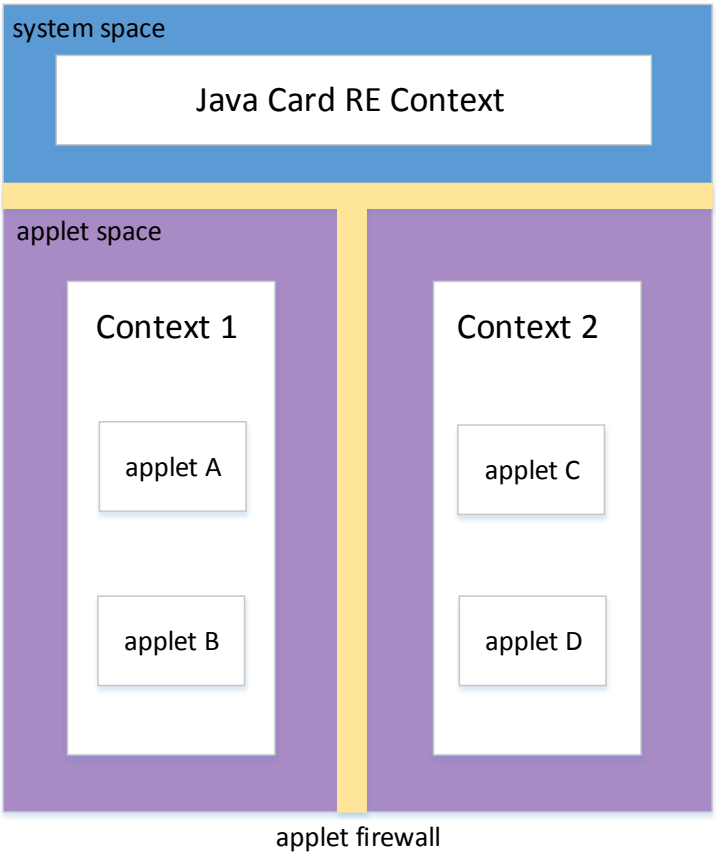


Figure 2.8: Mode of operation of Java Card firewall [22]

### 2.5.2 Separation with the Java Card Firewall

The Java Card firewall (also called applet firewall) is a software solution for memory separation and protects against the most common security concern like malicious applets. As you can see in Figure 2.8, the applet firewall build different separated memory spaces. This spaces are called "context". Every time an applet creates an instance, the Java Card Virtual Machine assigns it to an existing or a new context. Now the applet firewall is set between the system space and the different group contexts. So if an object wants to access another object in the same context, it is allowed because there is no firewall in between. But if an object tries to access an object in another context, this action is blocked by the firewall. The most important context is the system space which holds the Java Card Runtime Environment context. This JCRE context has other privileges than a normal group context. It is possible from the system to access every other context. But in the reverse situ-

ation, no group context is able to access the system space because it is blocked by the firewall. All those facts leads to one important question: How does the JCVM choose, which context is trying to access?
At any time, the JCVM has just one active context. This can be a group context or the JCRE context. Thereby, it is easy to check the permission. But it is also possible to access another context over shared interfaces. If such an access occurs, the JCVM performs a context switch. This works in the following steps:

1. The active context is pushed to the stack.
2. The context of the called interface is loaded and gets the new active context.
3. Like implemented in the interface, it is now possible to access every object in this context.
4. After the return value is provided by the interface, the JCVM performs a restoring context switch.
5. The originally context gets popped from the stack and gets the currently active context.

With this scheme the applet firewall allows to access one context from another at the same time it prevents accessing objects in another context which are not shared.

### 2.5.3 Memory Protection Devices

Memory protection is a part of the most modern OS. The goal is to prevent unauthorized access to memory. The three most common ways for memory protection are:

- segmentation
- virtual memory paging
- protection keys

There are also some other technologies but they are rather unused in modern systems. In this thesis it was a goal to design a memory protection for the java card firewall. After analysis of this goal there were different possibilities to implement this. The two most common devices are a Memory Protection Unit (MPU) which uses segmentation, and a Memory Management Unit (MMU) which is based on virtual memory paging.

### Memory Protection Unit

A typically Memory Protection Unit is set between Central Processing Unit (CPU) and memory. This memory gets segmented into memory regions. After that it is possible to write access permissions for every single region. Because of that simple behavior, this type of device has a really low overhead. This is good for the performance of the system. Out of that there are also some disadvantages.
Normally the permissions are fixed and can not be changed during runtime. This decreases the dynamic use of the MPU. But to solve this problem, newer devices are built with a rewritable permission register. So it is possible to change the usage of the regions and make the behavior more dynamic. Next to this new type can also be integrated directly in the CPU, the more common type is the static one.

### Memory Management Unit

The most common use for a Memory Management Unit is using virtual memory management. Every address which is requested from a process is a virtual address. This virtual address is sent to the MMU. With the "help" of a Translation lookaside buffer (TLB), this address is converted in a physical address which can be used to access the memory. But before this address gets translated, it is checked for the access authorization. If a process do not have the privilege to access an address, a page fault occurs. This error message leads to the next advantage of a MMU. Every access and management is handled by the OS. Through that, the device can react dynamic because at every time, it is possible to change the permissions. On the other hand it reduces the memory fragmentation. If a new memory gets allocated, the OS can search in the whole address space for a matching segment.
As you can see now, the MMU has some more advantages than a MPU. The big problem with that is the generated overhead in the system. That results in a loss of performance.

### Protection Keys

A system which is secured by protection keys divides the physical memory into a fixed size. Now every memory region gets an associated protection key (this is a numerical value). Further, each process has a associated protection key value. Every time a process accesses a memory region, it is checked by the hardware, if the protection key value of the current process suits to the protection key of the memory region. If yes, access is granted. When it fails, a memory exception occurs. Because of this simple behavior, the implementation is simple and during runtime it has less overhead compared to MPU and MMU. But it is a completely non-dynamic system. A process is only able to access a region or not. Even simple read and write privileges are not used in this system.

# Chapter 3

# Design

## 3.1 System Overview

For implementing a hardware memory protection the first step is to specify a system, where this device can be tested and used on. Such a system can be seen in Figure 3.1. Even there are more parts possible to specify the system, in that case are only the main ones included. A main part is needed to test the protection unit or is needed for I/O.
All components are connected over a Advanced Microcontroller BUS Architecture (AMBA) Advanced High-Performance BUS (AHB) Binary Unit System (BUS) system. The Cortex M1 is the CPU of this system and is connected over a AHB master interface to the other parts. Four other parts are connected as slaves to the AMBA BUS.
First is the interface between the AHB and the Advanced Peripheral BUS (APB), called the bridge. The reason for this is, that the two I/O devices are connected over the APB. The GPIO interface shows, if the system is in normal or remap mode. The UART interface is used in debugging mode, when the system run with a "OS".
The second part is the memory controller. Normally this is directly connected to the AHB. In our system this was changed and it is now connected to the MMU over a AHB interface.
The next part is a Static RAM (SRAM) block. This is used as quick but small memory. It is not protected by the MMU.
The last block is the MMU. It has two AHB slaves and one AHB master interfaces. Because of that, the MMU is directly set between the AHB and the memory controller. It also has a direct connection to the CPU over the interrupt pin. The structure is described in Section 3.2.2.

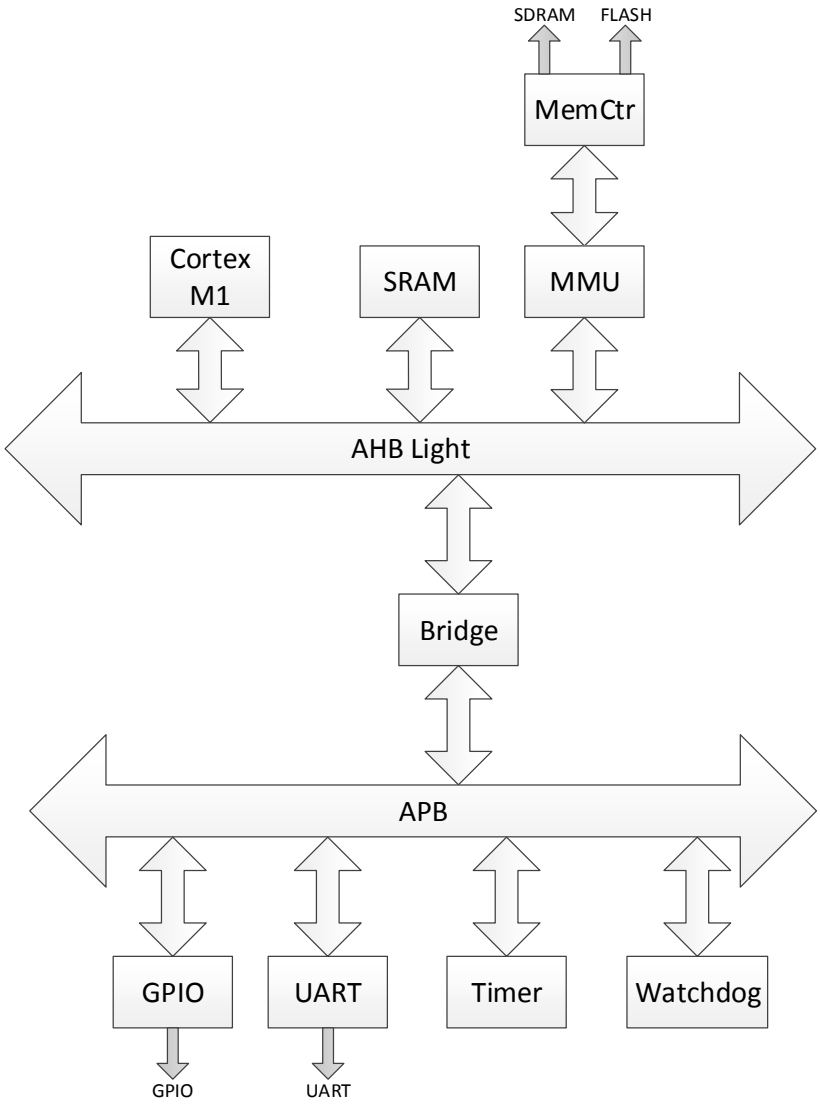


Figure 3.1: System overview

## 3.2 Chosen Memory Protection Device

### 3.2.1 Memory Management Unit

In Section 2.5.3 the most common protection systems were shown. After weighting the pros and cons of the single systems, it was possible to exclude the protection keys method, because the system should be at least dynamic enough to differ between

read and write access. Now only the Memory Protection Unit and Memory Management Unit were left. The big disadvantage of the MPU is that the permissions are fixed. So this system needs to be adapted for the wanted usage because access privileges should be able to be changed during run-time.
Finally the MMU was chosen as protection device because of following reasons:

- Less memory fragmentation because memory management is handled by OS.
- Virtual memory paging prevents the following:
  - If attacker can hack into the OS, he can only get the virtual addresses and is not able to make conclusions of the internal structure of the MMU.
  - With using virtual addresses, it is possible to prevent some known attacks like described in Section 2.4.3.
- Dynamic access privileges for every page which can be changed
- Make it easier for future projects to adapt the design for new tasks because it is more powerful than a MPU.

### 3.2.2 Internal structure

In the final system, the MMU will be the most important part for the JCRE firewall. The non-volatile writeable memory is protected by it. In this part of the memory the CAP-Files and objects from the Java Card is stored. Figure 3.2 shows the inner architecture of the MMU. There are two parts: The config part and the memory part. This two parts work together over the lookup table.
The config part consist over the AHB slave where the data which is needed for the lookup table are transmitted. Also the context Identification (ID), which is stored in the MMU and used to check the access rights, is saved over the config part as well.
The memory part is for transmitting the data into the memory controller. This part also consist of an AHB slave interface. After that, the memory logic is located. Addresses are controlled and then compared to the lookup table. If an error occurs, a page fault interrupt is invoked. Otherwise, if the lookup table provides a hit of the addresses, the data is transmitted over a AHB master interface which is connected to the memory controller. This architecture was chosen to make the integration into the existing system easier.

## 3.3 Memory Map

For each application which is running in the Java Card OS, a virtual memory is created. This virtual addresses have to be mapped into the physical memory as you can see in Figure 3.3.

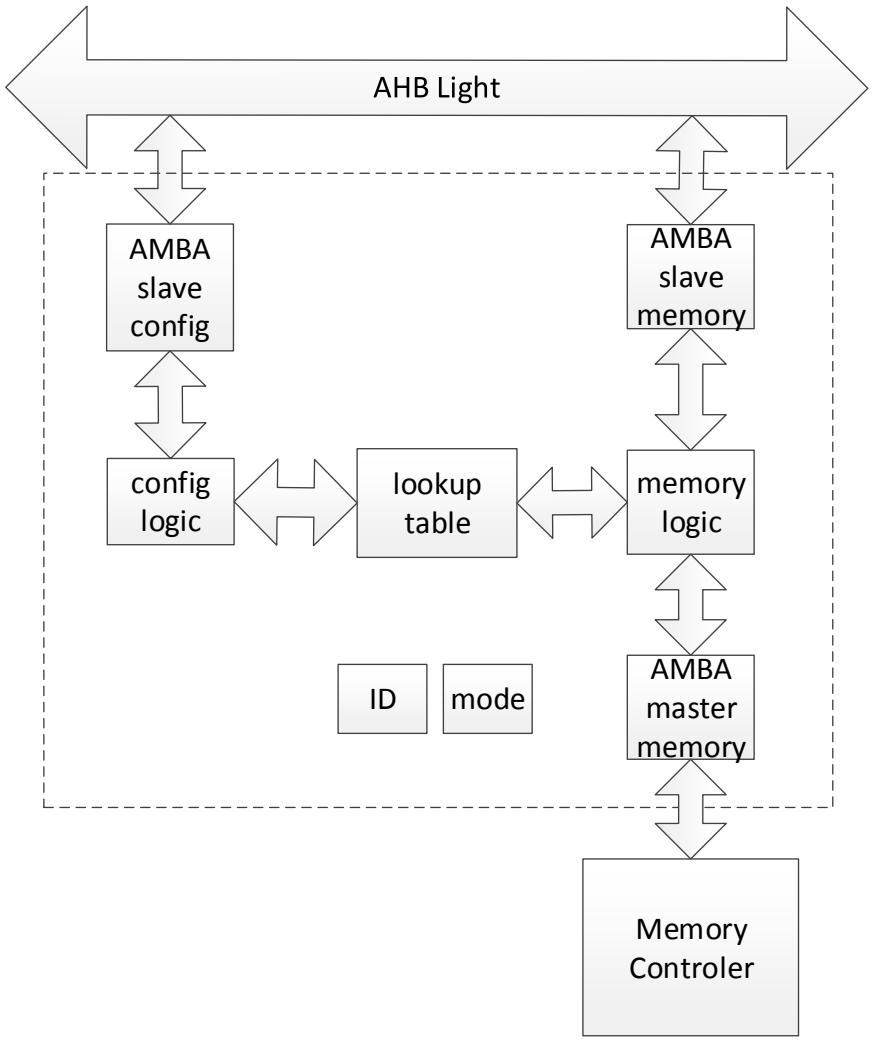


Figure 3.2: The Memory Management Unit

Virtual addresses will start at entry 0x04000000 and go to 0x07FFFFFF. This is because the MMU has to check, if a address is a physical or a virtual. This solution is done for two reasons:

1. The MMU can easily check if a address is virtual or physical. Just a bit has to be controlled to find this out. In future this bit is declared as VIRT. When HIGH, the address is virtual, when LOW, it is a physical address.

2. That this type of addresses is also working, when debug-mode is activated.

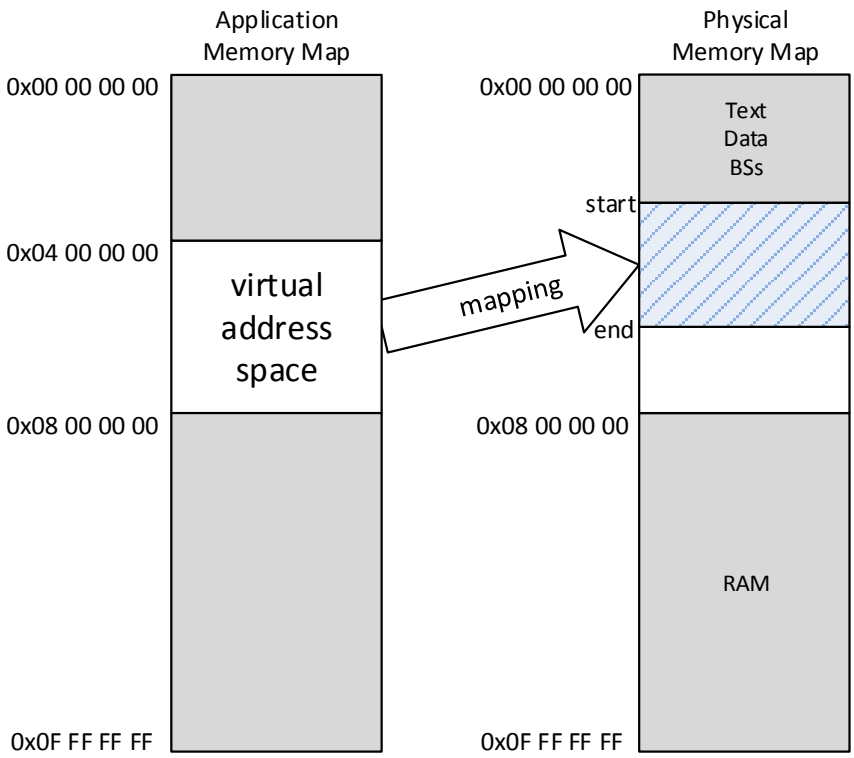


Figure 3.3: The different types of memory maps

Debug mode means, that on the used Field Programmable Gate Array (FPGA) board the non-volatile memory is deactivated and everything is stored in the RAM. But because of the size of the memory, which is under the virtual address area, this solution also works here.

