Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

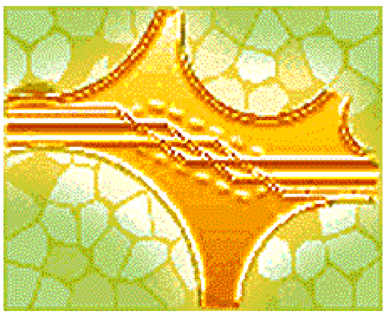
2011

MSc THESIS

A Co-processor for a Secure Implantable Medical Device

Siskos Dimitrios

Abstract



CE-MS-2011-03

This work describes an energy efficient solution for secure communication between an Implantable Medical Device (IMD) and a user. For the security part a communication protocol has been selected which supports entity-authentication, message-authentication and confidentiality. For achieving low power and energy consumption the following setup has been chosen. The IMD processor has been partitioned into two modules. The first module supports the main implant function, while the second module is responsible for secure communication with the outside world. In this work we have implemented the latter module: we have designed a 5-stage-pipeline RISC, application specific processor along with its compiler and optimized them to efficiently support the IMD security workload. The processor has been synthesized for a 90nm CMOS ASIC technology. The comparison between the base-line processor and its optimized version resulted in the following improvements: 41.4% decrease in execution time, 11.8% increase in the executed instructions per cycle (IPC) and 37% decrease in energy consumption. These improvements came at the slight cost of 8% increase in power and 7% increase in area.

A Co-processor for a Secure Implantable Medical Device

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Siskos Dimitrios
born in Athens, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

A Co-processor for a Secure Implantable Medical Device

by Siskos Dimitrios

Abstract

This work describes an energy efficient solution for secure communication between an Implantable Medical Device (IMD) and a user. For the security part a communication protocol has been selected which supports entity-authentication, message-authentication and confidentiality. For achieving low power and energy consumption the following setup has been chosen. The IMD processor has been partitioned into two modules. The first module supports the main implant function, while the second module is responsible for secure communication with the outside world. In this work we have implemented the latter module: we have designed a 5-stage-pipeline RISC, application specific processor along with its compiler and optimized them to efficiently support the IMD security workload. The processor has been synthesized for a 90nm CMOS ASIC technology. The comparison between the base-line processor and its optimized version resulted in the following improvements: 41.4% decrease in execution time, 11.8% increase in the executed instructions per cycle (IPC) and 37% decrease in energy consumption. These improvements came at the slight cost of 8% increase in power and 7% increase in area.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2011-03

Committee Members :

Advisor:	Ioannis Sourdis
Advisor:	Christos Strydis
Advisor:	Georgi Nedeltchev Gaydadjiev
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Jan van der Lubbe

Contents

List of Figures

List of Tables

Acknowledgements

First, I would like to thank my advisors I. Sourdis, C. Strydis and G.N Gaydadjiev for their guidance and patience.

I would also like to thank Hans van Someren, P. Peris Lopez and especially Danny P. Riemens for spending their valuable time and helping me with their experience. This work would not be completed without Danny's help.

Finally, I would like to acknowledge my family and friends for their support and especially: my brother, Dimitris, Giorgos, Antonis, Peggy, Petros, Nikiforos and Stefanos.

Siskos Dimitrios
Delft, The Netherlands
March 26, 2011

1 Introduction

The main functionality of an Implantable Medical Device (IMD) is to treat and measure physiological conditions within the body. IMDs are implanted 2-3 cm under the skin. Millions of people are carrying IMDs on their bodies. Examples of such devices are listed below:

- **Pacemakers:** Causes the heart to beat when the heart rate slows down.
- **Implantable Cardiac Defibrillators:** Managing cardiac arrhythmia.
- **Neurostimulators:** Managing Parkinson disease.
- **Implantable Drug Pumps:** Treating diabetes.

Most of the Implantable Medical Devices (IMDs) use a battery as their energy source. This leads us to understand that the device needs to be ultra-low-power in order to increase the battery's life. Most of the times the battery's discharge leads to a surgery. In early days an implant's low-power design was translated into a device including only the main functionality and was implemented hardwired. Over the last decades the fast development in batteries, shrinking of transistors and low-power designs enabled the usage of microcontrollers or even processors while increasing the battery's life. Nowadays, IMDs have the potential to monitor, store data and treat physiological conditions within the body. Now the physician has the opportunity to read the medical history of his patient, to update the software, or to turn off the device by wireless communication.

As every electronic device, implants are vulnerable to attacks, especially when they are communicating with the outer world remotely. Furthermore, this gives the motivation for our work which is explained below in detail. It is easy to understand that IMDs need a level of security against adversaries and this is the goal of our thesis. The difficulty in achieving that goal is that there are more than a couple of parameters that must be taken into account. First, an IMD because of its nature should be small-size, low-power and must not limit at all the remaining functionalities. So, a formula should be found that provides security on IMDs while maintaining the above -IMD- characteristics.

Summarizing, this introduction explains why processors are used on IMDs, presents some examples of them, and motivates the need of security in IMD communication, which is the main problem addressed in this thesis. In the remainder of this chapter follows: the problem statement in section 1.1, the thesis objectives in section 1.2 and finally the thesis overview in section 1.3.

1.1 Problem Statement

In this section, we present the problems addressed while securing an IMD. The main problem is the power and energy consumption of the IMD. The IMD must be ultra-low power and energy in order for its battery to last longer. By adding hardware and software security the power and energy problems become even bigger. Hence, a scheme must be found that will be low energy and power consuming. In addition to the main problem, there are problems which are caused by possible attacks. These attacks are shown bellow:

- Impersonation of the implant or of a reader in order to prevent treatment.
- Altering the messages during communication.
- Stealing and reading the messages which comprise personal data.
- Jam Denial-of-Service (DoS) attack. Jam DoS attack occurs when an intruder blocks the communication channel of an implant by sending repeatedly messages.
- Battery DoS attack. Battery DoS attack occurs when the attacker discharges the battery. For instance this can be done, if the attacker asks repeatedly a specific operation from the implant. In such situation the IMD will run repeatedly some software for analyzing the received request. Thus the implant will run out of battery.
- The Doctor alters the saved data in the implant when he makes a mistake to cover himself.
- The doctor denies that he did a specific action to the implant which finally leaded to unpleasant results. For example, if the doctor has provided a wrong medicine to his patient, probably it could be shown in the implant's log files. Thus, if the doctor has the authority to reset some information from the IMD's memory, then he has the chance to hide evidence of wrong treatment.

Some of the above attacks may cause battery's discharge, treatment prevention, human segregation (for instance, if an employer can identify that someone carries an IMD, maybe he will not hire him). It is obvious that most of the above attacks can cause severe harm or even death. Furthermore, the above attacks could also be used for blackmailing someone who carries an implant. Consequently, the problem is how to secure the implant against these attacks. Probably, because of power limitation, it is not possible to deal with all of them, thus the most severe ones should be selected for defending against. While working on defending these attacks, a balance must always be kept between *safety and utility* with *security and privacy* criteria, as it is mentioned in [12]. These criteria have many attributes that are opposite to each other, which makes the problem more difficult to solve. Both criteria and attributes are explained in detail below:

Safety and Utility: Safety for an IMD means that the IMD should not cause harm and utility means that the device must be useful for both patients and clinicians.

To achieve this we need the following: *data availability* to appropriate entries, *data accuracy*, *device identification* -maybe an IMD need to be deactivated before a surgery-, *device configurability* -authorized entities change settings-, *updatable software*, *auditable* and *resource efficiency*.

Security and Privacy: In order to achieve security, we need *authorization*. This means that specific groups of people can perform specific tasks. Here, we have to keep in mind that in emergency situations the authorization policies should change (eg. Criticality-aware access-control model [17]), in order to allow treatment by a physician who, in normal conditions, does not have access. We also need *careful availability* of the device so an adversary won't be able to perform DoS. In terms of privacy, the device *should not reveal its existence* to unauthorized parties, neither its *type*, its *id*, or the *personal data* of the patient.

As we can easily understand by reading the above analysis there are many attributes which are trade-offs. The dilemmas are the following: *security versus accessibility, security versus device resources and security versus usability*, as they are identified in [12].

In the current section we stated the problems for designing a secure IMD. Summarizing, the main problem of this thesis is to keep the IMD low-power while securing it. The remaining problems are related to defending possible attacks, without compromising the utility and the safety of the IMD.

1.2 Thesis Objectives

The main objective of the thesis is to design a low-power architecture for the communication part of a secure Implantable Medical System. The communication is considered to be wireless and is taking place, between the implant and the outside world (reader). This work is part of the SiMS (Smart Implantable Medical Systems) project, which is described in detail in [3]. The main goal of the SiMS project is to provide the IMD designers with reliable, stable and small-sized implantable components in order to build novel IMDs. The importance of this is that the designers will not have to build the implants from scratch anymore which is time-consuming and risky.

For achieving the main object of the thesis, this work has been partitioned in the following tasks:

- Define the possible attacks against the IMD.
- Describe the security-system architecture and the security protocol, used for protecting the IMD against the defined attacks. The system architecture defines the IMD modules.
- Design an Application-Specific-Instruction-Set Processor (ASIP) and its compiler, to support secure communication suited for IMDs.
- Apply optimizations to the proposed processor and compiler, so as to improve power efficiency and performance.

- Evaluate the proposed solutions in terms of security, area-cost, power efficiency and performance.

1.3 Thesis Overview

In this chapter, we presented the reason for using processors into IMDs, the thesis motivation, the thesis problems and the thesis objectives.

Chapter 2 presents the background and the related work of this thesis. There we show some well known cryptographic techniques, the prior work on implant security and a brief description on the SiMS project.

Chapter 3 describes the design and the implementation of the proposed system. We present the attack scenarios, the communication protocol, the system architecture and the security co-processor and compiler which we have built. That chapter combines knowledge of the previous chapters and proposes solutions.

In chapter 4, we present and explain the thesis results. Furthermore, we show and explain some decisions taken in chapter 3 about the processor and compiler design.

Chapter 5 contains the thesis conclusions. It includes a brief summary of this thesis, the thesis contributions in IMDs and in the SiMS project and the future work.

Background and Related Work 2

This chapter contains background information on secure implantable systems that the reader should be familiar with in order to better understand this text. We present some essential aspects for security, which are the base of our security-protocol proposal, further explained in chapter 3. We also describe the SiMS project, which includes this thesis work. Finally, prior work on IMD security is presented in order to gain insight on access control and security methods, and to identify open issues where we could contribute.

This chapter is organized in the following four sections: Section 2.1 gives an overview of the basic methods of cryptography. Section 2.2 describes the overall SiMS project, which includes our work. In section 2.3, we describe prior work on secure implants. Finally, in section 2.4, we explain the selection of the MISTY1 cryptographic algorithm and then we describe this algorithm.

2.1 Basic Methods for Security

As can be seen in book [29], a security system can be divided in the components shown in table 2.1. These are Key Management, Entity Authentication, Data Integrity, Message Authentication and Confidentiality, which are explained in the upcoming subsections. The above mechanisms can be implemented in different ways. For instance, there is no standard secure-communication protocol, which can be efficient for every system. For each system the protocol should be selected, regarding its available resources and constraints, in addition with the attacking-types on which the system is vulnerable.

Key Management	
Entity Authentication	Message Authentication
Data Integrity	
Confidentiality	

Table 2.1: Typical components of a security system.

In the following subsections we explain what is the purpose of each mechanism and how it can be implemented. Particularly, in subsection 2.1.1, we present Confidentiality. In subsection 2.1.2, we describe Entity Authentication, Data Integrity and Message Authentication. Finally, in subsection 2.1.3, we give an overview on Key Management. The current section is written according to the book [29].

2.1.1 Confidentiality

Confidentiality is defined as the prevention of specific information from being accessed by unauthorized entities [18]. In cryptography this is achieved by encrypting this information with a secret-key.
The cryptographic algorithm of a secured-cipher (cryptographic) system should try to minimize the following:

- The probability with respect to finding the key, given the ciphertext.
- The probability with respect to retrieving the plaintext, given the ciphertext.
- The probability with respect to finding the key, given both the ciphertext and the plaintext.

There are two main categories of systems in cryptography, namely the public-key (or asymmetrical cipher) and the private-key (or symmetrical cipher) systems. A system is named after the cryptographic algorithm it uses. If the algorithm is symmetrical, then the system is called symmetrical otherwise, if the algorithm is asymmetrical, the system is asymmetrical as well. Symmetrical is a system where all parties use the same key for encryption and decryption, while asymmetrical is a system where each party acquires different keys for encryption and decryption.

Private Key System: A private system -also known as symmetrical-cipher system-needs the same key for encrypting and decrypting a message. This kind of ciphers are distinguished into block and stream ciphers. A block cipher divides the text into a fixed group of bits (blocks) and operates with each block (The block cipher can encrypt the blocks separately or not and that depends to its *mode of operation*). A stream cipher partitions the text into bits and plaintext bits are encrypted one at a time -however, the key-stream should be at least the same length with the plaintext-. An important issue in a symmetrical system, is how to ensure that the same secret-key is available to both the transmitter and the receiver. One solution is to transfer the secret-key physically. Another solution is to transmit the key by another transmission line than the one used for the ciphertext. However, most of the times the one entity generates the secret key, signs it, encrypts it by using an asymmetrical algorithm and then sends it to the other entity. A main advantage of a private system against a public one is that the private is nearly always much less computationally intensive and this makes it desirable for usage in low-power systems. However, the Elliptic-Curve public cryptographic algorithm is not so computationally intensive and for that reason is already used in low-power devices. A disadvantage of the private system is that each entity-pair needs a unique key for maintaining communication confidentiality. Consequently, as the number of the communicating entities increase, the same is true for the storage requirements.

In order to minimize the probabilities mentioned in the beginning of this section, the private cipher algorithms uses techniques such as:

- **Permutation:** Altering the bit-positions of a bit-sequence. The bit-sequence can be a n-bit block which is going to be enciphered.
- **Expansion:** Increasing the bit-number of a group of bits by repeating some bits.
- Using **substitution boxes**: These boxes are tables with publicly known contents. These tables get as input a location (a column and a row) and return the contents of it.
- **Iteration:** Specific functions of the algorithm should be repeated and for each repetition different subkeys could be calculated.

Examples of private ciphers are the Data Encryption Standard (DES), Advanced Encryption Standard (AES), Blowfish, MISTY1 and RC5. Cipher selection is very important because each cipher has different characteristics. For instance, ciphers can be distinguished according to security, power, energy and computational complexity.

Public-Key System: A public system -also known as asymmetrical-cipher system- uses a key for encrypting and a different one for decrypting a message. An advantage of this system over the private-key system is that, there is no need anymore for a secret key-exchange or for storing the keys of all the communicating parties. A single key-pair is sufficient. The explanation for this key-convenience is the following.
Every entity has its own private and public keys. The private key is not known to anyone else but the entity itself. On the other hand, the public key is given to anyone who wants to communicate with the specific entity (the public key is not assumed to be secret). Assuming that A and B want to communicate with each other:

1. A should know the public key of B -U_b- and B should know the public of A -U_a-. Also A and B have their own private keys P_a and P_b respectively.
2. A sends a message to B encrypted by U_b. Now the only one who can decrypt the specific message is the one who owns P_b. The only one who has that key is B, because it is B's own private key and B has not given it to anybody. Hence, confidentiality is guaranteed.

It is also important to mention that asymmetrical-cipher systems depend on one-way and "trapdoor" functions. These functions are easy to calculate themselves, but their inverse function is very difficult to be computed. Such a function is the power function ($f(x) = x^a$). It is easy to compute the product of it, but it is relatively more difficult to find the root of a number.

Examples of public ciphers are: RSA, Elliptic-Curve, El Gamal and Cramer-Shoup. The public-cipher selection is as important as the private-cipher selection.

In the current subsection we have defined confidentiality and the ways it can be achieved. The next issue we must solve is the authentication of the person who has sent the message.

2.1.2 Authentication and Integrity

The identification of the communicating parties and the integrity of a message are of great importance in a confidential communication. Without identification many serious problems can occur. For instance, in a pacemaker situation, nobody else except the doctor should be able to send a shut-down command. This can only be prevented by using an authentication protocol. These issues are described in detail in the current subsection. Firstly, the following terms should be defined: *entity authentication*, *message integrity* and *message authentication*.

- **Entity authentication:** This means identification of a person, or of a system. In cryptography when A sends B a message, it is important for B to be sure that A had sent the message. Also it is important for A to be sure that B received the message.
- **Message integrity:** Message integrity is achieved when the message is not altered through its way to the receiver and is not replayed by an intruder.
- **Message authentication:** Message authentication is the combination of data integrity and entity authentication. This means that both entities that communicate are sure about the identity of the other party and the integrity of the message itself.

In the remainder of this subsection we describe the techniques and the protocols for achieving entity authentication, message integrity and message authentication. First, we describe two authentication protocols, one for a symmetrical and one for an asymmetrical system, because they are distinctly different.

Entity Authentication protocols:

Private-Key System: Here we describe an entity-authentication protocol for a private system. As we have mentioned, this is not the only safe protocol, because security requirements change according to the system for which security is implemented. For example, the protocol below achieves mutual authentication, while for another system a one-entity authentication would be enough. Consider the protocol shown in figure 2.1, which is also described below in steps:

1. A sends a random number R_A to B. Now A expects from B to encrypt R_A with key, K, and send it back. The only ones that could encrypt a message with the key K are A and B. Hence, a successful comparison between the sent and received random numbers, can ensure A that A is talking with B.
2. B responds with $R_B//eK(B//R_A)$. eK is the encryption function which encrypts, the parenthesis-enclosed data with key K. B sends R_B to A for the same reason that A had sent R_A. B does not encrypt only R_A, but also its address. B does that against reflection attacks. A reflection attack exists when an intruder C establishes two parallel sessions with A and B, and misleads them into believing that they talk to each other, when they actually talk to C. For instance assume that A tries to send R_A to B while an intruder C captures it. C then immediately sends R_A to A. A then, responses $ek(R_A)$

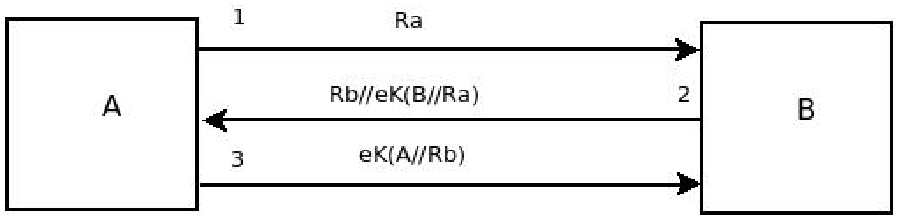


Figure 2.1: Mutual-entity authentication protocol for a symmetrical system. (**Where:**
R_I: *Random number, which is generated and sent by the party* I.
I: *It is an identification (the address), of the party* I.
$eK(X)$: *Describes the encrypting result of message* X*, with the symmetrical key,* K.
$eU_I(X)$: *Describes the encrypting result of message* X*, with the public key of the party* I.
$dP_I(X)$: *Describes the signing result of message* X*, with the private key of the party* I.
"//": concatenation.)

to C, which is also the response that A expected from B. Then C sends back to A, $ek(R_A)$. In the meanwhile C does similar things to mislead B.

3. A responds with $eK(A//R_B)$ to B in order for B to make sure that A received the message.

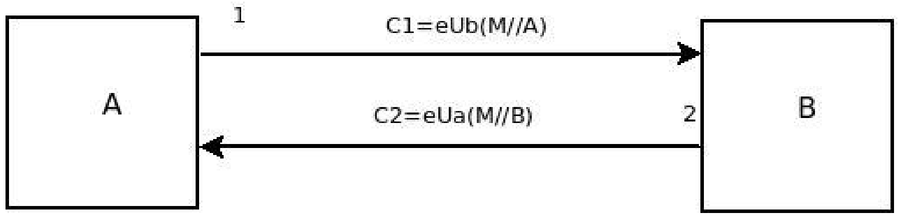
That is how the symmetrical mutual entity-authentication is achieved.

Public-Key System: Now we are going to describe an entity authentication protocol for an asymmetrical system. Consider the protocol shown in figure 2.2(a).

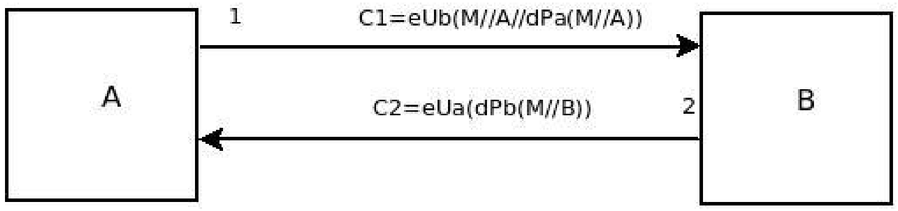
1. A sends $C1$ to B, which is data encrypted by the public key of B, U_B. This data includes a message M and the address of A. The only one who can decrypt $C1$ is the one who owns P_B. P_B is the private key of B, hence only B can decrypt it.
2. Entity B receives the message and decrypts it. Then for authenticating itself to A, B encrypts the message M and its address B with the public key of A , U_A. The only one who can decrypt that message is A, with A's private key P_A. Then, B sends $C2$ to A.

In the protocol we have just described only B has been authenticated, so it is a one-way entity authentication. The reason is very simple. Everybody has the public key, U_B, of B, hence, anyone can create $C2$. For mutual authentication, B should include in $C2$ another message $M2$ and then A should send back $M2//A$ encrypted by U_B. In that case, the protocol would become a three-way protocol. A two-way protocol to achieve mutual authentication is given in the message authentication description.

Message Integrity:
Next, we continue by explaining how message integrity can be achieved. Using a hash function is the most easy way to do it, as is explained later in detail. An original hash



(a) One-way entity-authentication protocol



(b) Message & mutual-entity authentication protocol

Figure 2.2: Asymmetrical systems protocols

function is a publicly known algorithm, which gets as input a variable-length sequence of values and produces a constant-length sequence, which is completely different from the input. Some examples of such hash functions are CRCs and checksums.
Let's assume that A wants to send a message to B. A first sends the message to B and then, via another channel A sends to B the hashed message. Then B calculates the hash of the message and compares the result with the hashed message that A sent. If they are equal, message integrity has been achieved. We have mentioned that A sends to B, the hashed message through a *different* channel than the original message. A does that because the hash function is publicly known and subsequently, if someone captures the message, he could easily calculate its hashing result.

Message Authentication:
Message authentication is a combination of entity authentication and message integrity. This can be achieved with the Message Authentication Code (MAC) or with digital signatures. The former is used when the two parties that communicate trust each other and the latter when they do not.

MAC: In this case we use a symmetrical encryption algorithm as a hash function. A difference between this and an original hash function is that the symmetrical algorithm is secret because a secret key is needed as input. Hence, it is possible to send together -and not through a different channel- the message and its hash result (MAC). Examples of symmetric algorithms used for that purpose are DES Hash and MD5.

Let's assume that A sends a message M, and its MAC, mgK(M), to B. B then computes the MAC of message M' that he receives. If the computed MAC and the MAC he had received are equal then the message is authenticated. There are two reasons that this works. First, nobody without possessing the secret key, can create the same MAC from the same message, so A is authenticated to B. Second if the message or the MAC are changed on their way to B, then the two MACs are not going to be equal. Thus, message integrity is maintained. But as we said before, the use of MAC requires that the two entities should trust each other. If this is not the case, then entity B -after receiving a message M- could declare that it received a message M', which has generated itself. So, it is obvious that there is no security between the communicating entities.

Digital Signature: A digital signature consists of a group of bits and its purpose is to prove the authenticity and integrity of specific data. Digital signatures are used when the communicating parties do not trust each other. The characteristic that makes them necessary in such situations is that they can easily be verified by anybody, but they can only be generated by one specific entity. For the signing procedure a public cryptographic algorithm like RSA or a dedicated algorithm like DSS can be used. In the case of the public cryptographic algorithm, a hash function is recommended to be used before signing in order to decrease the bit-width; otherwise, the signing computation and the transmission are going to be very slow. To achieve digital signing, signing and verifying algorithms along with two keys (a public and a private key) are needed. Let's assume that A sends a digital signature of a message M to B. In contrast to encryption, where A would encrypt the message M with the public key of B, U_B, now A is signing with its own private key, P_A, which means that only A can produce the signature and everybody possessing A's public key, U_A, can verify it. Hence, B and also a third person can easily justify if the message that B has received was truly from A. So message authentication has again been achieved. Examples of situations where digital signatures are used are software authentication and digital bank checks.

Now, let's return to the public entity authentication protocol in figure 2.2(a). There, we had a protocol which was not mutual, because only B, was authenticated by A. Let's see now the protocol in figure 2.2(b). We can observe the difference between the two figures: in figure 2.2(b) the encrypted messages include also digital signatures. Which means that only the receiver, who owns the private key, can decrypt the message and then can verify the signature with the public key of the sender. So we achieve message authentication and confidentiality.

2.1.3 Key Management

So far now we have discussed cryptographic methods to achieve confidentiality, authentication and integrity. Another important aspect in cryptography is key management. Key management consists of the following features:

- generation,
- distribution,
- storage,
- replacement,

- usage,
- and destruction,

of the keys. Key generation can be achieved by using a pseudorandom generator, such as the DES. Some people select to generate a key pattern on their own by, for instance, tossing a coin. For achieving security in session-key storage, a storage-key is needed. The storage key is used for enciphering the session key. Another important issue in key management is the frequency with which the key is replaced. This frequency depends on the importance of the enciphered data, the period of validity of the protected data -there are cases where data must be kept secret for just a small period- and the strength of the cryptographic algorithm. For key destruction a simple deletion from memory is sufficient. The only part remaining to complete the key-management brief description is key distribution. Key distribution differs between symmetrical and asymmetrical systems:

Private-Key System: Initially, a physical distribution of the keys is needed. Then comes the session-keys distribution by the distribution center, which can be done on-line or off-line. For the off-line case, the distribution center first determines which parties should communicate with each other and supplies the corresponding session-keys. In an on-line distribution each entity has its own and unique key for communication with the distribution center. When, for instance, A wants to communicate with B, A first authenticates itself to the center and the center returns a session key for A and B. The on-line in comparison with the off-line technique needs less stored-keys because there is no need for saving the session keys.

Public-Key System: Consider two parties -asymmetrical systems- A and B, who want to exchange a session key, K. Hence, A sends to B the message $eU_B(eP_A(K))$. The only one who can decrypt it and get the key is B. The only problem here is the authentication of B's public key, U_B, which can be solved by using a trusted authority i.e. someone who issues authentic keys U_A and U_B. A problem to this solution is the large number of stored keys, which adds also complexity in managing them. The number of keys is $n * (n - 1)$, because each party has $n - 1$ session keys, one for each communicating party. For that reason, the Diffie-Hellmann protocol is used, which reduces the total number of keys to n, one key for each user. In this case, the session key is always calculated when it is needed, hence there is no need for storing it. For more information about the Diffie-Hellmann protocol see [29].

In the current section the terms confidentiality, entity authentication, message authentication and key management were defined. Subsequently, methods for achieving them were explained in detail. After reading this section the reader should have acquired sufficient cryptographic background to understand this work. For further information on cryptography see [29].

2.2 The SiMS Project

This work is part of the Smart Implantable Medical Systems (SiMS) project which takes place in the Technical University of Delft. More precisely our work is part of the SiMS

digital architecture and the compiler tools, which are described below.
The main goal of the SiMS project is to construct a framework that provides medical researchers with the modules that are required to build variable implantable medical devices. The framework must be designed according to the following requirements:

- high dependability
- reliability and safety
- small size
- ultra-low power consumption

Nowadays, a typical IMD consists of a central unit -microcontroller, microprocessor-, sensors, actuators, a wireless transceiver and a power source. The majority of IMDs today are used to measure, process and regulate a single parameter of the body. A problem with today's implants is the procedure which is followed for building them. Usually such systems are made by researchers with electrical or mechanical engineering background who collaborate with medical researchers. That happens because it is not always feasible for researchers from all different fields, involved in building the device, to meet and work together. That approach probably lacks the selection of the best techniques from the different fields. That problem as well as that implantable devices are built from scratch -which results to a large number of tests- explain the large delay and risk which is involved, until the product is ready for use.
On the other hand, the existence of a platform can decrease the risk and the design time of medical implantable devices, because the platform provides the designer with already tested and used modules which have been improved by a lot of people. In figure 2.3 the platform concept is graphically shown. Also, it can easily be realized that if the design time is small then the device itself should be cheaper; consequently, it would be affordable for more people.
IMDs, because of their direct relation with human life, are characterized by high dependability, autonomy and self-awareness. Contrary to the majority of current IMDs, a major goal of SiMS is to monitor and regulate more than one biomedical parameter. The described system is shown in figure 2.4.
The various SiMS building blocks are as follows:

- **Digital architecture:** The digital architecture forms the main processing and control unit of the implant. It is also interacting with the radio transmitter. The requirements of this unit are, to be low-power, small-sized, and reliable. For achieving reliability fault-tolerant schemes are needed.
- **Compiler and design tools:** A compiler will be built for generating machine code for the specific Instruction Set Architecture (ISA) of the designed processor. As in every compiler code optimizations will be needed, which in this case are very important. Good code optimizations lead to less instructions, thus smaller execution time, thus less power consumption. Dependability is also treated in the compiler level and is responsible for finding out whether specific requirements

in area, timing and power could be met by the design. This is done by feeding the compiler an application-specific constraint file together with the application bitstream. This file contains the area, timing and power constraints. Then, the compiler determines whether a realistic solution exists in the SiMS platform -under these constraints-.

- **Sensors and actuators:** The idea about sensors and actuators, is to either develop new ones or to improve existeing ones. The goal is to boost their sensitivity and performance and to make them modular for serving diverse implant applications.

- **Wireless transceiver:** A wireless transceiver will be needed, which will receive and transmit various types of information with Quality of Service (QoS). The challenges here are again: small-size, reliable, low-power and ultra wide-band transmission.

- **Chip interfaces:** Standard interfaces for the system will be needed, in order to achieve the desired modularity, interoperability and re-usability. All sub-blocks of the SiMS framework should adhere to the same interfaces.

Existing work on the SiMS project includes:

- **"Implantable microelectronic devices" [24]:** This work performs a broad survey of existing IMDs -over a period of 20 years- and then classifies them according to the findings of that survey (see also [9])

- **"A generic digital architecture and compiler for implantable devices" [26]:** This work describes the idea of SiMS and its general frame.

- **"A New Digital Architecture For Reliable, Ultra-Low-Power Systems" [8]:** This work is an extension of work [26]. Namely it presents specifically what must be done on the SiMS project.

- **"Suitable cache organizations for a novel biomedical implant processor" [6]:** This work evaluates different instruction- and data-cache organizations in terms of perormance, power, energy and area.

- **"Profiling of Lossless-Compression Algorithms for a Novel Biomedical-Implant Architecture" [25]:** This work profiles a large set of compression algorithms and evaluates them regarding power, energy, compression rate and program-size.

- **"Profiling of Symmetric-Encryption Algorithms for a Novel Biomedical-Implant Architecture" [28]:** This work profiles a large set of symmetrical encryption algorithms and finally indicates the best algorithm in terms of area, power, energy, program-code size, encryption rate and safety. In section 2.4.1 this profiling is described in detail.