So when an address is in the virtual address space it is mapped into the physical address space. This is done by the OS. It has a declared start and end point there. Important is, that the endpoint of the physical addresses is before 0x04000000 to occur problems with the virtual address detection. Every application should only try to access the memory over such a virtual address. Otherwise a page fault is given.

## 3.4 Parsing of lookup table entries

As seen as in Section 3.2.2, the configuration logic hast access to a lookup table to save the physical address and other information. These information is received over

the AMBA interface. As seen in Figure 3.4, there is the address and the data BUS and each of them is 32-bit width. The address BUS is split up into:

- 4-bit AMBA for decoding the slave
- 1-bit RAM for showing if access to RAM or flash
- 1-bit VIRT for showing if virtual address or physical address
- 5-bit TLB for decoding the address for lookup table
- 12-bit Page number
- 9-bit Offset

The data BUS is split up into:

- 12-bit RFU (Reseved for Future Use)
- 18-bit Physical address
- 1-bit VALID declares entry for as valid or invalid
- 1-bit RW for showing read and write privileges

When this information is available, the configuration logic saves it into the lookup table. The size of one entry is also 32-bit and there are 32 entries available. For identifying, the TLB bits are used for addressing the lookup table entry. After that, the configuration logic provide the data at the 32-bit CONF_In interface. This is partitioned in the following order:

- 12-bit Page number
- 18-bit Physical address
- 1-bit VALID
- 1-bit RW

**Example:**
At the address BUS is the value is "04 C9 70 13" and the data BUS value is "00 00 03 17", both sent in hexadecimal. From the address side, this means that a virtual address is accessed. The TLB entry has the number 6. The page number is 1208 and the offset is 19. From data side there can be seen the following information. The physical address is 197, it is a valid entry and has read and write access.
The result of that means that at CONF_In the value is "4B 80 03 17". To prove that this is true, in Section 5.1.1 a test was made with this values.

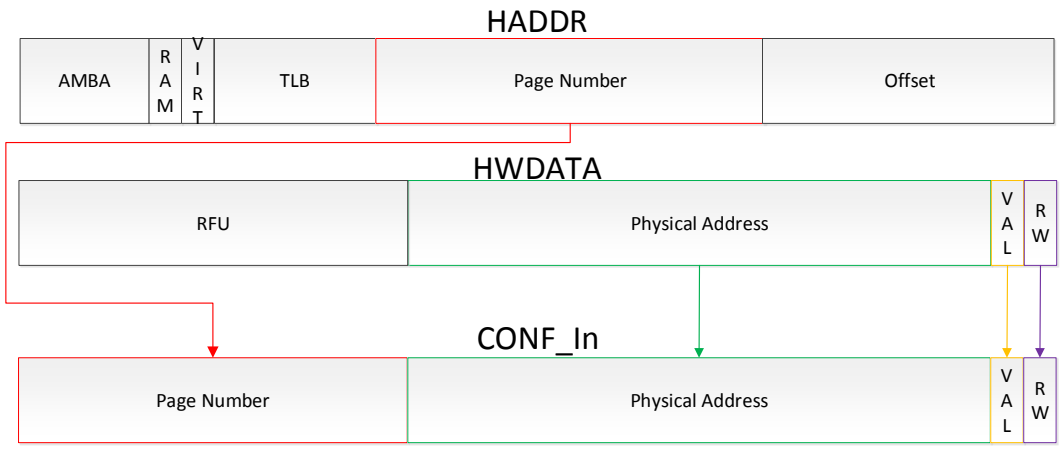


Figure 3.4: Parsing of lookup table entry

## 3.5 Use Cases

This section describes the most common processes are described. This will happen on three different layers. The first are the use cases from the perspective of the Memory Management Unit on hardware level. The next is on the level of the Operating System. And the last is from the Java Card Virtual Machine.

### 3.5.1 Use Cases of the Memory Management Unit

In Figure 3.5 the possible use cases of the Memory Management Unit are shown. Each use case is triggered by the OS. There are five different possible use cases:

- Memory Access
- Add Table Entry
- Delete Table Entry
- Context Switch
- Switching between System and User Mode

#### Memory Access

In this use case the behavior of the MMU is listed, when there is a read or write access to the memory. This happens in the following steps:

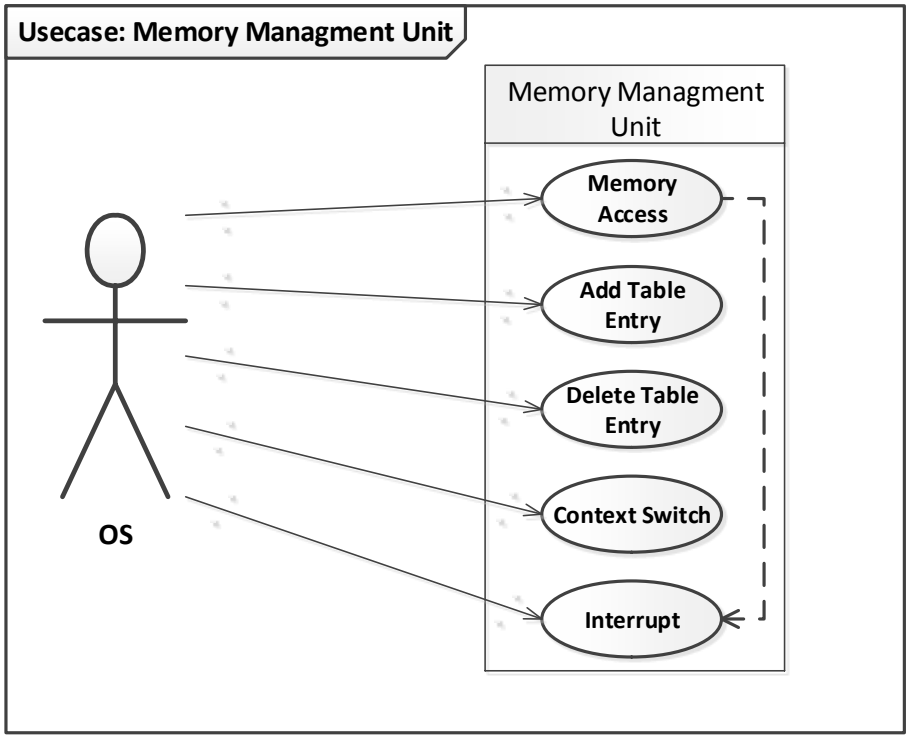


Figure 3.5: Use cases of the Memory Management Unit

1. Access to an address.
2. Checking from the memory logic which memory is accessed. There are three types of memory:
   - Transient memory
   - Read Only Memory (ROM)
   - Non-volatile memory: here all loaded applets and the JCRE are stored
3. Depending of the type of the address the access is granted in two different ways:
   (a) Access to a physical address is mapped 1:1 to the memory.
   (b) If it is a virtual address, then there are more steps to do:
      - Searching the lookup table for an entry.

- Checking if the entry is valid. Then there are two different possibilities:
  i. The lookup table entry is valid. The virtual part of the address is replaced with the physical page frame number. The access to the memory is granted.
  ii. In the lookup table is no valid entry or the privileges for access are not given. In this case this leads to a page fault interrupt, which is handled by the OS.

### Add Table Entry

This use case happens, when new memory needs to be allocated. Then a new entry in the lookup table has to be created. This is needed for a new table entry:

1. At first the data which is needed to create a table entry. This consists of four parts:
   (a) Virtual Page Number
   (b) Physical Page Frame
   (c) Metadata
   (d) Context ID
2. In the next step this data has to be stored at the right location within the lookup table.

### Delete Table Entry

IF an application do not use a page anymore (application free memory or has not used the page in a certain amount of time), it can be deleted. For this the entry just has to be set to "invalid" in the page table. This happens in the following steps:

1. MMU has to switch to System Mode.
2. First the data for the entry is needed. This consists of:
   (a) Virtual Page Number
   (b) Context ID
3. Check the context ID to see if the permission for deleting is given
4. Setting the entry to invalid
5. The MMU switches back to User Mode.

#### Context Switch

Context Switch means, that a new application has been selected and the system has to switch from one virtual memory to another. Every application has its own lookup table in the memory which replaces the current table in the MMU. The different lookup tables are choosen by the context ID. This happens in two step:

1. All entries in the lookup table of the MMU has to be set to invalid
2. The new context ID has to be set in the MMU. Otherwise every memory access produces a page fault interrupt because the wrong access rights are given.

#### Switching between System and User Mode

Swichting between System and User Mode is very important. This is especially needed when there is a context switch (Section 3.5.1). In this case the MMU has to be reconfigured. This is only allowed in system mode. To set the MMU into this mode, an activation code has to be sent over the AHB configuration interface. The same has to be done for deactivating it. There are two cases:

- Switching from User to System Mode
- Swichting form System to User Mode

Both cases are pretty similar to each other. As you can see in Figure 3.2, there is a register which is called "Mode". This register has a boolean value and means "false" for System Mode and "true" for User Mode. For switching User to System Mode the following has to be done:

1. The code is received over the AHB configuration interface.
2. If the code is valid, the logic in the configuration part of the MMU sets the register.

The activation code is "4F 4E" and "4F 46 46" for deactivation, which means, when converted to ASCII, "ON" and "OFF".

### 3.5.2 Use Cases of Operating System

Our system is on a microcontroller, that means we have a lightweight version of an OS. We do not have any system calls or security rings. This is shown in Figure 3.6. Here the behavior of this OS is described. We have six different use cases which are handled:

- Memory Access
- Context Switch

- Page Fault
- Allocate Memory
- Free Memory
- Switching between System and User Mode

Except of the use case Page Fault, each of the above mentioned cases describes the MMU use cases from the view of the OS.

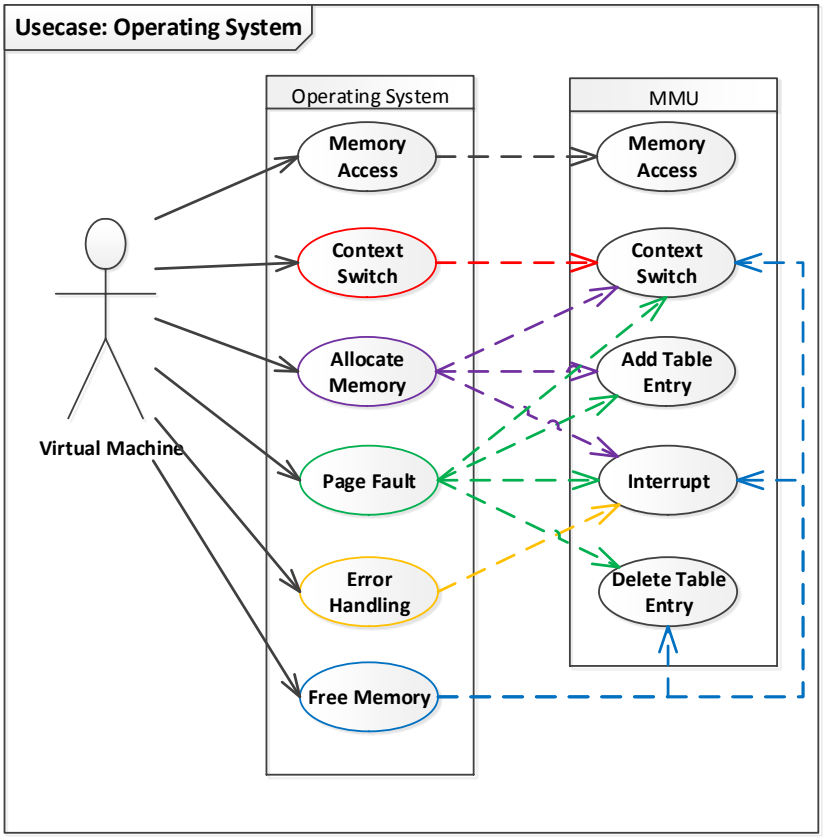


Figure 3.6: Use cases of the OS

### Memory Access

Here the memory access from the OS is described. This is needed, when an application needs to access to the memory for different reasons e.g. loading variables.

1. An application needs to access memory.
2. Now the memory access is handled from the MMU (see Section 3.5.1)

### Context Switch

For some possible reason (e.g. an interrupt), the context has to be switched. The OS has to do the following steps to prevent errors:

1. The lookup table for the new context has to be located in the memory.
2. Switching the MMU to System Mode (described in Section 3.5.2).
3. Load the new lookup table into the MMU.
4. Load Context ID into MMU.
5. Switch the MMU back to User Mode (Section 3.5.2)

### Page Fault

A page fault only occurs during a memory access and it is given by the MMU. This is described in Section 3.5.1. For this one of this reasons is given:

1. An application tries a write access on the code section.
2. An application tries to access a physical address. This type of access is only allowed in system mode.
3. An application tries to access a virtual address. When the MMU has no entry in the lookup table there could be two reasons for that:
   (a) The page does not exist and so it can not be loaded into the memory.
   (b) The page has not been loaded into the memory. In this case the OS load the page and modifies the lookup table in the MMU and in the persistent memory.

If the page is not loaded yet, the OS has to load the page into the memory and modify the page table in the MMU. In every other case an exception with an error code is given.

### Allocate Memory

Additional memory is requested in the OS in case an application needs more resources. Then the following steps are made:

1. The OS is switching to System Mode like in Section 3.5.1.

2. The available memory is searched for a free space and the OS reserves it.
3. The physical address of this page has to be stored in the persistent memory and also in the lookup table of the MMU.
4. The OS switches back to user mode.

The main problem here is to find a memory space with the needed size. When there are memory gaps which are to small, they can not be used. At a certain point the OS must execute a "Free Memory" like in Section 3.5.2 to get new memory.

#### Free Memory

This case is important, when a page is not longer needed by an application. For this the lookup table in the MMU and in the persistent memory has to be changed. This is easy because only the valid byte has to be deleted. Additional to be sure that no error occurs, this page can be deleted. But this is not necessary. These changes are made in system mode.

#### Switching between System and User Mode

Switching between this two modes is important because with that only the OS is able to configure the MMU. Four different situations are possible:

- Switching from User to System Mode
  The activation code is sent over the AHB interface. Then it proceeds like described in Section 3.5.1.
- Switching from System to User Mode
  The same like before. In this case the deactivation code is sent and then it continues processing as described in Section 3.5.1.
- Switching from User to User Mode
  This type of use case occurs, when there is a switch from one context to another. First there is a switch User to System Mode. After that the MMU can be configured. In the end it has to be switched back to User Mode.
- Startup process from the OS
  In the startup process the MMU is set to System Mode. This is done to prevent page fault interrupts during startup. After the process finished, the MMU is set from System to User Mode.

## 3.6 Future Changes of given Architecture

The used JCVM, developed by Michael Lafer [20], has to be modified. Without that, the software is not able to work with the new hardware. In Figure 3.7, the software architecture is shown. The red marked squares are the parts which has to be changed or created:

- In the Hardware Abstraction Layer (HAL) Interface it is necessary to create a Advanced RISC Machines support. This is needed because the developed test system is running on such a architecture. Without that communication between JCVM and hardware would not be possible.
- In the Operating System layer, there is the memory manager. The existing one must be updated, to make it possible to use the virtual address management. This includes:
  - Virtual addressing from applications
  - Management of providing free memory
  - Management of context tables entries
  - Context switch
  - Changing between System and User Mode
- Changing of the Object Manager in the Virtual Machine layer. The change here is needed because in Java Card Objects are stored in the persistent memory. This memory is protected with the new developed Memory Management Unit.
- The last change is in the Virtual Machine layer as well. It is the bytecode interpreter. Here a function can be called by a index in a static array. Every single of these functions exits with a specific code. Each code it possible to indicate if an error occured or if it was executed correctly. Here some platform support changes have to be made.

This are the parts of the given software architecture which are sure have to be changed. But it is also possible, that other modules has to be updated during the developing process.

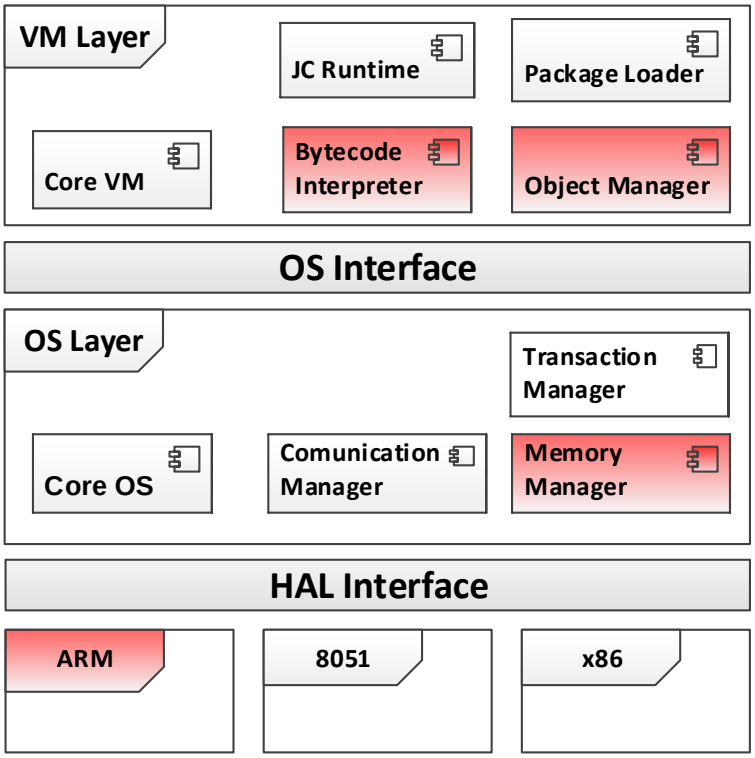


Figure 3.7: Changes in software architecture

# Chapter 4

# Implementation

This chapter shows everything about the implementation process. At first used hardware and software tools are shown. The next step is to show the implementation in detail. At the end, there is shown how to set-up the tools, to simulate the hardware again.

## 4.1 Development Environment

### 4.1.1 Field Programmable Gate Array Board

For this thesis a ARM ®Cortex ™-M1-enabled ProASIC ®3L was used. This board is shown in Figure 4.1. With this board it is possible to simulate the implemented hardware in our test system. The following features of this board were used:

- Oscillator for providing system clock
- The LEDs were used as a visual sign, that the implemented system has been synthesized. This was possible because of a test software which was stored in the flash. So after startup of the system, the LEDs started to blink.
- USB Connector providing a RS232 interface for programming the FPGA.
- USB Connection is used for debugging. This was used for example to search for errors while changing the Java Card Virtual Machine.
- The switches were used as General-purpose input/output (GPIO). With the switch number nine, it is possible to change between normal and debugging mode.

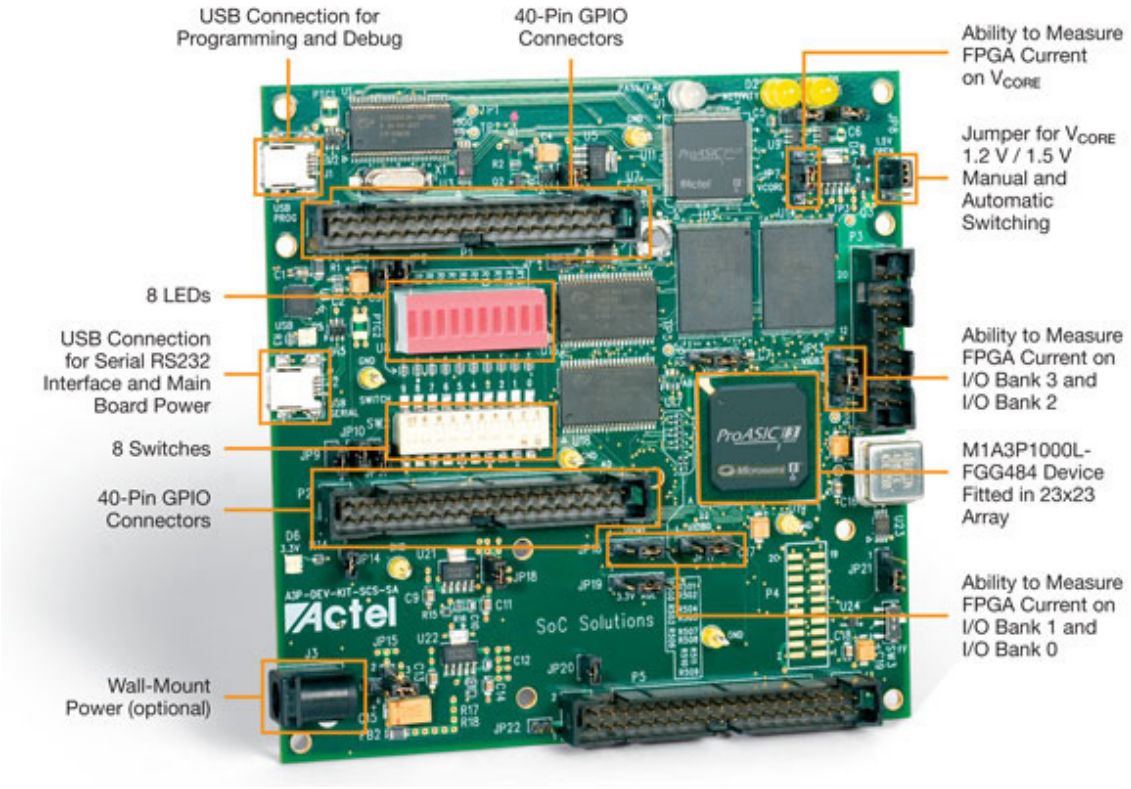


Figure 4.1: FPGA board [5]

### 4.1.2 Software Environment

#### Hardware Implementation

The used software for hardware implementation was Libero SoC (version 11.4 SP1) from Microsemi [1] . The reason for using this software was because the FPGA board is also from Microsemi and was delivered with a license license for the Libero SoC software. Furthermore, kickstarting into development with the Libero SoC software was easy, due to numerous available demo software for this particular FPGA. This included a running test system implemented in verilog and a few software programs for using that system in combination with the computer (using the RS232 interface). These programs were executed in debug mode of the board. Also an additional memory loader was available, which was used to program the flash memory of the FPGA.

[1] `http://www.microsemi.com/products/fpga-soc/design-resources/design-software/libero-soc` for downloading and free licensing for 1 year

### Hardware Simulation

For simulation of the implemented hardware the program Modelsim Microsemi [2] (version 10.3a) was used. The software was running in combination with Libero. Modelsim is a widely used tool for hardware description language simulation like verilog and Very High Speed Integrated Circuit Hardware Description Language (VHDL) simulation. To run the simulation more efficiently, own configuration files were created. So the used modules were first loaded and debugged. After that, the simulation with the given constraints were started.

### Software Implementation

For implementing the software, the used program was the Eclipse IDE with a special development kit. This tool is named Microsemi SoftConsole IDE [3] (version 3.4.0.5). This development kit is necessary to guarantee the communication between the computer software and the USB ports of the FPGA board. The most important "feature" is the possibility to use the debug mode. In debug mode it is possible to start the software on the written hardware while even debugging it. This was important for testing the system with the JCVM.

## 4.2 Implementation of the System in VHDL

The test system was used to familiarize with the board and development tools, but there is a test system available for Libero. This system was used to get known with the board and the development tools. But there were two issues with this demo model:

1. The test system from Libero is not up to date. The first step was to update all IP-Cores to a current version. IP stands for intellectual property core, which describes a functional block of a chip design. To prevent future errors, all cores were put to the current version offered by Microsemi.

2. After updating and testing this system, the testing tools were set up. This was working without any problems. Then the first parts of the MMU were implemented. The occurred problem was that Modelsim is not able to simulate a model which contains verilog and VHDL files. So the whole system was build again from scratch. After that every file was written in VHDL and also the simulation of the system was working with Modelsim.

As seen in Figure 4.2 the implemented system is pretty similar to the designed system in Section 3.1. In Table 4.1 the used IP-cores are listed. The only difference

[2] this is downloaded and installed together with Libero SoC

[3] `http://www.microsemi.com/products/fpga-soc/design-resources/design-software/softconsole` for downloading

between the designed and implemented systems are the watchdogs. They have not been used in the implementation and thats why the core was deleted. Instead of that, it was necessary to have the possibility to signal an interrupt and start with a interrupt service routine. That is why the interrupt core is used. There are two different kinds of interrupts used. The normal interrupt request like timer or UART and a fast interrupt request like it is used from the MMU. In Table A.1 are more details about version number and class of the used IP-cores available.

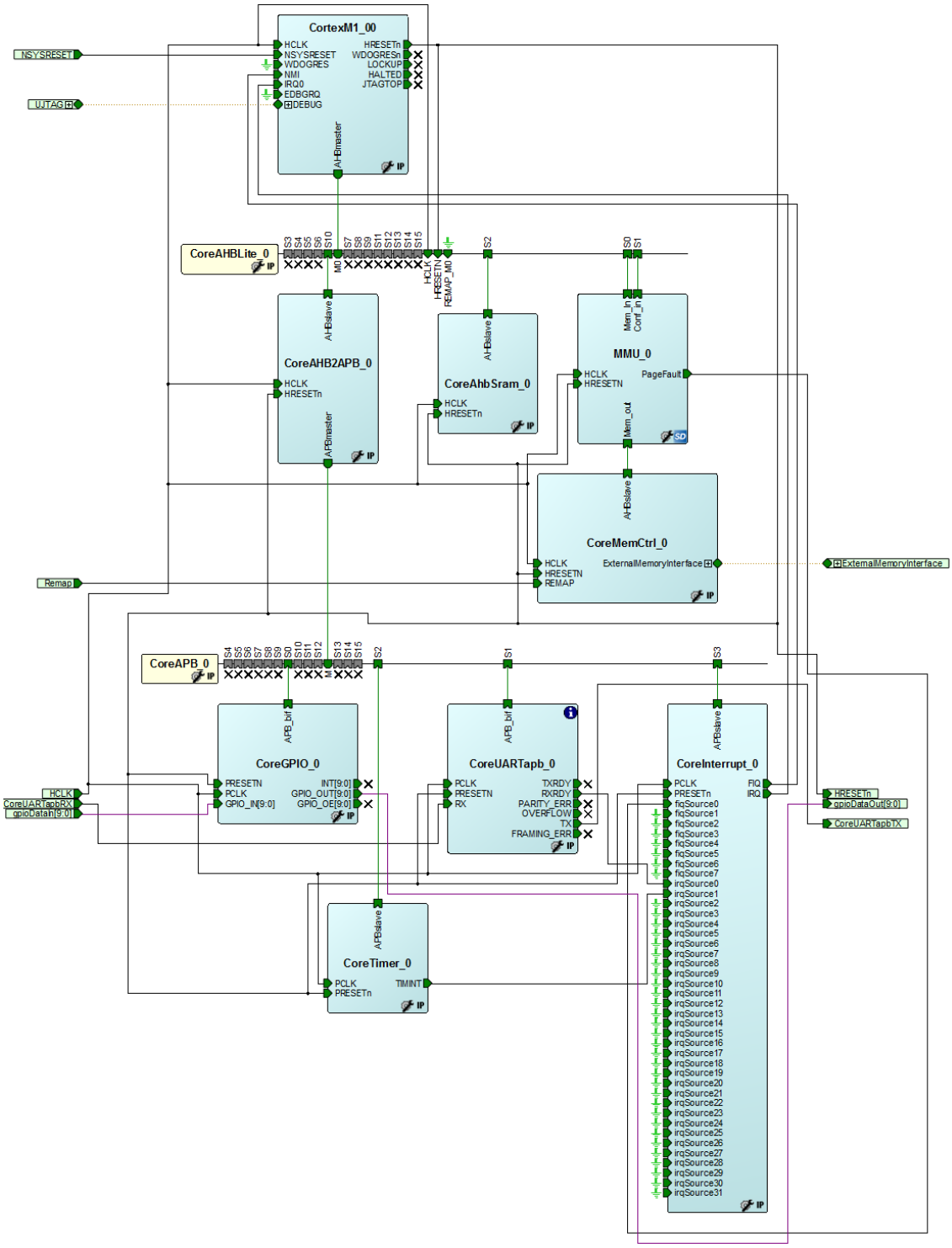


Figure 4.2: VHDL Test System

| IP-Core | Description |
|---|---|
| CortexM1Top | This is the Cortex M1CPU of the system. |
| CoreAHBLite | The AHB BUS controller. |
| CoreMemCtrl | Memory controller for writing into RAM or Flash. |
| CoreAhbSram | A SRAM which is directly accessable. |
| CoreAHB2APB | Bridge between AHB and APB BUS. |
| CoreAPB | The APB BUS controller. |
| CoreTimer | 16 or 32 Bit timer which generates a interrupt. |
| CoreUARTapb | A standard UART interface for using the USB port of the board. Can produce a interrupt. |
| CoreGPIO | GPIO for using the I/O pins from the board. |
| CoreInterrupt | Interrupt Controller which sends a flag to CPU if a interrupt occurs. |

Table 4.1: Used IP-Cores for implemented system

## 4.3 Memory Management Unit

In this chapter the Memory Management Unit is described with all parts. In Figure 4.3 the internal parts are shown. It consists of:

- Two AHB slave interface
- Configuration logic block
- Memory logic block
- One mirrored AHB slave interface
- Dual port memory used as TLB
- D-Flip-Flop
- Two logical gates

### 4.3.1 The Advanced High-Performance BUS slave interface

To enable communication between CPU and MMU two AMBA AHB slave interfaces are needed. Two because there is a configuration and a memory path. For full functionality this part acts like a real AHB slave. The configuration path is used for:

- Changing between User and System Mode
- Reseting the TLB

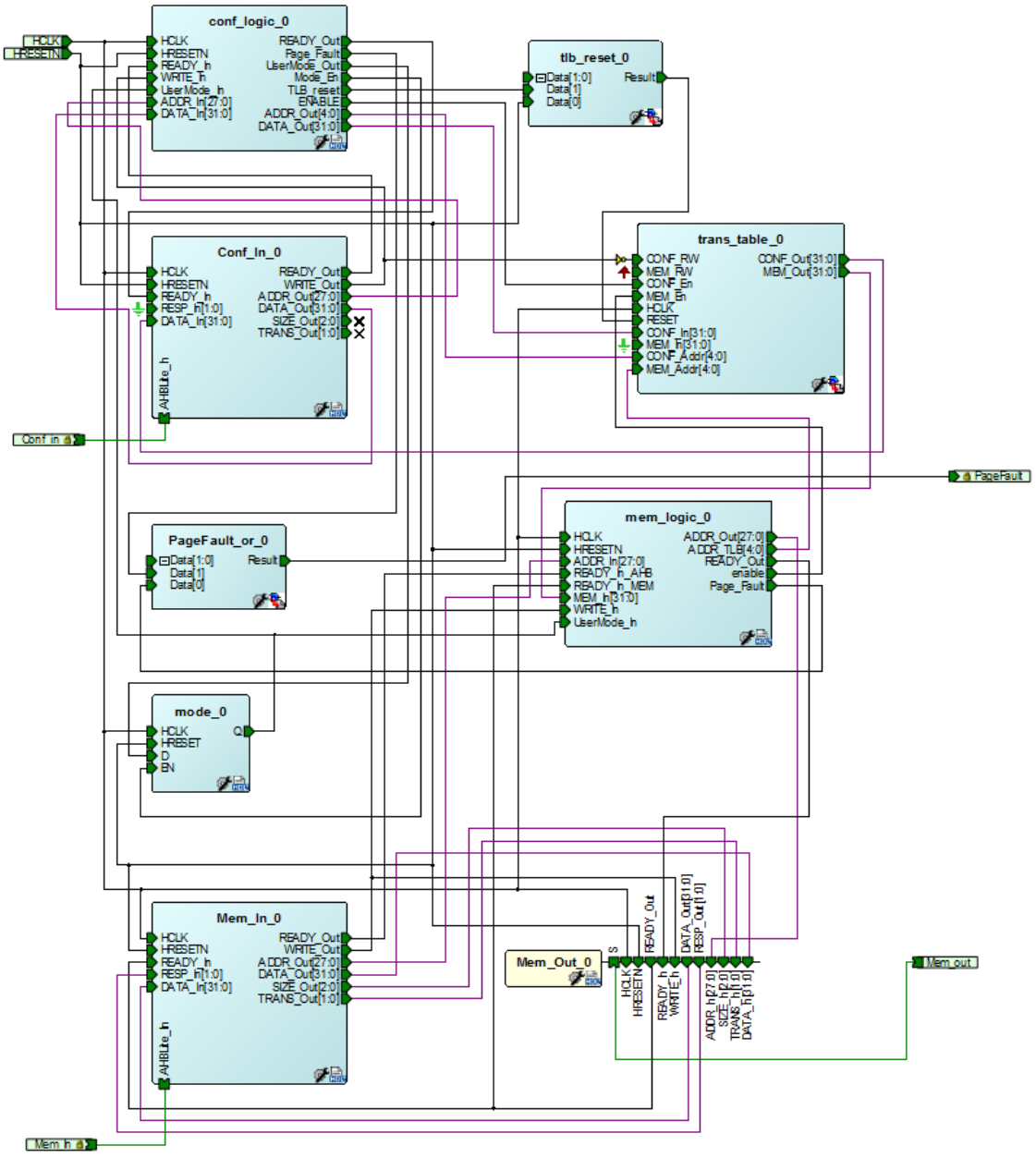


Figure 4.3: VHDL Memory Management Unit

- Managing TLB entries

The memory path is less complex than the configuration path. The only thing it has to do is changing the virtual address to the physical address and write this to the memory controller.

**In and Out Ports**

In Figure 4.4 the picture of all inputs and outputs are shown and the functionality is described in Table 4.2. Every signal starting with "H" comes from the AMBA AHB. The other signals are internal signals of the MMU.

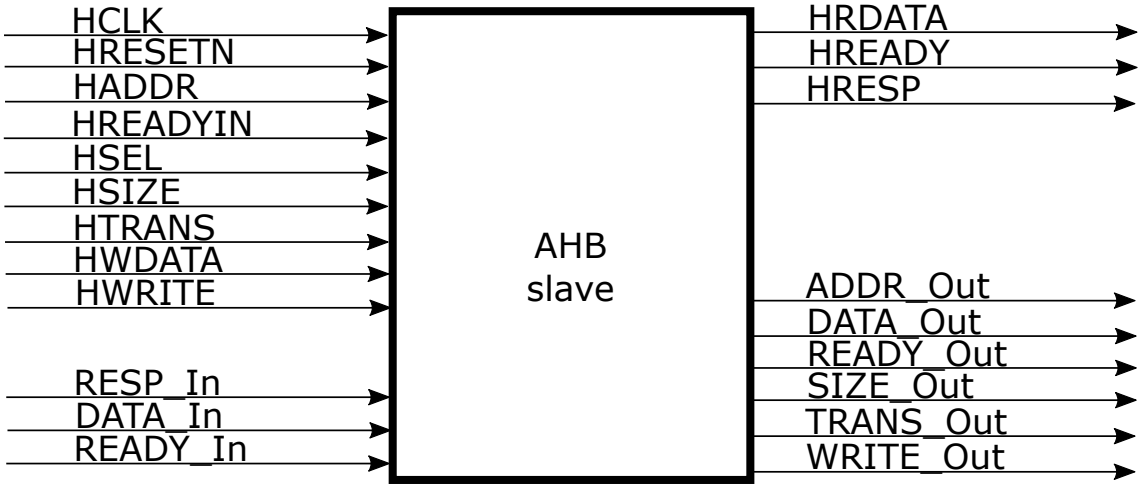


Figure 4.4: Ports of AHB slave interface

**Functionality**

In Figure 4.5 the state diagram of the AHB slave interface is shown. After booting or a reseting, the interface is set to IDLE state and as long as HSEL or HREADYIN are logically "0", it will remain in this state.
Once selecting this device with HSEL = 1, it is listening to the HREADY signal. When this signal turns "1", the transfer can start. Now it depends if it is a read or write transfer. When HWRITE = 1, the next state is WRITING. This is needed because the data which is written arrives one clock cycle after the begin of the transfer.
After that, or when it is a read transfer, the device changes to the state BUSY. As long as the READY_In signal is LOW, the interface will remain in this state. When it changes to HIGH, the interface goes back into IDLE state.