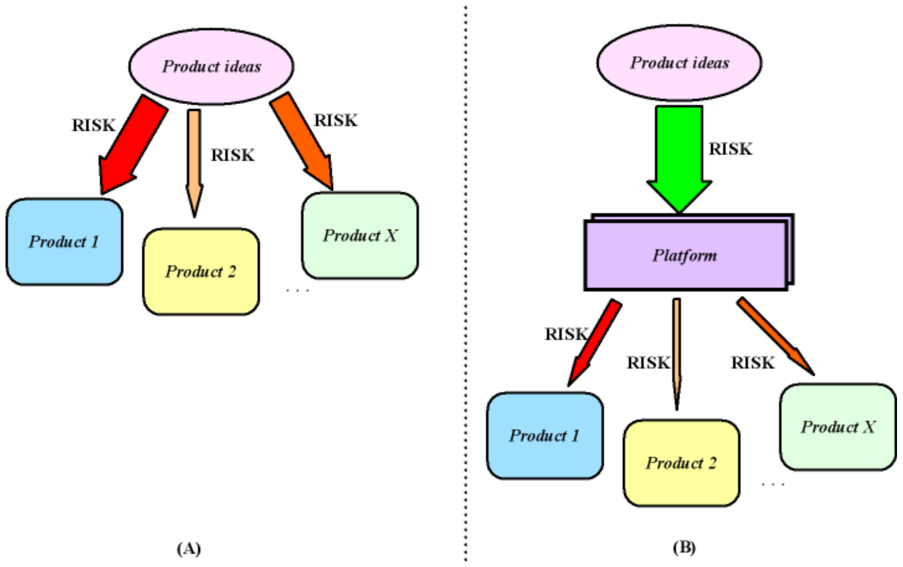


Figure 2.3: Involved risk factors with (A) and without (B) a design platform. (The thickness of the arrows is proportional to the risk weight).

- **"ImpBench: A novel benchmark suite for biomedical, microelectronic implants" [27]:** This work presents ImpBench, a novel benchmark suite chosen for designing and evaluating new digital processors for microelectronic implants.
- **"The Case for a Generic Implant Processor" [7]:** In this work a biomedical-application scenario is presented , which is executed on a processor simulator. The findings of that simulation are presented and analyzed.
- **"Exploring suitable adder designs for biomedical implants" [23]:** This work explores suitable adder designs for biomedical implants along with fault tolerant schemes.
- **"Automated Implant-Processor Design" [10]:** This work implements and describes ImpEDE, a framework that optimizes processor parameters. The optimization procedure is based on a genetic algorithm.

In this section we presented the goals, the contribution, the design details and all the works on the SiMS project. We can see that besides work 2.4.1 which selects an efficient symmetrical algorithm, there is no other work done on securing the IMD during communication with an external device. So, in this thesis we try to secure the IMD while at the same time to maintain the implant small-size, low-power, low-energy. Furthermore, the security should not prevent treatment and should not disrupt the IMD's main functionality. For these reasons we decided to design a secure communication co-processor in order to fulfill the requirements which we have just stated. More detail on the system architecture, the processor design and how they fulfill the above requirements is given in chapter 3.

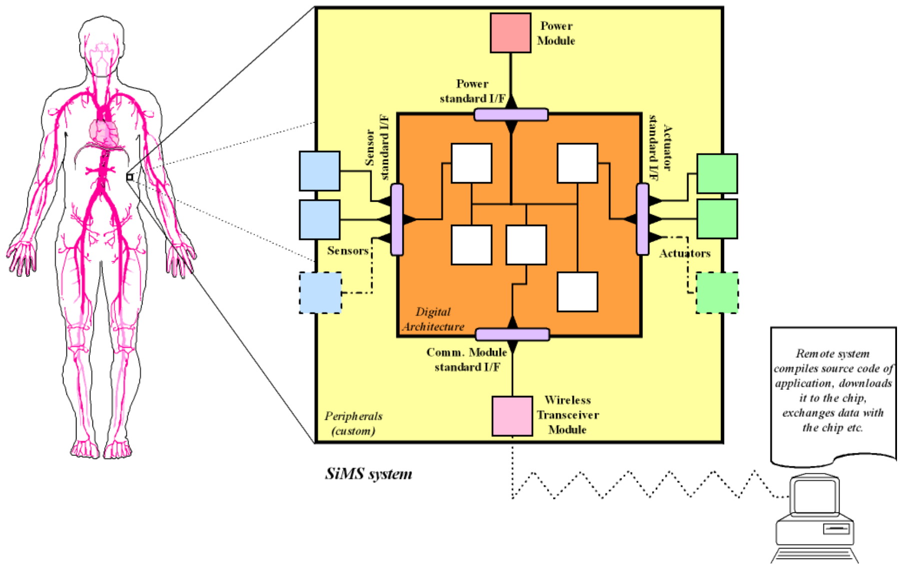

Power Module
Power standard I/F
Sensor standard I/F
Actuator standard I/F
Sensors
Actuators
Digital Architecture
Comm. Module standard I/F
Wireless Transceiver Module
Peripherals (custom)
SiMS system
Remote system compiles source code of application, downloads it to the chip, exchanges data with the chip etc.

Figure 2.4: Abstract view of the SiMS concept.

2.3 Prior Work on Secure Implants

In this section, we describe access-control, switching-mode and security methods upon implantable medical devices. Access-control is responsible for enabling access to authorized parties and to block unauthorized ones. In other words, access-control is the authentication protocol and it can be implemented by a typical cryptographic scheme, or by an alternative scheme such as the one described in subsection 2.3.1. The switching-mode method is responsible for switching between the emergency and normal mode. The emergency mode is activated when the patient is in an emergency situation, such as is a heart attack or falling unconscious. In such situations, security must not be the reason for preventing treatment. Hence, it is obvious that the access-control is different between the two modes. Switching modes can be achieved by using a typical magnetic-switch or by using another method such as the one described in subsection 2.3.2 which uses a Cloaker. The following works, have inspired us for selecting an access-control method, a switching-mode method and a suitable system setup. An open issue in these works, is that none of them propose a new processor architecture for the main implant or for the implant's communication part. In our work we have implemented an ASI co-processor for securing communication. A custom processor architecture is important, because it can save great amount of power and energy.

2.3.1 Proximity-based Access Control for Implantable Medical Devices

In [16], a method for securing an IMD in a normal and in an emergency mode is described. The security -as it is explained in detail below- is based upon cryptographic techniques and the proximity between the IMD and the user. The attack scenarios that are mentioned in [16], are the following:

1. The attacker wants to get access to medical data stored in the IMD or change device settings. This could be done for blackmailing and it can be achieved by replay attacks.

2. The attacker impersonates an IMD and makes the reader talk to him, instead of the real IMD. A motivation could be to prevent treatment.

Before continuing with the description of the normal and emergency modes, we will explain the proposed access-control proximity-based protocol, which is used in both modes. This protocol enables communication between entities, which are located inside a predefined range. Thus, both the IMD and the reader can be sure that the other side is in a specific range, which protects them against a large number of attackers. Now follows the description of the normal and the emergency modes:

- **Normal mode:** The security in this mode is achieved by using a credential (held by the reader) which shares a secret-key with the IMD and by the proximity based-protocol. The secret-key is used for cryptographic reasons. The proximity based protocol ensures that both sides are located in a specific range to each other.

- **Emergency mode:** In this mode only the proximity-based protocol is used. So whoever is in the range of the implant ($< 10cm$) can have access to it. The difference in comparison with the normal mode, is that in emergency mode the valid range is much smaller. The range is small, so that the risk involved for attacking the implant is significantly reduced.

However, by reading [16] is not clear how the switching-mode is done. Even so, we believe that using a magnetic-switch (the mode is changed if a magnet is passed over the IMD,) for that purpose is enough. Both modes are relying on the precise distance between the IMD and the reader. Here we explain the concept of the distance accurate measurement. The prover (reader) sends a hello message to the verifier (IMD) to start the communication. Then, the verifier sends back a single bit by the radio channel and records the start time $t1$. The prover receives it at time $t1'$ and sends back a bit at time $t1''$ via the sound channel which is much slower than the radio. Finally, the verifier receives the bit at time $t2$ and calculates the bit-travel time from the prover to the reader. We assume that $t1 = t1' = t1''$, because the bit-transmission through the radio channel and the computations in the prover's side, are negligible in comparison with transmission through the sound channel. Hence, the distance is calculated by the well known formula $d = v_s(t2 - t1)$. Where d, is the distance between the communicating entities, v_s is the speed of sound, $(t2 - t1)$ is the travel time form the prover to the verifier.
To test the concept the researchers have built proof-of-concept prototypes of both the prover and verifier. The prototypes are respectively a prover and a verifier with analog circuity for the RF and sonic communication and ATMega644p microcontrollers running at 20MHz for computation and control. The power consumption measurements of the microcontrollers during peak computations were, 0.15W at 5V DC for the receiver and 0.17W at 5V DC for the transmitter. The analog portion of the receiver consumed 0.13W at 10V DC.
In this thesis we have decided not to use this access-control method. A great advantage

of this approach is that provides security in the emergency mode without causing safety problems. The only problem is that it would probably need more complex computation than a typical access-control method. Consequently, we should reconsidered this concept for a future work, if the typical access-control-implementation energy consumption measures allow more computation, .

2.3.2 Switching IMD Modes using Cloakers

In [19], a security method is described which is based on Cloakers. Cloakers are externally worn devices, the functionality of which is described later in detail. The authors of [19] aim at achieving the following goals:

1. Provide safety and open access in emergencies: IMDs in an emergency situation should allow caregivers to have complete access over them.

2. Prove security and privacy under adversarial conditions.

3. Increase the battery's life: The IMD should be power efficient.

4. Limit the response time: The IMD's functionality should not be blocked or delayed by the security functionality because such a delay could be life critical.

So, the solution needs to balance the first two goals, above, while simultaneously protecting the battery life and response time of an IMD under both normal and adversarial conditions. The solution that is given in [19], is to use communication Cloakers. The setup requires an IMD, a Cloaker and one or more reading devices (malicious or not). The idea is that the Cloakers prevent connection with the IMD when they are present and allow connection when they are *not* present. The Cloaker can work as a proxy or as a system, that just establishes the communication between a reader and the IMD.
In that work no hardware detail is given. It is only mentioned that the system has been written in Java in order to maximize the portability of the code base onto different hardware implementations.
The Cloaker approach is very attractive for its switching-mode way and that is why we were thinking of including the Cloaker into our implementation. However, as it is explained in more detail in section 3.1, we have decided not to use one in our work. Even so, Cloaker usage should be reconsidered in a future work. In subsection 2.3.5, we proceed with a comparison between the switching-mode method by using a Cloaker and the typical one by using a magnetic-switch. For reasons explained in subsection 2.3.5 we have selected the magnetic-switch method.

2.3.3 Practical Techniques for limiting disclosure of RF-equipped medical devices

Radio links among IMDs are becoming very important in health care because they provide convenient power transmission. However, devices that respond to unauthorized queries may disclose their presence. So, the goal of [13] is to combine both facts and create an efficient protocol in terms of power and security. Before continuing to the protocol properties, we should explain what the Certificate authorities (CA), Access-control lists (ACLs) and revocation lists (RL) are, because the protocol is based on them. The CA is an organization that it is trusted by an IMD device and permitted to distribute authorization certificates to readers. A certificate shows if its holder has full or a set of permissions to a device. Certificates are used in public key systems. An ACL is a list of the authorized readers to access the IMD. An RL is a list of readers that used to have access permissions to an IMD but not any more. These readers, most of the times, have been stolen or lost, which means that there is a probability that a malicious person posseses them. The protocol properties are:

- Undetectable: By unauthorized persons. The devices should only respond to authorized readers. This is achieved by the RL and ACL.
- Replay resistant: By using a timestamp the devices resist in previous valid authentications.
- Global, group and individual authentication: A CA should authorize one or a group of readers to access one or a set of IMDs. This is useful for normal and emergency situations.
- One-time or sustained authentication: A reader can be authorized for a limited or an unlimited period of time.
- Audit trail: The device should keep a record of privileged readers.
- Revocation: Each IMD should keep a revocation list.

This is only a theoretical work, so no implementation or simulation exists. For further information see [13]. In our work we do not use a CA scheme because we have decided not to use an asymmetrical-cryptographic system for the reasons explained in subsection 2.3.2 and in sections 3.1 and 3.2. The main reason is that an asymmetrical-cryptographic algorithm is more computational intensive than a symmetrical one. However, the Elliptic-Curve asymmetrical algorithm could be used but this is left for a future work on this project.

2.3.4 Zero-Power Defenses for an IMD

In [11], three zero power defenses against radio based attacks are presented. The defenses are based on RF-power harvesting. The three goals of the authors of [11] are:

1. To detect and prevent attackers using commercial or custom equipment.
2. Not drawing energy from the primary battery for security purposes.
3. Security sensitive events should be effortlessly detectable by the patient.

The three defenses based on RF-power harvesting (RFID is added to the IMD) are:

- **Zero-powered notification for patients:** When potentially malicious activities are taking place the patient is alerted. A prototype was built by using the WISP [15] with an attached piezo-element. A WISP is a small embedded system which contains an RFID circuitry, hence it harvests energy from an RFID reader. Assuming that the WISP with the attached piezo-element is connected with an IMD; if someone tries to communicate with the IMD, the WISP will alert the patient without draining the prime battery.

- **Zero-power authentication:** Verifying that the reader is authorized to gain access. The device implements a simple challenge-response protocol based on RC5-32/12/16. In this model the IMD has a key K and an identity I. All the commercial programmers (readers) have a master key K_M. There exists a relation between the two keys, $K = f(K_M, I)$, where f is a strong cryptographically pseudorandom algorithm. So, the protocol starts by a challenge sent by the programmer. The IMD-using the WISP- responds with its identity I and a random value. The programmer computes K and sends back $R = RC5(K, N)$. Then, the WISP computes R by itself and compares it with the received R. If they are equal the programmer is authenticated.

- **Zero-power sensible-key exchange:** This technique combines both previous techniques. The goal is to distribute a symmetric-cryptographic key over a human-perceptible sensory channel, by the above described method. When the key is distributed the patient is alarmed using the first method. The key is distributed through sound waves which are received by locating a microphone on the skin, because the waves are very weak (hence secured). The key's computation and the authentication is achieved by the *zero-power authentication* method.

The only thing that is missing in [11] is key management. However, the idea of gaining power from the reader to power-up and work the IMD is very interesting. This idea is used in our system too, as it is going to be described in section 3.1.

2.3.5 Comparison of mode-switching methods

Switching-mode schemes are responsible for switching between the normal and the emergency modes. Usually in normal mode the security is on, while in the emergency mode the security is deactivated. The first comparing method is by using Cloakers which has been described in subsection 2.3.2 and the second one is by simply using a magnetic-switch. The advantages and disadvantages of both methods are shown in table 2.2. This comparison is important because our work's switching-mode method has been selected according to it. We have selected the method which uses the magnetic-switch for the following reasons. The system consists of a single device which makes it handy and limits the possible problems. It also balances the security and safety tensions, because in the normal mode the security is activated while in the emergency mode we can deactivate it easily by passing a magnet over it. On the other hand, maybe the magnetic switch is too sensitive to magnetic fields which causes mode switch. A big advantage of the Cloaker is that along with its switching-mode use, it can act as an additional processing unit.

Thus, the Cloaker enables the use of asymmetrical cryptography while in the magnetic-switch case, the power limitation allows only symmetrical cryptography. For instance, the Cloaker can communicate with the reader side with an asymmetrical protocol and with the IMD side with a symmetrical protocol. Additionally, the Cloaker can act as a proxy for monitoring the transferring data. On the other hand the Cloaker approach has disadvantages as well. The system is not handy anymore because it consists of two modules; the implant and the Cloaker. This introduces two practical problems: first, the patient must be aware of the Cloaker's battery and second, because it is an external wearing devices, it reminds to the patient his/her condition. Additionally, the Cloaker adds new possible attacks, such as covering the presence of the Cloaker to the implant (by a Jam DoS) which leads into switching to emergency mode. In section 3.1 the reasons for selecting the magnetic-switch access-control method for our implementation are analyzed further.

	Cloaker	**Magnetic Switch**
advantages	• Only the Cloaker knows the secret key of the IMD which means higher security. • In normal mode the IMD talks only to the Cloaker which enables the usage of symmetric cryptography • The Cloaker can use asymmetrical cryptography for communication with the Readers. • The Cloaker provides us with modularity because we can update the security software and keys easier without changing the IMD settings, but only the Cloaker's. • The Cloaker could be used as a proxy for analyzing the data coming from outside	• The system contains only 1 device. Which means that the IMD is more handy and there are less problems that could occur. • Balances security-safety.
dis-advantages	• The system contains 2 devices. It is not handy and we have to care about 2 things. • The patient has to be aware also about the Cloaker's battery. • Jam DoS, for covering the presence of the Cloaker. • The patient will be reminded about his condition.	• Maybe the magnetic switch is too sensitive. • Because a symmetric encryption algorithm is used, the doctor must store all the keys of his patients who use these implants. This would not happen in an asymmetrical system where the doctor would only need his own pair of keys.

Table 2.2: Advantages and Disadvantages of switching-mode methods encountered in the literature.

2.4 Symmetrical Cryptographic Algorithm selection

In this section the selection of the MISTY1 cryptographic algorithm is presented. First, we describe a profiling study on symmetrical encryption algorithms which concludes that the MISTY1 algorithm is the most efficient of the evaluated ones. Then, we describe the MISTY1 algorithm.

2.4.1 Profiling of Symmetric-Encryption Algorithms

In [28], a profiling study has been done on symmetric-encryption algorithms. The decision, of making the profiling on symmetric algorithms and not on asymmetric, was taken based on the fact that the symmetric algorithms are less power-, energy- and area-demanding than asymmetric ones. Such parameters are very important because this profiling has been done in the context of implantable medical devices. The authors of [28] aimed at achieving the following goals:

- To identify symmetric-encryption algorithms which consume the lowest average power.
- To identify symmetric-encryption algorithms which need the lowest energy to encrypt variable text sizes.
- To identify symmetric-encryption algorithms which can encrypt at the highest rate or -at least- to satisfy the sampling rate of the biological-plaintext data.
- To find the instruction mixes and frequencies of the most efficient algorithms in order to have a first idea of how to design a processor architecture for implant security.
- To identify the safest algorithms by using the well known security margin metric.
- To identify the less demanding algorithms in terms of binary-code size.

The compared symmetric ciphers were the folowing: 3WAY, BLOWFISH, DES, GOST, IDEA, LOKI91, RC5, SKIPJACK, XXTEA, MISTY1, RC6, TWOFISH, RIJNDAEL. The final results are shown in table 2.3.

av. power consumption	peak power consumption	total energy cost	encryption efficiency	encryption rate	program-code size	security margin
IDEA	IDEA	RC6	RC6	RC6	XXTEA	GOST
LOKI91	MISTY1	RC5	IDEA	RC5	3WAY	MISTY1
SKIPJACK	LOKI91	IDEA	RC5	MISTY1	LOKI91	BLOWFISH
MISTY1	TWOFISH	MISTY1	MISTY1	RIJNDAEL	RC6	IDEA
RIJNDAEL	RIJNDAEL	BLOWFISH	RIJNDAEL	BLOWFISH	RC5	RC5

Table 2.3: Five best-performing encryption algorithms (in descending order of performance). Source from [28].

By interpreting the results in table 2.3 the writers of [28] concluded that the best algorithm for an IMD is MISTY1, because is the only one that appears in 6 out of 7 metrics and it captures good positions.

2.4.2 Description of MISTY1

In our work we rely on the profiling method presented in [28] and use MISTY1 as our symmetrical encryption algorithm. MISTY1 has been designed in 1995 by Mitsuru Matsui and others for Mitsubishi Electric. The main characteristics of MISTY1 are the following:

- Type: symmetric, block cipher.
- key size: 128 bits.
- Block size: 64 bits.
- Rounds: There is a variable number of rounds used for the MISTY1 algorithm. However 8-rounds is recommended and used in most cases. That's why we also use 8-rounds.

MISTY1 can be divided in two parts. The first part is the "key scheduling part", where the algorithm gets as input the 128-bit key and expands it into 256-bit key. The second part is the "data randomizing part", where the algorithm gets 64-bit of data and encrypts them. In figure 2.5, the flowchart of encrypting a single block with MISTY1 is shown. It receives as input a 64-bit block and divides it into two 32-bit blocks $d0$ and $d1$. The encryption result is written into $d0$ and $d1$. In the flowchart in figure 2.5 we see the $fo()$ and $fl()$ functions which encrypt and randomize a data block.
For more informations see [1].

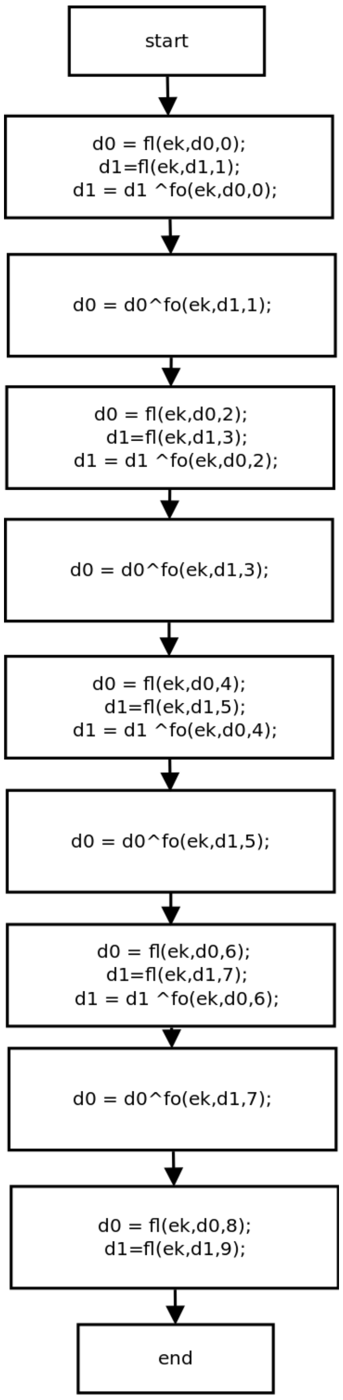


Figure 2.5: MISTY1 flow-chart.

2.5 Conclusions

In this chapter, the background and related work of the thesis, were presented. First, some essentials on security theory, were described. Then we continued by presenting the

SiMS project, which includes the current work. Then we showed some interesting works on secure IMDs and a comparison of two switching-mode methods. We decided to select the magnetic-switching switching-mode approach. None of these works have proposed a new processor architecture for securing an IMD. A new secure processor architecture is this work's challenge. A custom processor can lead in power, energy and area decrease. In the next step we presented a profiling study on symmetric-cryptographic algorithms. Symmetric algorithms instead of asymmetric were chosen, because they are usually less computationally complex and consequently less power consuming. This profiling indicated that the MISTY1 algorithm is the most efficient one, thus we selected it to be our work's cryptographic algorithm. MISTY1 was also described in this chapter. As we have mentioned the Elliptic-Curve asymmetrical algorithm which is not computationally intensive could be reconsidered in a future work of this project.

3 Proposed Secure Implantable System

Achieving security for an implant can be divided in two parts. The first part, as in every secure system, is to make the system tolerant to attacks. The second part, which is not so frequent in other secure systems, is that the implant must be power and energy efficient. High power and energy consumption and a miscalculation in the security protocol could lead to a patient's death. Thus, we can understand how important the system and the communication-protocol selection is, which is the purpose of this chapter. Specifically, the chapter's purpose is the suitable selection of the system architecture, the communication protocol, the cryptographic algorithm and the ISA of the implemented co-processor, in order to result in a secure and power-efficient system. In order to select or build a secure system we should start by listing the possible attacks that such a system could face. For our implantable system these attacks are shown below. The attacks that we deal with are shown in boldface text:

- **Impersonation of the implant or of a reader in order to prevent treatment.**
- **Altering the messages during communication.**
- **Stealing and reading the messages (personal data).**
- **Battery DoS attack.**
- Jam DoS attack.
- The doctor alters the saved data in the implant when he/she makes a mistake, to avoid responsibility.
- The doctor denies that he/she did a specific action to the implant which finally leaded to unpleasant results.

The above selection does not mean that the rest of the attacks are not important, but according to restrictions in power and area (because of the implant's requirements), we must choose only some of them. Hence, taking into consideration the tight limitations on power and area and a looser limitation on performance, we conclude that we need to cover the essential attacks which are the ones that have to do with power and energy consumption. After getting power, area and timing results of a basic design which covers the above attacks, it is going to be evaluated whether it is possible to implement more defenses. For now, we assume that the doctor is always a trusted person and a Jam DoS attack shall not occur. The majority of this work is taking place in the presentation layer of the OSI model, where encryption and decryption algorithms are used.

The remainder of this chapter is organized as follows: Section 3.1 describes the system architecture which is selected based on the knowledge acquired in chapter 2. Section 3.2 presents the communication protocol using again the knowledge of chapter 2. Section 3.3 presents the communication scenarios between the implant and a malicious or not reader. This analysis gives strong hints towards implementing implant security for a co-processor system as will be explained later on. In section 3.4 a co-processor baseline design and its optimized implementations are described. Finally, in section 3.4 the coprocessor's compiler implementation is presented.

3.1 System Architecture

The system architecture and the communication protocol are strongly related to each other. A decision taken in the system architecture can make the implementation of the protocol more difficult or easier and the other way around. This means that the system architecture and the protocol are worked on concurrently. However, in order for the thesis to be readable we have tried to separate them in two different parts. In this section we describe our decisions upon the system architecture. With respect to the communication protocol we make the important decision to use a symmetrical algorithm. The reason is that a symmetrical cipher algorithm is significantly less computationally intensive, as we have mentioned in section 2.1.1, thus less power consuming. Consequently, this enables importing the whole security protocol in the IMD, without the help of a Cloaker. In the asymmetrical case, the Cloaker could lighten the security-workload of the implant.

Energy efficiency: A main issue when one builds security for an implant is the energy consumption of the security part. It is not acceptable for the implant to run out of battery because of the security-protocol's execution. A battery-DoS attack on the implant can drain its battery. For instance, when someone tries to connect to the implant, the implant runs the authentication protocol, thus consumes energy. If that is repeated for a long time by a malicious or not reader the battery will end. The first part of the solution to that problem is to use the idea from work [11], where the IMD has an RFID attached to it and receives the RF energy of the reader in order to communicate. The second part of the solution is to partition the implantable system in two modules. The first module is the main processor of the implant, which is responsible for the implant's main functionality and operates with battery energy. The second module is a co-processor, which is responsible for the communication between the implant and the outside world and operates with the induced RF energy. In order for that idea to work, a co-processor implementation is needed which operates in the power limits that the RF provides. The reasons for this implant separation are two. First, the security-part should not use the same resources as the main-functionality of the implant. That is, because it is not acceptable that the main functionality program (it is life-critical) stalls for the sake of the security program. Second, we do not want the security part to drain the main-implant battery. Running out of battery energy would lead to potentially life-

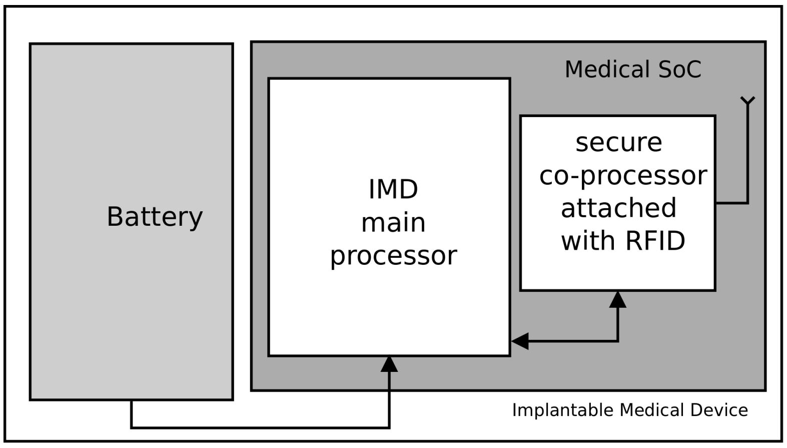

Medical SoC
Battery
IMD main processor
secure co-processor attached with RFID
Implantable Medical Device

Figure 3.1: IMD energy efficient system architecture.

threatening situations. In the above we described an energy efficient IMD architecture which is shown in figure 3.1.

Mode-switching methods: After solving the energy issue, we move to the next important issue which is the mode-switching procedure. An implantable medical device should have at least two operation modes. The first mode is the *normal mode* where the security protocols are activated. The second mode is the *emergency mode* where the security protocols are deactivated for safety reasons. For instance, if a patient who carries an IMD passes-out, a doctor -even if he does not have or obtain the cryptographic-key- should be able to switch the device off. In the literature -as we have mentioned in section 2.3- we found two approaches for switching modes. The first one is by using a Cloaker as is suggested in [19] (see figure 3.2). The second one is by simply using a magnetic switch (see figure 3.3). In the Cloaker approach, the IMD works in the normal mode when the Cloaker is present, which means that it blocks any direct connection attempt from a reader to the IMD. Switching to the emergency mode is done by removing the Cloaker. In table 2.2 were shown the advantages and disadvantages of both designs. The purpose of figures 3.2 and 3.3 is to show the different setups and not the detailed protocols. The protocol of the selected scheme is shown and described in detail in section 3.2. Taking into account the advantages and disadvantages of the two schemes, we select the magnetic-switch approach because it is simpler to design and more secure in the normal mode (because it consists of only one device). For instance, an important issue in the Cloaker approach is how to make the implant aware of the Cloaker's presence. Additionally, if someone jams the communication channel between the implant and the Cloaker, then their connection will be lost. This will result in switching to the emergency mode since the implant will not be aware of the Cloaker presence. These problems do not appear when a single device is used. Also by implementing the magnetic-switch approach, it is relatively easy to jump to the Cloaker one, because the communication protocol between the IMD and the Cloaker in some parts could be the same with the direct IMD-reader protocol. The following hold for the variables shown in the figure 3.2:

- ***SKr***: private-key of the reader.
- ***PKr***: public-key of the reader.
- ***SKc***: private-key of the Cloaker.
- ***PKc***: public-key of the Cloaker.
- ***Kc-r***: session-key between the Cloaker and the reader (if it is needed)
- ***Ki-c***: symmetrical-key for the IMD-Cloaker communication

Additionally the following holds for the variable shown in figure 3.3:

- ***Ki-r***: symmetrical-key for the IMD-Reader communication

Summarizing, the selected system architecture is presented in figure 3.1. It is shown that the IMD is partitioned in: the main-function processor and the communication co-processor. The energy source for the former is the battery and for the latter is the induced RF-energy of the reader. The contribution of that scheme is that the security and communication part of the implant is not draining the prime battery, hence there is no fear of unexpected battery's drop.