### 4.3.2 The configuration logic

The configuration logic is part of the MMU which manages access to the TLB over the AHB interface. The following functionality is provided:

- Switching between User and System Mode

| Name | In / Out | Description |
|---|---|---|
| HCLK | In | The global clock signal. Related to the rising edge. |
| HRESETN | In | The global reset signal. Managed by CPU. Every part is LOW active. |
| HADDR | In | The 32-bit address bus signal. Top 4-bits are used for decoding. |
| HREADYIN | In | This signal is sent from AHB master to slaves to signal with HIGH that the BUS is free and LOW for occupied. |
| HSEL | In | Select signal for showing a AHB slave that the current transfer is for it. Must be combined with HREADYIN. |
| HSIZE | In | 3-bit signal which indicates the size of the actual transfer. |
| HTRANS | In | 2-bit signal which indicates the transfer type |
| HWDATA | In | Data BUS with 32-bit width for writing data. |
| HWRITE | In | Signal for write (HIGH) or read (LOW). |
| HRDATA | Out | Data BUS with 32-bit width for reading data. |
| HREADY | Out | This signal is sent from AHB slave to show the master that a transfer has been finished. |
| HRESP | Out | Signal for giving a response if a transfer succeeded (LOW) or had an error (HIGH) |
| RESP_In | In | Internal signal for forwarding to HRESP. |
| DATA_In | In | Internal signal for forwarding to HRDATA. |
| READY_In | In | Internal signal for forwarding to HREADY. |
| ADDR_Out | Out | Forwarding HADDR for internal use. |
| DATA_Out | Out | Forwarding HWDATA for internal use. |
| READY_Out | Out | Ready signal which goes from HIGH to LOW when all data has arrived. |
| SIZE_Out | Out | Forwarding HSIZE for internal use. |
| TRANS_Out | Out | Forwarding HTRANS for internal use. |
| WRITE_Out | Out | Internal write signal, similar to HWRITE. |

Table 4.2: Description of AHB slave interface ports [1]

- Reset of TLB
- Add / Remove entries
- Read entries
- Throwing a page fault on wrong access try

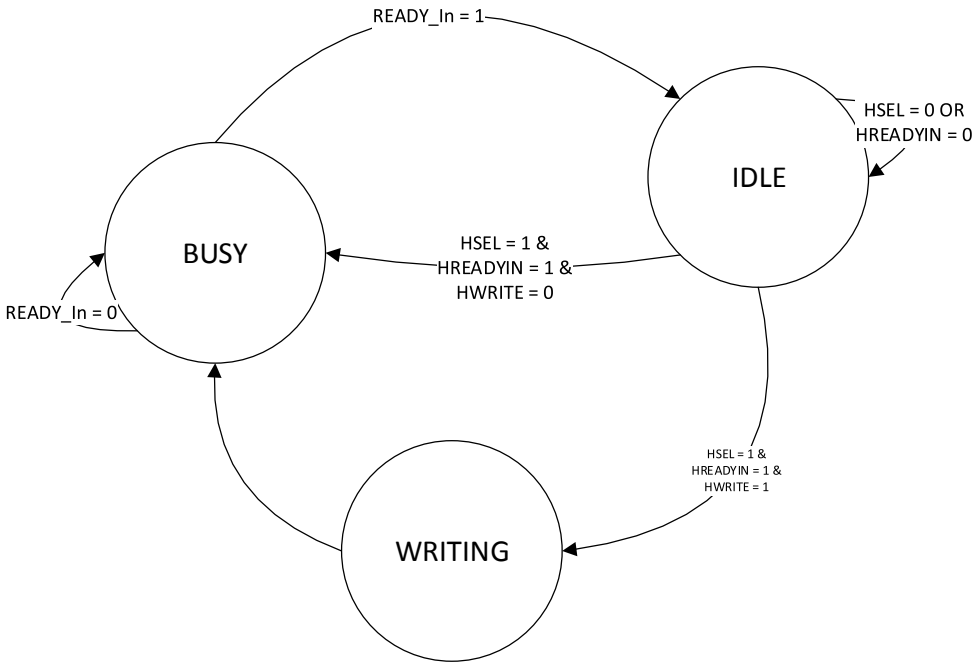


Figure 4.5: State Diagram of AHB slave interface

### In and Out Ports

In Figure 4.6 all in- and outputs are shown and the functionality is described in Table 4.3.

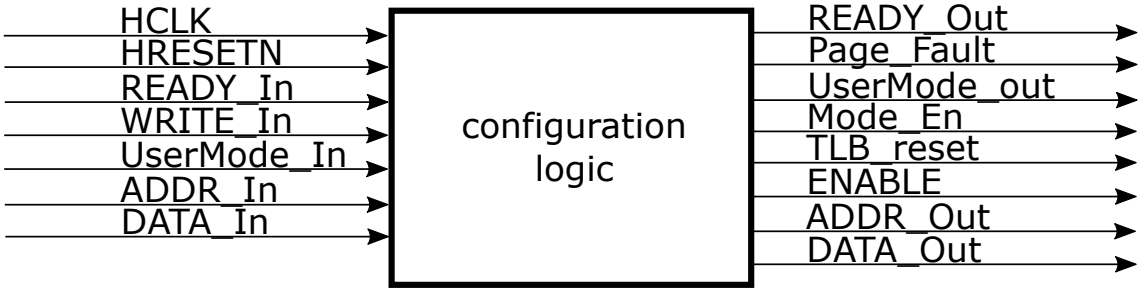


Figure 4.6: Ports of configuration logic

| Name | In / Out | Description |
|---|---|---|
| HCLK | In | The global clock signal. Related to the rising edge. |
| HRESETN | In | The global reset signal. Managed by CPU. Every part is LOW active. |
| READY_In | In | Sensitive on a falling edge of this signal to start processing with the input data. |
| WRITE_In | In | Shows if there is a read (LOW) or write (HIGH) access. A change in the TLB is always a write access. |
| UserMode_In | In | Shows the actual mode of the MMU. HIGH stands for user mode, LOW for system mode. |
| ADDR_In | In | Here the 27-bit address from the AHB slave interface is shown. |
| DATA_In | In | When writing into the TLB, the physical address and the control bits are sent over this signal. |
| READY_Out | Out | Ready signal which goes from HIGH to LOW when all data has arrived. |
| Page_Fault | Out | With an occurring error a page fault is signalized (HIGH signal). |
| UserMode_Out | Out | The wanted mode is sent out. LOW for system and HIGH for user mode. |
| Mode_En | Out | If the mode is changed, this signal has to be HIGH to enable the D-Flip-Flop were the value is saved. |
| TLB_RST | Out | If necessary, the TLB can be reset over this signal (LOW active). |
| ENABLE | Out | Enable signal for the TLB (HIGH active). |
| ADDR_Out | Out | 4-bit address for addressing the TLB. |
| DATA_Out | Out | 32-bit data which is a combination of virtual and physical address and the control bits. |

Table 4.3: Description of configuration logic ports

**Functionality**

In Figure 4.7 the state diagram of the configuration logic is shown. As long as READY_In = 1, the logic will remain in IDLE state. When READY_In = 0, four possibilities are remaining.
The first is to change the mode. If both (ADDR_In and DATA_In) have the value “4F 4E” or “4F 46 46” (in ASCII this means ON and OFF), the logic goes into the MODE state. The MODE_En signal goes HIGH and the desired mode is sent out over the UserMode_Out port.
Next possibility is the reset of the TLB. This is like changing of the mode and called TLB_RST. ADDR_In and DATA_In have to be “52 53 54” (means RST in ASCII).

The third possible state is the error check. Access to the TLB is only given in system mode and with the virtual address which is created, edited or deleted. Like told in Section 3.3 the virtual addresses are appropriated by the VIRT bit. So if this bit is not set, or the user mode is selected, the logic changes to the IRQ state, where the page fault signal is set by the MMU.
If no other state gets selected, the logic goes into the RW state. When there is a read access, the information is sent out. When a TLB entry is created or changed (deleting is changing the valid control bit), the information is parsed here and sent out. After that is changes to BUSY state to wait one clock cycle to write the data into the TLB. After that it returns to the IDLE state. TLB entries is parsed. This parsing is described in Section 3.4.

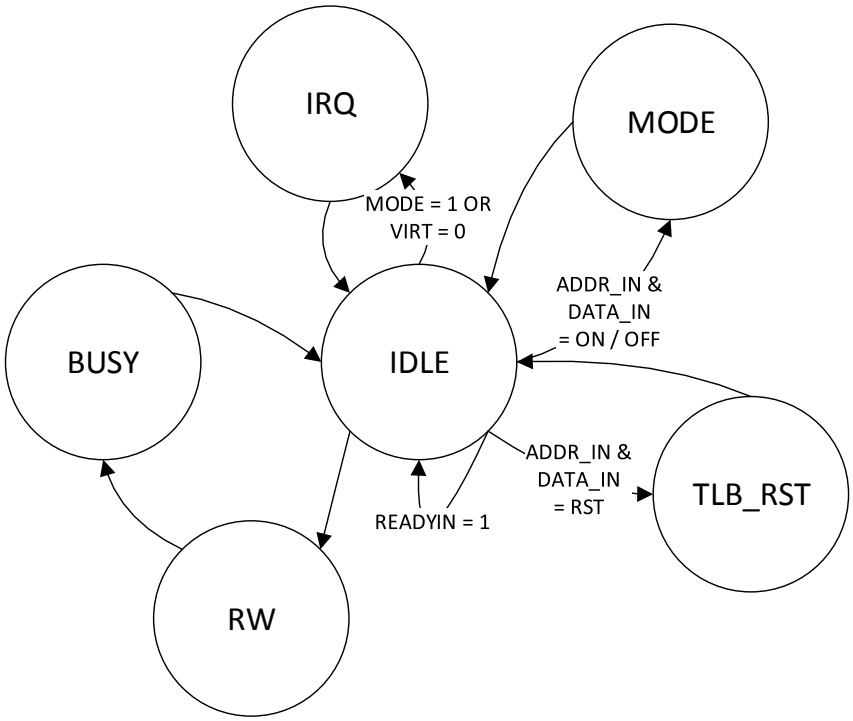


Figure 4.7: State Diagram of configuration logic

### 4.3.3 The memory logic

The memory logic is that part of the MMU which translates the virtual address into the physical address. This provides the following functionality:

- Direct forwarding of physical addresses in system mode

- Changing a virtual address into a physical address
- Throwing a page fault on wrong access attempts

### In and Out Ports

In Figure 4.8 all in- and outputs are shown, the functionality is described in Table 4.4.

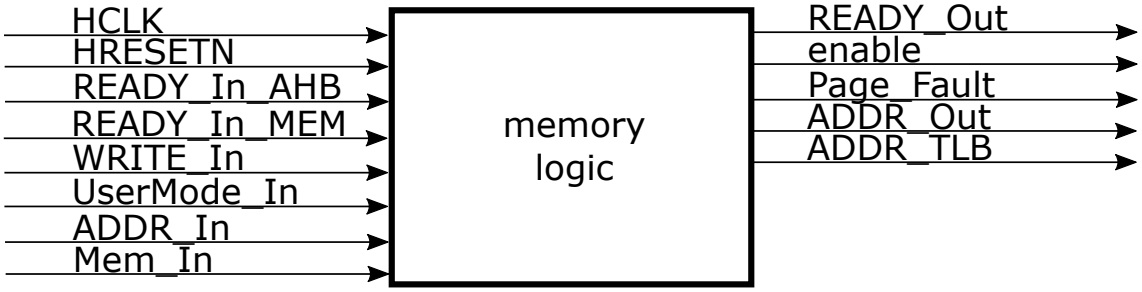


Figure 4.8: Ports of memory logic

### Functionality

In Figure 4.9 the state diagram of the memory logic is shown. Like in the other devices, it will stay in IDLE state as long as READY_In_AHB = 1. When READY_In_AHB changes to LOW, there are two possibilities.
The first one is to change to the IRQ state. This happens when the MMU is in user mode and the VIRT bit is not set. This means an application tries to access directly to a physical address.
If there is no forbidden access, the logic changes into the PREPARE state. Here the logic checks, if there is a virtual or a physical address. When the address is physical, the next state is the WAITING state. If not, the physical address has to be loaded from the TLB.
This happen in the next states, LOAD and LOADED. This states are to ensure that the data is available. After this two loading states, the logic is into the TRANSLATE state. Here the TLB entry gets checked with the control bits. First of all the VALID bit has to be set. The second thing is the read / write permission. If a write access is performed on a readable address, this is an invalid transfer. If one of these two cases occur, the next state is IRQ state again. Otherwise the logic switches to the WAITING state.
In the WAITING state the READY_Out bit goes to LOW to signalize, that a valid address is available. It will remain in this state until the READY_In_MEM will become HIGH. After that the logic return in IDLE state.

| Name | In / Out | Description |
|---|---|---|
| HCLK | In | The global clock signal. Related to the rising edge. |
| HRESETN | In | The global reset signal. Managed by CPU. Every part is LOW active. |
| READY_In_AHB | In | Sensitive on a falling edge of this signal to start processing with the input data. |
| READY_In_MEM | In | Sensitive on a rising edge of this signal to see that transfer has been finished. |
| WRITE_In | In | Shows if there is a read (LOW) or write (HIGH) access. |
| UserMode_In | In | Shows the actual mode of the MMU. HIGH stands for user mode, LOW for system mode. |
| ADDR_In | In | Here the 27-bit address from the AHB slave interface is shown. |
| Mem_In | In | The data which was read from the TLB is available here. |
| READY_Out | Out | Ready signal which goes from HIGH to LOW when transfer to memory can begin. |
| enable | Out | This is the enable signal for activating the TLB. |
| Page_Fault | Out | With an occurring error a page fault is signalized (HIGH signal). |
| ADDR_Out | Out | When READY_Out goes on LOW, the physical address is available here. |
| ADDR_TLB | Out | The address for accessing the TLB to gain the information about the requested address. |

Table 4.4: Description of memory logic ports

### 4.3.4 The mirrored AHB slave interface

To enable communication between MMU and the memory controller, a mirrored AMBA AHB slave interface is needed. For the full functionality, this part has to act like a real AHB master. This part is in the memory path. Like in the AHB slave interface, all ports with a "H" are the AMBA signals. All ports are similar to this interface from the signals, but the direction changed from in to out and vice versa.

#### In and Out Ports

In Figure 4.10 the picture of all inputs and outputs are shown and the functionality is described in Table 4.5. Every signal with has a "H" at the beginning comes from the AMBA AHB. The other signals are internal signals of the MMU.

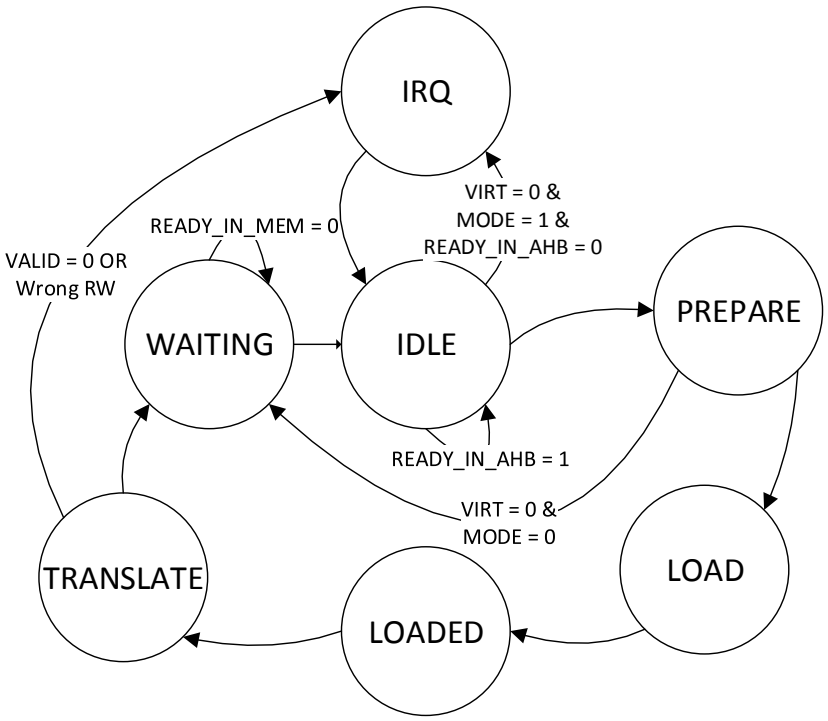


Figure 4.9: State Diagram of memory logic

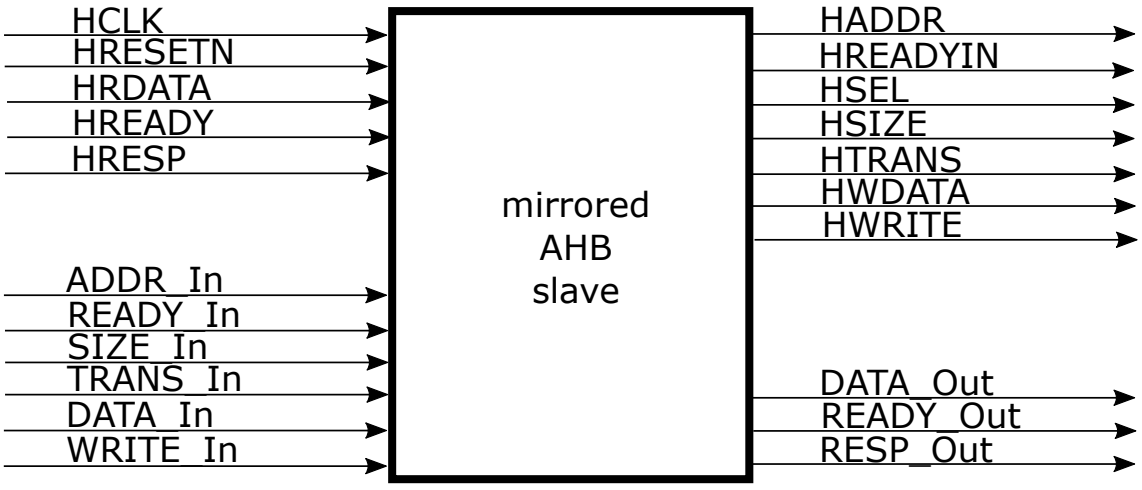


Figure 4.10: Ports of mirrored AHB slave interface

| Name | In / Out | Description |
|---|---|---|
| HCLK | In | The global clock signal. Related to the rising edge. |
| HRESETN | In | The global reset signal. Managed by CPU. Every part is LOW active. |
| HRDATA | In | Data BUS with 32-bit width for reading data. |
| HREADY | In | This signal is sent from AHB slave to show the master that a transfer has been finished. |
| HRESP | In | Signal for giving a response if a transfer succeeded (LOW) or had an error (HIGH) |
| HADDR | Out | The 32-bit address bus signal. Top 4-bits are used for decoding. |
| HREADYIN | Out | This signal is sent from AHB master to slaves to signal with HIGH that the BUS is free and LOW for occupied. |
| HSEL | Out | Select signal for showing a AHB slave that the current transfer is for it. Must be combined with HREADYIN. |
| HSIZE | Out | 3-bit signal which indicates the size of the actual transfer. |
| HTRANS | Out | 2-bit signal which indicates the transfer type |
| HWDATA | Out | Data BUS with 32-bit width for writing data. |
| HWRITE | Out | Signal for write (HIGH) or read (LOW). |
| ADDR_In | In | The physical address which get forwared to HADDR. |
| READY_In | In | The interface is sensitive to this signal. When it is LOW, the transfer process starts. |
| SIZE_In | In | Internal signal, forwarded to HSIZE. |
| TRANS_In | In | Internal signal, forwarded to HTRANS. |
| DATA_In | In | Data BUS which is forwarded to HWDATA. |
| WRITE_In | In | Internal signal, forwarded to HWRITE. |
| DATA_Out | Out | Forwarding HRDATA for internal use. |
| Ready_Out | Out | Forwarding HREADY for internal use. |
| RESP_Out | Out | Forwarding HRESP for internal use. |

Table 4.5: Description of mirrored AHB slave interface ports [1]

### Functionality

In Figure 4.11 the state diagram of the mirrored AHB slave interface is shown. After booting or a reset, the interface is set to IDLE state and as long as READY_In = 1, it will remain in this state.
With changing it to LOW, the transfer can begin. First the WRITE_In signal is checked. When it is HIGH, the next state is WRITING, if it is low, the next state is BUSY. Like for the normal AHB interface, this WRITING state is needed because

the data has to be at the data BUS one clock cycle later.
After that, the interface stays in the BUSY state until the HREADY signal gets HIGH. After that the transfer has been completed and the interface returns in IDLE state.

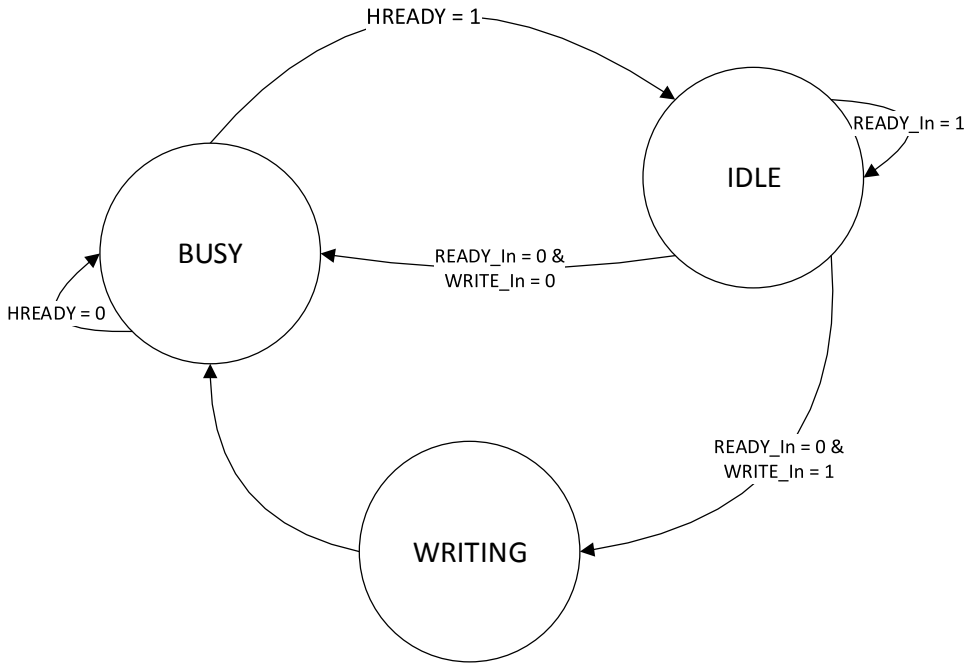


Figure 4.11: State Diagram of mirrored AHB slave interface

### 4.3.5 The dual port memory

The dual port memory is used as TLB. As illustrated in Figure 3.1, the TLB is located between the configuration and the memory path. And with this kind of memory, it is able to have read and write access from both sides, clocked by one single signal. Every entry has 32-bit width and the table contains 32 entries. It is even possible to define the width and entries for every port.

#### In and Out Ports

In Figure 4.12 the picture of all inputs and outputs are shown and the functionality is described in Table 4.6. This is the only part withing the MMU, which is sensitive on a falling clock edge.

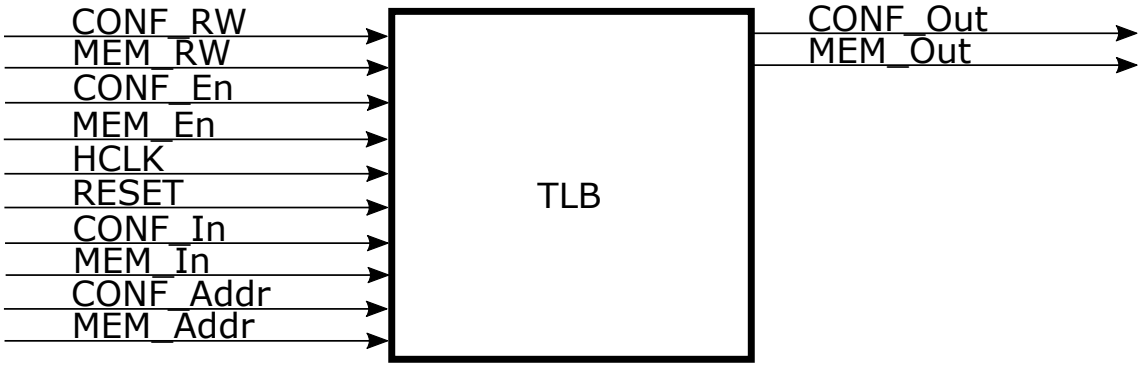


Figure 4.12: Ports of TLB

| Name | In / Out | Description |
|---|---|---|
| CONF_RW | In | Read / write signal of configuration side. Read is HIGH and write is LOW. That is inverted to the AHB protocol. So this port has a upstream inverter |
| MEM_RW | In | Read / write signal of memory side. Because there is no write access allowed from this side, it is constant high. |
| CONF_En | In | Enable signal for configuration side. Sensitive to a HIGH Signal. |
| MEM_En | In | Enable signal for memory side. Sensitive to a HIGH Signal. |
| HCLK | In | The global clock signal. Related to the falling edge. |
| RESET | In | The global reset signal. Managed by CPU. Every part is LOW active. |
| CONF_In | In | The data which is written into the TLB from the configuration side. |
| MEM_In | In | The data which is written into the TLB from the memory side. Because no write access is allowed on this side, it is constant LOW. |
| CONF_Addr | In | 4-bit address from configuration side. |
| MEM_Addr | In | 4-bit address from memory side. |
| CONF_Out | Out | Data BUS with 32-bit output for configuration side. |
| MEM_Out | Out | Data BUS with 32-bit output for memory side. |

Table 4.6: Description of TLB ports

### 4.3.6 The D-Flip-Flop

The D-Flip Flop is used to store if the MMU is in user mode or system mode. When the enable signal gets HIGH, the input is saved to the output at the next rising clock edge. When the enable signal is LOW, nothing happens to the output. Because at booting the VM has to do a lot of memory accesses, the Flip-Flop starts in system mode.

#### In and Out Ports

In Figure 4.13 the picture of all inputs and outputs are shown and the functionality is described in Table 4.7.

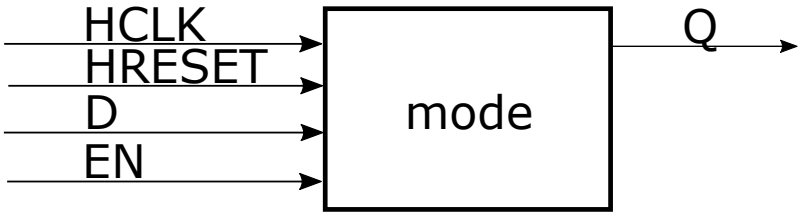


Figure 4.13: Ports of D-Flip-Flop

| Name | In / Out | Description |
|---|---|---|
| HCLK | In | The global clock signal. Related to the rising edge. |
| HRESET | In | The global reset signal. Managed by CPU. Every part is LOW active. |
| D | In | Input signal of the Flip-Flop. |
| EN | In | Enable signal for Flip-Flop. When HIGH, the input is saved. |
| Q | Out | Out signal of the Flip-Flop. |

Table 4.7: Description of D-Flip-Flop ports

### 4.3.7 The logic gates

This are the “easiest” devices of the MMU. It has two inputs and one output (described in Table 4.8. One is a logical “or”, which means if one input is is HIGH, the output is HIGH. The other one is a logical “and”, which means both inputs have to be HIGH, to get a HIGH at the output. The two logical gates are:

1. The first one is the logical "or" for the page fault signal. When there occurs an error in the memory logic or the configuration logic, the signal to the output port of the MMU is lead to the interrupt controller (seen in Figure 4.2). The "or" function was used because the signals are normally LOW.

2. The second one is the logical "and" for a TLB reset. There are two possibilities for a reset. First is system reset from the CPU, the second is a reset of the TLB from the configuration logic. Here the "and" function was used because the reset signal is normally on a HIGH level.

| Name | In / Out | Description |
|---|---|---|
| IN_1 | In | First input signal. |
| IN_2 | In | Second input signal. |
| OUT | Out | Output signal. |

Table 4.8: Description of logic gate ports

## 4.4 Test cases

To ensure that the hardware is working without any problems, some tests were made during the implementation. Only if the test finished without errors, the next part of the hardware was implemented. The described tests here are in chronological order as implemented. The results of these tests are shown in the Chapter 5.

### 4.4.1 Testbench for Advanced Microcontroller BUS Architecture model

First of all a testbench for testing our hardware was necessary. This testing environment should simulate the AMBA signals and include the memory controller, flash and SRAM. As second part the existing memory controller of the system must be able to communicate over the simulated AMBA interface. This include read an write access over the memory controller to the memory. This was also really important after implementing the VHDL model like described in Section 4.2. After this test were successful, it was possible to begin with the implementation of the MMU.

### 4.4.2 Testing of the Advanced High-Performance BUS slave interface

The next step was to establish the connection over the AHB interface. For this reason three parts were needed:

1. AHB slave interface
2. A reduced version of the configuration logic
3. Working TLB

With this parts it is possible to check the whole function of the AHB interface and the TLB in one step. The configuration logic has only one function: to ensure that the TLB is enable and disable at the right moment.
The testbench from the AMBA model (Section 4.4.1) was taken and modified for this test. The MMU component has been included in the test file and was connected to the AMBA simulator. Because of the reason, that the MMU has no AHB out interface at the moment, it was possible to ignore the memory controller. The test completed with the function of writing data into the TLB and also being able to read it after that.

### 4.4.3 Testing of the mirrored Advanced High-Performance BUS slave interface

The next step was implementing the mirrored AHB slave interface to connect the MMU between the AMBA controller and the memory controller. The functionality of the normal interface was granted because of the previous test.
In this test, the AHB interface of the memory path was connected directly to the mirrored AHB slave interface. After that, the memory controller had to be connected within the testing file. And because now there are already two AHB interfaces (for configuration and memory path) the right one had to be connected to the AMBA simulator, the other one had to be declared as "open" without function. This test was declared as successful after writing and reading through the MMU and the memory controller.

### 4.4.4 Testing the memory logic

Because the test files were prepared to test the memory logic, the decision was made to implement the memory logic. For simulation, it is possible to fill TLB entries with useful data. So some test entries were made to check the full function of this path. This included three cases:

- Behavior for normal function with translating of valid addresses
- Behavior for wrong access
- Behavior for accessing a invalid entry

With completing this test series, the memory path was completed and had fully functionality.

### 4.4.5 Testing the configuration logic

The last test drive completes the MMU functions. The testing files gets prepared like in Section 4.4.2. After that the configuration logic gets implemented. The following functions had been tested:

- Adding and changing of a TLB entry
- Generatin a page fault
- Changing of the MMU mode
- Reset of the TLB

This was the last needed test for the hardware. The full functionality of the Memory Management Unit is tested and available.

## 4.5 Setting up of the Test environment

For running the test, the software from Section 4.1 must be installed on the system. Also a working license file has to be installed [4]. The project files are put together into a .zip file. This has to be extracted into a chosen folder. After that Libero SoC has to be started.

In Figure 4.14 you can see, that in the working menu, the left enty "Project" → "Open Project" has to be chosen. Change to the folder where the .zip file was extracted. There choose the "CortexM1.prjx" file and click open (Figure 4.15).

In the next step on the right side the COREMEMCtrl has to be made the root file. This is made with right click on it and "Set As root". After that it looks like in Figure 4.17. If this file is already root, this option is not available anymore.

The next step is to start the simulation. For this a own test file has to be created like it is available in A.4. The project paths have to be changed to the current folders. After that the simulation with Modelsim can be started like shown in Figure 4.16. After loading Modelsim, just enter "do test.do" (where test has to be replaced with the real file name you created). The simulation will start. If you want to change the AMBA parameters, this can be done in the file "corememctrl_usertb.bfm" in the folder simulation. In Section A.5, there is the content of this file shown.

[4] `http://soc.microsemi.com/Portal/DPortal.aspx?v=24`


Figure 4.14: Libero SoC start screen

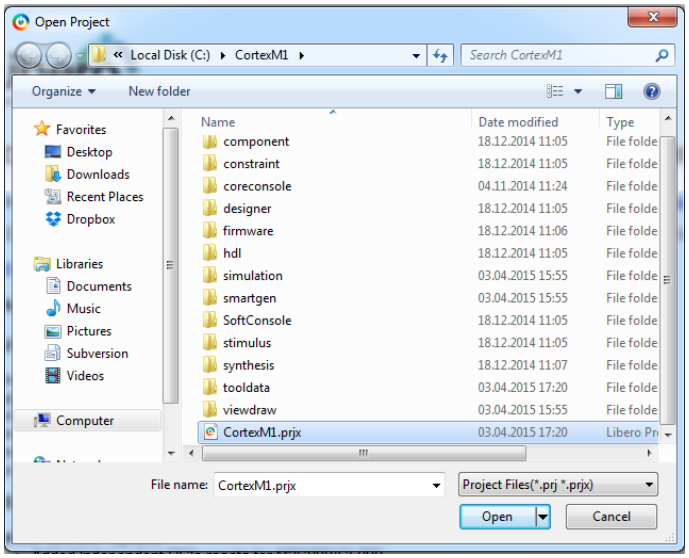


Figure 4.15: Choosing the project file

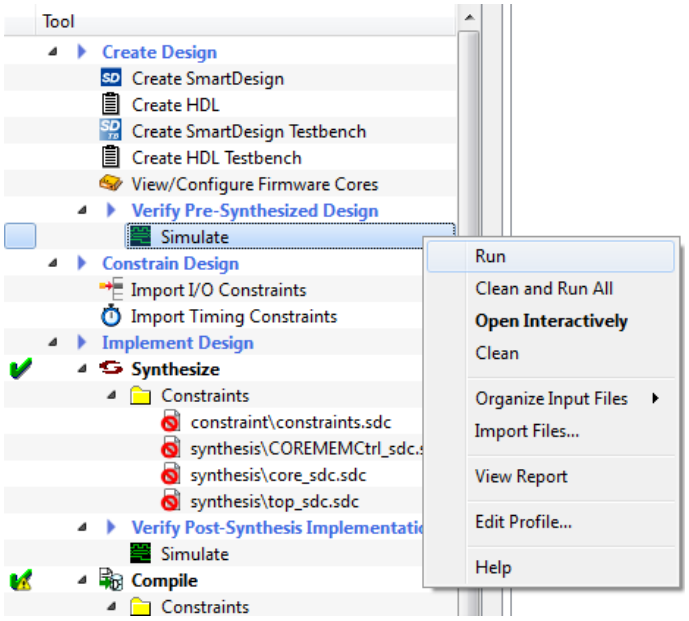


Figure 4.16: Starting the simulation with ModelSim

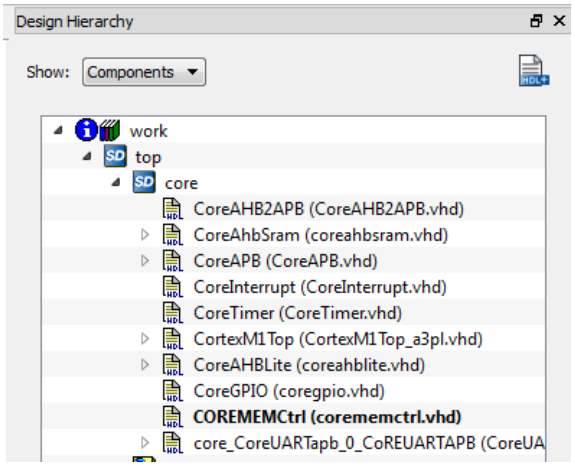


Figure 4.17: Choosing the root file in Libero SoC

# Chapter 5

# Results

This chapter contains the final results of the test cases, which were made during the implementation. We are just taking a look on the final hardware. First the test of the configuration path are shown. The second is the result of the memory path. At last there is a before and after speed analysis.