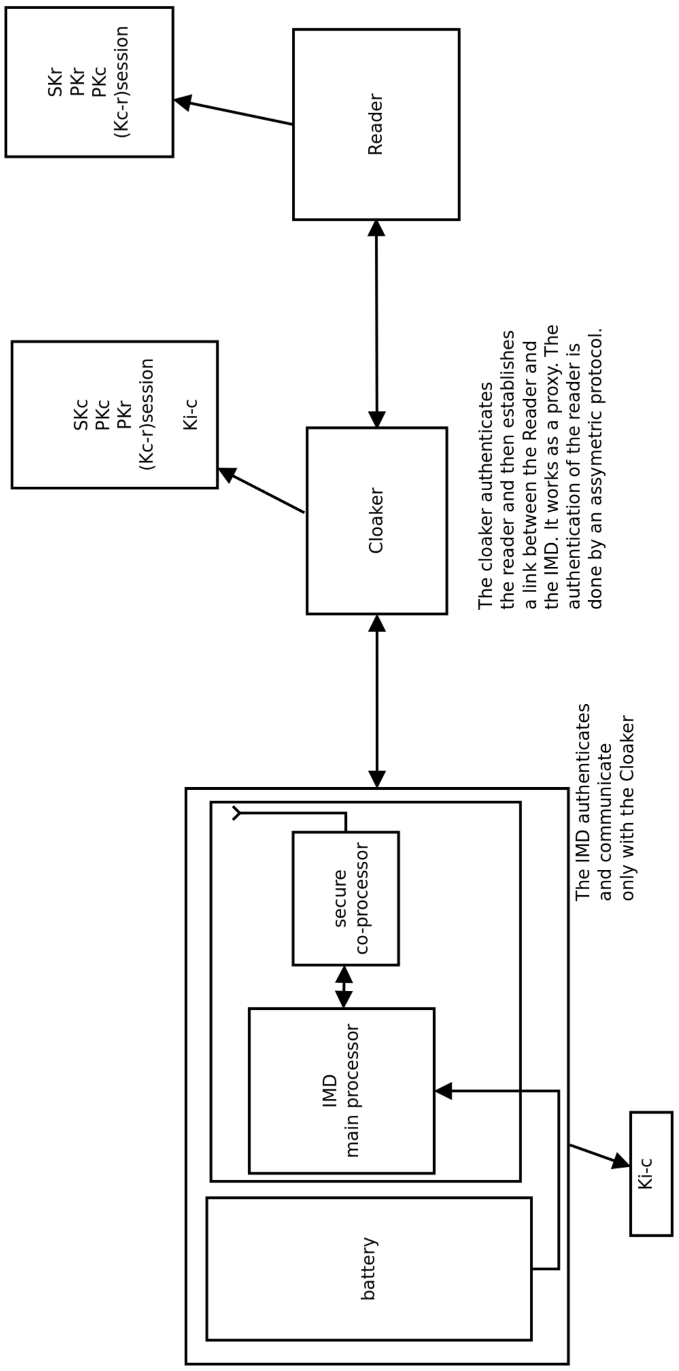

SKr
PKr
PKc
(Kc-r)session
Reader
SKc
PKc
PKr
(Kc-r)session
Ki-c
Cloaker
The cloaker authenticates the reader and then establishes a link between the Reader and the IMD. It works as a proxy. The authentication of the reader is done by an assymetric protocol.
The IMD authenticates and communicate only with the Cloaker
secure co-processor
IMD main processor
battery
Ki-c

Figure 3.2: Cloaker approach

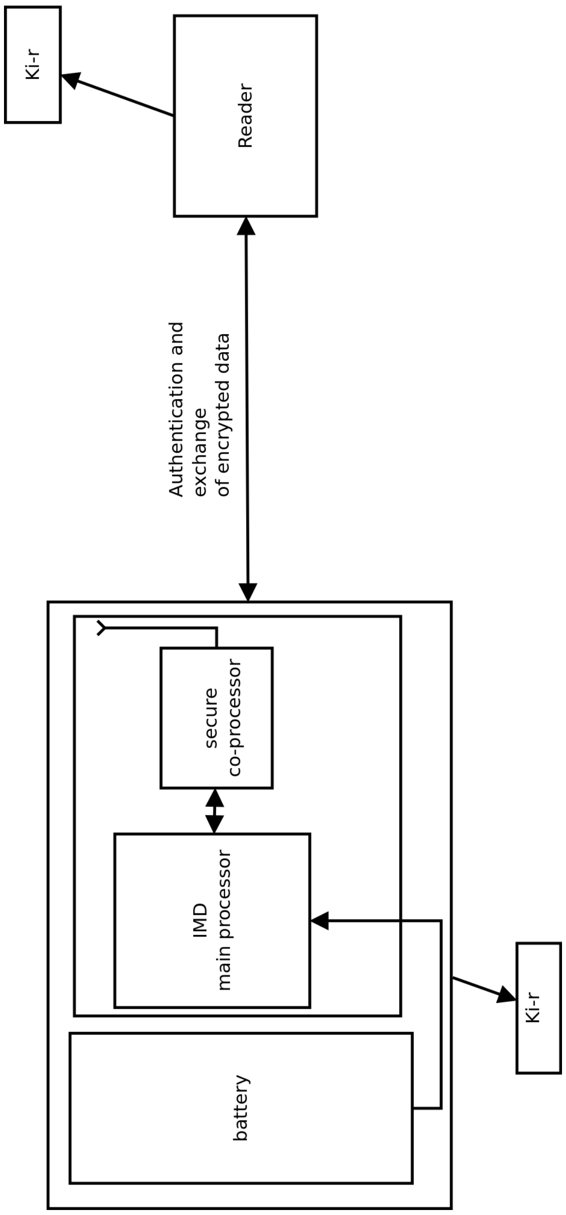


Figure 3.3: Magnetic-switch approach

3.2 Security Approach

In this section, we analyze the communication protocol based on the selected scheme (magnetic switch, figure 3.3) in section 3.1. That scheme solves the energy DoS-attack problem, thus the protocol must take care of the remaining attacks which have been stated in the beginning of the chapter. As we have already said in section 3.1, we choose for a symmetrical system, which means that we use a symmetrical algorithm for encryption (one shared secret key). The realistic assumption of having only a small number of valid readers (maybe 3: doctor, patient, relative of the patient), along with the low processing requirements of such systems indicate the use of a symmetrical algorithm. In the following subsection we present the alternative designs for each goal shown in table 2.1 along with our design choices. The alternative designs shown in table 2.1 are for a symmetrical-security.

3.2.1 Symmetrical-System alternative designs

In table 3.1, we present the possible choices for every layer of a security system. The final decisions for our system appear in boldface text. A brief explanation of our decisions is given next. First, in a symmetrical system the key distribution can be done by a distribution center. For our system it is obligatory to use an off-line distribution system, because the on-line requires communication with the distribution center. Moreover, in our case there are only three readers which all have the same privileges and use the same key. So, there is no reason for a more complicated key-distribution scheme. Finally, replacement of the key is not needed unless a reader is compromised.

The next step is the selection of the entity-authentication and message-integrity methods. We have decided to cover both by selecting a message-authentication protocol. We describe how data integrity and entity authentication can be achieved by using the proposed scheme. The following hold for the proposed message authentication scheme and the protocols described in figure 3.4:

- ***A* and *B***: Are the entities.
- ***Msg***: Is the transmitted message.
- ***MsgR***: Is the message which has been sent by the reader.
- ***MsgIM***: Is the message which has been sent by the IMD.
- ***CRC(X)***: Is the CRC result of data *X*.
- ***eK(X)***: Is the encrypted result of data *X*.
- ***len(X)***: Is the length in bytes of data *X*.
- ***RN***: Is a random number.
- ***count***: Is the result of a counter. The encrypted *count* can hold for random number.

Key Management	• **We always need an initial key distribution.** • **Off-line distribution system**/ on-line distribution system (distribution of session key). (This actually describes who can communicate with whom) • Replacement/**No replacement**
Entity Authentication	• One side authentication (2-way challenge response protocol). • Mutual authentication (3-way challenge response protocol). • Trusted Party.
Data Integrity	• A sends to B a message M, and then B sends it back for verification. • Hash function.
Message authentication (combination of entity authentication and data integrity)	• MAC (Message authentication Code if the two sides trust each other). • Digital signature (if the two sides doesn't trust each other). • **Application of a hashing algorithm (eg. CRC), RN generation, followed by encryption with the message.**
Confidentiality	• **Encryption.**

Table 3.1: Security System.

Assume that A sends $eK(Msg//RN//CRC(Msg//RN))$ to B. Then B:

- can **authenticate** A because: The only one who could encrypt the data with that specific key is A. Also the $CRC(Msg//RN)$ part ensures us that the decrypted data are not garbage. B is going to decrypt the $ek((Msg//RN)'//CRC(Msg//RN))$ and then he is going to calculate $CRC((Msg//RN)')$, on his own. If $CRC((Msg//RN)') = CRC(Msg//RN)$, then the data are valid.
- can be sure of **data integrity**: As we said previously, B is going to calculate if

$CRC((Msg//RN)') = CRC(Msg//RN)$. If this holds then B is sure that the message has not been altered.

The above scheme description without the RN (random number) would be vulnerable to replay-attacks. A replay-attack is a network attack in which a valid data transmission is maliciously or fraudulently repeated or delayed. A replay-attack could lead the implant to reply and send information to a non-valid person. The RN is a number which is different in every message for ensuring no repetition of a message.

Finally, as it is shown in table 3.1, confidentiality is achieved by using a symmetrical cryptographic algorithm.

3.2.2 Communication Protocol

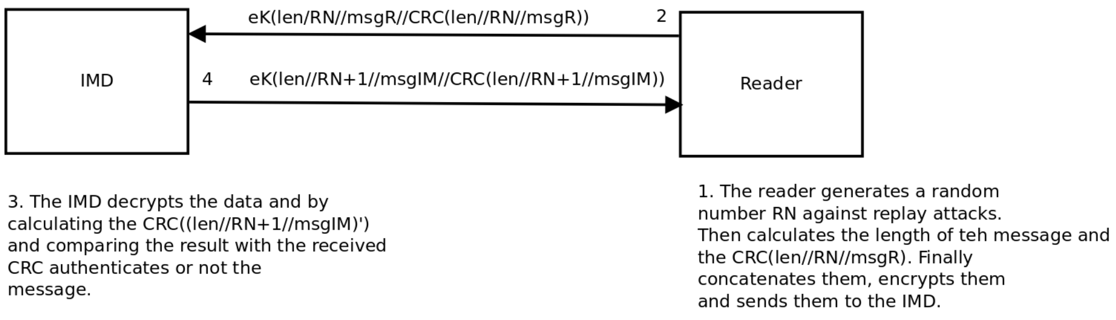
In this subsection, we present in detail our communication protocol according to the design choices made in subsection 3.2.1. We discuss two protocols which are presented in figure 3.4. Their difference is that in the protocol shown in figure 3.4(b) a modification related with the RN generation has been made which did it more efficient in terms of memory and energy. Their comparison is shown in table 3.4. That table lists the advantages and disadvantages of the protocols presented in figure 3.4. According to that table we choose to implement the protocol depicted in figure 3.4(b) , because the memory requirements -in the protocol shown in figure 3.4(a) - for keeping log of RNs would be very large for our IMD system. Clearly, this comes with the RN computation overhead and the possibility of a Jam-DoS attack by repeated connection efforts. Jam-DoS attacks should be studied further in a future work of this project.

In table 3.2, we present the functionalities of the IMD and the reader sides of the protocol shown in figure 3.4(a). Respectively, in table 3.3, we present the functionalities of the IMD and the reader sides of the protocol shown in figure 3.4(b).

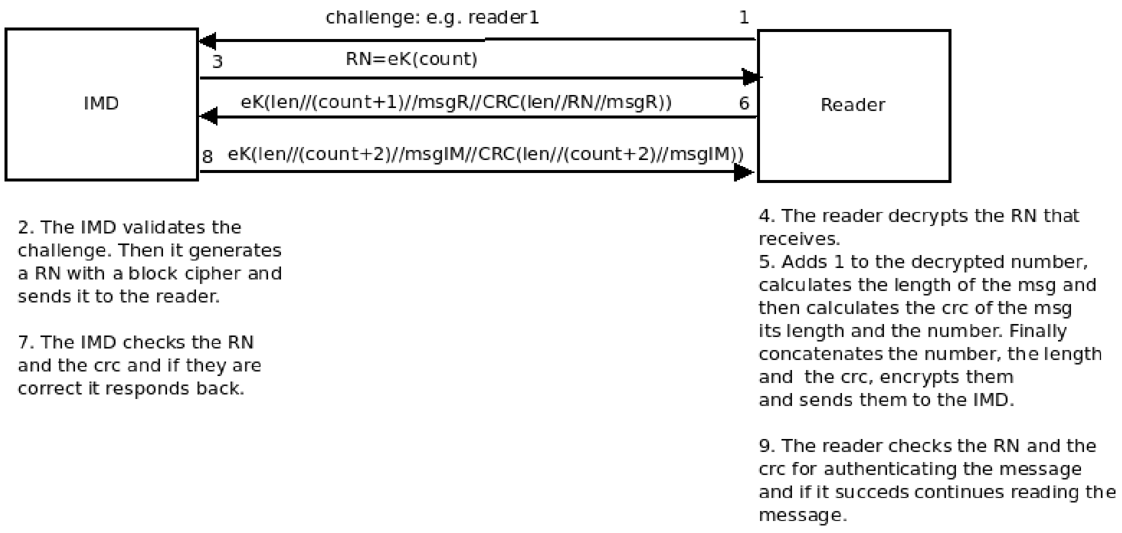
Before continuing we should define which hashing and symmetrical encryption algorithms we use. The symmetrical algorithm is the block cipher MISTY1 for the reasons explained in 2.4.1. The hashing algorithm we want to use is a CRC-variant, which is not computationally intensive. Now remains to specify which CRC we use. The CRC selection depends on the size of the data we want to transfer. The relationship is the following: If r is the degree of the CRC polynomial then the maximal total blocklength is equal to $2^r - 1$. For example, if we want to transmit a message of length 1KB=8192 bits then we need a polynomial with: $2^r - 1 \geq 8192 \Rightarrow r \geq 14$.
Hence, in that case the CRC-16 (17 bits) or the CRC-32 (33 bits) would be accepted CRC algorithms. We have selected to use the CRC-32 hash function because for our application a 10KB transmission is also possible.

3.2.3 Message and command formats

In this subsection we define the formats of the exchange messages. Table 3.5 shows these formats. As wee can see, the message consists of the length of the data followed by a RN, followed by the data followed by the CRC of the sent message. When the message is sent from the IMD side the data (3rd field of the message format in table 3.5) is the medical history of the patient (information). When the message is sent from a reader to


IMD
Reader
2 eK(len/RN//msgR//CRC(len//RN//msgR))
4 eK(len//RN+1//msgIM//CRC(len//RN+1//msgIM))
3. The IMD decrypts the data and by calculating the CRC((len//RN+1//msgIM)') and comparing the result with the received CRC authenticates or not the message.
1. The reader generates a random number RN against replay attacks. Then calculates the length of teh message and the CRC(len//RN//msgR). Finally concatenates them, encrypts them and sends them to the IMD.

(a) Protocol where the RN is generated by the reader


challenge: e.g. reader1 1
3 RN=eK(count)
eK(len//(count+1)//msgR//CRC(len//RN//msgR)) 6
8 eK(len//(count+2)//msgIM//CRC(len//(count+2)//msgIM))
IMD
Reader
2. The IMD validates the challenge. Then it generates a RN with a block cipher and sends it to the reader.
7. The IMD checks the RN and the crc and if they are correct it responds back.
4. The reader decrypts the RN that receives.
5. Adds 1 to the decrypted number, calculates the length of the msg and then calculates the crc of the msg its length and the number. Finally concatenates the number, the length and the crc, encrypts them and sends them to the IMD.
9. The reader checks the RN and the crc for authenticating the message and if it succeds continues reading the message.

(b) Protocol where the RN is generated by the IMD

Figure 3.4: Communication Protocols.

the IMD the data is a command and the command format is shown in table 3.5 as well. Each field of the command format is described in table 3.6

3.2.4 Software Implementation

After determining the protocol, we proceeded with its implementation. The implementation is done by using the C programming language and is executed on the XTREM-simulator platform. XTREM is a performance and power simulator for the Intel XScale microarchitecture (see [14]). We selected the XTREM simulator because it was available, suitable, accurate and was used by previous work on the SiMS project. Additionally, it is easy of use.

In figure 3.5 we present the flow-chart of our implementation. In this flow-chart is shown that the program starts and ends with the construction and destruction of the keys. Probably, in the real system the key-construction and destruction-functions should run only once because the software of the protocol should not end after accomplishing

IMD SIDE	READER SIDE
• Receives a message and authenticates it by the following: • Decrypts the command. • Checks if the RN has already been used. • Checks the CRC (message authentication completed). • The implant executes the command which can be either sending information to the reader or execute an action (switch off, switch on, reset...). • If the implant is asked to send back information, which is the most usual, it does the following: • Create a message with a command, its length and the RN. • Calculates the CRC of that message. • Encrypts all the data. • Sends them to the reader. The information can be of the size of 1KB or 10KB. We can understand that this is the most processing and time consuming part.	• Generates a RN. • Create a message with a command, its length and the RN. • Calculates the CRC of that message. • Encrypts all this data with a symmetrical encryption algorithm. • Sends the data to the implant. • If the reader asked for information he receives them. • Decrypts the data using MISTY1 (this is the most time and processing consuming part because there are a lot of data).

Table 3.2: Functionality of the protocol shown in figure 3.4(a)

a single communication. It should only get in sleep mode, until it is awakened by an RF signal from a reader. However, for our profiling purposes we wrote a software which at some point ends by necessity. On the one hand, we would get more accurate results if we had removed the key construction and destruction functions. On the other hand their execution does not affect much the selection of a representative scenario since the same execution, time and energy overhead is present in all scenarios. As we will see in section 3.3 we have defined 20 communicating scenarios on which we did a profile study in order to select the most representative one in terms of power and energy to be our main benchmark for our designing processor.

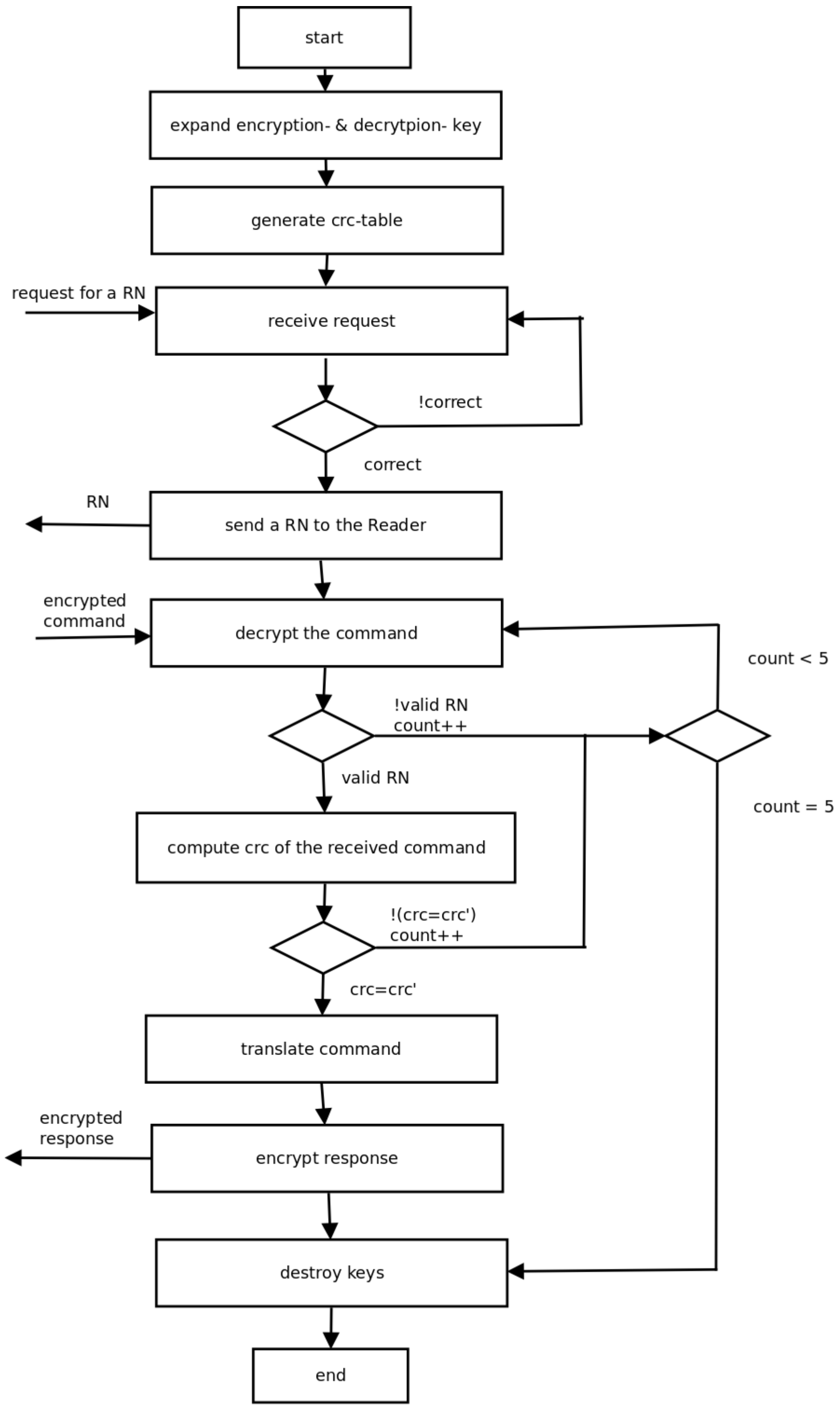


Figure 3.5: Flow Chart of the protocol shown in figure 3.4(b).

IMD SIDE	**READER SIDE**
• Receives challenge. • Check if the challenge was correct. • Generate a random number with a block cipher. • Send the respond containing the RN. • Receive the encrypted command from the reader. • Decrypts the command. • Check the RN. • Check the CRC (after this the message authentication has been achieved). • The implant executes the command which can be either sending information to the reader or execute an action (switch off, switch on, reset...). • If the implant is asked to send back information which is the most usual, it does the following: • Calculates the CRC of that message. • Encrypts all the data. • Sends them to the reader. The information can be of the size of 1KB or 10KB. We can understand that this is the most processing and time consuming part.	• Sends challenge. • Receives respond (RN). • Decrypts the respond. • Creates a message with a command, its length and the RN. • Calculates the CRC using CRC32 of that message. • Encrypts the whole message using a symmetrical algorithm. • Sends the message to the implant. • If the reader asked for information he receives them. • Decrypts the data using MISTY1 (this is the most time and processing consuming part because there are a lot of data).

Table 3.3: Functionality of the protocol shown in figure 3.4(b)

	advantages	**disadvantages**
Protocol depicted in figure 3.4(a)	The RN is generated by the reader which means less processing requirements for the implant.	All the RNs should be stored in the memory of the implant for comparing them with the new ones. This requires too much execution time for comparisons and additional memory for storing the RNs (area increase).
Protocol depicted in figure 3.4(b)	The implant does not have to keep track of the RNs since they are generated by it, so there is no memory overhead for storing the RNs. Furthermore, as RN generator we use the encryption algorithm of the cryptographic system. Thus, we do not need additional software -or hardware- for RN generator.	• The Implant has a slight time-execution overhead to calculate the RN. • The implant can become a victim of a Jam-DoS attack by someone who sends challenges continuously (Becoming ideal for a while after some fault challenges could be a solution).

Table 3.4: Advantages and Disadvantages of the Communication Protocols.

message format

length of data(4B)	RN(4B)	data(command or information from the implant)	CRC(4B)

command format

Data mode (1b)	source sensor (4b)	length of data asked (3b)
State mode (1b)	state (4b)	

Table 3.5: Message format.

Data-request cmd: 0
State-change cmd: 1
If Data-request mode is selected:
Source Sensor:
1xxx: asking data from all the sensors
0000: BP
0001: AEP
0010: ECGI
0011: EEGI
0100: EMGII
0101: PFI
0110: RCI
Size of data:
000: 1 KB
001: 10 KB
100: ALL
If the change-state mode is selected
command:
0000: power down
0001: power up
0010: rst
0011: sleep
0100: system check
0101: sensor calibration

Table 3.6: Explanation of the command format in table 3.5

3.3 Attack and Normal Scenarios

In this section, the attack and normal scenarios are defined and have been implemented in C. Scenarios are the use-cases which the described communication-protocol shown in figure 3.4(b) can face. The communication protocol has been implemented at it is shown in subsection 3.2.4. By using the scenarios as inputs to the implemented protocol, we have created a number of benchmarks. These benchmarks were executed on the XTREM simulator which provided us with important profiling (energy, power and performance) measures. These measures led us to select the most representative benchmark (scenario) in terms of power and energy, which is used as the main benchmark for our designing processor presented in section 3.4. The measures and the scenario-selection procedure are shown in section 4.1.

We have tried our scenarios to cover as many cases as possible, and furthermore not to overlap with each other in order to keep their number limited. The idea was to divide the scenarios in the following way:

- unsuccessful scenarios
- successful scenarios where the IMD transmits 1KB of data
- successful scenarios where the IMD transmits 10KB of data

First, for defining the unsuccessful scenarios we should be aware of the problems that could corrupt the normal flow of the protocol. These problems are stated below.
Wrong challenge:

- wrong challenge size
- unexpected challenge data

Wrong command:

- wrong command size
- wrong Random Number
- wrong CRC

These possible problems are checked in the above order. The scenarios are the following:

Scenarios which lead to unsuccessful transactions: The reader asks for 1 or 10 KB (the size does not matter in the unsuccessful scenarios)

- **scenario1:** challenge(x) $\rightarrow$ invalid data eg. $reader4$
- **scenario2:** challenge(x)$\rightarrow$ challenge with smaller size eg. $rea1$
- **scenario3:** challenge(x) 6 times$\rightarrow$ timeout eg. $reader1 * 6$
- **scenario4:** challenge(v) command(x)$\rightarrow$ smaller command (like scenario2)
- **scenario5:** challenge(v) command(x)$\rightarrow$ unexpected RN

- **scenario6:** challenge(v) command(x)$\rightarrow$wrong CRC

Scenarios which lead to successful transactions: The reader asks for **1KB**

- **scenario7:** challenge(v) command(v) $\rightarrow$ normal eg. $reader1$ $RN//1//00000000//CRC$
- **scenario8:** challenge(v) challenge*4(x) command(v) $\rightarrow$ DoS attack or many readers try to connect $reader1 * 5$ $RN//1//00000000//CRC$
- **scenario9:** command(x) challenge(v) command(v) $\rightarrow$ $kkkk$ $reader1$ $RN//1//00000000//CRC$
- **scenario10:** challenge(x) challenge(v) command(v) $\rightarrow$ non existing challenge eg.$reader4$ $reader1$ $RN//1//00000000//CRC$
- **scenario11:** challenge(x) challenge(v) command(v)$\rightarrow$ smaller challenge size eg. $read1$ $reader1$ $RN//1//00000000//CRC$
- **scenario12:** challenge(v) command(x) command(v) $\rightarrow$wrong RN eg. $reader1$ $RN - 1//1//00000000//CRC$ $RN//1//00000000//CRC$
- **scenario13:** challenge(v) command(x) command(v) $\rightarrow$wrong CRC eg. $reader1$ $RN - 1//1//00000000//wrongCRC$ $RN//1//00000000//CRC$

The reader asks for **10KB**
In that case the scenarios are the same as above but instead of the command being 00000000 it is 00000001. These scenarios are the following: **scenario14, scenario15, scenario16, scenario17, scenario18, scenario19, scenario20**.

The simulation of the receiving and sending procedure in the above scenarios has been done by using files. The receiving procedure was simulated by reading a text file. Respectively, the sending procedure was simulated by writing into a file. The profiling results are shown and analyzed in chapter 4. According to these results we have selected the worst behaving scenarios in terms of power and energy. We have used this scenario to drive to a co-processor design. Regarding a profiling study of this scenario on the baseline co-processor (which we designed), we proceeded to optimizations of the baseline processor. The baseline design and its optimizations are shown in section 3.4.

3.4 Security Coprocessor

In this section, we describe our baseline processor design and its two optimizations. The optimizations are done for achieving faster execution of our benchmark (less execution cycles), which subsequently decreases the total energy. As will be shown later, this comes at the cost of a small increase in power and area. First, we present the baseline-architecture characteristics, its Instruction Set Architecture (ISA) and its datapath. Then, the ISAs of the optimized designs follow. We have implemented our processor by using the Processor-Designer tool of CoWare, which is described further in the Appendix. The Processor Designer combines C and a tool-native language in order to describe the hardware. After compiling, the tool generates synthesizable VHDL. The main convenience that the Processor Designer provides is the fast and simple hardware description. Additionally, it is very easy to add or remove instructions, without the need to change the whole datapath, as it would happen in the case of VHDL. However, this convenience comes at the cost of not having a detailed hardware description. So, it is obvious that this tool is good for determining an ISA for a processor and to have a first estimate of power, energy, area, and delay metrics. After concluding on an ISA, if the system has tight power/energy/timing requirements, probably it would be a good idea to rewrite the processor in VHDL. Furthermore, the Processor-Designer tool provides us with the means for an easy assembler and disassembler implementation.

Processor Specifications: Now, let's continue on to the architecture characteristics of our processor. Since we wanted our design to be low-power, we have chosen to implement a Reduced Instruction Set Computing (RISC) architecture. Furthermore, we have selected to design a 5 pipeline-stages processor which consists of the Instruction Fetch stage (FE), the Decode stage (DEC), the Execution stage (EX), the Memory stage (MEM) and the Write-Back stage (WB). This co-processor pipeline design is shown in figure 3.6. It consists of a 16-bit instruction set along with 16 registers of 32-bits each. These specifications are used very often in embedded processors, as it is shown in the AppendixJ of [20]. In our case, the use of 32-bits registers is important because the encryption and decryption algorithms of our benchmark are operating on 32-bits quantities. Finally, the data-memory size is 16KB and consists of 4K address space, while the program-memory is also 16KB and consists of 8K address space. For keeping the processor low-power, we have not included floating-point units and forwarding. Additionally, the selected branch-prediction scheme is the very simple "always not taken" one.

The baseline instruction formats are shown in table 3.7 and the baseline ISA in table 3.8. From the ISA, can be easily observed that it is a very simple RISC processor, without multiplication, division or modulo units, which would result in large area amount and would consume a lot of power. If the benchmark needs to do such an expensive operation, the compiler will take care of it, as we will see in section 3.5.