## 5.1 Configuration tests

In this section the results of the test cases of the configuration logic are shown. This contains the following cases:

- Write / Read access
- IRQ
  - VIRT not set
  - MMU in user mode
- Mode Changing
- TLB reset

### 5.1.1 Write access

Figure 5.1 shows a write access to the lookup table. Transfer begins at 0.43us, with the positive clock edge. The state signals are the current state of the devices. The first one is the AHB interface, the second one is the configuration logic.
As you can see, after the HREADY signal goes low, the Conf_In state (that is the AHB interface) changes to WRITING. The logic is changing into RW state. The signal CONF_In is the input interface for the lookup table. After successful write CONF_Out follows CONF_In.

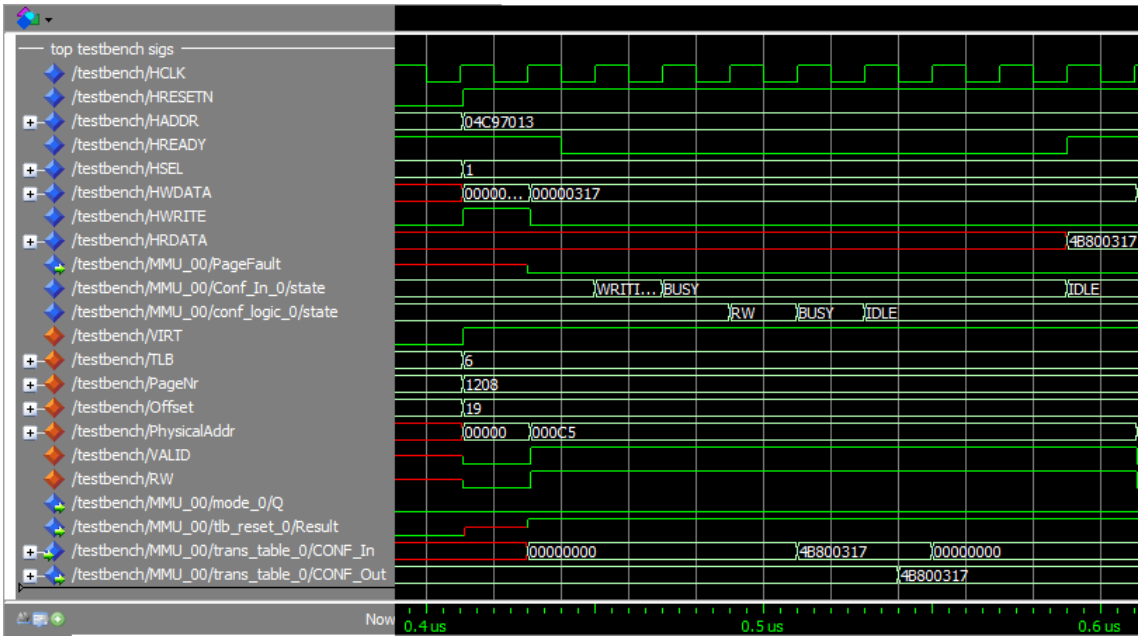


Figure 5.1: Write access of configuration logic

### 5.1.2 IRQ

In Figure 5.2 there are two test cases were a page fault occurs. Normally the OS would join an interrupt service routine. But because we are only testing the hardware, here only the signal changes from LOW to HIGH. This is illustrated in Figure 5.2 using the state named IRQ.
The upper transfer shows a page fault when an access is tried in user mode. The signal Q is the output of the mode register. It is HIGH, so this means user mode. The transfer is startet at 0.73us. The logic changes directly from IDLE to IRQ and the page fault signal rises.
In the lower transfer a page fault occurs, because an access is tried without having a virtual address. The VIRT signal shows that this is a physical address and even though the MMU is in system mode, the page fault occurs.

### 5.1.3 Mode Changing

Figure 5.3 show the change between user and system mode. Because the simulation starts in Boot-Mode, the signal Q (output from mode register) is LOW which means system mode. At 0.43us the signal for changing to user mode is sent. The mode changes to HIGH. The READY signal goes HIGH and signals the AMBA master that the transfer has ended. Directly after that, the code for changing to system mode is sent. The mode register changes back to LOW.

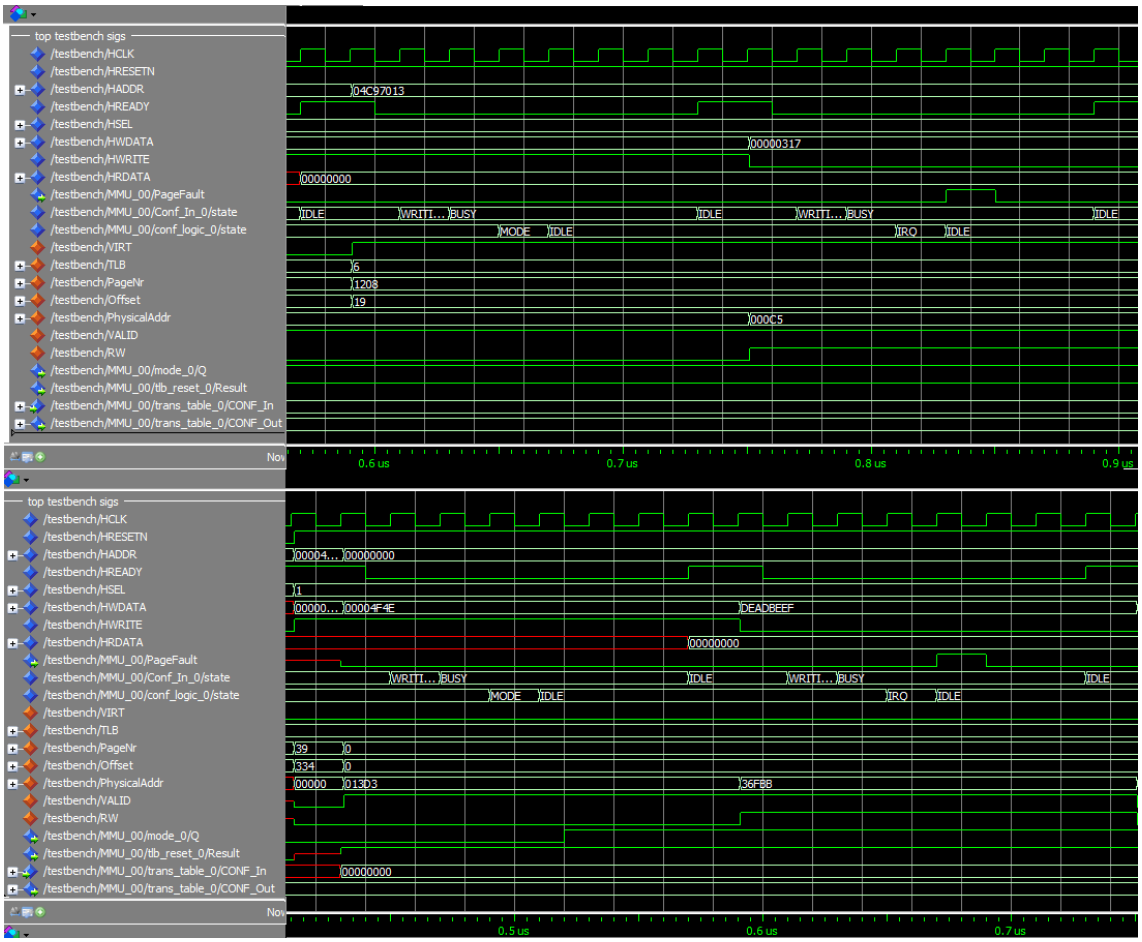


Figure 5.2: Occuring page fault from configuration logic

### 5.1.4 Reset

The last test in Figure 5.4 shows the TLB reset. At 0.43us the reset signal is sent. The configuration logic changes to state TLB_RST. The reset signal changes to LOW, all entries in the lookup table are deleted. After that the MMU returns into IDLE state.

## 5.2 Memory tests

In this section the results of the test cases of the memory logic are shown. This contains the following cases:

- Physical address access
- Virtual address access

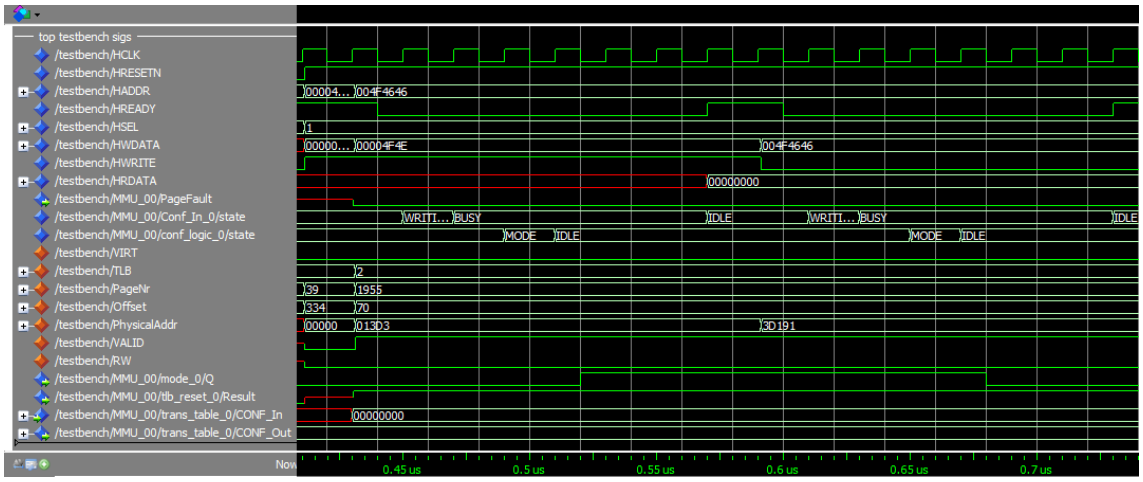


Figure 5.3: Changing between user and system mode

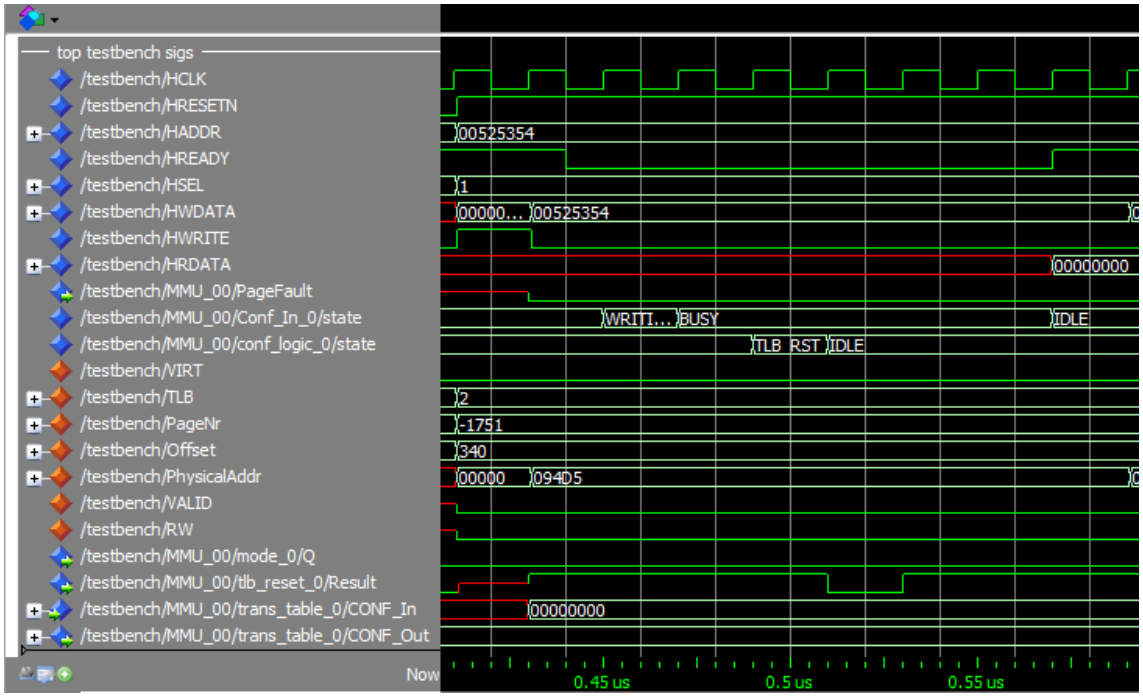


Figure 5.4: Reset of lookup table

- IRQ
  - Entry invalid
  - Wrong permission

### 5.2.1 Physical address access

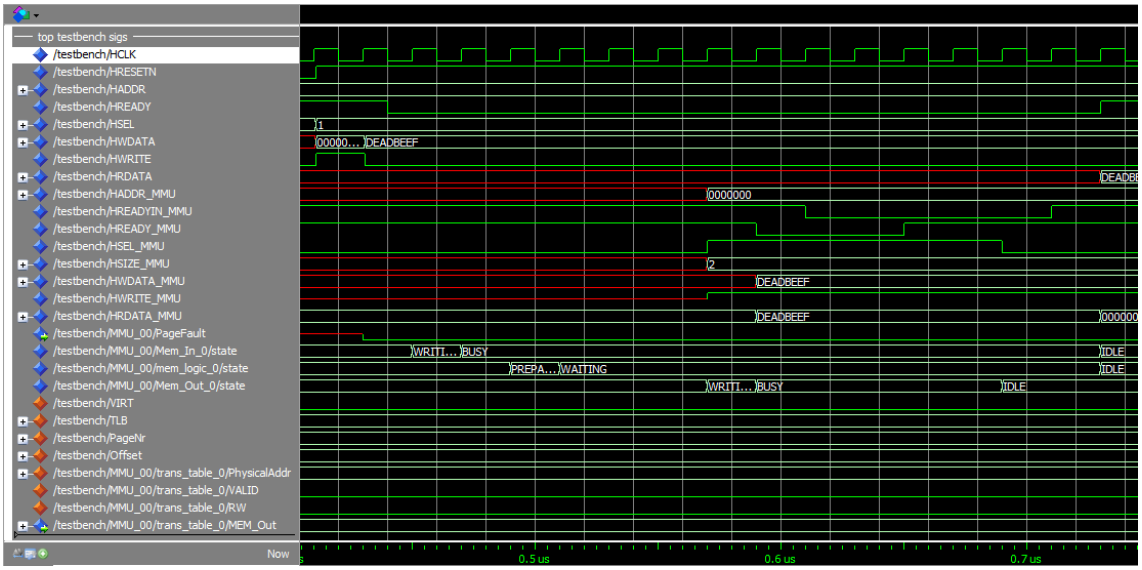


Figure 5.5: Memory access with physical address

In Figure 5.5 a memory access with a physical address is done. The information is sent over the AHB interface. The logic changes into the PREPARE state. Because the address is physical and the MMU is in system mode, the logic directly changes to the WAITING state. So the address is directly sent to the mirrored AHB interface. HADDR_MMU is the address BUS to the memory controller. After successful write, every device returns to IDLE state.

### 5.2.2 Virtual address access

In Figure 5.6 a memory access with a virtual address is done. This is seen at the VIRT signal. The information is sent over the AHB interface. The logic changes into the PREPARE state. Because the address is virtual and the MMU is in system mode, the logic changes to the LOAD state. In state LOADED the output of the lookup table (MEM_Out signal) changes to the saved table entry. This entry is parsed and the physical address is read and parsed. The physical address is 0 so it changes to this. After the logic changes to the WAITING state, the changed address is sent to the mirrored AHB interface. In HADDR_MMU it is possible to see that the address has been changed.

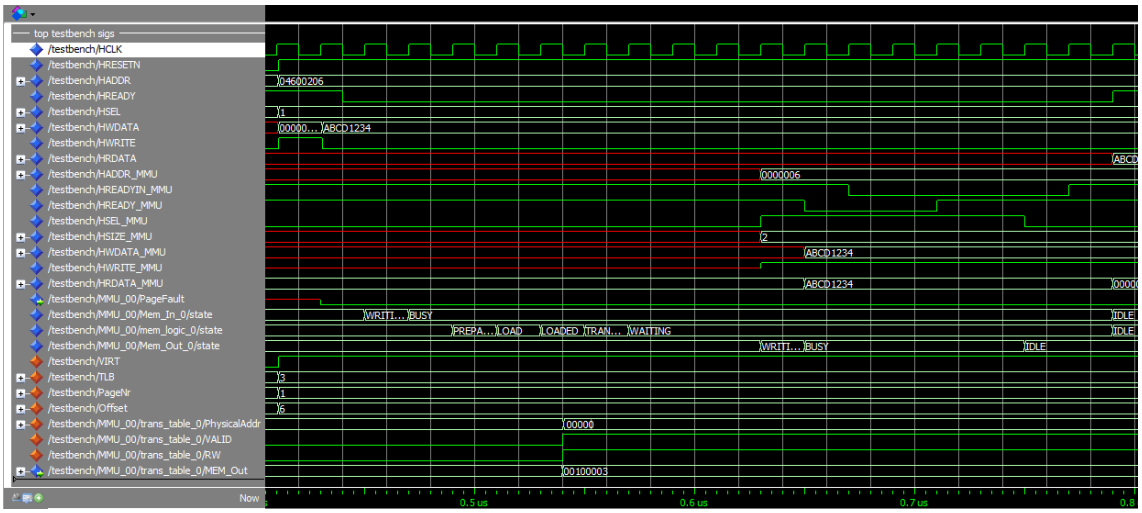


Figure 5.6: Memory access with virtual address

### 5.2.3 IRQ

#### Wrong permission

In Figure 5.7 memory access is shown. The upper transfer is a writing and the lower a reading access. Both to the same address. As you can see, the RW bit is LOW, this means that on this address only reading access is allowed. That is why in the upper picture a page fault occur. Because this is a simulation of the hardware, no interrupt service routine is called. In the lower transfer the reading access granted, because everything is alright with the permissions. The MMU allows read access from the memory address.

#### Invalid Entry

Figure 5.8 show a memory access to an address with an invalid lookup table entry. In the TRANSLATE state, the VIRT bit is checked. Because it is LOW, that means that the Entry is invalid, the next state is IRQ were the page fault signal changes to HIGH.

## 5.3 Speed Analysis

In Figure 5.9 a normal write access is made in the test system. The transfer starts at 0.43us and ends at 0.49us, this means that the access has a duration from 0.06us. Now compared with Figure 5.9 there can be seen. That the duration rises rapidly to 0.36us. This means one access is six times longer than before. This is the worst

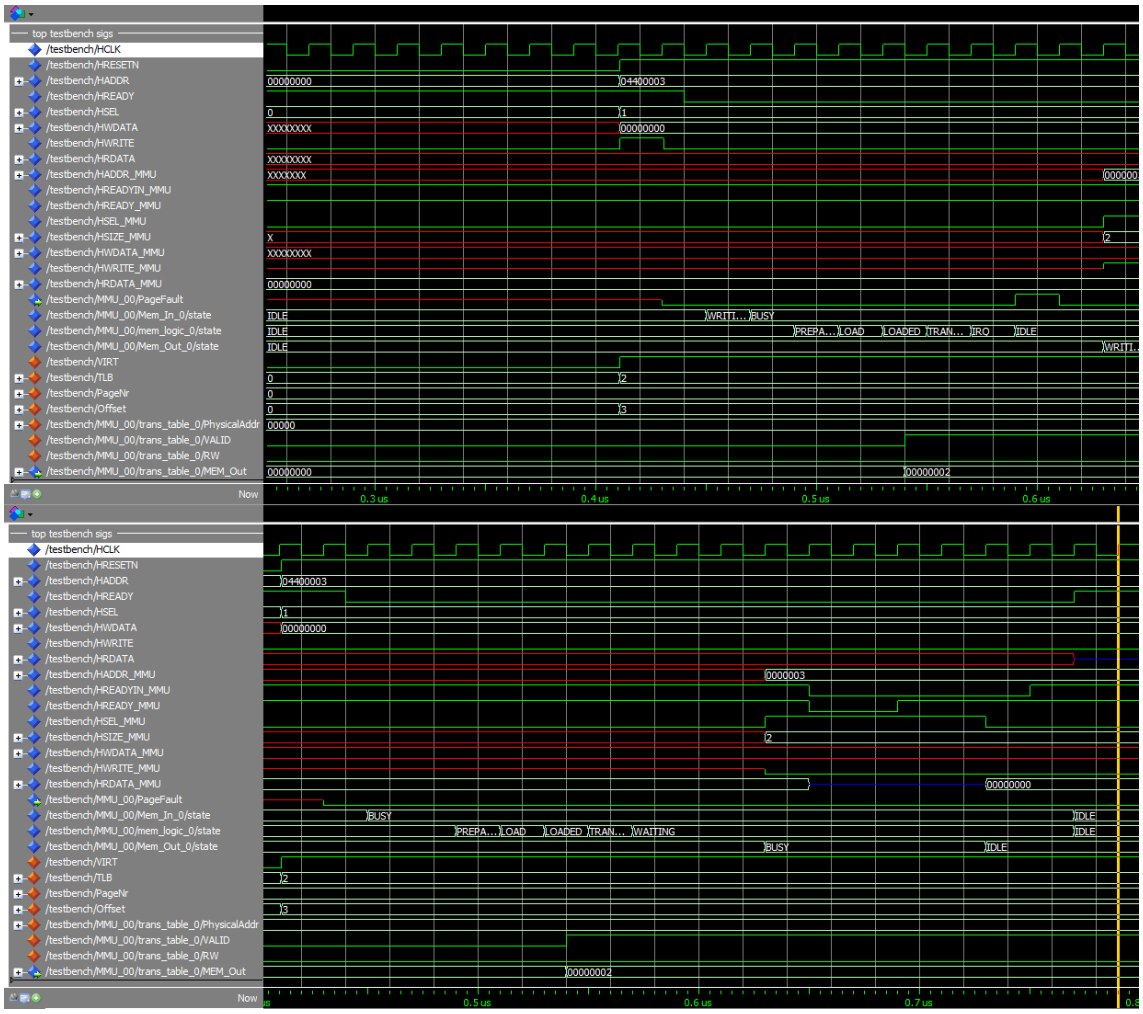


Figure 5.7: Memory access with permission errors

case for a reading access and occurs because of the devices which has to be passed through into the MMU.

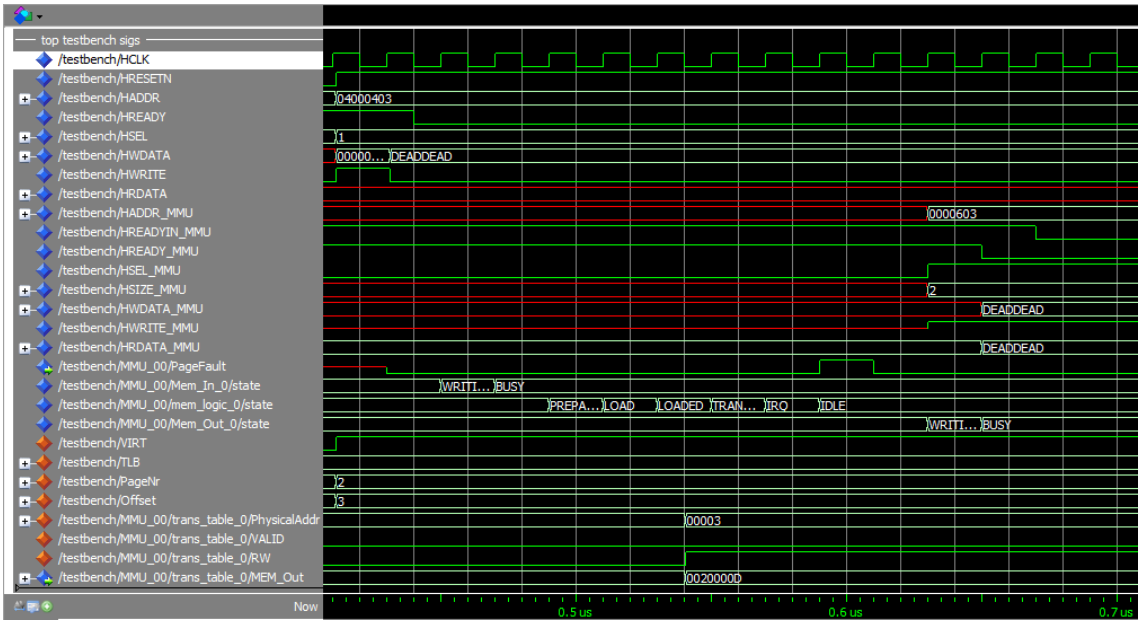


Figure 5.8: Memory access with invalid entry

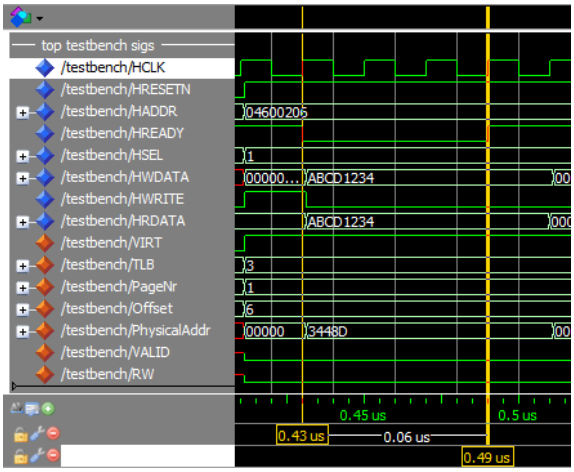


Figure 5.9: Transfer time without MMU

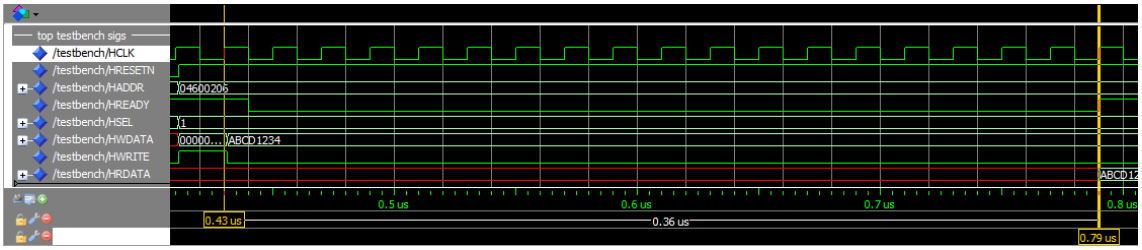


Figure 5.10: Transfer time with MMU

# Chapter 6

# Conclusion

This chapter includes a short conclusion about this project and what new important things were learned during working on this master thesis. Furthermore there is a short outlook about projects what can be worked on in further projects.

The goal of the project was to develop a hardware module to support the Java Card firewall. This goal was reached. Attacks which try to access another context than their own are blocked. Also the access to the system space is not possible from inside an applet context. Another success is that an attacker is not able to make reverse engineering because the only thing he is able to find are the virtual addresses which are managed by the Operating System.
The only negative aspect is that the performance of the systems gets lower. This is because more devices are included in one memory access. This loss of performance is justified with the win of security.

## 6.1 Outlook

Using this thesis as a base, the next step will be to adapt the Operating System to make it possible to use the new developed hardware. Another interesting point after adapting is to reduce the overload of the Memory Management Unit. By doing this, it would be possible to improve the speed performance of the system.
Furthermore another idea is to change the position of the MMU inside the used system. At the moment it is located between the AHB BUS and the memory controller. The repositioning between CPU and AHB will change the security level because with that it is possible to check every access which is made to every available memory. But in this case, the complete protocol of the MMU has to be changed as well.

# Appendix A

# Appendix

## A.1 Acronyms

**AHB** Advanced High-Performance BUS . . . . . 24
**AMBA** Advanced Microcontroller BUS Architecture . . . . . 24
**APB** Advanced Peripheral BUS . . . . . 24
**API** Application Programming Interface . . . . . 9
**ARM** Advanced RISC Machines . . . . . 20
**BUS** Binary Unit System . . . . . 24
**CAP** Composite Application Platform . . . . . 9
**CPU** Central Processing Unit . . . . . 23
**FPGA** Field Programmable Gate Array . . . . . 28
**GPIO** General-purpose input/output . . . . . 39
**ID** Identification . . . . . 26
**I/O** input/output . . . . . 12
**JC** Java Card . . . . . 6
**JCVM** Java Card Virtual Machine . . . . . 12
**JCRE** Java Card Runtime Environment . . . . . 9
**JDK** Java Development Kit . . . . . 6
**JRE** Java Runtime Environment . . . . . 6
**JVM** Java Virtual Machine . . . . . 6
**MMU** Memory Management Unit . . . . . 22
**MPU** Memory Protection Unit . . . . . 22
**OS** Operating System . . . . . 6

**PIN** Personal Identification Number . . . . . 12
**RAM** Random-Access Memory . . . . . 11
**RFID** Radio-frequency identification . . . . . 12
**ROM** Read Only Memory . . . . . 31
**SoC** System-on-Chip . . . . . 20
**SRAM** Static RAM . . . . . 24
**TLB** Translation lookaside buffer . . . . . 23
**VHDL** Very High Speed Integrated Circuit Hardware Description Language . . . . 41
**VM** Virtual Machine . . . . . 5

## A.2 Core Information

| IP-Core | Version Number | Class |
|---|---|---|
| CortexM1Top | 3.1.101 | CPU |
| CoreAHBLite | 5.0.100 | BUS |
| CoreMemCtrl | 2.0.105 | Regular |
| CoreAhbSram | 1.4.104 | Regular |
| CoreAHB2APB | 1.1.101 | Bridge |
| CoreAPB | 1.1.101 | BUS |
| CoreTimer | 1.1.101 | Regular |
| CoreUARTapb | 5.2.2 | Regular |
| CoreGPIO | 3.0.120 | Regular |
| CoreInterrupt | 1.1.101 | Regular |

Table A.1: IP-Cores version numbers

## A.3 Testbench VHDL Code

Here some code elements from the testbench.vhdl file are shown. The available testbench file was changed in a few parts. The first one was at the beginning. Here the component of the own hardware had to be added.

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
  use ieee.numeric_std.all;

library work;
  use work.corememctrl_pkg.all;
  use work.coreparameters.all;

entity testbench is
    generic (
        VECTFILE    : string := "corememctrl_usertb.vec"
    );
end entity testbench;

architecture test of testbench is

-- component declaration for MMU
component MMU is
port(
```

```
  -- Inputs
  HADDR      : in  std_logic_vector(27 downto 0);
  HADDR_1    : in  std_logic_vector(27 downto 0);
  HCLK       : in  std_logic;
  HRDATA_0   : in  std_logic_vector(31 downto 0);
  HREADYIN   : in  std_logic;
  HREADYIN_1 : in  std_logic;
  HREADY_0   : in  std_logic;
  HRESETN    : in  std_logic;
  HRESP_0    : in  std_logic_vector(1 downto 0);
  HSEL       : in  std_logic;
  HSEL_1     : in  std_logic;
  HSIZE      : in  std_logic_vector(2 downto 0);
  HSIZE_1    : in  std_logic_vector(2 downto 0);
  HTRANS     : in  std_logic_vector(1 downto 0);
  HTRANS_1   : in  std_logic_vector(1 downto 0);
  HWDATA     : in  std_logic_vector(31 downto 0);
  HWDATA_1   : in  std_logic_vector(31 downto 0);
  HWRITE     : in  std_logic;
  HWRITE_1   : in  std_logic;
  -- Outputs
  HADDR_0    : out std_logic_vector(27 downto 0);
  HRDATA     : out std_logic_vector(31 downto 0);
        HRDATA_1   : out std_logic_vector(31 downto 0);
  HREADY     : out std_logic;
  HREADYIN_0 : out std_logic;
  HREADY_1   : out std_logic;
  HRESP      : out std_logic_vector(1 downto 0);
  HRESP_1    : out std_logic_vector(1 downto 0);
  HSEL_0     : out std_logic;
  HSIZE_0    : out std_logic_vector(2 downto 0);
  HTRANS_0   : out std_logic_vector(1 downto 0);
  HWDATA_0   : out std_logic_vector(31 downto 0);
  HWRITE_0   : out std_logic;
  PageFault  : out std_logic
);
end component;
```

The next step was to add a few signals to create a connection between the MMU and the memory controller.

```
-- signal for MMU to CoreMemCtrl
signal HADDR_MMU        : std_logic_vector(27 downto 0);
signal HREADYIN_MMU     : std_logic;
```

```
signal HSEL_MMU          : std_logic;
signal HSIZE_MMU         : std_logic_vector(2 downto 0);
signal HTRANS_MMU        : std_logic_vector(1 downto 0);
signal HWDATA_MMU        : std_logic_vector(31 downto 0);
signal HWRITE_MMU        : std_logic;
signal HRDATA_MMU        : std_logic_vector(31 downto 0);
signal HREADY_MMU        : std_logic;
signal HRESP_MMU         : std_logic_vector(1 downto 0);
```

The last part was to change the port map of the MMU and the memory controller in dependency from the chosen test case.