Processor ISA: In the instruction set we can see the following logical operations: *and, or, not, xor, sftl, sftr* and *sftru.* A *xor* unit is not an essential logic unit for a simple GPP, but our benchmark uses it a lot as it will be seen in chapter 4. The shifting operations are very important, because they are used a lot by the compiler for type casting and for other procedures as is presented in section 3.5. However, we see that there is no shifting-instruction by a register amount (no rr-type), but only by a fixed immediate. This would be a problem if in our benchmark appeared a case like $a = b << c$, but it does not. We also have the essential arithmetic operations: *add, sub, addi* and *subi.* We could do without *addi* and *subi* instructions but there are a lot of cases where *addi* and *subi* appear, with most frequent the stack and frame pointers movement. Thus, if we did not have them we should use two instructions instead of one, every time (eg. $addi \rightarrow li, add$). Then we have the very useful comparison operations *se, sgt, sgtu and beqz* (*beqz* is also a branch instruction). These instructions are used for implementing the if-else statements as we will see in section 3.5. Afterwards, we see the load and store instructions *lw, sw* and *lb.* We see that the *sb* instruction is missing

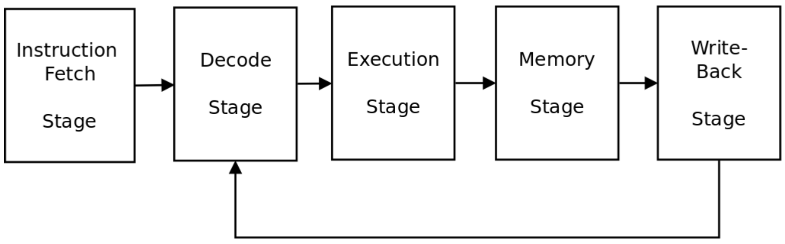


Figure 3.6: Datapath's abstract view.

since it cannot be implemented because we can only read and write a whole word (32-bits) to the data memory. For assisting the compiler in storing a single byte we have implemented the *cb* instruction. This instruction gets the *rd* register and changes its $rs2^{th}$ byte (the 2 LSb of *rs2* indicates the byte that will be changed) by the LSB of *rs1*. The next very important instruction is *li*, which stores an 8-bit immediate to a register. We can realize the drawback of loading only 8-bits every time, because a 32-bit quantity will need calling 1 *li*, 3 *addi* (is used for loading the remaining 3 bytes), plus the corresponding left shifting. Finally, we have the jump instructions *j, jal* and *jr*. The j instruction should be used when we want to jump to an address which can be illustrated by 12-bit (4K address space), but when the address is larger than 4K then the *jr* should be used that can potentially branch to the address 4G-1. Additionally, *jr* is important for returning from a function. The return address is stored in register 15 (*r15*). *Jal* is responsible for jumping to a function and storing its address to register 15. We would also need a *jalr* if we wanted to have the capability to branch to a function, which is located into an address larger than 4K. However, by running our benchmark, we have realized that all the functions were located into the 4K address space despite the fact that our benchmark used more addresses. This happened because in C all the functions are defined above the main function. Thus the *jalr* instruction is not needed for this design and this benchmark, but for a future design where the program size is larger it should be included into the ISA.

By studying table 3.7 with the instruction formats we can see that there is the potential for adding more instructions. This can be done by using the remaining unused opcodes for single instructions or even for new instruction formats. These free opcodes are recommended when designing a processor, because many times some extra instructions, either for optimizations or for forgotten instructions are needed.

In figure 3.6, an abstract view of our 5-stage processor is shown. This consists of the following stages: Instruction-Fetch (IF), Decode (DEC), Execution (EX), Memory (MEM) and Write-Back (WB). In figures 3.7, 3.8, 3.9, 3.10, 3.11 the design of each stage is presented respectively. The datapath is very abstract, because the hardware has been described with the Processor-Designer tool, which uses the abstract hardware-description language LISA. A brief explanation of every stage in the pipeline follows.

Instruction Fetch Stage: In this stage the instructions are fetched. The program memory consists of 16KB and it has a 4K address space (12 bits). In front of the memory's address input, a multiplexer is located, which selects between the program

Baseline Formats																
Formats:																
rrr_type	opcode				rd				rs1				rs2			
rr_type	opcode				rd				rs1				funct			
ri_type	opcode				rd				imm							
jump_type	opcode				imm											
nop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rr_type:																
beqz	0	0	0	1	rd				rs				0	0	1	1
sgtu	0	0	0	1	rd				rs				1	1	1	1
sgt	0	0	0	1	rd				rs				1	1	1	0
se	0	0	0	1	rd				rs				1	1	0	1
lb	0	0	0	1	rd				rs				1	0	1	1
jr	0	0	0	1	rd				rs				1	0	1	0
and	0	0	0	1	rd				rs				0	0	1	0
lw	0	0	0	1	rd				rs				1	0	0	0
mov	0	0	0	1	rd				rs				0	1	1	1
not	0	0	0	1	rd				rs				0	1	0	0
or	0	0	0	1	rd				rs				0	1	0	1
sw	0	0	0	1	rd				rs				1	0	0	1
xor	0	0	0	1	rd				rs				0	1	1	0
sub	0	0	0	1	rd				rs				0	0	0	1
add	0	0	0	1	rd				rs				0	0	0	0
ri_type:																
subi	0	1	1	1	rd				imm							
addi	0	1	1	0	rd				imm							
li	0	1	0	0	rd				imm							
sftru	0	1	0	1	rd				imm							
sftl	0	0	1	1	rd				imm							
sftr	0	0	1	0	rd				imm							
rrr_type:																
cb	1	1	0	1	rd				rs1				rs2			
jump_type:																
jal	1	0	0	1	imm											
j	1	0	0	0	imm											

Table 3.7: Baseline co-processor Instruction Formats

counter (PC), the branch address or the previous address. The multiplexer selects: the PC when the execution is normal, the branch address when a branch is taken and the previous address when a data dependency is detected by the processor. The multiplexer's selection is a relation of the signals branch_taken and stall_en. Stall_en becomes 1 when a data dependency has been realized which is checked in the control_unit of the decode

name	format	assembly	action
jr	rr	jr rd	branch to addr in rd
and	rr	and rd,rs	rd $\leftarrow$ rd and rs
lw	rr	lw rd,rs	rd $\leftarrow$ mem[rs]
sw	rr	sw rd,rs	mem[rs] $\leftarrow$ rd
mov	rr	mov rd,rs	rd $\leftarrow$ rs
not	rr	not rd,rs	rd $\leftarrow$ not rs
or	rr	or rd,rs	rd $\leftarrow$ rd or rs
xor	rr	xor rd,rs	rd $\leftarrow$ rd xor rs
sub	rr	sub rd,rs	rd $\leftarrow$ rd-rs
add	rr	add rd,rs	rd $\leftarrow$ rd+rs
lb	rr	lb rd,rs	rd $\leftarrow$ mem[rs] (byte)
se	rr	se rd,rs	if(rd==rs) then rd$\leftarrow$1 else rd$\leftarrow$0
sgt	rr	sgt rd,rs	if(rd>rs) then rd$\leftarrow$1 else rd$\leftarrow$0(sgn)
sgtu	rr	sgtu rd,rs	if(rd>rs) then rd$\leftarrow$1 else rd$\leftarrow$0(uns)
beqz	rr	beqz rd,rs	if(rd==0) then branch to Imem[rs2]
subi	ri	subi rd,imm	rd$\leftarrow$rd-imm
addi	ri	addi rd,imm	rd$\leftarrow$rd+imm
li	ri	li rd,rs	rd$\leftarrow$imm
sftl	ri	sftl rd,rs	rd$\leftarrow$rd<<imm
sftr	ri	sftr rd,rs	rd$\leftarrow$rd>>imm(sign extension)
sftru	ri	sftru rd,rs	rd$\leftarrow$rd>>imm(zero extension)
cb	rrr	cb rd,rs1,rs2	exchanges the $rs2^{th}$-byte of rd by the LSB of rs1
j	jump	j imm	branch to Imem[imm]
jal	jump	jal imm	branch to Imem[imm] and r15$\leftarrow$PC+1

Table 3.8: Baseline processor ISA

stage.

Decode Stage: In this stage the instruction which has been fetched in the previous stage is decoded. For the decoding procedure the information in table 3.7 is used. The register file (RF) can read 3 registers concurrently and write to 1. The addresses of operand1, operand2 and operand3, are located in IR(11 downto 8), IR(7 downto 4) and IR(3 downto 0) respectively. The destination address is the same with operand1.

The writing-data and address-signals are coming from the WB stage. After the RF are located two multiplexers. The top one chooses which data is going to be opr1. The selection is done between the data in register rd_addr1 (rrr-type, rr-type, ri-type) and the branch address (jump-type). The bottom one produces opr2 and chooses between the data in register rd_addr2 (rrr-type, rr-type) and an 8-bit immediate value (ri-type). Then, we have the very important control unit which produces signals informing:

- if there is a branch pending or taken (br_pending, branch_taken),
- if this instruction is going to write or read from the memory in memory stage (wr_mem_en, rd_mem_en) -and if it is going to read a word or a byte (byte_mem_en)-,
- if there was any data dependency (stall_en), what kind of operations will be executed in the execution stage (alumode) and finally,
- if this instruction is going to write in the RF in WB stage (wr_en).

The EX_alumode signal (coming from the execution stage), informs us if a *beqz* instruction is executing, while the equality_z (also from the execution stage) signal informs us if the comparison in *beqz* is true. If it is true then the branch_taken signal is asserted.

Execution Stage: In this stage we see 2 main components: the ALU component and CB component. The ALU component gets as inputs the operands opr1, opr2 and the alumode from the decode stage and returns a result. The ALU component consists of the following units: an adder, two subtracters, a NOT, a XOR, an AND, an OR, comparators, a left shifter, a right-signed shifter and a right-unsigned shifter. In a VHDL implementation we would make one component for subtraction and addition but that is not possible with the Processor-Design tool. The second subtracter is for subtracting 0x2000 from the data address. This is because the linker understands the program and data memory as one memory, with the data memory located above the program memory. On the other hand, in our processor design the two memories are separated. The CB (unit for the **cb** instruction) component gets as inputs all the operands opr1, opr2, op3 and uses them to produce the result as is explained in table 3.8 at the *cb* instruction.

Memory Stage: The memory stage handles the cases of the instructions *lw, lb* and *sw*. The *lw* and *lb* read 32-bit data from a specific address of data memory. On the other hand, the *sw* writes data to that location. The address is provided by EX_opr2 and the data by Ex_opr1. If there is a *lb*, then the whole word is read from memory and then the desired byte of this word is kept. This is done in the component after the data-memory.

Write-Back Stage: In the write-back stage the processor computes the signal to feed the RF. The top multiplexer selects the data field if the current instruction is a load or otherwise the MEM_res. The bottom multiplexer selects number 15 if the current instruction was a *jal*, which needs to write the PC into register 15, otherwise chooses the MEM_opr_des.

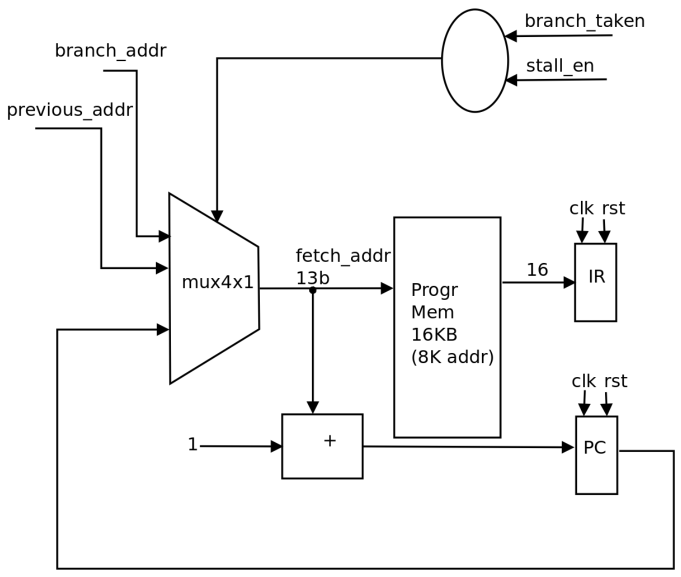


Figure 3.7: Instruction Fetch stage.

Optimizations Now, we move to the optimization phase. We have implemented two optimizations. For the first one we have inserted a very simple modulo unit in our processor. The reason for selecting that optimization was the large number of modulo operations present in the benchmark as shown in chapter 4. Subsequently we have observed that all the modulo operations are a power of 2, hence the implementation becomes very easy as is shown next. Additionally, this optimization decreases considerably the number of execution cycles at the cost of a slight increase in area. We are taking advantage of the rule which says that if we want to calculate a $mod = a\%2^k$, then our result is the k LSb of a. The implementation is done in the following way. Assuming the instruction, $mod = opr1\%opr2$ then:

- $tmp = opr2 - 1$. With that operation we create a quantity with 1s in the k LSb. $k = log(opr2)$.
- $mod = opr1\&tmp$. With this operation we keep the k LSb of a, which is actually our result.

For implementing the *mod* (modulo) instruction, we have selected to add new dedicated hardware instead of using the existed subtracter and AND units. We did that because we did not want to make our stage slower by adding multiplexers before the AND unit, in which we should feed the result of the subtraction. Table 3.9 shows the instruction formats of optimization1 design and table 3.10 presents the optimization1 ISA.

The second optimization has been done by using the collapsing technique. This means to implement a group of operations into a single operation. This procedure consists of finding the single-instruction mix (instructions frequency) along with the pair-instruction mix of the processor. By pairs we mean sequential instructions that have data dependencies between them. As shown in section 4.2, the *mov-and-beqz* group is the most

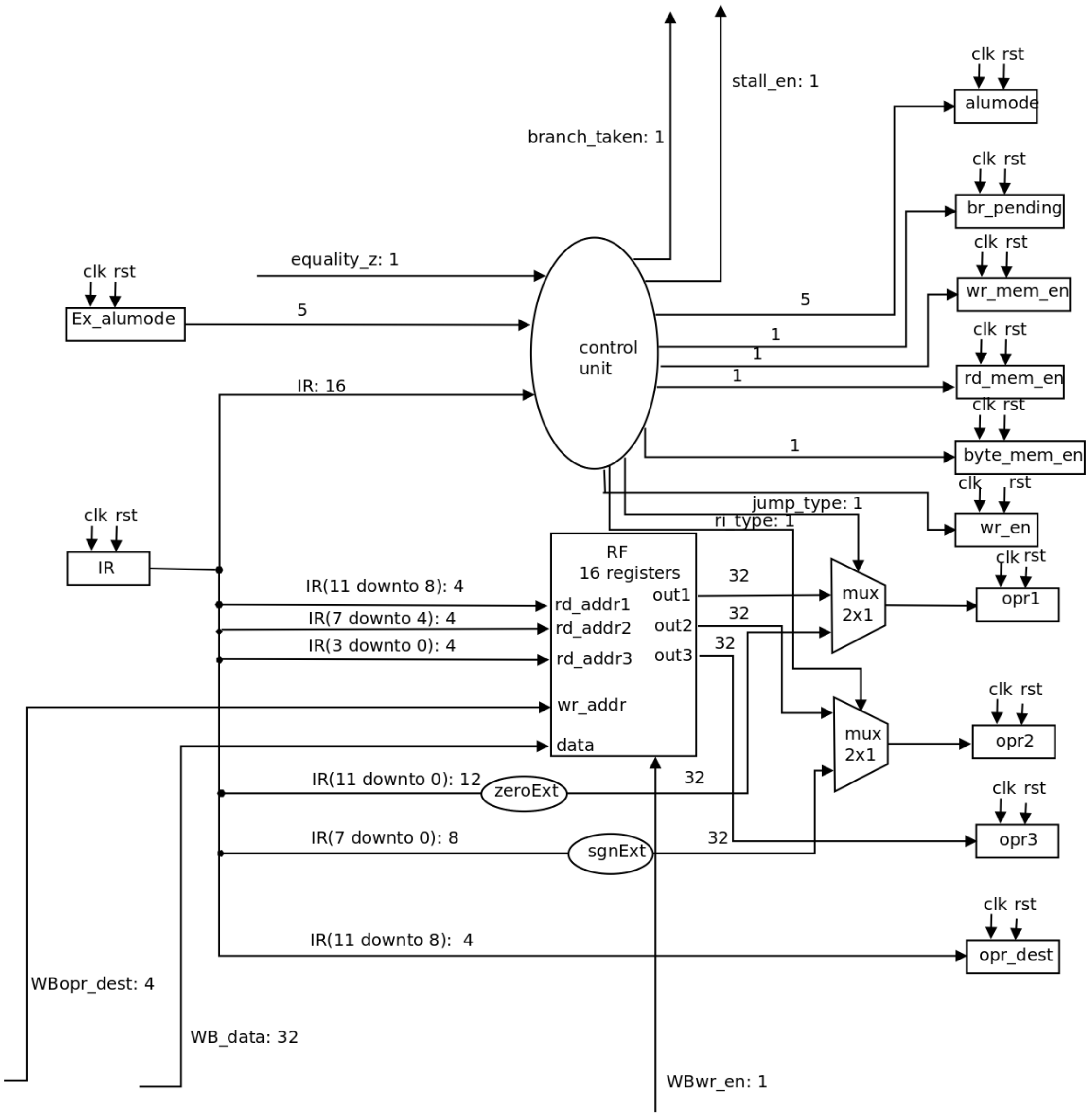


Figure 3.8: Decode stage.

frequent one. The reason for such a high frequency is the big amount of multiplications, where this sequence appears. The implementation in hardware has been done again by adding dedicated hardware, which means one more AND unit and one more comparator are needed. Table 3.11 shows the instruction formats of optimization1+2 design and the table 3.12 presents the optimization1+2 ISA. This section has described our processor-design procedure. The compiler-design of the current processor ISAs follows.

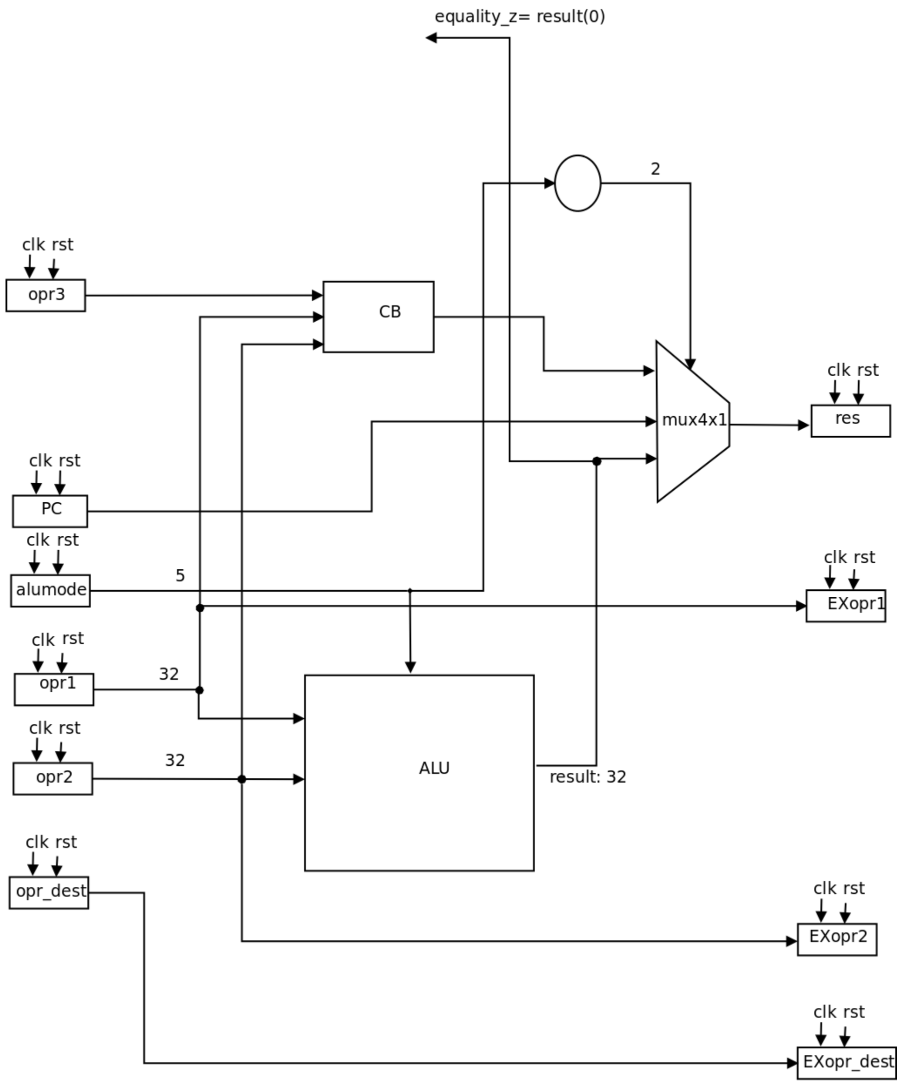


Figure 3.9: Execution stage.

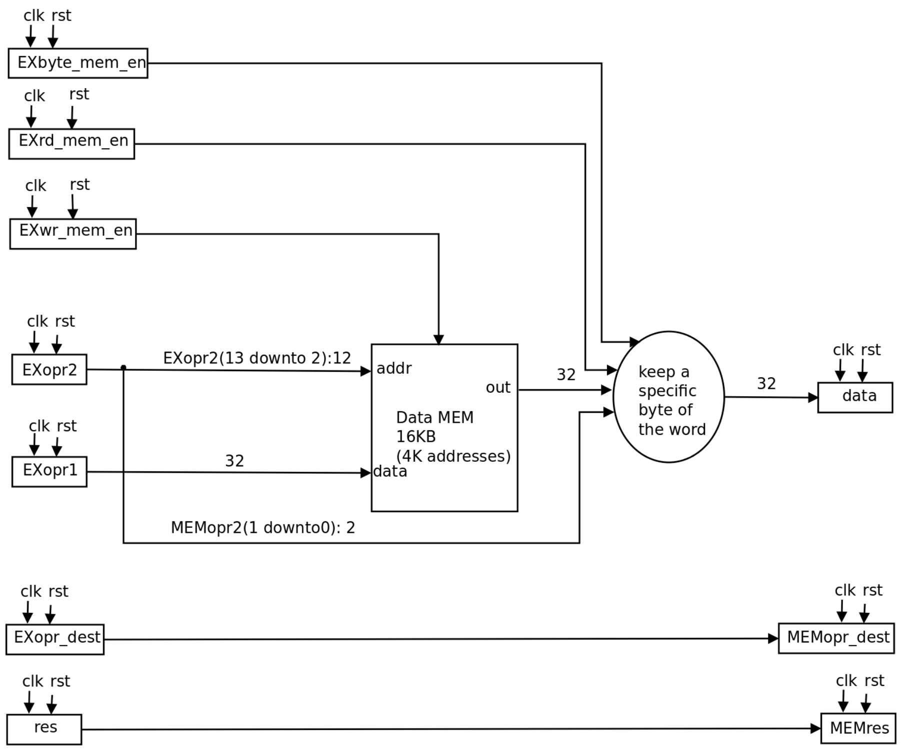


Figure 3.10: Memory stage.

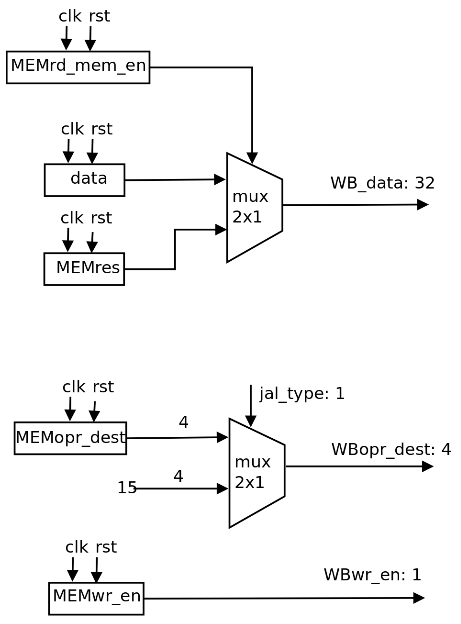


Figure 3.11: Write-Back stage.

Optimization 1 Formats																
Formats:																
rrr_type	opcode				rd				rs1				rs2			
rr_type	opcode				rd				rs1				funct			
ri_type	opcode				rd				imm							
jump_type	opcode				imm											
nop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rr_type:																
beqz	0	0	0	1	rd				rs				0	0	1	1
sgtu	0	0	0	1	rd				rs				1	1	1	1
sgt	0	0	0	1	rd				rs				1	1	1	0
se	0	0	0	1	rd				rs				1	1	0	1
lb	0	0	0	1	rd				rs				1	0	1	1
jr	0	0	0	1	rd				rs				1	0	1	0
and	0	0	0	1	rd				rs				0	0	1	0
lw	0	0	0	1	rd				rs				1	0	0	0
mov	0	0	0	1	rd				rs				0	1	1	1
not	0	0	0	1	rd				rs				0	1	0	0
or	0	0	0	1	rd				rs				0	1	0	1
sw	0	0	0	1	rd				rs				1	0	0	1
xor	0	0	0	1	rd				rs				0	1	1	0
sub	0	0	0	1	rd				rs				0	0	0	1
add	0	0	0	1	rd				rs				0	0	0	0
mod	0	0	0	1	rd				rs				1	1	0	0
ri_type:																
subi	0	1	1	1	rd				imm							
addi	0	1	1	0	rd				imm							
li	0	1	0	0	rd				imm							
sftru	0	1	0	1	rd				imm							
sftl	0	0	1	1	rd				imm							
sftr	0	0	1	0	rd				imm							
rrr_type:																
cb	1	1	0	1	rd				rs1				rs2			
jump_type:																
jal	1	0	0	1	imm											
j	1	0	0	0	imm											

Table 3.9: Optimization 1 co-processor Instruction Formats

name	**format**	**assembly**	**action**
jr	rr	jr rd	branch to addr in rd
and	rr	and rd,rs	rd $\leftarrow$ rd and rs
lw	rr	lw rd,rs	rd $\leftarrow$ mem[rs]
sw	rr	sw rd,rs	mem[rs] $\leftarrow$ rd
mov	rr	mov rd,rs	rd $\leftarrow$ rs
not	rr	not rd,rs	rd $\leftarrow$ not rs
or	rr	or rd,rs	rd $\leftarrow$ rd or rs
xor	rr	xor rd,rs	rd $\leftarrow$ rd xor rs
sub	rr	sub rd,rs	rd $\leftarrow$ rd-rs
add	rr	add rd,rs	rd $\leftarrow$ rd+rs
lb	rr	lb rd,rs	rd $\leftarrow$ mem[rs] (byte)
se	rr	se rd,rs	if(rd==rs) then rd$\leftarrow$1 else rd$\leftarrow$0
sgt	rr	sgt rd,rs	if(rd>rs) then rd$\leftarrow$1 else rd$\leftarrow$0(sgn)
sgtu	rr	sgtu rd,rs	if(rd>rs) then rd$\leftarrow$1 else rd$\leftarrow$0(uns)
beqz	rr	beqz rd,rs	if(rd==0) then branch to Imem[rs2]
mod	rr	mod rd,rs	rd$\leftarrow$rd%rs (rs must be a power of 2)
subi	ri	subi rd,imm	rd$\leftarrow$rd-imm
addi	ri	addi rd,imm	rd$\leftarrow$rd+imm
li	ri	li rd,rs	rd$\leftarrow$imm
sftl	ri	sftl rd,rs	rd$\leftarrow$rd<<imm
sftr	ri	sftr rd,rs	rd$\leftarrow$rd>>imm(sign extension)
sftru	ri	sftru rd,rs	rd$\leftarrow$rd>>imm(zero extension)
cb	rrr	cb rd,rs1,rs2	exchanges the $rs2^{th}$-byte of rd by the LSB of rs1
j	jump	j imm	branch to Imem[imm]
jal	jump	jal imm	branch to Imem[imm] and r15$\leftarrow$PC+1

Table 3.10: Optimization 1 co-processor ISA

Optimization 2 Formats																
Formats:																
rrr_type	opcode				rd				rs1				rs2			
rr_type	opcode				rd				rs1				funct			
ri_type	opcode				rd				imm							
jump_type	opcode				imm											
nop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rr_type:																
beqz	0	0	0	1	rd				rs				0	0	1	1
sgtu	0	0	0	1	rd				rs				1	1	1	1
sgt	0	0	0	1	rd				rs				1	1	1	0
se	0	0	0	1	rd				rs				1	1	0	1
lb	0	0	0	1	rd				rs				1	0	1	1
jr	0	0	0	1	rd				rs				1	0	1	0
and	0	0	0	1	rd				rs				0	0	1	0
lw	0	0	0	1	rd				rs				1	0	0	0
mov	0	0	0	1	rd				rs				0	1	1	1
not	0	0	0	1	rd				rs				0	1	0	0
or	0	0	0	1	rd				rs				0	1	0	1
sw	0	0	0	1	rd				rs				1	0	0	1
xor	0	0	0	1	rd				rs				0	1	1	0
sub	0	0	0	1	rd				rs				0	0	0	1
add	0	0	0	1	rd				rs				0	0	0	0
mod	0	0	0	1	rd				rs				1	1	0	0
ri_type:																
subi	0	1	1	1	rd				imm							
addi	0	1	1	0	rd				imm							
li	0	1	0	0	rd				imm							
sftru	0	1	0	1	rd				imm							
sftl	0	0	1	1	rd				imm							
sftr	0	0	1	0	rd				imm							
rrr_type:																
cb	1	1	0	1	rd				rs1				rs2			
mandb	1	1	0	0	rd				rs1				rs2			
jump_type:																
jal	1	0	0	1	imm											
j	1	0	0	0	imm											

Table 3.11: Optimization1+2 co-processor Instruction Formats

name	**format**	**assembly**	**action**
jr	rr	jr rd	branch to addr in rd
and	rr	and rd,rs	rd $\leftarrow$ rd and rs
lw	rr	lw rd,rs	rd $\leftarrow$ mem[rs]
sw	rr	sw rd,rs	mem[rs] $\leftarrow$ rd
mov	rr	mov rd,rs	rd $\leftarrow$ rs
not	rr	not rd,rs	rd $\leftarrow$ not rs
or	rr	or rd,rs	rd $\leftarrow$ rd or rs
xor	rr	xor rd,rs	rd $\leftarrow$ rd xor rs
sub	rr	sub rd,rs	rd $\leftarrow$ rd-rs
add	rr	add rd,rs	rd $\leftarrow$ rd+rs
lb	rr	lb rd,rs	rd $\leftarrow$ mem[rs] (byte)
se	rr	se rd,rs	if(rd==rs) then rd$\leftarrow$1 else rd$\leftarrow$0
sgt	rr	sgt rd,rs	if(rd>rs) then rd$\leftarrow$1 else rd$\leftarrow$0(sgn)
sgtu	rr	sgtu rd,rs	if(rd>rs) then rd$\leftarrow$1 else rd$\leftarrow$0(uns)
beqz	rr	beqz rd,rs	if(rd==0) then branch to Imem[rs2]
mod	rr	mod rd,rs	rd$\leftarrow$rd%rs (rs must be a power of 2)
subi	ri	subi rd,imm	rd$\leftarrow$rd-imm
addi	ri	addi rd,imm	rd$\leftarrow$rd+imm
li	ri	li rd,rs	rd$\leftarrow$imm
sftl	ri	sftl rd,rs	rd$\leftarrow$rd<<imm
sftr	ri	sftr rd,rs	rd$\leftarrow$rd>>imm(sign extension)
sftru	ri	sftru rd,rs	rd$\leftarrow$rd>>imm(zero extension)
cb	rrr	cb rd,rs1,rs2	exchanges the $rs2^{th}$-byte of rd by the LSB of rs1
mandb	rrr	mandb rd,rs1,rs2	tmp$\leftarrow$(rd and rs), if (tmp==0) then branch to Imem[rs2]
j	jump	j imm	branch to Imem[imm]
jal	jump	jal imm	branch to Imem[imm] and r15$\leftarrow$PC+1

Table 3.12: Optimization1+2 co-processor ISA

3.5 Compiler Design

A compiler is a program that accepts as input a program text in a specific language and translates it into another program text. The input language is called source language and the output is called target language (see book [2]). In the current work, we need a compiler to translate our benchmark -which is written in C- to the machine code of our co-processors. Thus, we need a C-compiler to translate the benchmark into the ISA of each of the 3 co-processor. In this section, the compiler implementation of each processor instance is presented. The differences between the compilers are minor, because the ISAs of the processor versions are similar.

The tool we have used for the implementation is the C Compiler Designer. C Compiler Designer is actually a very simplified version of the Cosy compiler designing tool, which is compatible with the Processor Designer tool. The advantage of using this tool is that the designer has to deal only with the back-end part of the compiler. The front-end part is processor independent- so it is created automatically by the Compiler Designer tool- and provides to the back-end an internal representation (IR). Then the back-end uses the IR to generate the processor's assembly. The different phases of the back-end implementation follow:

- **Matching:** Mapping assembly instructions to given C code. This is the most demanding and difficult part of our design and we will explain it in detail later. The matching procedure is the following: The matcher receives an internal representation (IR), in the form of a tree (CCMIR tree). The CCMIR tree consists of the mir (Medium-level Internal Representation) nodes, which can represent an operation (eg. mirPlus, mirDiff) or an object (eg. mirIntConst, mirObjectAddr). The matcher's job is to detect patterns of mir nodes and to map them to assembly. The patterns are detected according to rules, which are specified by the designer.

- **Register Allocation:** Assigning registers to variables and temporary values. In this part we state to the compiler which registers should be allocated and for what purpose. We define that register 14 is the stack pointer (sp), register 13 the frame pointer (fp) and register PC of our processor design is the program counter.

- **Scheduling:** Scheduling the code by changing the instruction order and inserting the corresponding latencies. All the instructions are scheduled in order, without any delay slot. Our processor is responsible for checking the data dependencies and stalling the execution if a dependency appears.

- **Emit:** Writing the code to an assembly file.

From the above, it can be concluded that the compiler back-end consists of the register allocator, the matcher and the scheduler. The matcher and the register allocator are the basic blocks of the back-end, so, we mostly worked on them. We elaborate on the matcher later in this chapter. In the current work we did not use any ready library of C and also we did not use double types (64-bits), which would require two registers to save one value. We did not use doubles because the benchmark did not include such type. Additionally, as we have already mentioned, we did not work on the scheduler and the compiler optimizations, which would lighten significantly the processor's workload. The scheduler and the compiler optimizations could be part of future work on this project. Compiler optimizations are a basic part of the SiMS project because they are transferring work outside the implant, which results into saving power and area. Finally, with good scheduling implementation we can decrease the processor's area by checking the dependencies at the compiler level and not at the hardware. This would remove a lot of multiplexers.

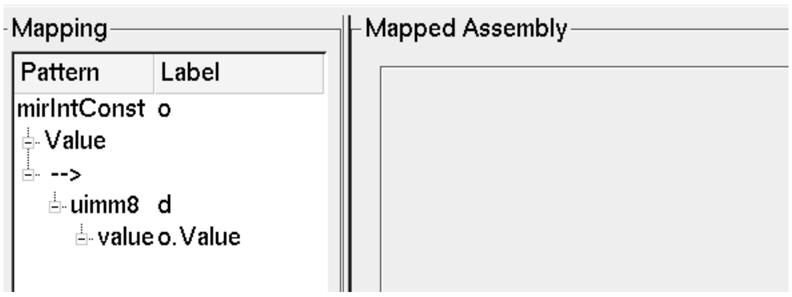
The matcher presentation follows which is the largest part of the compiler work. We begin with the baseline-processor's matcher. The matchers, of optimization1 and optimization1+2 processors designs, only require small modifications which are shown later. First, our matching rules are divided into groups as shown below:

- Object Conversions
- Load and Store→Basic Load/Store, Address Load/Store
- Casts
- Stack Management
- Comparisons
- Control Flow→Function Call Support, Conditional Jumps
- Arithmetic→Common Arithmetic, Address Arithmetic
- Moves

Now follows the explanation of these groups of rules. We do not show and explain every rule, but only some main ones, which are enough for understanding the procedure:

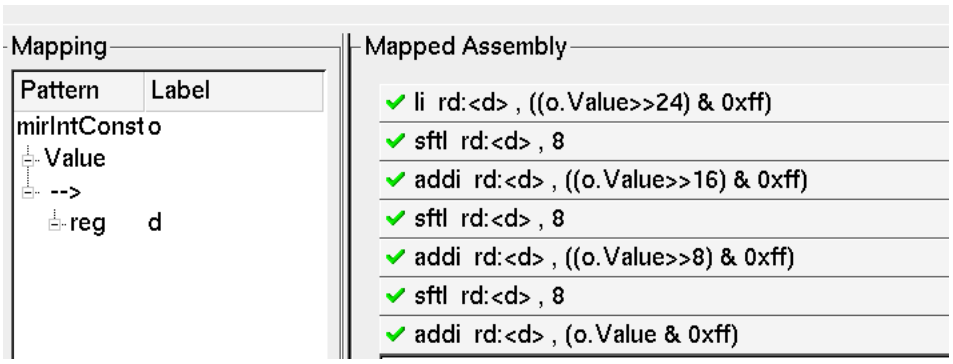
Object Conversions: This group of rules is responsible for transforming some objects into a form, which is useful for the available assembly instructions. Without these transformations many of the instructions could not be used and the compiler would return errors for not achieving to map part of the C code to assembly. The objects can be addresses, constants and non-terminals. Here are presented some rules:

- **Constant to a non-terminal:** These rules are used when a constant has specific characteristics, which are defined in a non-terminal. For instance non-terminals can be an uimmd8 (8-bit unsigned immediate), an uimmd12 (12-bit unsigned immediate), a zero and many others. Hence, when an immediate with the above characteristics is observed, then it is transformed into the relative non-terminal. Figure 3.12 illustrates the conversion rule of a constant -which is 8-bit unsigned- to the uimmd8 non-terminal. This conversion is important to us because we need this non-terminal for *addi* and *subi* instructions. Without this non-terminal in combination with that rule, we could not use the *addi* and *subi* instructions. In the case where no such terminal exists, we should load every constant into a register which means more instructions.



Pattern	Label
mirIntConst	o
Value	
-->	
uimm8	d
value	o.Value

Figure 3.12: Covert Constant to unsigned immediate with 8-bit width.



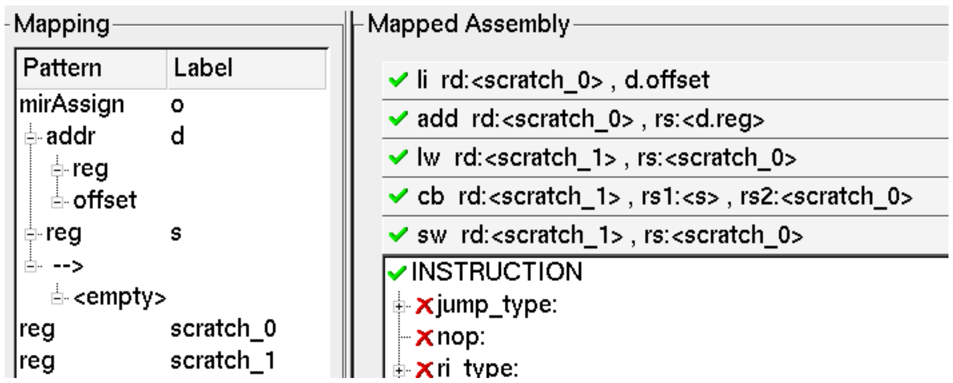
Pattern	Label
mirIntConst	o
Value	
-->	
reg	d

```
✓ li  rd:<d> , ((o.Value>>24) & 0xff)
✓ sftl  rd:<d> , 8
✓ addi  rd:<d> , ((o.Value>>16) & 0xff)
✓ sftl  rd:<d> , 8
✓ addi  rd:<d> , ((o.Value>>8) & 0xff)
✓ sftl  rd:<d> , 8
✓ addi  rd:<d> , (o.Value & 0xff)
```

Figure 3.13: Covert Constant to register.

- **Constant -or non-terminal- to register:** These rules load a constant -or a non-terminal, as the ones mentioned above- into a register. These rules are used, when a command, as $a = 5$, is detected in the C code. The mir node of such command is the mirIntConst (eg. 5), which is linked with the destination register. In figure 3.13 the mapping assembly and the pattern described above are shown . Lets explain a bit the mapping assembly: first a *li* instruction is called, for loading the MSB of the constant into the destination register. Then, follows one byte left shift. Then an other load of the 2nd MSB is done by using an *addi*. The same procedure continues until the whole 32-bit constant is loaded. The reason for converting a non-terminal -for instance uimmd8- into register, is that no instruction can match to a specific mir pattern. For instance, if a *xor* is appeared in C and one of the two operands is uimmd8, we will need to make a non-terminal to register conversion, because a *xori* does not exist in our ISA.
- **Global Address object into register:** In these rules, we follow the same procedure as at the constant-to-register rule. There are different rules for the same thing because the mir-tree creates different nodes for address and constant objects.

Load/Store: This group is responsible for loading -and storing- values from -to- memory. This group is divided into two sub-groups. The one sub-group is for loading


Mapping

Pattern	Label
mirAssign	o
addr	d
reg	
offset	
reg	s
-->	
<empty>	
reg	scratch_0
reg	scratch_1

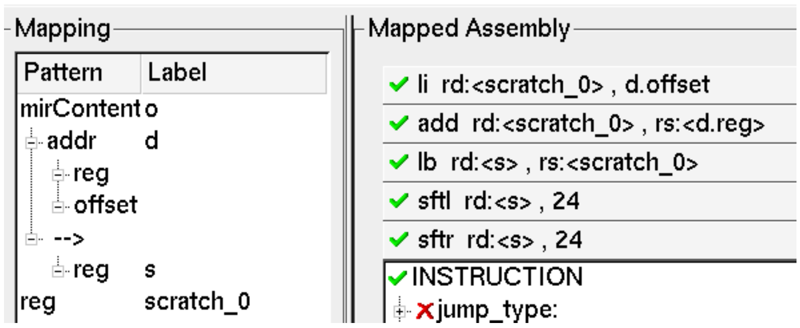
Mapped Assembly

```
✓ li  rd:<scratch_0> , d.offset
✓ add  rd:<scratch_0> , rs:<d.reg>
✓ lw  rd:<scratch_1> , rs:<scratch_0>
✓ cb  rd:<scratch_1> , rs1:<s> , rs2:<scratch_0>
✓ sw  rd:<scratch_1> , rs:<scratch_0>
✓INSTRUCTION
  ✗jump_type:
  ✗nop:
  ✗ri_type:
```

Figure 3.14: Store singed or unsigned byte.

-or storing- by using an address-non-terminal (an address non-terminal consists of a register and an offset), while the other sub-group is responsible for loading and storing by using an implicit address (the whole address is located into a register). The first sub-group is more complicated and for that reason we describe it. It consists of the following rules:

- **Store signed/unsigned word:** In that case the whole register is stored in memory. In storage the values are not divided into signed and unsigned because we transfer the value of a register to an equal or smaller space which means that there is no need for sign or zero extensions. Thus, it does not matter if the value is signed or unsigned.
- **Store signed/unsigned byte:** Here one byte is stored in memory. It is interesting at that point to explain the mapped assembly shown in figure 3.14. When a storage into memory appears at C level, then the mirAssign node appears in the mir-Tree. This node must be connected with an address (destination) and a register (source). The mapping procedure is more tricky because a *sb* instruction does not exist, so we have to achieve that functionality by using other instructions. Hence, we use the *cb* instruction which has been explained in section 3.4. Briefly, this instruction inserts a specific byte into a specific byte location of a register. Hence, the procedure is the following. First, a *li* is needed for loading the address-offset into a scratch register (a scratch register is a temporary register, of which the compiler must be aware in order to allocate it). Then, an *add* instruction is used for adding the scratch register (offset) and the address register. Then, we proceed by loading the whole address, that we want to store the byte, into another scratch register by calling *lw*. Then, the *cb* instruction follows, for inserting the desired byte into the desired byte-location of the previously loaded register (it is loaded again into the 2nd scratch register). Finally, we store the second scratch register into the location specified by the first scratch register.
- **Store signed/unsigned half-word:** Here a half-word is stored in data-memory. In that case we use twice the store signed/unsigned-byte procedure.


Mapping

Pattern	Label
mirContent	o
addr	d
reg	
offset	
-->	
reg	s
reg	scratch_0

Mapped Assembly

✓ li rd:<scratch_0> , d.offset
✓ add rd:<scratch_0> , rs:<d.reg>
✓ lb rd:<s> , rs:<scratch_0>
✓ sftl rd:<s> , 24
✓ sftr rd:<s> , 24
✓ INSTRUCTION
✗ jump_type:

Figure 3.15: Load singed byte.

- **Load signed/unsigned word:** This rule loads a whole word from the memory into a register.
- **Load signed byte:** In the loading procedure, a sign and unsigned discrimination is needed because a quantity in memory might be smaller than the register size (than a word). Hence, a zero extension or a sign extension is needed. In figure 3.15 is shown the sign-byte loading procedure. First, we load the byte which is located in the address non-terminal. Then, we shift left 3 bytes and finally we sign-shift 3 bytes to the right, to complete the sign extension.
- **Load unsigned byte:** The same as above but without the shifts because we only need zero extension.
- **Load signed half-word:** Here, we repeat the same technique as in Load signed-byte.
- **Load unsigned half-word:** Here, we repeat the same technique as in Load unsigned-byte.

Casts: This group is very important because it does all the type conversions. Our compiler supports char (1 byte), short (2 bytes), int (4 bytes) and long (4 bytes) types (signed and unsigned). For instance, the compiler performs a type conversion when the following cases appear:

$unsigned\ int\ a = 0;$
$int\ b = 1;$
$a = b$

or

$unsigned\ int\ a = 0$
$a = 10$ (constants are usually seen as signed integers)

Obviously, there are many other cases where type conversions are preformed.
The conversions are done according to table 3.13 where:

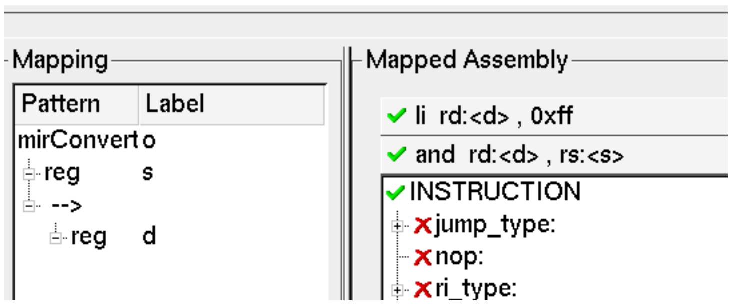


Figure 3.16: Conversion from a signed char/short/int to an unsigned char type.

- SEx: sign extension from bit x to 31.
- ZEx: zero extension from bit x to 31.
- -: identical types do not need conversion.
- DT: Direct translation of value which means no modification of value.

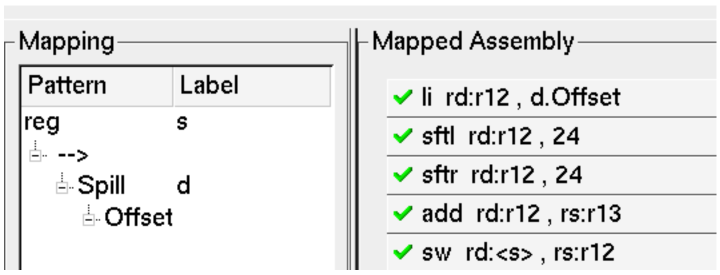
When the compiler detects a type conversion, it adds to the mir-tree the mirConvert node. In figure 3.16, the conversion from a signed char/short/int to unsigned char is presented. The procedure is very simple. In the *s* register exists the value which we want to convert into another type and *d* register is the destination register. First, we load 1s into the 8 LSb of *d*. Then, we *AND d* with *s*. Thus, we have the LSB of *s* in the LSB position in *d* and the rest of the bytes of *d* are zero.

	SINT8	**UINT8**	**SINT16**	**UINT16**	**SINT32**	**UINT32**
SINT8	-	SE8	SE8	SE8	SE8	SE8
UINT8	ZE8	-	ZE8	ZE8	ZE8	ZE8
SINT16	DT	DT	-	SE16	SE16	SE16
UINT16	ZE16	DT	ZE16	-	ZE16	ZE16
UINT16	DT	DT	DT	DT	-	DT
UINT32	DT	DT	DT	DT	DT	-

Table 3.13: Conversions from a size (horizontal) into another size(vertical).

Stack Management: Our stack has as start address the end of the addressable memory space, and it grows downwards (The static data memory starts at the beginning of the memory address space). The stack is responsible for saving the context of each function. For instance, when inside a function is called another function, then some registers (callee changed registers) must be stored in the stack by the caller, in order not to lose any information. Stack management needs a stack-pointer. The use of the frame-pointer is not obligatory but it makes the compiler implementation easier and safer. For that reason we decided to include the frame-pointer into our compiler design. This group consists of the following rules:

- **Save register using the frame pointer:** Let's assume that a function calls (caller) another function (callee). The callee must save all the caller-changed



Pattern	Label
reg	s
-->	
Spill	d
Offset	

```
✓ li  rd:r12 , d.Offset
✓ sftl  rd:r12 , 24
✓ sftr  rd:r12 , 24
✓ add  rd:r12 , rs:r13
✓ sw  rd:<s> , rs:r12
```

Figure 3.17: Save a register in the stack.

registers, which are going to be used, for not losing them. The same must be done by the caller for the callee-changed registers. In such a case, the current rule is used. This rule is presented in figure 3.17 to help the reader understand this group of rules. The Spill non-terminal which is shown at the left of the figure provides us with the offset that must be added to the fp in order to find the storing address in the stack-frame.

- **Reload register using the frame pointer:** Above, we have mentioned that the the callee function must save the caller-changed registers at its beginning (prologue). When the callee function reaches its end (epilogue), must reload the caller-changed registers with their previous values. That is achieved by the current rule.
- **Decrement the stack pointer:** The compiler in every callee function calculates the frame size (the stack size needed for the current function) that the function is going to use. Then, the sp is decreased by the frame-size value. So the sp indicates to the end of the stack-frame space (Hence, the fp points at the start and the sp at the end of the stack-frame space). This decrement is done by the current rule.
- **Save the frame pointer:** At the beginning of the callee function the fp must be saved to the address that the sp currently points, because the fp is going to be moved where the sp is. This movement must be done in order for the fp to indicate at the beginning of the new function (callee). If the old-fp address is not saved, it won't be possible to restore it easily. Storing the fp is done by this rule.
- **Restore the frame pointer:** This rule restores the old fp by loading the value of the address, which the fp points at. This is done in order to return to the previous (caller) function (at the end of the callee function).
- **Moving the stack to the frame pointer:** This rule is also used at the end of the callee function (obviously before restoring the old fp), in order to point at the end of the caller function. This is done by moving the sp to point where the fp indicates.
- **Moving the frame to the stack pointer:** This is done after the old fp has been stored to the stack and before the sp has been decreased.

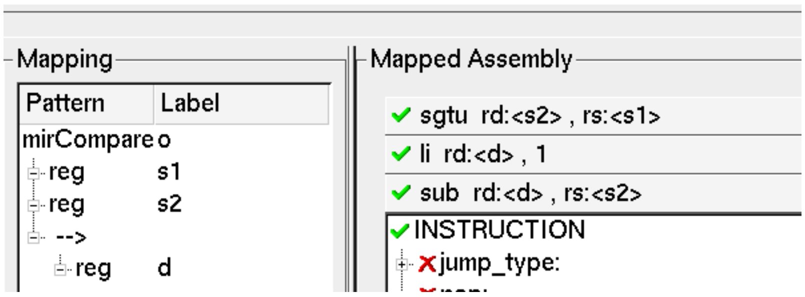
Comparisons: This group of rules is responsible for the comparisons in a program. Comparison results are also used in control-flow statements like if-else. The implemented comparisons between two variables are the following:

- **Equal.**
- **Not equal.**
- **Less-than.**
- **Equal or less-than.**
- **Greater than.**
- **Equal or greater-than.**
- **Less-than (between unsigned quantities).**
- **Equal or less-than (between unsigned quantities).**
- **Greater-than (between unsigned quantities).**
- **Equal or greater-than (between unsigned quantities).**

When a comparison is detected, the compiler adds a mirCompare node into the mir-tree. The different kinds of comparisons which are listed above are distinguished by the condition rule-clause. In figure 3.18, is shown the equal or greater-than comparison for unsigned operands. In that rule we have also specified a condition for selecting it. That condition says, that this comparison is "IS_GREATER_EQUAL" and that also the operands are unsigned -IS_UINT-. Let's explain the comparison in figure 3.18. The mir-node as we have already mentioned is the mirCompare, which receives as inputs two registers *s1* and *s2* and produces a register *d* as an output. Our ISA does not support instructions such as equal or greater-than(seqgtu). Thus, we do that work by using the *sgtu* instruction and the procedure is the following:
$seqgtu\ s1, s2 \rightarrow$
$!(sgtu\ s2, s1) \rightarrow$
$sgtu\ s2, s1$
$li\ d, 1$
$sub\ d, s2$
So, the last two instructions are reversing the result for us. First, we load a 1 in the result register, and then we subtract from it, the result of the *sgtu* comparison. It is now obvious, that if the *sgtu* comparison is 0, then the final result will be 1; otherwise it will be 0.

Control Flow: The control-flow group is responsible for the flowing of the program and it is divided into 3 main categories: function call support, conditional jumping and unconditional jumping.

- **Function call support:** This category of rules is responsible for functions. Which means jumping to a specific function, passing parameters by registers- or by memory if the registers are not enough- and returning from a function. The registers which are used for passing parameters are 1 to 8, while the result



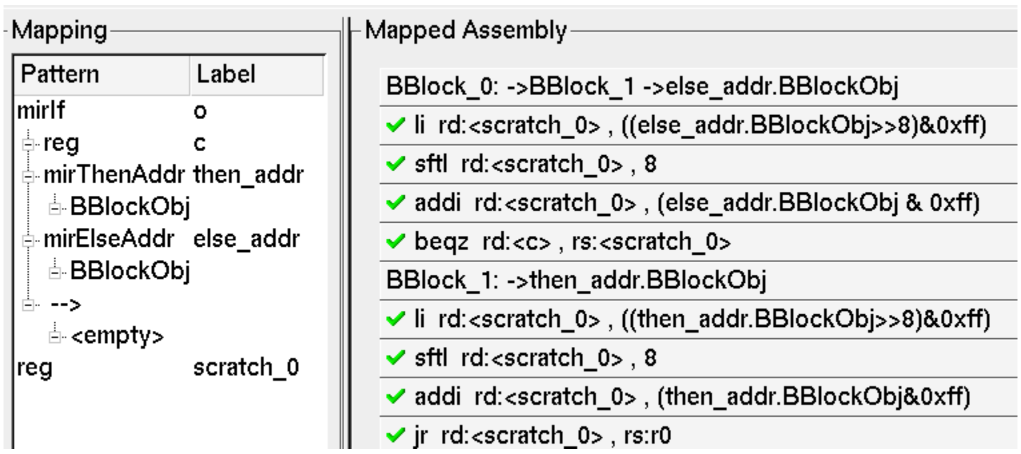
Mapping

Pattern	Label
mirCompare	o
reg	s1
reg	s2
-->	
reg	d

Mapped Assembly

```
✓ sgtu  rd:<s2> , rs:<s1>
✓ li  rd:<d> , 1
✓ sub  rd:<d> , rs:<s2>
✓ INSTRUCTION
  ✗ jump_type:
```

Figure 3.18: Equal or grater-than comparison between unsigned operands.



Mapping

Pattern	Label
mirIf	o
reg	c
mirThenAddr	then_addr
BBlockObj	
mirElseAddr	else_addr
BBlockObj	
-->	
<empty>	
reg	scratch_0

Mapped Assembly

```
BBlock_0: ->BBlock_1 ->else_addr.BBlockObj
✓ li  rd:<scratch_0> , ((else_addr.BBlockObj>>8)&0xff)
✓ sftl  rd:<scratch_0> , 8
✓ addi  rd:<scratch_0> , (else_addr.BBlockObj & 0xff)
✓ beqz  rd:<c> , rs:<scratch_0>
BBlock_1: ->then_addr.BBlockObj
✓ li  rd:<scratch_0> , ((then_addr.BBlockObj>>8)&0xff)
✓ sftl  rd:<scratch_0> , 8
✓ addi  rd:<scratch_0> , (then_addr.BBlockObj&0xff)
✓ jr  rd:<scratch_0> , rs:r0
```

Figure 3.19: If-else control flow statement.

of a non-void function is loaded into register 1 (*r1*). The mapping assembly-instructions of a function call must include the *jal* instruction which jumps and saves the returning address in register 15 (*r15*).

- **Conditional jumping:** This category supports the if-else statements of the program. This category consists of one rule, which actually uses the results of the Comparison group. In other words, first the comparison is made and its result is loaded into a register. Afterwards, this register is taken as an input into the conditional-jump rule. In figure 3.19 this rule is shown.
- **Unconditional jumping:** This category contains rules which supports unconditional jumps to addresses.

Arithmetic: This group is responsible for all the arithmetic operations which we list in a while. This group is partitioned into two categories: the address and the common arithmetic. The first category uses only addition and subtraction as -pointer- operations. However, different rules are implemented for addition and subtraction in the common arithmetic category. So, an obvious question is, why do we use different rules for the same thing? The answer is that the front-end compiler creates different objects for addresses and different for the rest values, thus, they must be manipulated separately. The list of operations for the second category are the following:

- **Addition of two operands:** We have two kind of additions. If the second operand is an unsigned 8-bit constant, then the *addi* instruction is used for mapping.
- **Subtraction of two operands:** Also here we have two kinds of subtractions for the same reason as above.
- **Or-ing two operands.**
- **And-ing two operands.**
- **Xor-ing two operands.**
- **Not-ing a operand.**
- **Unsigned right shifting.**
- **Signed right shifting.**
- **Left shifting.**
- **Negating an operand.**
- **Dividing two operands.**
- **Modulo operation.**
- **Multiplying two operands.**

The most interesting operations from above are the division, modulo and multiplication. As we have mentioned in section 3.4 that the compiler takes care of the multiplication, division and modulo operations, for saving area and power from the processor. The implementation of these operations supports only unsigned numbers, because we have observed that our benchmark has only unsigned multiplications, divisions and modulo. If we would like to add signed operations we could do it in the following ways: either by changing a signed number into unsigned before calling the operation -explicit cast in C-, or by adding some more assembly instructions in their -multiplication, division, modulo- matching rules.
Let's proceed to the division implementation. The division of a 2k-bit dividend by a k-bit divisor can be calculated in k-cycles of shifting and subtracting. This pseudocode is shown in Algorithm 1.

First, the divisor is left-shifted 16 positions. Then, for 16 loops the algorithm checks if the remainder is equal or greater than the divisor. If so, the divisor is subtracted from the remainder and then, 1 is added to the quotient which has already been left-shifted once. The quotient is the result of the division. For more information about division see in [22]. The division is the same for all three processors we have designed. The division implementation with the Compiler-Designer tool is presented in figure 3.20.

The modulo operation uses the same algorithm as the division. The only difference is that the result is the remainder and not the quotient. This algorithm has been used only in the baseline processor because in the second design -Optimization1-

Algorithm 1 Radix-2 unsigned division algorithm

```
$Dividend \rightarrow z$
$Divisor \rightarrow d$
$Quotient \rightarrow q$
$Remainder \rightarrow s$

$tempd = d << 16;$
$s = z;$
for $(i = 0; i < 16; i++)$ do
  $q << 1;$
  $tempd = tempd << 1;$
  if $(tempd \leq s)$ then
    $q = q + 1;$
    $s = s - tempd$
  end if
end for
```

the modulo operation has been added in the ISA of the processor, as has been mentioned in section 3.4. In Optimization1+2 design, this optimization still remains. In chapter 4, the advantages of doing are optimization is shown. In figure 3.21, we show the baseline modulo, while in figure 3.22 we show the optimized version of the remaining two designs.

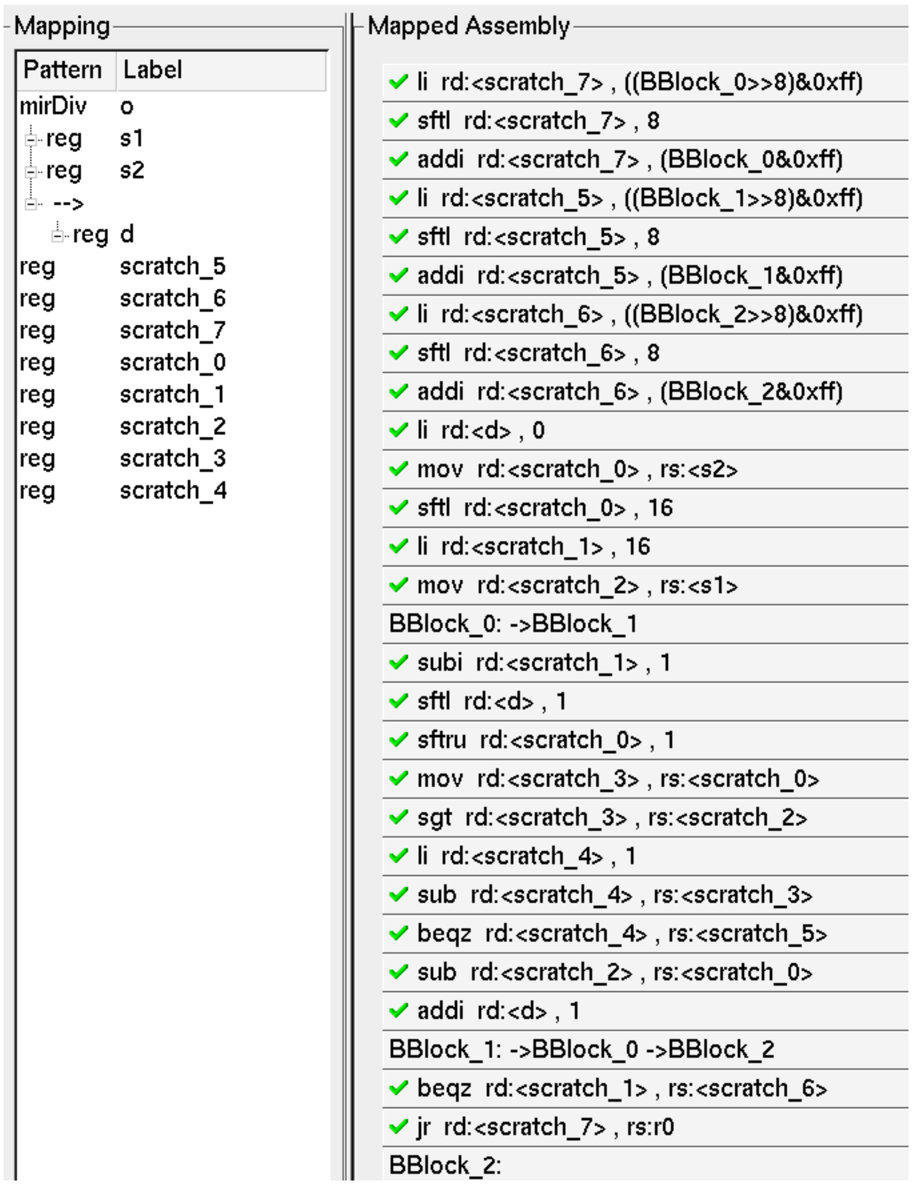
The multiplication operation uses the scheme shown in figure 11.8a of the book [22]. It is a radix-2-unsigned algorithm. When we have a multiplier and a multiplicand of n bits, the maximum product that we can get is 2n bits. Hence, in our case the multiplicand and the multiplier are 16-bits wide and the product 32-bits. This pseudocode is shown in Algorithm 2.