```
-- MMU (device under test)
MMU_00 : MMU
port map (
  HCLK        => HCLK,
  HRESETN     => HRESETN,
  -- AHB_Mem_In
  HADDR           => HADDR(27 downto 0),
  HREADYIN        => HREADY,
  HSEL            => HSEL(0),
  HSIZE           => HSIZE,
  HTRANS                  => HTRANS,
  HWDATA                  => HWDATA,
  HWRITE                  => HWRITE,
  HRDATA                  => HRDATA,
  HREADY                  => HREADY,
  HRESP           => HRESP,
  --AHB_Mem_Out
  HADDR_0                 => HADDR_MMU,
  HREADYIN_0      => HREADYIN_MMU,
  HSEL_0                  => HSEL_MMU,
  HSIZE_0                 => HSIZE_MMU,
  HTRANS_0        => HTRANS_MMU,
  HWDATA_0        => HWDATA_MMU,
  HWRITE_0        => HWRITE_MMU,
  HRDATA_0        => HRDATA_MMU,
  HREADY_0        => HREADY_MMU,
  HRESP_0                 => HRESP_MMU,
  -- AHB_Conf_In
  HADDR_1                 => (others => '0'),
  HREADYIN_1      => '0',
  HSEL_1                  => '0',
  HSIZE_1                 => (others => '0'),
```

```
  HTRANS_1       => (others => '0'),
  HWDATA_1       => (others => '0'),
  HWRITE_1       => '0',
  HRDATA_1       => open,
  HREADY_1       => open,
  HRESP_1                => open
);

-- Memory controller
COREMEMCTRL_00 : CoreMemCtrl
generic map (
  -- Configuration parameters
  SYNC_SRAM            => SYNC_SRAM,
  FLASH_16BIT          => FLASH_16BIT,
  NUM_WS_FLASH_READ    => NUM_WS_FLASH_READ,
  NUM_WS_FLASH_WRITE   => NUM_WS_FLASH_WRITE,
  NUM_WS_SRAM_READ     => NUM_WS_SRAM_READ,
  NUM_WS_SRAM_WRITE    => NUM_WS_SRAM_WRITE,
  SHARED_RW            => SHARED_RW,
  FLOW_THROUGH         => FLOW_THROUGH,
  FLASH_ADDR_SEL       => FLASH_ADDR_SEL,
  SRAM_ADDR_SEL        => SRAM_ADDR_SEL
)
port map (
  -- Inputs
  -- Global
  HCLK                 => HCLK,
  REMAP                => REMAP,
  HRESETN              => HRESETN,
  -- Testing with MMU
  HADDR                => HADDR_MMU,
  HREADYIN             => HREADYIN_MMU,
  HSEL                 => HSEL_MMU,
  HSIZE                => HSIZE_MMU,
  HTRANS               => HTRANS_MMU,
  HWDATA               => HWDATA_MMU,
  HWRITE               => HWRITE_MMU,
  HRDATA               => HRDATA_MMU,
  HREADY               => HREADY_MMU,
  HRESP                => HRESP_MMU,
  --Testing without MMU
  --HADDR                => HADDR(27 downto 0),
  --HREADYIN             => HREADY,
```

```
  --HSEL                  => HSEL(0),
  --HSIZE                 => HSIZE,
  --HTRANS                => HTRANS,
  --HWDATA                => HWDATA,
  --HWRITE                => HWRITE,
  --HRDATA                => HRDATA,
  --HREADY                => HREADY,
  --HRESP                 => HRESP,
  --Testing MEMCTRL
  --HADDR                 => (others => '0'),
  --HREADYIN              => '1',
  --HSEL                  => '0',
  --HSIZE                 => (others => '0'),
  --HTRANS                => (others => '0'),
  --HWDATA                => (others => '0'),
  --HWRITE                => '0',
  --HRDATA                => open,
  --HREADY                => open,
  --HRESP                 => open,
  MEMADDR                => MEMADDR,
  MEMREADN               => MEMREADN,
  MEMWRITEN              => MEMWRITEN,
  FLASHCSN               => FLASHCSN,
  FLASHOEN               => FLASHOEN,
  FLASHWEN               => FLASHWEN,
  SRAMBYTEN              => SRAMBYTEN,
  SRAMCSN                => SRAMCSN,
  SRAMCLK                => SRAMCLK,
  SRAMOEN                => SRAMOEN,
  SRAMWEN                => SRAMWEN,
  -- Inouts
  MEMDATA                => MEMDATA
);
```

## A.4 Code simulation File

Here the code of the *.do file for runnning the simulation is shown. The paths have to be changed to the current folder and installation. This is important because otherwise it will not work.

```
quietly set ACTELLIBNAME proasic3l
quietly set PROJECT_DIR "D:/TU/Master/Diplomarbeit/SVN/
   CortexM1_System"
```

```
source "${PROJECT_DIR}/simulation/bfmtovec_compile.tcl";

if {[file exists presynth/_info]} {
   echo "INFO: Simulation library presynth already exists"
} else {
   file delete -force presynth
   vlib presynth
}
vmap presynth presynth
vmap proasic3l "C:/Microsemi/Libero_v11.4/Designer/lib/
   modelsim/precompiled/vhdl/proasic3l"
vmap proasic3 "C:/Microsemi/Libero_v11.4/Designer/lib/
   modelsim/precompiled/vhdl/proasic3l"
vmap COREAHBLITE_LIB "${PROJECT_DIR}/component/Actel/
   DirectCore/CoreAHBLite/5.0.100/mti/user_vhdl/
   COREAHBLITE_LIB"
vcom -work COREAHBLITE_LIB -force_refresh
vlog -work COREAHBLITE_LIB -force_refresh
if {[file exists COREGPIO_LIB/_info]} {
   echo "INFO: Simulation library COREGPIO_LIB already
      exists"
} else {
   file delete -force COREGPIO_LIB
   vlib COREGPIO_LIB
}
vmap COREGPIO_LIB "COREGPIO_LIB"
vmap COREMEMCTRL_LIB "${PROJECT_DIR}/component/Actel/
   DirectCore/CoreMemCtrl/2.0.105/mti/user_vhdl/
   COREMEMCTRL_LIB"
vcom -work COREMEMCTRL_LIB -force_refresh
vlog -work COREMEMCTRL_LIB -force_refresh
if {[file exists COREUARTAPB_LIB/_info]} {
   echo "INFO: Simulation library COREUARTAPB_LIB already
      exists"
} else {
   file delete -force COREUARTAPB_LIB
   vlib COREUARTAPB_LIB
}
vmap COREUARTAPB_LIB "COREUARTAPB_LIB"

vcom -93 -explicit  -work COREMEMCTRL_LIB "${PROJECT_DIR}/
   component/Actel/DirectCore/CoreMemCtrl/2.0.105/rtl/vhdl/
   core_obfuscated/corememctrl.vhd"
```

```
vcom -93 -explicit  -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreMemCtrl/2.0.105/rtl/vhdl/
    test/user/corememctrl_pkg.vhd"
vcom -93 -explicit  -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreMemCtrl/2.0.105/
    coreparameters.vhd"
vcom -93 -explicit  -work COREAHBLITE_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreAHBLite/5.0.100/rtl/vhdl/
    core_obfuscated/components.vhd"
vcom -93 -explicit  -work COREAHBLITE_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreAHBLite/5.0.100/rtl/vhdl/
    core_obfuscated/coreahblite_addrdec.vhd"
vcom -93 -explicit  -work presynth "${PROJECT_DIR}/component
    /Actel/DirectCore/CoreAhbSram/1.4.104/rtl/vhdl/u/
    components.vhd"
vcom -93 -explicit  -work COREGPIO_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreGPIO/3.0.120/rtl/vhdl/
    core_obfuscated/components.vhd"
vcom -93 -explicit  -work COREGPIO_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreGPIO/3.0.120/rtl/vhdl/
    core_obfuscated/coregpio_pkg.vhd"
vcom -93 -explicit  -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreMemCtrl/2.0.105/rtl/vhdl/
    core_obfuscated/components.vhd"
vcom -93 -explicit  -work presynth "${PROJECT_DIR}/component
    /Actel/DirectCore/CortexM1Top/3.1.101/rtl/vhdl/o/
    CortexM1Top_a3pl.vhd"
vcom -93 -explicit  -work COREUARTAPB_LIB "${PROJECT_DIR}/
    component/work/core/CoreUARTapb_0/rtl/vhdl/
    core_obfuscated/components.vhd"
vcom -93 -explicit  -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreMemCtrl/2.0.105/rtl/vhdl/
    test/user/async_memory.vhd"
vcom -93 -explicit  -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreMemCtrl/2.0.105/rtl/vhdl/
    test/user/sync_memory.vhd"
vcom -93 -explicit  -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreMemCtrl/2.0.105/test/
    amba_bfm/vhdl/bfm_main.vhd"
vcom -93 -explicit  -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreMemCtrl/2.0.105/test/
    amba_bfm/vhdl/bfm_ahbl.vhd"
```

```
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    hdl/AHB_In.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    hdl/AHB_Out.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    hdl/conf_logic.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    hdl/mode.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    hdl/mem_logic.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    smartgen/tlb_reset/tlb_reset.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    smartgen/UserMode/UserMode.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    smartgen/PageFault_or/PageFault_or.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    smartgen/trans_table/trans_table.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    component/work/MMU/MMU.vhd"
vcom -93 -explicit -work COREMEMCTRL_LIB "${PROJECT_DIR}/
    component/Actel/DirectCore/CoreMemCtrl/2.0.105/rtl/vhdl/
    test/user/testbench.vhd"

vsim -L proasic3l -L presynth -L COREAHBLITE_LIB -L
    COREGPIO_LIB -L COREMEMCTRL_LIB -L COREUARTAPB_LIB -t 1
    ps COREMEMCTRL_LIB.testbench
do "${PROJECT_DIR}/simulation/wave.do"
run -all
```

## A.5 AMBA control File

The last part here is the code from the test file for the AMBA controlls. Here it is easily possible to change the controls for the master device.

```
#////////////////////////////////////////////////////////////
#
# Syntax:
# ~~~~~~~
#
# memmap      resource_name base_address;
#
# write       width resource_name byte_offset data;
```

```
# read       width resource_name byte_offset;
# readcheck  width resource_name byte_offset data;
#
#-------------------------------------------------------------


#-------------------------------------------------------------
# Memory Map
# Define name and base address of each resource.
#-------------------------------------------------------------

memmap FLASH     0x00000000;
memmap SSRAM     0x08000000;

procedure main
    call mem_test
return


procedure mem_test
    # Set debug level (controls verbosity of simulation
       trace)
    debug 3;

#-------------------------------------------------------------
# Test FLASH access
#-------------------------------------------------------------
print "=============================================";
print "FLASH test";
print "=============================================";
# print "=============================================";
# print "Set USERMODE ON";
# print "=============================================";
# write      w FLASH    0x4F4E      0x00004F4E;

# print "=============================================";
# print "Set SYSTEMMODE";
# print "=============================================";
# write      w FLASH    0x4F4646    0x004F4646;

# print "=============================================";
# print "Set RESET";
# print "=============================================";
```

```
# write      w FLASH    0x525354    0x00525354;

# print "=================================================";
# print "Set USERMODE ON";
# print "=================================================";

# write      w FLASH    0x4C97013    0x00000317;
# read       w FLASH    0x4600200;
# readcheck w FLASH    0x00         0x0000dead;
# write      w FLASH    0x00         0xdeadbeef;
# write      w FLASH    0x4F4E       0x00004F4E;
# write      w FLASH    0x4C97013    0x00000317;
write       w FLASH    0x4600206    0xabcd1234;
# write      w FLASH    0x4000403    0xdeaddead;
# write      w FLASH    0x4400003    0x00000000;
# read       w FLASH    0x4400003;
# readcheck w FLASH    0x00         0xdeadbeef;
# readcheck w FLASH    0x4600206    0xabcd1234;

# write      w FLASH    0x200000     0x12345678;
# readcheck w FLASH    0x200000     0x12345678;
# readcheck w FLASH    0x00         0xdeadbeef;
# readcheck w FLASH    0x04         0x12345678;
# readcheck w FLASH    0x02         0x45454545;
# write      w FLASH    0x10         0xabcd1234;
# write      w FLASH    0x44         0x00abcdef;
# write      w FLASH    0x50         0x3d681acf;
# write      w FLASH    0x54         0x82db74a9;
# write      w FLASH    0x58         0xe81d93be;
# readcheck w FLASH    0x50         0x3d681acf;
# readcheck w FLASH    0x54         0x82db74a9;
# readcheck w FLASH    0x58         0xe81d93be;
# readcheck w FLASH    0x44         0x00abcdef;

#-------------------------------------------------------
# Test SSRAM access
#-------------------------------------------------------

print "=================================================";
print "SRAM test";
print "=================================================";

# write      w SSRAM    0x00         0xD00FD00F;
```

```
# write      w SSRAM   0x03        0x22222222;
# write      w SSRAM   0x05        0x33333333;
# readcheck  w SSRAM   0x00        0xD00FD00F;
# readcheck  w SSRAM   0x03        0x22222222;
# readcheck  w SSRAM   0x05        0x33333333;

# write      b SSRAM   0x24        0x41;
# write      b SSRAM   0x3c        0xa5;
# write      b SSRAM   0x28        0xcd;
# readcheck  b SSRAM   0x3c        0xa5;
# readcheck  b SSRAM   0x24        0x41;
# readcheck  b SSRAM   0x28        0xcd;

# write      b SSRAM   0x42        0xae;
# write      b SSRAM   0x43        0x67;
# write      b SSRAM   0x4a        0x14;
# readcheck  b SSRAM   0x42        0xae;
# readcheck  b SSRAM   0x43        0x67;
# readcheck  b SSRAM   0x4a        0x14;

# write      w SSRAM   0x50        0x3d681acf;
# write      w SSRAM   0x54        0x82db74a9;
# write      w SSRAM   0x58        0xe81d93be;
# readcheck  w SSRAM   0x50        0x3d681acf;
# readcheck  w SSRAM   0x54        0x82db74a9;
# readcheck  w SSRAM   0x58        0xe81d93be;

# write      h SSRAM   0x12        0x368c;
# write      h SSRAM   0x10        0x8dba;
# write      h SSRAM   0x16        0xe1db;
# readcheck  h SSRAM   0x12        0x368c;
# readcheck  h SSRAM   0x10        0x8dba;
# readcheck  h SSRAM   0x16        0xe1db;

return
```

The "#" character has to be removed in the lines which are going to be used in the simulation.

# Bibliography


[1] AMBA®3 AHB-Lite Protocol v1.0 Specification. ARM Limited.

[2] Java Card Technology. Oracle Corporation. Accessed on 27.12.2014.

[3] TrustZone. ARM Limited `http://www.arm.com/products/processors/technologies/trustzone.php/`. Accessed on 05.03.2015.

[4] Identification System, 1969. `http://worldwide.espacenet.com/publicationDetails/biblio?CC=GB&NR=1317915A&KC=A&FT=D&ND=4&date=19730523&DB=EPODOC&locale=en_EP#`.

[5] Cortex-M1-enabled ProASIC3L Development Kit. Microsemi Corporation `http://www.microsemi.com/products/fpga-soc/design-resources/dev-kits/proasic3/cortex-m1-enabled-proasic3l-development-kit`, 2002. Accessed on 25.03.2015.

[6] RFID cards. Kaardiekspert Ltd. `http://www.kaardiekspert.ee/en/rfid-kaardid`, 2002. Accessed on 23.03.2015.

[7] Write once, run everywere. Computer Weekly `http://www.computerweekly.com/feature/Write-once-run-anywhere`, 2002. Accessed on 27.12.2014.

[8] `http://microtoad.free.fr/?p=15`, Accessed on 10.04.2005.

[9] Bar-el, H. Known Attacks again Smartcards. Tech. rep., Discretix Technologies Ltd., Reviewed 18.02.2015.

[10] Chan, S.-c. C. An Overview of Smart Card Security, captured on Jan. 27, 2001. Aug 17 (1997), 1–7.

[11] Chen, Z. Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley Professional, 2000.

[12] Daryatmo, B. J2ME. Weblog : Budi Daryatmo (November 2007).

[13] Ding, J.-H. ARMvisor: System Virtualization for ARM. Tech. rep., National Tsing Hua University, In: Linux Symposium (2012).

[14] Goldberg, R. P. Architectural Principles for Virtual Computer Systems. Tech. rep., Harvard University, 1973.

[15] Hogenboom, J., and Mostowski, W. Full Memory Read Attack on a Java Card. Tech. rep., 2010.

[16] Iguchi-Cartigny, J., and Lanet, J.-L. Developing a Trojan applets in a smart card. Tech. rep., J. Comput. Virol., 2010.

[17] ISO/IEC. Identification cards – Contactless integrated circuit cards – Proximity cards – Part 4: Transmission protocol. ISO/IEC 14443-4., International Organization for Standardization / International Electrotechnical Commission, 2008.

[18] Joan, B. Memory protection units(MPU). A look at ARM MPUs., September 2013.

[19] Lackner, M., and Irauschek, M. Codesign for Countermeasures against Malicious Applications on Java Cards. Work Package 1: Threat Analysis. Tech. rep., Institute for Technical Informatics Graz University of Technology, 2011.

[20] Lafer, M. Design and Implementation of a Java Card Operating System for Design Space Exploration on Different Platforms. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2014.

[21] Mostowski, W., and Poll, E. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. Tech. rep., 2008.

[22] Oracle Corporation. Java Card 3 Platform: Runtime Environment Specification, Classic Edit 3.0.4 ed., September 2011.

[23] Popek, G., and Goldberg, R. Formal requirements for virtualizable third generation architectures. Communications of the ACM (1974).

[24] Smith, J., and Nair, R. Virtual Machines: Versatile Platforms For Systems And Processes. Morgan Kaufmann Publishers Inc., 2005.

[25] Tunstall, M. Attacks on Smart Cards. Gemplus, 2003. Reviewed 17.01.2015.

[26] Witteman, M. Advances in Smartcard Security. Information Security Bulletin (2002).