Algorithm 2 Radix-2 unsigned multiplication algorithm

```
$Multiplicand \rightarrow a$
$Multiplier \rightarrow x$
$Product \rightarrow p$

for $(i = 0; i < 16; i++)$ do
  if $(x_i == 1)$ then
    $p = p + a;$
  end if
  $a = a << 1;$
end for
```

The algorithm shown needs 16 loops. In every loop the next bit of the multiplier is checked and if it is 1 then the multiplicand is added to the current product. At the end of every loop the multiplicand is left-shifted once.
The algorithm for all three processor designs is the same. But in the third pro-



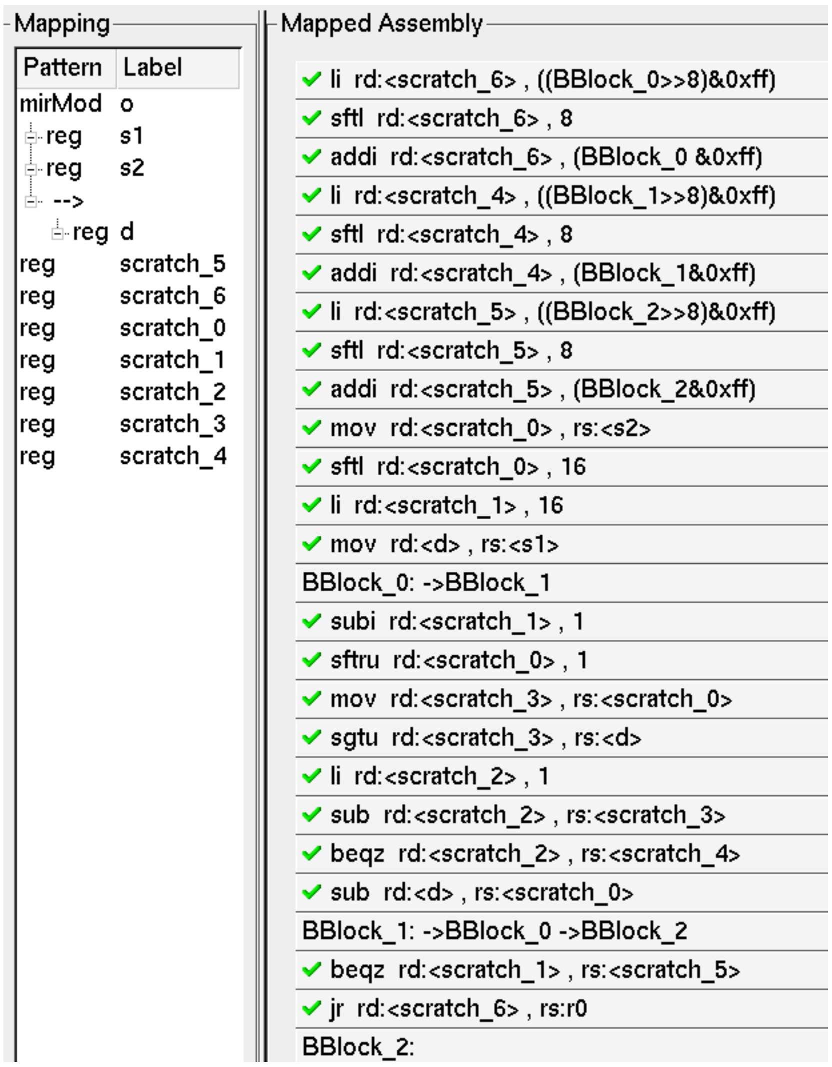
Pattern	Label
mirDiv	o
reg	s1
reg	s2
-->	
reg	d
reg	scratch_5
reg	scratch_6
reg	scratch_7
reg	scratch_0
reg	scratch_1
reg	scratch_2
reg	scratch_3
reg	scratch_4

```
✓ li  rd:<scratch_7> , ((BBlock_0>>8)&0xff)
✓ sftl  rd:<scratch_7> , 8
✓ addi  rd:<scratch_7> , (BBlock_0&0xff)
✓ li  rd:<scratch_5> , ((BBlock_1>>8)&0xff)
✓ sftl  rd:<scratch_5> , 8
✓ addi  rd:<scratch_5> , (BBlock_1&0xff)
✓ li  rd:<scratch_6> , ((BBlock_2>>8)&0xff)
✓ sftl  rd:<scratch_6> , 8
✓ addi  rd:<scratch_6> , (BBlock_2&0xff)
✓ li  rd:<d> , 0
✓ mov  rd:<scratch_0> , rs:<s2>
✓ sftl  rd:<scratch_0> , 16
✓ li  rd:<scratch_1> , 16
✓ mov  rd:<scratch_2> , rs:<s1>
BBlock_0: ->BBlock_1
✓ subi  rd:<scratch_1> , 1
✓ sftl  rd:<d> , 1
✓ sftru  rd:<scratch_0> , 1
✓ mov  rd:<scratch_3> , rs:<scratch_0>
✓ sgt  rd:<scratch_3> , rs:<scratch_2>
✓ li  rd:<scratch_4> , 1
✓ sub  rd:<scratch_4> , rs:<scratch_3>
✓ beqz  rd:<scratch_4> , rs:<scratch_5>
✓ sub  rd:<scratch_2> , rs:<scratch_0>
✓ addi  rd:<d> , 1
BBlock_1: ->BBlock_0 ->BBlock_2
✓ beqz  rd:<scratch_1> , rs:<scratch_6>
✓ jr  rd:<scratch_7> , rs:r0
BBlock_2:
```

Figure 3.20: Division Implementation.

cessor design -Optimization1+2- the algorithm is implemented by using different instructions in a specific part. In that part, three instructions -*mov, and, beqz*- are collapsed together in a single instruction (*mandb*). We did that because -as it is also shown in chapter 4 in detail- these three instructions appeared very frequently. The new multiplication implementation of Optimization1+2 is shown in figure 3.24 while the baseline one is presented in figure 3.23.

Moves: This group just consists of one rule, which maps into the *mov* instruction.



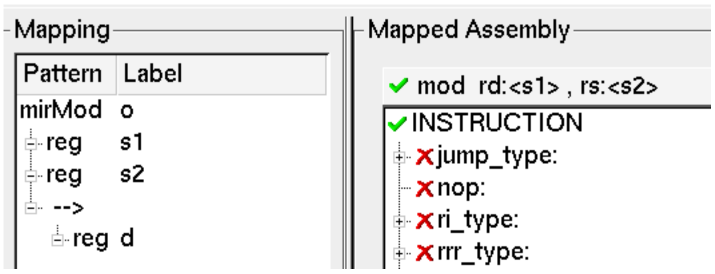
Mapping

Pattern	Label
mirMod	o
reg	s1
reg	s2
-->	
reg	d
reg	scratch_5
reg	scratch_6
reg	scratch_0
reg	scratch_1
reg	scratch_2
reg	scratch_3
reg	scratch_4

Mapped Assembly

```
✓ li  rd:<scratch_6> , ((BBlock_0>>8)&0xff)
✓ sftl  rd:<scratch_6> , 8
✓ addi  rd:<scratch_6> , (BBlock_0 &0xff)
✓ li  rd:<scratch_4> , ((BBlock_1>>8)&0xff)
✓ sftl  rd:<scratch_4> , 8
✓ addi  rd:<scratch_4> , (BBlock_1&0xff)
✓ li  rd:<scratch_5> , ((BBlock_2>>8)&0xff)
✓ sftl  rd:<scratch_5> , 8
✓ addi  rd:<scratch_5> , (BBlock_2&0xff)
✓ mov  rd:<scratch_0> , rs:<s2>
✓ sftl  rd:<scratch_0> , 16
✓ li  rd:<scratch_1> , 16
✓ mov  rd:<d> , rs:<s1>
BBlock_0: ->BBlock_1
✓ subi  rd:<scratch_1> , 1
✓ sftru  rd:<scratch_0> , 1
✓ mov  rd:<scratch_3> , rs:<scratch_0>
✓ sgtu  rd:<scratch_3> , rs:<d>
✓ li  rd:<scratch_2> , 1
✓ sub  rd:<scratch_2> , rs:<scratch_3>
✓ beqz  rd:<scratch_2> , rs:<scratch_4>
✓ sub  rd:<d> , rs:<scratch_0>
BBlock_1: ->BBlock_0 ->BBlock_2
✓ beqz  rd:<scratch_1> , rs:<scratch_5>
✓ jr  rd:<scratch_6> , rs:r0
BBlock_2:
```

Figure 3.21: Modulo implementation of the Baseline design.



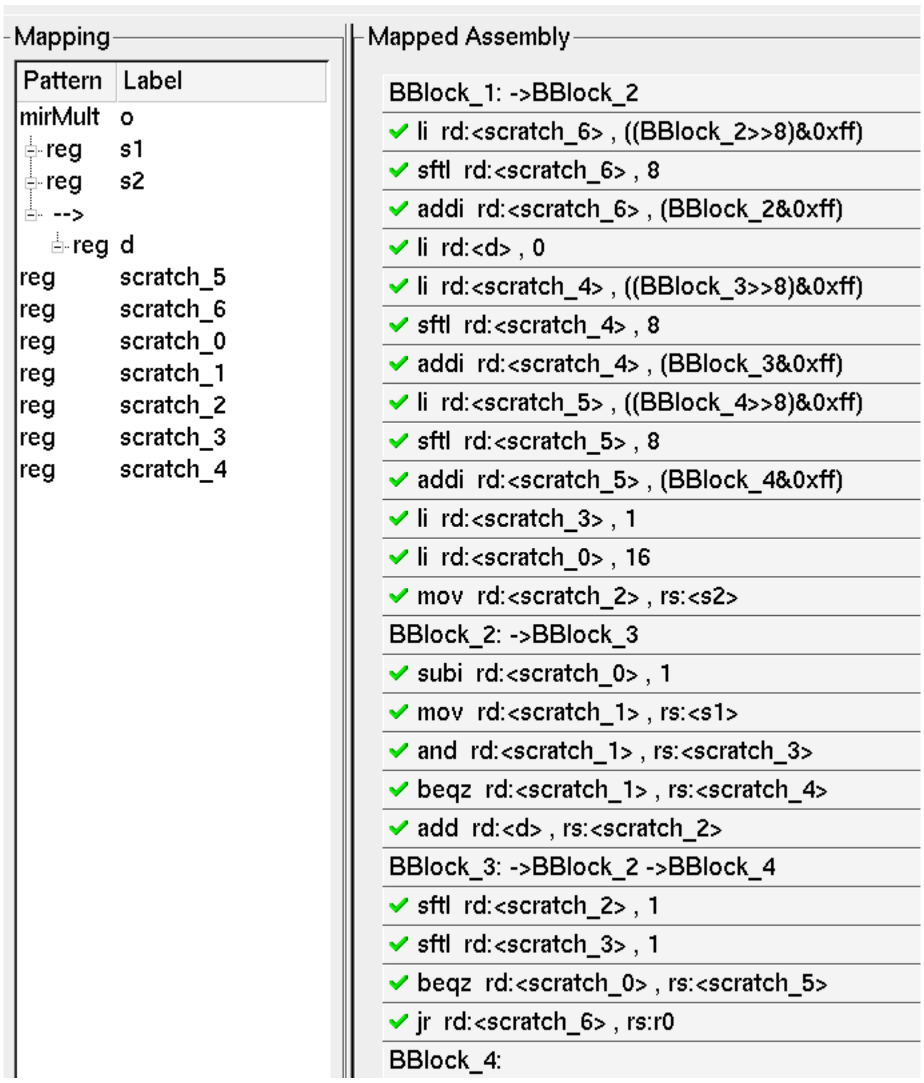
Mapping

Pattern	Label
mirMod	o
reg	s1
reg	s2
-->	
reg	d

Mapped Assembly

```
✓ mod  rd:<s1> , rs:<s2>
✓INSTRUCTION
  ✗jump_type:
  ✗nop:
  ✗ri_type:
  ✗rrr_type:
```

Figure 3.22: Modulo implementation of Optimization1 and Optimization1+2 designs.



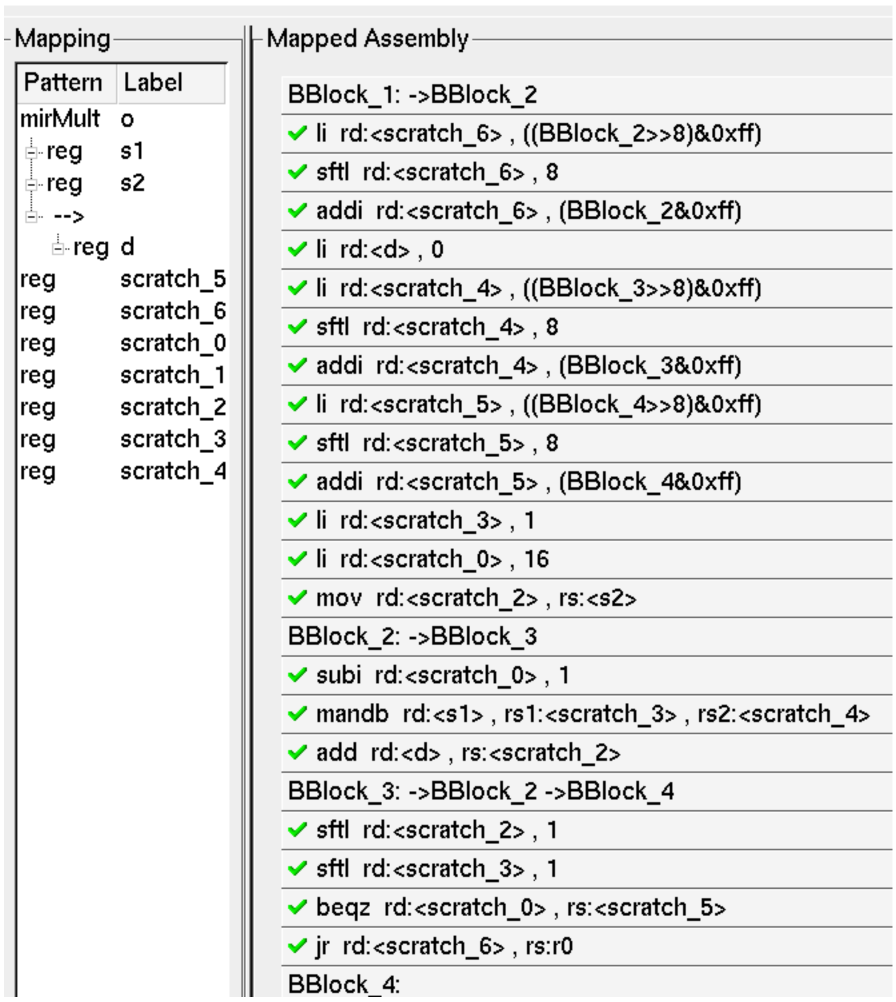
Mapping

Pattern	Label
mirMult	o
reg	s1
reg	s2
-->	
reg	d
reg	scratch_5
reg	scratch_6
reg	scratch_0
reg	scratch_1
reg	scratch_2
reg	scratch_3
reg	scratch_4

Mapped Assembly

```
BBlock_1: ->BBlock_2
✓ li  rd:<scratch_6> , ((BBlock_2>>8)&0xff)
✓ sftl  rd:<scratch_6> , 8
✓ addi  rd:<scratch_6> , (BBlock_2&0xff)
✓ li  rd:<d> , 0
✓ li  rd:<scratch_4> , ((BBlock_3>>8)&0xff)
✓ sftl  rd:<scratch_4> , 8
✓ addi  rd:<scratch_4> , (BBlock_3&0xff)
✓ li  rd:<scratch_5> , ((BBlock_4>>8)&0xff)
✓ sftl  rd:<scratch_5> , 8
✓ addi  rd:<scratch_5> , (BBlock_4&0xff)
✓ li  rd:<scratch_3> , 1
✓ li  rd:<scratch_0> , 16
✓ mov  rd:<scratch_2> , rs:<s2>
BBlock_2: ->BBlock_3
✓ subi  rd:<scratch_0> , 1
✓ mov  rd:<scratch_1> , rs:<s1>
✓ and  rd:<scratch_1> , rs:<scratch_3>
✓ beqz  rd:<scratch_1> , rs:<scratch_4>
✓ add  rd:<d> , rs:<scratch_2>
BBlock_3: ->BBlock_2 ->BBlock_4
✓ sftl  rd:<scratch_2> , 1
✓ sftl  rd:<scratch_3> , 1
✓ beqz  rd:<scratch_0> , rs:<scratch_5>
✓ jr  rd:<scratch_6> , rs:r0
BBlock_4:
```

Figure 3.23: Multiplication implementation of Baseline and Optimization1 designs.



Pattern	Label
mirMult	o
reg	s1
reg	s2
-->	
reg	d
reg	scratch_5
reg	scratch_6
reg	scratch_0
reg	scratch_1
reg	scratch_2
reg	scratch_3
reg	scratch_4

```
BBlock_1: ->BBlock_2
✓ li  rd:<scratch_6> , ((BBlock_2>>8)&0xff)
✓ sftl  rd:<scratch_6> , 8
✓ addi  rd:<scratch_6> , (BBlock_2&0xff)
✓ li  rd:<d> , 0
✓ li  rd:<scratch_4> , ((BBlock_3>>8)&0xff)
✓ sftl  rd:<scratch_4> , 8
✓ addi  rd:<scratch_4> , (BBlock_3&0xff)
✓ li  rd:<scratch_5> , ((BBlock_4>>8)&0xff)
✓ sftl  rd:<scratch_5> , 8
✓ addi  rd:<scratch_5> , (BBlock_4&0xff)
✓ li  rd:<scratch_3> , 1
✓ li  rd:<scratch_0> , 16
✓ mov  rd:<scratch_2> , rs:<s2>
BBlock_2: ->BBlock_3
✓ subi  rd:<scratch_0> , 1
✓ mandb  rd:<s1> , rs1:<scratch_3> , rs2:<scratch_4>
✓ add  rd:<d> , rs:<scratch_2>
BBlock_3: ->BBlock_2 ->BBlock_4
✓ sftl  rd:<scratch_2> , 1
✓ sftl  rd:<scratch_3> , 1
✓ beqz  rd:<scratch_0> , rs:<scratch_5>
✓ jr  rd:<scratch_6> , rs:r0
BBlock_4:
```

Figure 3.24: Multiplication implementation of Optimization1+2 design.

3.6 Conclusions

In this chapter we have described the design and implementation of a security co-processor for IMD's. The work consists of:

- **The system architecture**. In section 3.1 we described that the system consists of the IMD part and the reader. The IMD is distinguished into two parts, the main processor and the communication co-processor which is attached with an RFID for gaining RF energy. The IMDs must have a normal and an emergency mode. This switching-mode is done by a magnetic switch located inside the implant.
- **The communication protocol** between the IMD and the reader and its implementation in C language. The selected protocol is shown in figure 3.4(b).
- **The definition of the scenarios** and the selection of a representative one (scenario 10), which is shown in section 3.3.
- **The baseline-security-processor design** and two **optimized** designs, which are described in section 3.4.
- **The C compiler** implementation of the above processors, which is shown in section 3.5.

4 Co-processor Evaluation

In this chapter we present our work's experimental results. This chapter explains some decisions taken regarding our implementation described in chapter 3. It also presents a comparison between the different designs and a detailed explanation of the results.

This chapter is organized as follows: Section 4.1 contains the Scenarios profiling using the XTREM simulator [14]. These scenarios have been shown in section 3.3. Section 4.2 presents the instruction mixes and the execution-cycles number of all three designed processors. Finally in section 4.3 are described the synthesis results of our processor.

4.1 Profiling of the Normal and Attack Scenarios

In this section we show and analyze, the profiling results of the 20 attack and normal scenarios, described in section 3.3. The profiling has been done for two reasons. First, for selecting the most representative scenario according to power and energy, in order to use it as our main benchmark for our custom processor. We have selected only one scenario, because it would take too much time to run all the benchmarks in the synthesized processor. The second reason has been gaining useful information on designing our processors. The profiling analysis provided us with instruction number, micro-operation number, cycle number, performance, power and energy figures. As we have already mentioned, the profiling has been done on the XTREM simulator (see [14]). XTREM is a simulator for the Intel XScale core, which is compatible with the ARMv5TE instruction set. It has an average performance error of 6.5% and an average power error of 4%.

In table 4.1 is shown the scenario profiling by using XTREM. The scenarios can be divided into three groups: the unsuccessful ones, the 1KB-successful and the 10KB-successful ones. The first group consists of the first 6 scenarios (1-6), the second group consists of the next 7 scenarios (7-13) and the remaining 7 scenarios (14-20) form the third group. We expect the first group to complete its execution faster than the second one and the second one to be faster than the third one, because of their increasing workload sizes used. The cycles column of table 4.1, verifies our expectations. Also, the execution time increases according to the number of instructions -and uops- because we always run the same program (the only thing that changes in every scenario is the input data). The total energy consumption is increasing with respect to the execution time, as we have expected. We can observe that the performance for the first three scenarios is 0.07 -maximum of all three groups-, the fourth is 0.05 and the rest are 0.04. Let's explain this difference in performance. The first four scenarios, as we have seen in section 3.3, stop executing before the heavy and most time-consuming functions are executed. These are decryption and CRC calculation functions. They do not execute because, in the first three scenarios, the IMD receives wrong challenges while, in the fourth one, it receives a wrong command size, which actually means no execution of the decryption and CRC-calculation functions. In scenario5 only the decryption algorithm is executed once, while in scenario6 both decryption and CRC-calculation functions are running. That is why scenarios 5 and 6 also differ from each other. Obviously, the rest of the scenarios exhibit the same performance because the CRC-calculation and the decryption functions are running for most of the time. For remembering the flow chart of the program, see figure 3.5.

Scenarios	#uops	#instr	#cycles	perf(IPC)	power(mW)	Energy(mJ)
Scenario1	41412	32707	471512	0.07	75.1	17.71
Scenario2	41083	32471	466884	0.07	74.72	17.44
Scenario3	57287	41733	621036	0.07	84.12	26.12
Scenario4	50753	38147	567347	0.07	82.99	23.54
Scenario5	58434	42440	891589	0.05	61.53	27.43
Scenario6	67477	47510	1239684	0.04	51.94	32.19
Scenario7	999182	590085	16332237	0.04	76.22	622.42
Scenario8	1006050	594029	16384469	0.04	76.28	624.9
Scenario9	1000055	590463	14982910	0.04	83.26	623.74
Scenario10	1000351	590664	13595838	0.04	91.89	624.66
Scenario11	1000095	590501	13591824	0.04	91.89	624.48
Scenario12	1008653	595435	16672429	0.04	75.16	626.55
Scenario13	1017702	600507	15623253	0.04	80.89	631.88
Scenario14	7766854	4547818	125947516	0.04	77.86	4903.14
Scenario15	7773722	4551762	125999751	0.04	77.86	4905.17
Scenario16	7767727	4548196	114106644	0.04	86.12	4913.43
Scenario17	7767983	4548359	102652814	0.04	95.8	4917.07
Scenario18	7767727	4548196	102648804	0.04	95.8	4916.88
Scenario19	7776245	4553090	126237606	0.04	77.75	4907.49
Scenario20	7785301	4558169	114814453	0.04	85.73	4921.52

Table 4.1: XTREM profiling.

We have mentioned above that the execution cycles are increasing with respect to the uop increase. If we look closer at table 4.1, we can see that this is not always the case. For instance, let's compare scenario7 and scenario9 which have 999,182 and 1,000,055 uops, respectively. The execution cycles are 16,332,237 for scenario7 and 14,982,910 for scenario9 which is not what we would have expected. So, we have also ran the scenarios in the XEEMU simulator (see [21]) which is more accurate and newer than XTREM, to check if XTREM has been wrong. We did not use the XEEMU simulator from the beginning because on the one hand the XTREM simulator has been used in previous works of the SiMS project, thus, scripts for assisting the profiling study exist for the XTREM. On the other hand, the XEEMU simulator has not been properly incorporated in the SiMS toolflow, bulk experiments cannot be performed with it, yet. We have selectively used it for a few experiments as in this case. The XEMMU results showed that the XTREM was wrong at that point. The XEEMU evaluation has shown that the number of execution cycles increase according to the number of instructions. The problem of the wrong calculation of the execution cycles in XTREM might have a small effect on power and performance. That is, for a program with more instructions to execute faster than another, it must execute more instructions per cycle. This leads to an increase in performance and in power consumption.

Before continuing to the scenario-selection analysis, we should analyze a bit the energy consumption of the unsuccessful group. These energy results are, actually, the energy consumption of an IMD -using the XScale core- during possible attacks. We observe that the energy consumption of each attack is between 17.71 and 32.19 mJ. This is important to know, because it can show if this amount of energy is affordable for an RFID.

Now let's continue to the scenario-selection part. This scenario will be our benchmark for our implemented processor. We have selected a scenario from the successful-1KB transmission group, because a 10KB would need too much execution time to run in our processor simulator. In addition, we have not selected a scenario from the unsuccessful group because we have assumed that in most of the times, the communication between the IMD and a reader is going to be successful. The scenario that we have selected has been scenario10 (91.89 mW), because it has the highest power consumption together with scenario11 from all the others in that group. Then, we have chosen scenario10 instead of scenario11, because scenario10 has a higher energy consumption (than scenario11). Hence, for these reasons we believe that scenario10 is the most representative scenario of all. Scenario10 contains a wrong challenge, from the reader to the IMD, in the beginning and, then, a complete successful procedure, according to protocol B in figure 3.4(b).

Another interesting issue here is the instruction-mix analysis of single instructions or groups of them. In the XTREM evaluation when we say 'instruction' we mean 'uop'. The instruction-mix is defined as the % appearance of each instruction or groups of them in a program. Again here we are going to analyze the scenarios as groups (unsuccessful, 1KB-successful, 10KB-successful). Thus, we pick one scenario from each group, which are scenario1, scenario10 and scenario17.

In table 4.2, the results of the single-instruction mix are shown. The table is divided in three main columns, which represent the three groups. Each column consists of three smaller columns. The first column has the instruction names, the second column has the appearance number of the instructions and, finally, the third column has the % instructions appearance. In that table are included only the instructions, which the % appearance of which exceeds 1%.

In table 4.2 and in figure 4.1 we can see that the unsuccessful scenarios have a lot of "xor", "and", comparison and jump instructions. The "xor" and "and" large appearance can be explained by the fact that the key generation and the CRC generation table functions use these instructions many times. It can also be partly explained by the known fact that the compiler used for generating the XTREM code (ARM-GCC-2.95.3) does favor logical operations. The comparison appearance can be explained if we consider that in the unsuccessful scenarios are executed some comparisons and then the program ends without running either the cryptographic algorithms or the CRC. We can also see a lot of branches, which can be explained by the big percentage of comparisons and loops that occur inside the program.

The next two scenarios groups can be analyzed together because, as we can see in table 4.2 and in plots 4.2 and 4.3, their instruction appearances are similar. In comparison with the first group we observe an increase in ”xor” and ”and” instructions. This can be explained because the cryptographic and the crc-calculation algorithms are using them a lot. A decision that we can make about our targeting processors, by looking at the table 4.2, is to include the ”xor” instruction in our ISA. ”Xor” is not an essential instruction for a processor. On the other hand the ”and” instruction is an essential one, hence we would include it anyway. For instance, the ”xor” instruction needs ”and” instructions for its implementation.

Unsuccessful scenarios			1KB-successful scenarios			10KB-successful scenarios		
eor	9207	22.233%	eor	423140	42.299%	eor	3330273	42.872%
and	7506	18.125%	and	247159	24.707%	and	1952567	25.136%
cmp	5752	13.890%	mov	183613	18.355%	mov	1457577	18.764%
mov	5061	12.221%	add	37099	3.709%	add	281329	3.622%
b	3691	8.913%	sub	27303	2.729%	sub	215525	2.775%
add	2921	7.054%	cmp	14811	1.481%	rsb	85609	1.102%
ble	2597	6.271%	b	12999	1.299%	b	79594	1.025%
beq	2373	5.730%	rsb	10534	1.053%			
bne	579	1.398%						
sub	519	1.253%						

Table 4.2: Single-Instruction Mix.

In table 4.3, are shown the pair-instruction[1] mix of the grouped scenarios . At this point we can say that, if we could change the ISA of that specific processor, we should collapse together the pairs ”and-xor” and ”xor-xor” in order to gain in performance and probably in energy-consumption.

Unsuccessful scenarios			1KB-successful scenarios			10KB-successful scenarios		
and eor	3456	8.345%	and eor	144997	14.495%	and eor	1152578	14.838%
eor eor	2852	6.887%	eor eor	115253	11.521%	eor eor	900035	11.586%
b cmp ble	2304	5.564%	mov mov	56730	5.671%	mov mov	462626	5.956%
ble and cmp	2048	4.945%	and eor and	27130	2.712%	and eor and	217429	2.799%
beq mov	1036	2.502%	add mov	19162	1.916%	add mov	157629	2.029%
beq mov add	1025	2.475%	eor mov	15829	1.582%	eor mov	128928	1.660%
b add b	1024	2.473%	mov add	15633	1.563%	mov add	127691	1.644%
mov mov	573	1.384%	mov and	14165	1.416%	mov and	113535	1.462%
and eor and	445	1.075%	eor eor eor	11729	1.172%	eor eor eor	92113	1.186%
			eor and	10971	1.097%	eor and	88152	1.135%

Table 4.3: Pair-Instruction Mix.

In this section we have shown the profiling results of the 20 normal and attack scenarios, which are described in section 3.3. We have selected scenario10 because the

[1]”Instruction pairs are consecutive instructions whereby data generated by the first instruction is consumed by the second instruction; i.e. whereby data dependencies occur” ([25])

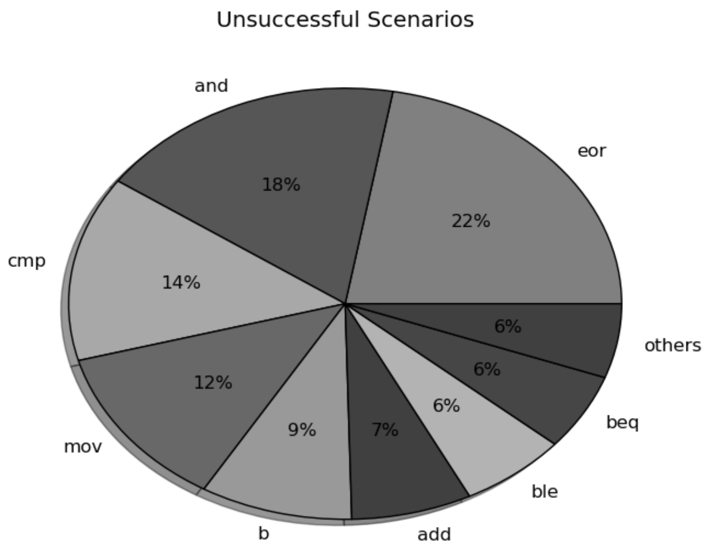


Figure 4.1: Single-instruction mix plot for the Unsuccessful scenarios.

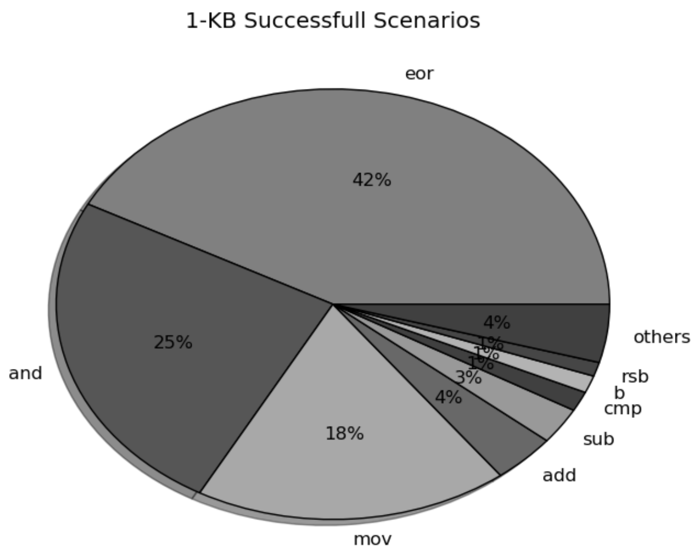


Figure 4.2: Single-instruction Mix plot for the 1KB Successful scenarios.

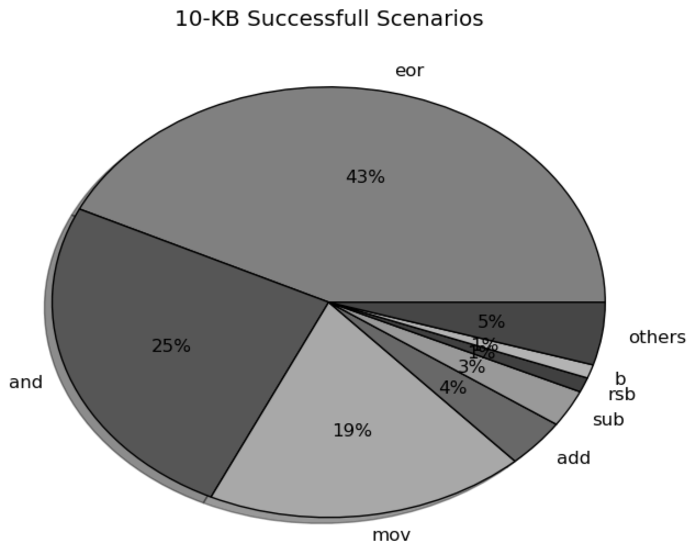


Figure 4.3: Single-instruction mix plot for the 10KB Successful scenarios.

evaluation show that it is the most representative one, in terms of power and energy. The instruction mixes of the scenarios show that we should include the ”xor” instruction in our ISA. Finally, by observing the energy results of the unsuccessful scenarios (scenarios where the communication has failed because of an attack), we can see that the energy consumption is not very large. So, we can conclude that this energy can be covered by the induced RF energy. We also believe that with a dedicated processor the energy and power consumption will be even less.

4.2 Instruction Mix and Execution Cycles

In this section we show the instruction mixes and the execution-cycle number of our selected benchmark running on Base-line, Optimization1 and Optimization1+2 processors, which were described in section 3.4. These are the ASIPs which we have implemented with the Processor Designer tool. The evaluation -in terms of instructions- of the Base-line processor is important because the decisions for optimizations (Optimization1 and Optimization1+2) are made according to it. Hence, we have generated and studied the instruction mixes of the Base-line processor and then we have taken decisions for optimizations. The instruction mixes have been generated by Python-scripts that we have implemented for that purpose. In tables 4.4, 4.5 and 4.8, are presented the instruction mixes of the Base-line, Optimization1 and Optimization1+2 processors. The tables have two main columns named Single-instruction mix and Pair-instruction mix. Each column is divided into 3 sub-columns. The first one lists the names of the instructions, the second one their absolute-number of appearances and the third one their appearance-frequency.

In the pair-mixes we only present the pairs that appear more than 1% in the instruction mix.

Lets examine our Base-line instruction mix. In the single-instruction mix we see the following appearances:

1. **beqz:** First in the list is the instruction *beqz*. The reason for that is that every comparison uses this instruction for branching to its final destination. In addition, as it has been shown in section 3.5, the multiplication, the division and the modulo -which appearing many times- have 16 loops inside them, which are translated into 16 *beqz*. This is the reason why the *beqz* instruction has such a high frequency.

2. **sftl:** This instruction is very frequent because it is used in casting (for zero and sign extension) and in loading constants and addresses. It is also used a lot into multiplication, division and modulo.

3. **li:** It is used for loading constants and addresses.

4. **mov:** This is frequent because our ISA uses a single register as source and destination. Which means that if the compiler wants to keep a register-value and not losing it, this value must be moved into another register.

5. **subi:** The relatively high *subi* frequency is explained by the fact that it appears in multiplication, modulo and division. Also it is used for decrementing the stack-pointer, to point at the end of the stack-frame.

6. **jr, j:** We see that *jr* has a high frequency, while *j* has 0. That happens because *j* can (describe and) jump to addresses up to 4095 (12-bits). Our program exceeds that bound that is why we use *jr* (32-bits). We do not use *j* for cases lower than 4095, because we did not find a way to inform the compiler about this discrimination.

7. **add:** Addition is normal to be used a lot in every program. For example, in address calculation.

8. **and, xor, or, not:** The *and* instruction is used so frequently because it is used in multiplication, in if-else C-statements and finally, it is also used (in addition to *xor*) in encryption, decryption and CRC-calculation functions. The *xor* instruction percentage we should expect to be higher according to section 4.1, but we must remember that the CRC-generation-table function (which includes many *xor*s) has been removed from our benchmark and it has been replaced by the already calculated CRC-table. This has been done for decreasing the size of the program memory. The *or* instruction is mostly used for comparison in if-else C-statements. The *not* instruction has not be used at all.

9. **sftru, sftr:** The *sftru* instruction is mostly used in zero-extensions. The *sftr* is mostly used for sign-extension.

10. **addi:** That instruction mostly appears in constant-loading. The reason is shown in figure 3.13.

11. **sgt, se, sgtu:** These instructions are used for comparisons, which appear especially in if-else statements. The *sgtu* is been used into the modulo operation.

12. **lw, sw, lb, cb:** The *lw* and lb instructions is logically more than *sw* and *cb* (it is part of the *sb* pseudo-instruction) because the writting is more frequent operation than reading.

13. **jal:** The *jal* is used for jumping into functions.

In the pair-instruction mix in table 4.4 we see the following occurrences:

1. **and beqz:** This pair appears in the multiplication.

2. **mov and** This pair appears in the multiplication and just before the *and-beqz* pair.

3. **li sftl:** It is used when we have to load an immediate into a register, which is wider than one byte, or we want to sign-extend a byte.

4. **li add:** This pair appears when we load from the stack. It is used for adding an offset to the stack-pointer.

5. **add lw:** The *lw* instruction is the instruction just below the *li-add* pair.

6. **li sub:** This pair appears in modulo, division operations and in compare-not-equal compiler-rule.

7. **sftru mov:** It appears in modulo and division operations.

8. **sub beqz:** Appears in modulo operation.

9. **sftl addi:** Usually it appears when we have to load more than one byte into a register. Hence, *addi* is mostly seen just after *li-sftl* pair.

10. **sftr add:** This pair is mostly seen when a value is stored into the stack (subtracting an offset). It is used for addition (subtraction) after a sign-extension.

11. **sftl sftr:** It is used for sign extension. The *sftl* is seen above the *sftr-add* pair.

12. **mov sgtu:** It appears in modulo operations.

13. **add sw:** This pair follows the pair *sftr-add* which stores into the program stack.

14. **sw li:** When we enter into a function, many sequential stores often are made into the stack. The last instruction of storing is *sw* while the first one is *li.* That is why we have that amount of appearances of this pair.

15. **addi sftl:** This pair appear when we need to store into a register a 4-byte immediate.

Single-Instruction Mix			Pair-Instruction Mix		
beqz	950499	15.290%	and beqz	296928	10.379%
sftl	942502	15.162%	mov and	296928	10.379%
li	779614	12.541%	li sftl	274921	9.610%
mov	533955	8.589%	li add	209947	7.339%
subi	479950	7.721%	add lw	206873	7.231%
jr	454274	7.308%	li sub	178214	6.229%
add	444448	7.150%	sftru mov	176880	6.183%
and	321246	5.168%	sub beqz	176880	6.183%
lw	213280	3.431%	mov sgtu	155600	5.439%
sftru	187564	3.017%	sftr add	155438	5.433%
sub	184334	2.965%	sftl sftr	155438	5.433%
addi	172759	2.779%	sftl addi	154223	5.391%
sgtu	155600	2.503%	add sw	141453	4.944%
sftr	155438	2.500%	sw li	56195	1.964%
sw	152532	2.454%	addi sftl	34740	1.214%
xor	29308	0.471%			
sgt	22823	0.367%			
lb	17059	0.274%			
or	6932	0.112%			
jal	6140	0.099%			
cb	4804	0.077%			
se	1340	0.022%			
not	0	0.000%			
nop	0	0.000%			
j	0	0.000%			

Table 4.4: Base line Instruction Mix.

In table 4.6 is given that the Base-line processor needs 18,805,011 execution cycles, from which the 6,216,401 are executing instructions and the rest are stalls. That means that there are 33.1% executing-instructions cycles and 66.9% stall cycles, which lead to a 0.331 IPC.

We continue on the first optimized processor, Optimization1. As we have mentioned in section 3.5, the first optimization was to insert a modulo (mod) instruction. The reason for doing so has been explained in detail in section 3.5. In table 4.5 is shown the instruction mix of our benchmark running on processor Optimization1. We see that the frequency of most instructions, has been decreased because these instructions have been used in the modulo operation. It is also shown that the modulo operation appears 9,725 times (0.21%) in total. Lets see some interesting results of table 4.5. The *beqz* decreases a lot and falls to the second most frequent position because it is called in every loop of the modulo operation, while the *sftl* instruction (most frequent instruction), which also exists in the modulo, does not appear in the loop of the modulo. The rest of the instructions that decrease a lot because they appear into the modulo's loop are: *subi, sub, mov, sgtu, li, jr.* The *sgtu* instruction drops to 0%, because modulo was the only operation where it appeared.

Now, by looking at the pairs-column in table 4.5, we can observe that the appearance of the *li-sub, sftru-mov* and *sub-beqz* pairs has been decreased in comparison with table 4.4. Actually, among the referred pairs, only the *li-sub* pair kept its appearance to more than 1%.

Single-Instruction Mix			Pair-Instruction Mix		
sftl	897134	19.185%	and beqz	296928	13.726%
beqz	639298	13.671%	mov and	296928	13.726%
li	578642	12.374%	li sftl	239280	11.061%
add	437984	9.366%	li add	209945	9.705%
mov	363181	7.767%	add lw	203642	9.414%
subi	324349	6.936%	sftr add	148976	6.887%
and	321246	6.870%	sftl sftr	148976	6.887%
jr	308398	6.595%	add sw	138220	6.390%
lw	210048	4.492%	sftl addi	125042	5.780%
sw	149298	3.193%	sw li	55088	2.547%
sftr	148976	3.186%	addi sftl	34738	1.606%
addi	143578	3.070%	lw xor	28260	1.306%
sftru	31964	0.684%	li sub	22614	1.045%
xor	29308	0.627%			
sub	23944	0.512%			
sgt	22823	0.488%			
lb	17059	0.365%			
mod	9725	0.208%			
or	6932	0.148%			
jal	6140	0.131%			
cb	4804	0.103%			
se	1339	0.029%			
not	0	0.000%			
sgtu	0	0.000%			
nop	0	0.000%			
j	0	0.000%			

Table 4.5: Optimization1 Instruction Mix.

Now, we proceed to the comparison of the Base-line and Optimization1 designs. The absolute numbers of cycles are shown in table 4.6. Their comparison is better shown in figure 4.4 and table 4.7. Table 4.7 shows a 25.6% decrease in execution cycles and a 24.8% decrease of the instructions number, in comparison to the baseline processor. The many executing instructions in the modulo function along with the modulo's high appearance, led to that large decrease. Table 4.7 also shows a 26% decrease in stalls, which is related to the big amount of data dependencies that appear in the Base-line's modulo operation. Clearly, these dependencies are causing stalls. The reduction in the number of stalls is the reason for the 0.9% increase in performance of the Optimization1 design.

	#cycles	#Instructions	#Stalls	Perf
Base-Line	18805011	6216401	12588610	0.331
Optimization1	13989619	4676170	9313449	0.334
Optimization1+2	11020338	4082314	6938024	0.370

Table 4.6: Processor-Design comparison.

	#cycles	#Instructions	#Stalls	Perf
Optimization1	-25,6%	-24.8%	-26%	+0.9%
Optimization1+2	-41.4%	-34.3%	-49.9%	+11.8%

Table 4.7: Comparison of the Optimization1 and Optimization1+2 designs, with the base line processor(%)

The next optimization (the second optimization) has been based completely on the instruction mix tables. We see that the group *mov-and-beqz* has the highest frequency in both tables 4.4 and 4.5. So, this fact led us to the decision to collapse these three instructions into one. Thus, we have included in the processor Optimization1+2, the instruction *mandb*. The instruction mix results of that new ISA are shown in table 4.8.

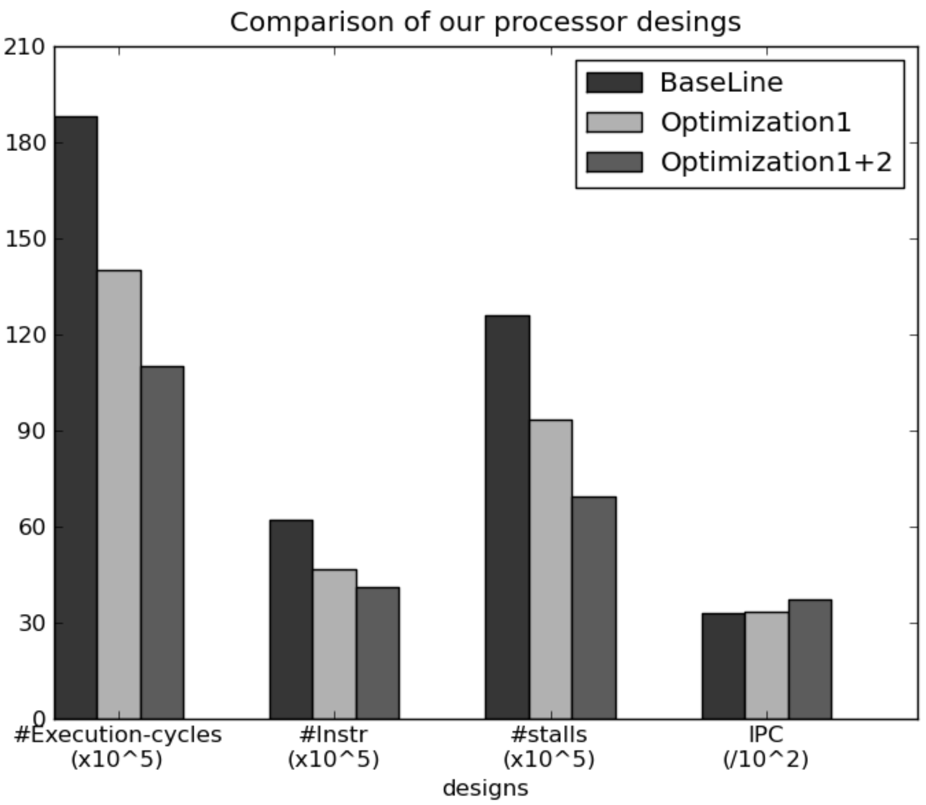


Figure 4.4: Cycle count, instructions count, stall count and performance of each design.

There, we can see that our decision has been correct, because the *mandb* instruction has a frequency of 7.274% of the total instruction mix.

Single-Instruction Mix			Pair-Instruction Mix		
sftl	897134	21.976%	li sftl	239280	15.247%
li	578642	14.174%	li add	209945	13.378%
add	437984	10.729%	add lw	203642	12.976%
beqz	342370	8.387%	sftr add	148976	9.493%
subi	324349	7.945%	sftl sftr	148976	9.493%
jr	308398	7.554%	add sw	138220	8.807%
mandb	296928	7.274%	sftl addi	125042	7.968%
lw	210048	5.145%	sw li	55088	3.510%
sw	149298	3.657%	addi sftl	34738	2.213%
sftr	148976	3.649%	lw xor	28260	1.801%
addi	143578	3.517%	li sub	22614	1.441%
mov	66253	1.623%	sftru mov	21280	1.356%
sftru	31964	0.783%	sub beqz	21280	1.356%
xor	29308	0.718%	mov sgt	21280	1.356%
and	24318	0.596%	add lb	17059	1.087%
sub	23944	0.587%	lw and	16842	1.073%
sgt	22823	0.559%			
lb	17059	0.418%			
mod	9725	0.238%			
or	6932	0.170%			
jal	6140	0.150%			
cb	4804	0.118%			
se	1339	0.033%			
not	0	0.000%			
sgtu	0	0.000%			
nop	0	0.000%			
j	0	0.000%			

Table 4.8: Optimization1+2 Instruction Mix.

In table 4.6 is shown that the Optimization1+2 design needs 2,969,281 less execution cycles than the Optimization1 design (21.2% less). We also have a 593,856 decrease in the instruction number in comparison with Optimization1 (12.7% less). The reduction in the execution cycles and the instruction number is explained by the 13.8% appearance of the *mov-and-beqz* group. So, by collapsing that instruction group, we reduce the number of the execution cycles and the instructions. Finally, we have a 2,375,425 less stalls in comparison to the Optimization1 design (25.5% less). That can be explained by the fact that, if two sequential instructions depend on each other, there occur 3 stalls (we do not have implemented forwarding). Thus, because in our case we had 3 instructions (*mov-and-beqz*) depending on each other, the stall number in the Base-line and Optimization1 processors were 6 every time. So, with the current collapsing we avoid 6 stalls every time that group appears. The above analysis can be shown graphically in figure 4.4.

The second row of table 4.7 shows the overall improvements of the Optimization1+2 design. The Optimization1+2 design reduced: the execution cycles by 41.4%, the instructions number by 34.3% and the stalls by 49.9%. Finally, table 4.7 shows an 11.8% improvement in performance.

Although the Intel XScale core's and the current results can not be compared, we will attempt some comments on them:

- Firstly, if we look at tables 4.6 and 4.1, we can see that all our processors have better performance than the XScale. The reason for that is that in our processors we do not have memory hierarchy, but we only have an instruction memory and a data memory, thus we need 1 cycle to load and store to memory. On the other hand the XScale needs 1 cycle for a cache-hit and 170 cycles for a cache-miss.

- Another interesting thing is that the XScale needs much fewer instructions than our processors for the same benchmark. Remember also that the benchmark has been simplified a lot in order to compile and execute in our processors. So, the question is: why does the XScale need fewer instructions? The answer is that the ISA of the XScale (ARMv5TE compatible) has 32-bit instructions and also consists of instructions with 3 registers (2 source and 1 destination registers). This means that the XScale processor does not need to change a source register to save its result. On the other hand our processor, which uses the one of the two available registers (in the instruction word) as a source-destination, needs more instructions to store and load registers into other registers or into memory.

- We can also observe a difference of the processors in the *xor* appearance-number. That could happen for two reasons. The first reason is that we do not know how the compiler of the XScale maps the C-code into assembly. The second reason is that we have removed the CRC-generation-table function from the second benchmark, which used a lot of *xor* instructions.

Summarizing, in the current section we have succeeded in decreasing the execution cycles to almost half of the Base-line processor and to also increase its performance. The % comparison between Optimization1 and Optimization1+2 designs with the base line design, is shown clearly in table 4.7. The power and energy results -after synthesizing- are following in section 4.3.

4.3 Synthesis Results

In this section we are performing the power, energy, area and timing analysis of our processors by running the selected benchmarks. In order to achieve that we have synthesized -with Synopsys tools- our processor designs which are the Base-line, Optimization1 and Optimization1+2 processors. We have used a 90-nm CMOS technology for ASIC. We have selected to run our processor at 20MHz, however we could run it for a lower frequency. The problem with lower frequencies would be that the leakage power would become a significant portion of the total power consumption. Thus, we have used 20MHz to obtain more meaningful dynamic power result[2]. Finally, we have used Modelsim for simulating our designs and executing our benchmark.

After synthesizing our designs we were able to get area and timing results which are shown in table 4.9. There we can see that the area increases slightly as we move from the Base-Line to the Optimization1 processor and from Optimization1 to Optimization1+2. This is what we have expected because we have added more hardware to Optimization1 and Optimization1+2 designs, as it has been explained in section 3.4. Regarding timing, we can see that the frequency increases slightly as we are optimizing our design.

	Base-Line	Optimization1	Optimization1+2
Area (Logic units)	55295	55959	59158
Critical path (ns)	11.31	11.11	10.83

Table 4.9: Area results and critical-path of the three processor designs.

Then, for getting dynamic power results, we have to run our benchmark in our processors in order to get the switching activity. The synopsys tool must be fed with that activity, which will produce the dynamic power. We have used Modelsim to simulate our processors and to run our benchmark, in order to check the functionality and get the switching activity. First, we have tried to run the whole benchmark in Modelsim, which became practically impossible due to memory and timing limitations. In order to run the whole benchmark and save its switching activity, we would need some hundreds of GB of memory space, which we did not have. Thus, we have broken our benchmark into smaller pieces. The pieces are the functions which are shown in table 4.10. In the first column are presented the executed functions, in the second column the times that each function is repeated in the complete benchmark and in the third, fourth and fifth column are shown the execution cycles of the Base-line, Optimization1 and Optimization1+2 designs, respectively, for a single function execution. From table 4.10, it can be realized that the keys, encryption and decryption functions have modulo included while the empty main and the CRC-calculation functions do not. We can realize that because the cycle number of them has been reduced. Also it can be observed that encryption and decryption algorithms have a 27.2% and 27.6% decrease in execution cycles, respectively, which is a lot. So, here it is shown that especially the encryption algorithm, which is repeated 131 times, is responsible for the large improvement in execution cycles achieved in Optimization1 design -in comparison with the Base-line-. Then, if we look at the third column we can see, that the *mov-and-beqz* group appears in all the functions besides the empty main. In comparison with Optimization1, the Optimization1+2 design has reduced its execution cycles by 29.0% for the key construction and destruction, 21.1% for the encryption, 20.9% for the decryption and 19.4% for the CRC-calculation functions. Although we had the largest decrease in the key-construction and -destruction functions, the improvement in execution cycles of the whole benchmark depends on the encryption algorithm which runs 132 times.

It is true that, dividing our benchmark in parts, does not give us completely accurate power and energy results. This is the price that we have to pay for acquiring synthesis

[2]Additionally, previous findings within the SiMS project indicate the 1-MHz or 10-MHz as the nominal order of magnitude for implants

	times repeated	Base-line #cycles	Optimization1 #cycles	Optimization1+2 #cycles
empty main	1	251	251	251
keys	2	70812-251=70561	66302-251=66051	47102-251=46851
encryption	131	132984-251=132733	96871-251=96620	76391-251=76140
decryption	1	131055-251=130804	94917-251=94892	75077-251=74826
crc calculation	132	8252-251=8001	8252-251=8001	6652-251=6401

Table 4.10: Functions for simulation.

results within a reasonable time frame. The more we increase the benchmark's execution cycles, the better accuracy we get because in the beginning of every program a power-peak is taking place. So, in order to eliminate that peak, the benchmark must run relatively long, for resulting in a better average power consumption. Another idea for getting power and energy results, would be to run the whole benchmark for 100B and to get exact power results. The problem in such a case would be in power and especially in energy consumption calculation. That is because the proportions inside the benchmark would change. For instance, the encryption function in such a situation would run much fewer times. As we know, the encryption function occupies the largest part of the execution cycles, because it is repeated 131 times. In other words the power and energy consumption are mostly depended on the encryption algorithm-and CRC-calculation function-, so, reducing its repetition will lead to inaccurate results.

In tables 4.11, 4.12 and 4.13 are shown the power, energy and execution time results of each design running at 20MHz. In the first column are shown the function names. In the second column are shown the runtime -in ns-, if they were executed once. Hence, we multiply 50ns (20MHz) with the number of execution-cycles of each function. The execution cycles are given in table 4.10, where we can see that we subtract always the execution cycles of the empty main in order to be more accurate. That is because all other the functions are always running in main. In the third column we can see each function's average power -in uW-. The total runtime of each function is located in the fourth column. The function's total runtime is calculated by multiplying its single-runtime (first column) by the repetition-times number. In the last row of this column, we add all the total runtimes, to find the total execution time of the program. In the last column the energy of each function, has been calculated by the known formula $W = P * t$. In the last row of the column is shown the total-energy consumption of the benchmark. Finally, in the last row of column three is given the total-average-power consumption which is calculated by using again the formula $P = W/t$, where W is the total energy of column five and t the total runtime of column four.

The comparison of the three designs can be seen clearer in figure 4.5. In that plot we can see that the power consumption from Base-line to Optimization1 design increases only by 2.2%. We were expecting to have an increase, because we have added more hardware to Optimization1 design. On the other hand, we can see that the overall energy decreases a lot, because the execution cycles have been reduced a lot. Specifically, the energy decreases by 24% from Base-line to Optimization1 design. From Optimization1 to Optimization1+2, we again observe a slight increase in power (5.6%), because we

	runtime(ns)	average power(uW)	total runtime(ns)	energy(uJ)
empty main	12550	220.6	12550	0.003
keys	3528050	231.5	7056100	1.634
encryption	6636650	232.7	869401150	202.310
decryption	6540200	232.5	6540200	1.521
crc calculation	400050	231.3	52806600	12.214
Total		231.577(Avg)	935816600	217.682

Table 4.11: Power, energy and execution-timing results for the Base-line design at 20MHz.

	runtime(ns)	average power(uW)	total runtime(ns)	energy(uJ)
empty main	12550	225.3	12550	0.003
keys	3302550	233.5	6605100	1.542
encryption	4831000	236.8	632861000	149.862
decryption	4744600	236.8	4744600	1.124
crc calculation	400050	233.9	52806600	12.352
Total		236.561(Avg)	697029850	164.883

Table 4.12: Power, energy and execution-timing results for the Optimization1 design at 20MHz.

added again some more hardware. But we can again observe a decrease in energy by 16.4%. So, we have an overall increase in power by 8% while we have a 37% overall decrease in energy (between Base-line and Optimization1+2).

The tables 4.11, 4.12 and 4.13 show that we have power and energy consumption of micro (u) order of magnitude, which is extremely low. However, this is a 5-stage pipeline processor which we expected to have a power consumption of some mW at least. We are going to give some reasons why the results are so low:

- The most important reason is that our processor is not yet packaged, which means that we have not considered power consumption due to input/output pins on the processor. This procedure would increase a lot the power and energy consumption of our processor.
- Our processor supports only essential instructions[3] and features, in order to maintain low-power and -energy consuming. It does not support: forwarding, complex branch-prediction (supports always not-taken), parallelism of functional units, floating-point operations and dedicated hardware for multiplication and division.
- Additionally, the processor does not consist of a complex memory system, so it does not need a memory manager which supports it, which would mean more hardware. There are only two 16-KB memories (instruction and data memories), where everything is stored and loaded.
- Our processor operates at the low frequency of 20MHz.

[3]additionally, the ISA consists of 16-bit instructions

	runtime(ns)	average power(uW)	total runtime(ns)	energy(uJ)
empty main	12550	234.6	12550	0.003
keys	2342550	248.8	4685100	1.166
encryption	3807000	250.1	498717000	124.729
decryption	3741300	249.9	3741300	0.935
crc calculation	320050	245.6	42246600	10.376
Total		249.925(Avg)	549402550	137.209

Table 4.13: Power, energy and execution-timing results for the Optimization1+2 design at 20MHz.

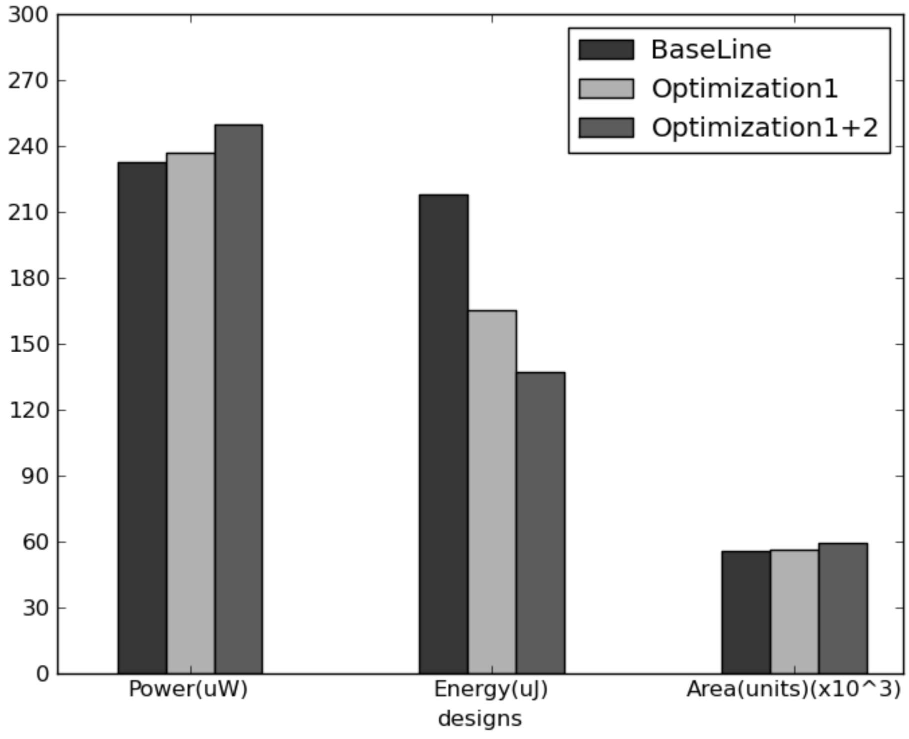


Figure 4.5: Power, energy and area plot.

- In comparison with the XScale core, which is manufactured with a 180-nm technology, our processor is using a 90-nm technology. The 90-nm is much less power/energy consuming, than the 180-nm technology.
- Finally our processor has also a lot of stalls, which means that during many cycles some processor-stages are idling. That means that the performance (IPC) is low which leads to low-power consumptions. However, this does not explain the low-energy consumption, because the energy consumption depends a lot to the execution time. When the execution time decreases the energy consumption

decreases.

4.4 Conclusions

In this chapter we have extensively discussed the synthesis results of this work. Firstly, the reason has been indicated of selecting scenario10 -as the most representative scenario- for being our benchmark. Then, has followed the evaluation of all three implemented processors. Their comparison showed an overall decrease in execution cycles between the base-line and the Optimization1+2 design of 41.4%, an overall decrease in executed instructions number of 34.3%, an overall performance improvement of 11.8% and an overall decrease in energy of 37%. All the above improvements came at the cost of only an 8% overall increase in power and a 7% increase in area. In table 4.14, are shown clearly the improvements of the Optimization1 and Optimization1+2, designs in comparison with the base line design.

	Area	Power	Energy	Execution time
Optimization1	+1.2%	+2.2%	-24.3%	-25.6%
Optimization1+2	+7%	+8%	-37%	-41.4%

Table 4.14: Comparison of the Optimization1 and Optimization1+2 designs, with the base line processor, in terms of area, power, energy and execution time.

Conclusions 5

In this thesis has been explained why Implantable Medical Devices (IMDs) use processors for their functionality and why they need wireless communication with the outside world. However, the wireless communication part adds the need for communication security-implementation. Of course, the processor usage in addition with the communication part increases the power and energy demands of the IMD. This work has given a solution to these problems. A secure-system architecture coupled with a communication protocol have been described. In this system architecture, the IMD has been divided in two parts: the main-function processor and the secure-processor. In this work have been implemented and described the secure-processor together with its C compiler. This chapter summarizes the contents of this work, outlines its contributions and proposes future research directions.

5.1 Summary

As it has been explained in chapter 1, like every electronic device, implants are vulnerable to attacks, especially when they are communicating with the outside world remotely. Thus our goal was to achieve sufficient implant-security with the least energy and power requirements.

Our work is part of the Smart Implantable Medical Systems (SiMS) project which has been described in chapter 2. The goal of this project is to provide to the medical engineers a standard framework for building an Implantable Medical Device. In chapter 2 some essential cryptographic-theory has been presented in detail which was important for our work. Additionally, in that chapter have been described some prior works on building secure IMDs. There, we have found interesting ideas like using Cloakers for changing access-modes and using an RF antenna attached to the IMD for draining energy from the external reader. When the Cloaker is present the IMD works in normal-mode which means that the communication-security is active, otherwise the IMD works in emergency-mode and the security is down. Finally, in chapter 2 is described the MISTY1 cryptographic algorithm, because it has been used in our implementation.

In chapter 3 our system implementation has been described. The implementation has been done in four phases. The first phase has been the system's architecture. The system consists of the IMD attached to an RF antenna for gaining RF power from a reader. The IMD is partitioned into the secure-processor and main-functionality processor. Our work has been done on the secure processor. The second phase has been the communication protocol selection. We have chosen our protocol to have the following characteristics; to provide confidentiality, message authentication and message integrity. We have achieved that by using a two-way communication-protocol in combination with the MISTY1 cryptographic algorithm, the CRC32 hash function and a random number. The third phase has been the secure-coprocessor design, by using the Processor-Designer tool and the implementation of its C-compiler, by using the Compiler-Designer. The fourth and final phase has been the optimization procedure of that Base-line coprocessor. There has been two optimized versions: Optimization1 and Optimization1+2. In Optimization1, the modulo instruction has been added in the ISA, while in Optimization1+2 a new instruction has been added according the instruction-collapsing technique.

Chapter 4 contains all the results of our work. This chapter has been divided into three sections. In the first section we have presented the profiling results of various benchmarks in order to select the most representative one, regarding energy and power, for using it as benchmark for our targeted coprocessor. These benchmarks were actually the communication protocol which accepted as inputs different scenarios. In the second section have been shown the comparison results of the coprocessor alternative designs in terms of execution-cycles, executed-instructions number and performance. The overall decrease in execution-cycles between the base-line and the Optimization1+2 design has been equal to 41.4%, the overall decrease in executed-instructions number has been equal to 34.3% and the overall performance improvement has been equal to 11.8%. In the third and final section the synthesis results have been presented. Thus, it consists of area, dynamic-power and total energy results. There has been slight increase in power and area, because more hardware has been added for the optimizations. The overall increase in power and area has been equal to 8% and 7%, respectively, and the overall decrease in energy has been equal to 37%.

5.2 Thesis Contributions

This thesis addressed several challenging issues regarding the design of the communication-part of Implantable Medical Devices (IMDs), aiming mainly at providing security and keeping low the power and energy consumption. The contributions of our work, while trying to achieve the main goal, are the following:

- First, in this work a system architecture has been defined, which aims to provide zero battery power-consumption. The system-design procedure has been separated in two phases. In the first phase, the communication-part and the main-implant module have been divided. In the second phase we have proposed attaching an RF antenna on the communication part for gaining RF-energy from the reader.

- Then, the communication protocol has been defined. The decisions about the communication protocol have been taken according the defined system architecture, the system's constraints and the possible attacks which we have determined.

- The described communication protocol has been implemented in the C language.

- We have implemented a 5-stage RISC processor to be our secure-processor. For that processor we have also implemented its C-compiler in order to run the implemented communication-protocol. The processor and its compiler have been implemented with the Processor-Designer and the Compiler-Designer which haven't been used previously in the TUDelft. Afterwards we have made two optimized version of the processor and we have synthesized our processors as well. The overall improvement has been equal to 41.4% in execution time, 11.8% in performance, 37% in energy while the power increased by only 8%.

- The final contribution has been that our work is going to be used in the SiMS project.

5.3 Future Work

In this section we are giving some ideas for further investigation regarding our work. These are ideas are as follows:

- More research should be done on the communication protocol. For instance, find an existing protocol which is used in passports and implement it. That would not involve less risk than designing a new one. For our protocol we could use a MAC instead of a CRC and remove the MAC from the encrypted data. The MAC should include the hashing of the encrypted data, which would not require any more the decryption of the whole message in order to compute the MAC. Additionally the Random number scheme should be reconsidered.

- In a future work of this project, should be reconsidered to use an asymmetrical system by using the elliptic curves cryptographic algorithm. We recommend this algorithm because it is not so computationally intensive as other asymmetrical cryptographic algorithms.

- More research should be done regarding the system security. For instance, to make the system secure against jam DoS attacks or against the doctor itself. Also research could be done for finding more potential attacks.

- More investigation is required regarding the commands the IMD receives from a valid reader. Thus, we have left some space into the command format, for adding more possible commands.

- If the program size becomes large, a jalr (*jump and link register* instruction), should be added to the processor's ISA. That is because the program would need to jump to an address that cannot be described by only 12-bits that jal provides.

- Another important improvement that could be done for future work is to apply compiler optimizations. This is a very important aspect, because a large amount of the workload can be transfered from the processor to the compiler. This is very useful because we do not care about how large the compiler's workload is while we care a lot about the processor's. More workload in the processor is translated into more area and more power consumption. For instance, imagine the dependencies being located in the compiler level and not into the processor. This would automatically mean the end of stalling and the removal of stalling's hardware components (lots of comparators). Consequently, that would lead in area and power improvement.

- The processor could also be packaged in order to measure its real power and energy consumption. If we do that we would be sure about the amount of power that the processor will demand to receive from the reader through the RF component.

- Continuing the above idea we should make an investigation for antenna selection. That would mean picking a specific antenna and measuring the receivied power amount from a specific distance. Afterwards a comparison between the received power and the processor's power consumption should occur and discover if the requirements are met.

- Finally, the cloaker scheme for changing from normal to emergency mode could be tried. That would probably add some more possible attacks and remove some others. This would probably differentiate the protocol, as well

Appendix A

This appendix is aiming to help a new user of the Processor- and Compiler-Designer tools to start with them faster and not to spend time at some specific points where we have delayed. This appendix will not give designing-details, because the manuals of the tools are well written and is strongly recommended to be read. The documentation of both tools is located in the ce-sims server at location "/opt/CoWare/V2009.1.1/PD/linux/doc/". Additionally at /opt/applics/cosy-2008/CoSy9701/doc/cover of ce-sims server is located a documentation of the CoSy compiler designing tool. The Compiler-Designer tool is a simplified version of the CoSy tool, thus there tool details can be found. The Processor- and Compiler- Designer tools are the CoWare's property. This appendix is written according to the manuals [5] and [4] and to our observations while working with the tools. The screen-shots are captured from the Processor- and Compiler-Designer tools

Processor Designer

Processor-Designer tool uses the "Language for Instruction-Set Architectures" (LISA) modeling language for describing hardware. LISA is suited to model any architecture that is driven by an instruction set. In other words is used for designing any kind of processors as GPs, RISC, DSPs, ASIPs, co-processors etc. Lets see how to open and start a project using Processor-Designer:

1. Connect to ce-sims server and type to the terminal "ldesigner" for opening the tool.

2. "File → New Project" for starting a new project or "File → Open Project" for opening an existed one.

3. If "New Project" has been selected, there are 3 different options:

    - Empty cycle accurate or empty instruction accurate model. The instruction accurate model it is used just for simulation and for checking functionality. On the other hand with the cycle accurate models real pipeline-processors can be produced. We have began our implementation by an empty cycle accurate model.
    - There also exist models written by CoWare designers. For instance there are RISC, ASIP, VLIW and DSP processors available. You can select to work and change one of them.

- Finally skeleton models are available. They are implemented to be used as starting points in a new design. There exist all the main parts of the processor. If these parts are filled in correctly the processor will work.

Then we are continuing to the part where a first version of the processor has been completed and it is ready for debugging. That could be a processor that supports fetching an instruction, decoding only one instruction (for example addition) and finally executing that instruction. So we are at the point where the processor is ready for its first compiling and its first vhdl(or verilog) generation. The steps are the following:

1. Press the "Processor-generator" button (or press F10), which appears at the panel below the "File, Edit, etc" panel. Then the processor generator will open.

2. Press the Add button. Then select the Create configuration for generic memory interface.

3. Put a configuration name and go to the generation properties. Then you could choose whatever we have chosen and which are shown in figures A.1, A.2, A.3, A.4, A.5. After finishing the configuration press OK.

4. Then select a configuration from the list in the left (probably the one that you have just generated). If everything has been syntactically correct the vhdl processor-implementation will be generated.

5. The way we have selected for simulating our designs has been by using modelsim.

6. For starting the execution we must fill in the program memory with instructions. For doing that we propose to build the assembler and the linker. In such a way you do not have to write any program in binary for testing. The procedure for simulating with modelsim is the following:

    - Configure the linker by the including the following text file(linker.cmd) in the bin folder of your project, which is located there where the assembler(lasm) will be appeared :

      ```
      MEMORY

      PROG : origin = 0x00000000, length = 0x2000
      DATA : origin = 0x00002000, length = 0x4000, bytes=1


      SECTIONS

      .text: load = PROG

      (.boot)
      (.text)
      ```

```
.data: load = DATA
.rodata: load = DATA
.bss: load = DATA

(.bss)
(COMMON)

.boot: load = PROG  *(.empty)
```

The important information that will change between different implementations are the PROG and DATA lines in MEMORY section. There you must specify to the linker where do the program and data memory begin and where do they end. In our case the program memory begins at address 0x0 and ends at address 0x2000. Which means that there are 0x2000 words for the program memory. The word width in our program memory is 16-bits(2-bytes), which results in 16KB memory. The alignment in that situation is the same with the words, namely 2-bytes. The data memory begins in address 0x2000 and contains 0x4000 words(1-byte) which results in 16KB memory. The above is very important for the alignment of the data memory. If this file is not included the program will probably overwrite data in data memory, especially if a compiler is built.

- After doing that press ”Build-Assembler+Linker” from the second panel of the main Processor-Designer’s window. Then in bin directory of your project, the assembler (lasm) and linker (llnk) will be generated. Then we can write an assembly code as the following:

```
.text

addi r1,0xff
sftl r1,8
addi r1,0xff
sftl r1,8
addi r1,0xff
sftl r1,8
addi r1,0xff
nop
nop
nop
sw r1,r0
```

Be careful to leave a whitespace at the start of each line of the assembly code.

- Then assemble the desired file by using:

```
./lasm yourFileName.s
```

./llnk yourFileName.lof linker.cmd

After that a a.out executable file will be generated.

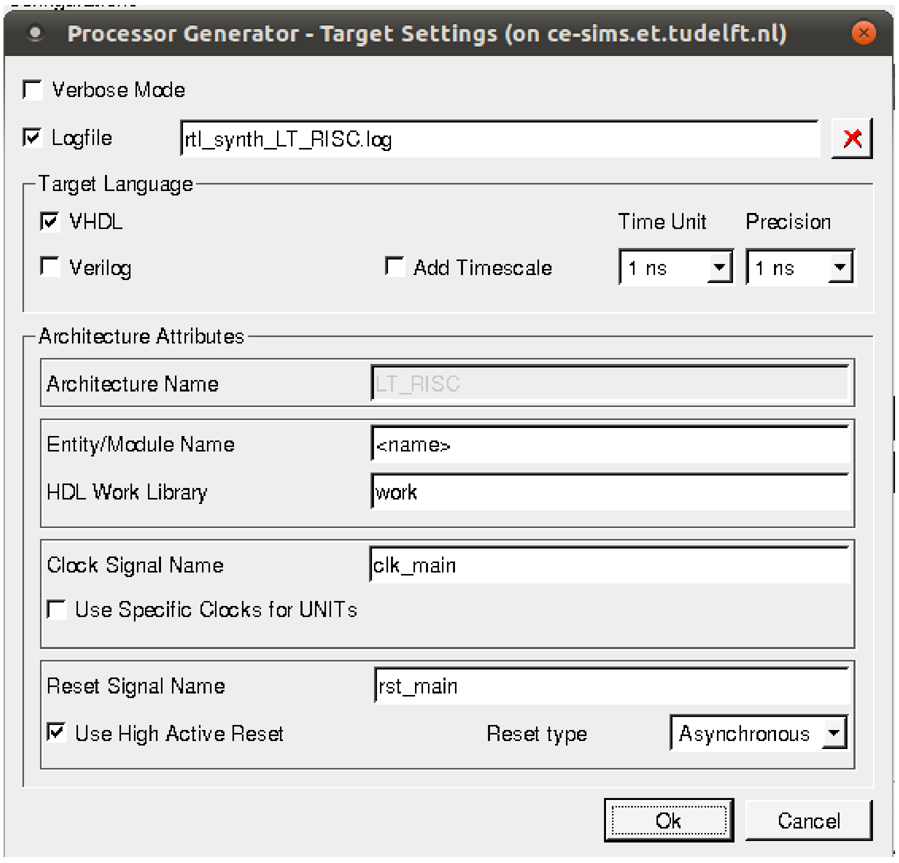
- Copy the a.out file in hdl_gen folder (is located, where bin is located).
- There you should run the command: "exe2txt -i hdl_memory_configuration.txt -e a.out". This command will produce from the executable, the program and data memories, which are called contents_prog_mem.mmap and contents_data_mem.mmap respectively.
- Then copy these two files into ".../hdl_gen/vhdl/sim_modelsim" in order to become your processor's program and data memories. Then go into this directory.
- Then run "./CreateMakefile" for generating the work directory. Run this command only at the first time and when the vhdl code has been changed.
- Finally run "make gui" for opening the modelsim simulator. There by watching into the register-file, in the data memory and the vhdl signal it is possible to debug your processor. If there is a problem you must return back and correct it at lisatek level and NOT in VHDL.

This is the procedure that we have followed for simulating and debugging our processor. Ofcourse if you read more carefully the manual you could find more and probably more efficient ways for that.

Now we are going to say a few words about designing a cycle-accurate processor with Processor-Designer tool. The basic structure of the LISA language is derived from the following distinction:

- The storage elements: The storage elements (also called resources in LISA) can be, (pipelined)registers, global variables, I/O pins and memories. The resources are global and they can be used by any operation in the program.
- The operations: They describe instructions and instruction-dependent functions such as the fetch mechanism. Each operation can have four discrete sections:
  - Declaration: in this section are declared LABELS, Instances, References, GROUPS, etc, which are used for exchanging information between discrete operations.
  - Syntax: This section specifies the assembly syntax and connects each instruction with Operations and its binary representation.
  - Coding: This section specifies the binary representation of the instruction. The syntax and the coding sections are used by the tool for building the assembler.
  - Behavior: In the section the functionality of each operation is implemented.

Follows an operation example:
$INSTRUCTION ri\_type IN pipe.DE$


Processor Generator - Target Settings (on ce-sims.et.tudelft.nl)

☐ Verbose Mode

☑ Logfile: rtl_synth_LT_RISC.log

Target Language

☑ VHDL

☐ Verilog

☐ Add Timescale

Time Unit: 1 ns

Precision: 1 ns

Architecture Attributes

Architecture Name: LT_RISC

Entity/Module Name: <name>

HDL Work Library: work

Clock Signal Name: clk_main

☐ Use Specific Clocks for UNITs

Reset Signal Name: rst_main

☑ Use High Active Reset

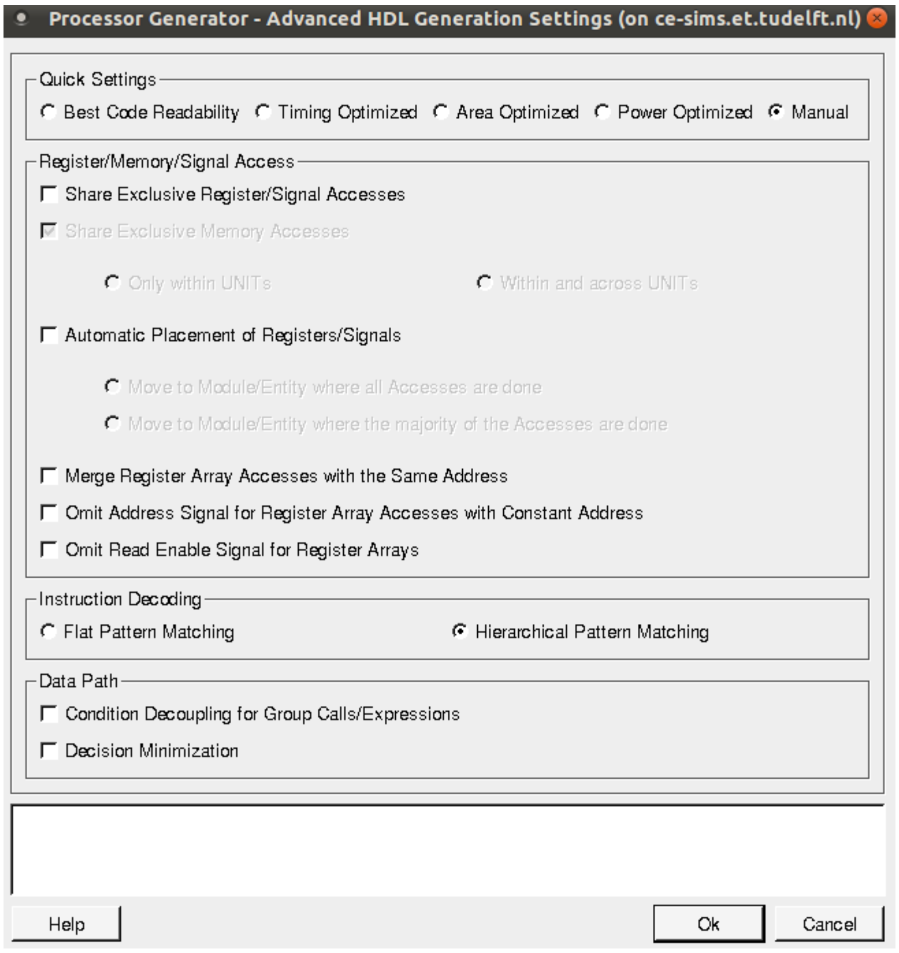
Reset type: Asynchronous

Figure A.1: Target Settings.

$\{$
$DECLARE$
$\{$
$GROUPopcode = \{sftr||sftl||sftru||li||addi||subi\};$
$GROUPrd = \{reg\};$
$GROUPimm = \{immd8\};$
$\}$
$CODING\{opcoderdimm\}$
$SYNTAX\{opcode\ ""rd","imm\}$

$BEHAVIOR$
$\{$
$OUT.opr1 = GPR[rd];$
$OUT.opr2 = imm;$
$OUT.opr\_dest = rd;$

$byte\_mem\_en = false; //isnotgoingtoread/writeabyte$



Processor Generator - Advanced HDL Generation Settings (on ce-sims.et.tudelft.nl)

Quick Settings

Best Code Readability Timing Optimized Area Optimized Power Optimized [✓ Manual]

Register/Memory/Signal Access

- Share Exclusive Register/Signal Accesses
- [✓ Share Exclusive Memory Accesses]
  - Only within UNITs Within and across UNITs
- Automatic Placement of Registers/Signals
  - Move to Module/Entity where all Accesses are done
  - Move to Module/Entity where the majority of the Accesses are done
- Merge Register Array Accesses with the Same Address
- Omit Address Signal for Register Array Accesses with Constant Address
- Omit Read Enable Signal for Register Arrays

Instruction Decoding

Flat Pattern Matching [✓ Hierarchical Pattern Matching]

Data Path

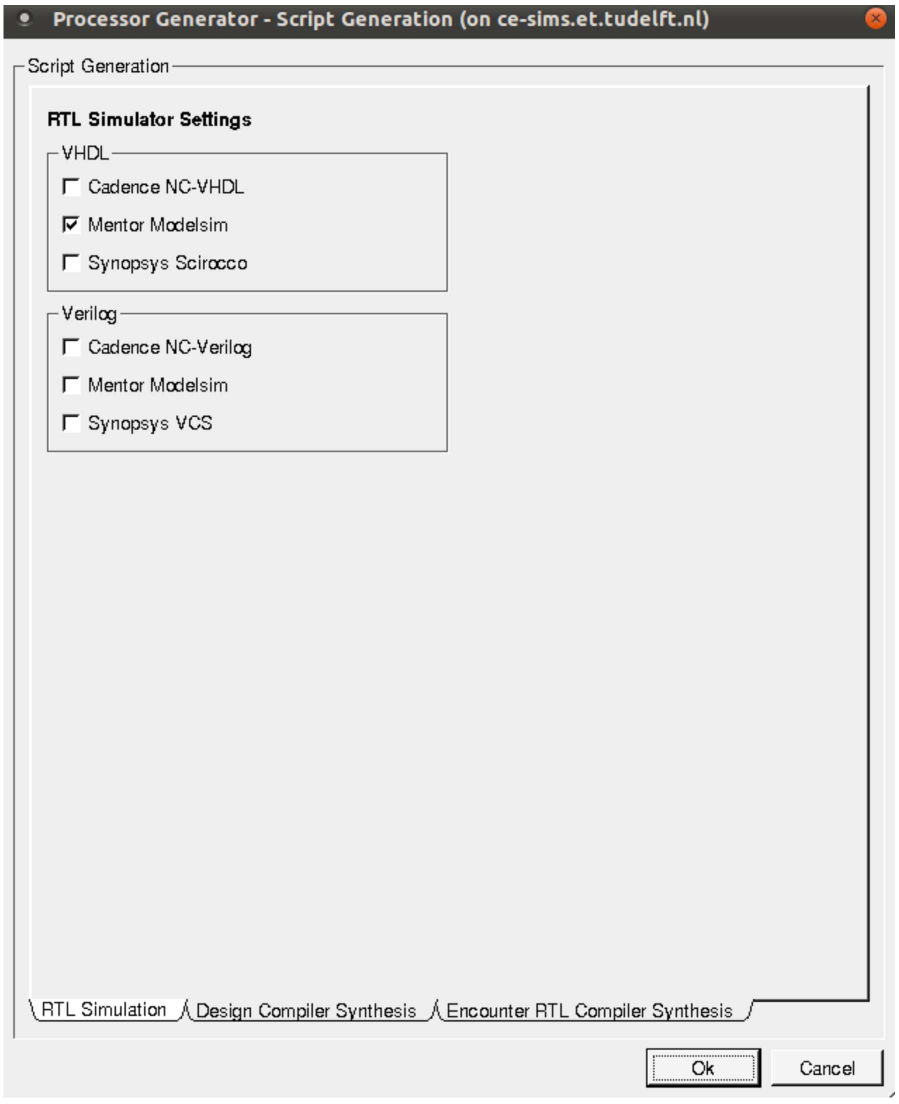
- Condition Decoupling for Group Calls/Expressions
- Decision Minimization

Figure A.2: Advanced HDL Generation Settings.

$OUT.wr\_en = true;$

$if((EX.IN.wr\_en \&\& (EX.IN.opr\_dest == rd)) || (MEM.IN.wr\_en \&\& (MEM.IN.opr\_dest == rd)) || (WB.IN.wr\_en \&\& (WB.IN.opr\_dest == rd)))$
$\{$
$stall\_en = true;$
$\}$
$else\{$
$stall\_en = false;$
$\}$

$temp\_wren = false;$
$temp\_rd\_mem\_en = false;$


Processor Generator - Script Generation (on ce-sims.et.tudelft.nl)

Script Generation

RTL Simulator Settings

VHDL
- [ ] Cadence NC-VHDL
- [✓] Mentor Modelsim
- [ ] Synopsys Scirocco

Verilog
- [ ] Cadence NC-Verilog
- [ ] Mentor Modelsim
- [ ] Synopsys VCS

Figure A.3: Script Generation, "RTL Simulation" tab.

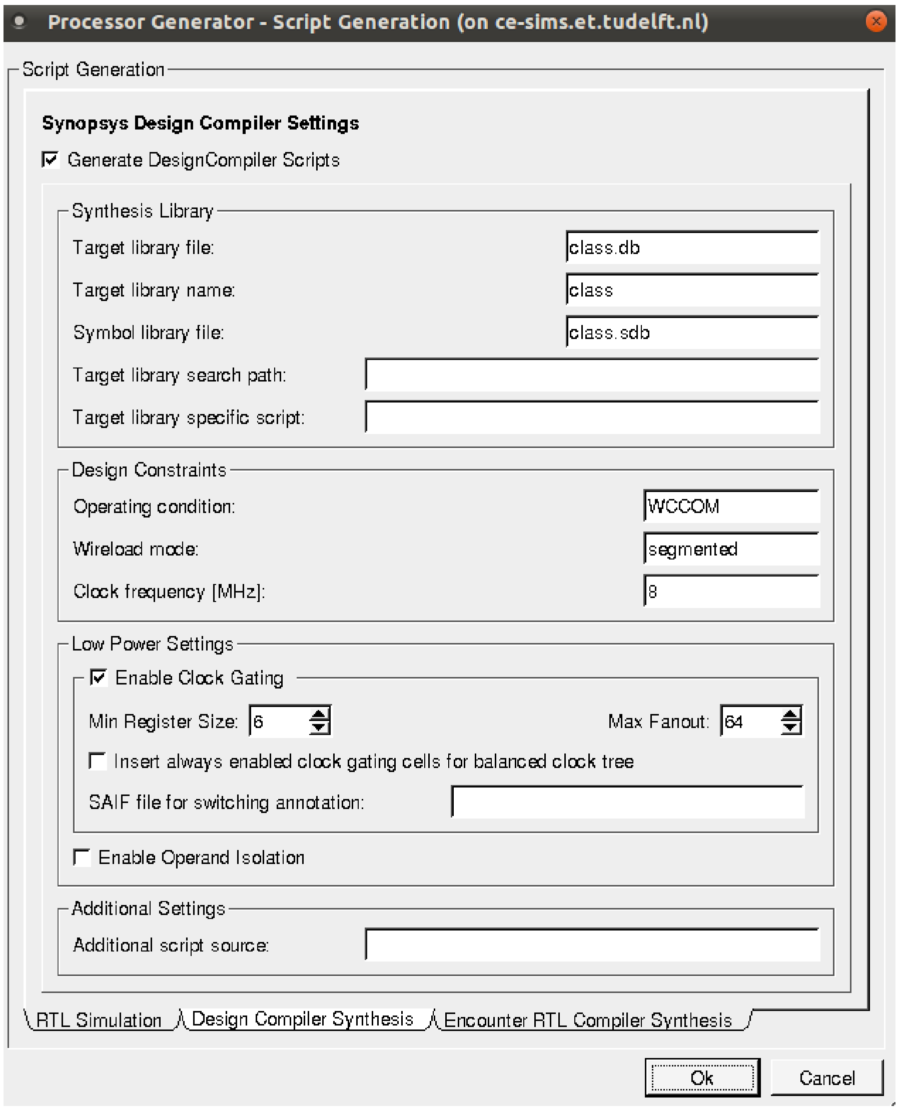
}

$ACTIVATION\{opcode\}$

}

Explanation:

GROUP: This word is used for building an operation hierarchy. For instance in our example we have a group called opcode which contains all the immediate format instructions. These instructions are described further in different operations. Activation is a keyword that is called for jumping in other operation. In our case we are jumping


Processor Generator - Script Generation (on ce-sims.et.tudelft.nl)
Script Generation
RTL Simulation
Design Compiler Synthesis
Encounter RTL Compiler Synthesis
Ok
Cancel

Synopsys Design Compiler Settings

☑ Generate DesignCompiler Scripts

Synthesis Library

Target library file:	class.db
Target library name:	class
Symbol library file:	class.sdb
Target library search path:	
Target library specific script:	

Design Constraints

Operating condition:	WCCOM
Wireload mode:	segmented
Clock frequency [MHz]:	8

Low Power Settings

☑ Enable Clock Gating

Min Register Size: 6 Max Fanout: 64

☐ Insert always enabled clock gating cells for balanced clock tree

SAIF file for switching annotation:

☐ Enable Operand Isolation

Additional Settings

Additional script source:

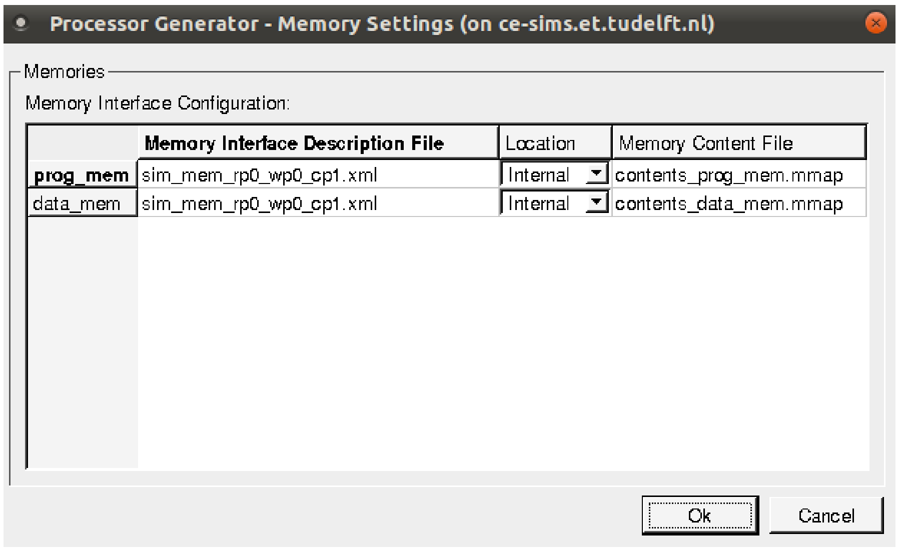
Figure A.4: Script Generation, "Design Compiler Synthesis" tab.

in one of the opcode operations.

For the processor implementation there must always exist a fetching mechanism as well as the main and reset operations. The fetching mechanism increases the Program Counter and reads an instruction from the program memory.

Compiler Designer

The Compiler-Designer tool automatically builds the front-end of a C-compiler. So, only the back-end is left to the user for implementation. The back-end consists of the following parts:


Processor Generator - Memory Settings (on ce-sims.et.tudelft.nl)
Memories
Memory Interface Configuration:
Ok
Cancel

	Memory Interface Description File	Location	Memory Content File
prog_mem	sim_mem_rp0_wp0_cp1.xml	Internal	contents_prog_mem.mmap
data_mem	sim_mem_rp0_wp0_cp1.xml	Internal	contents_data_mem.mmap

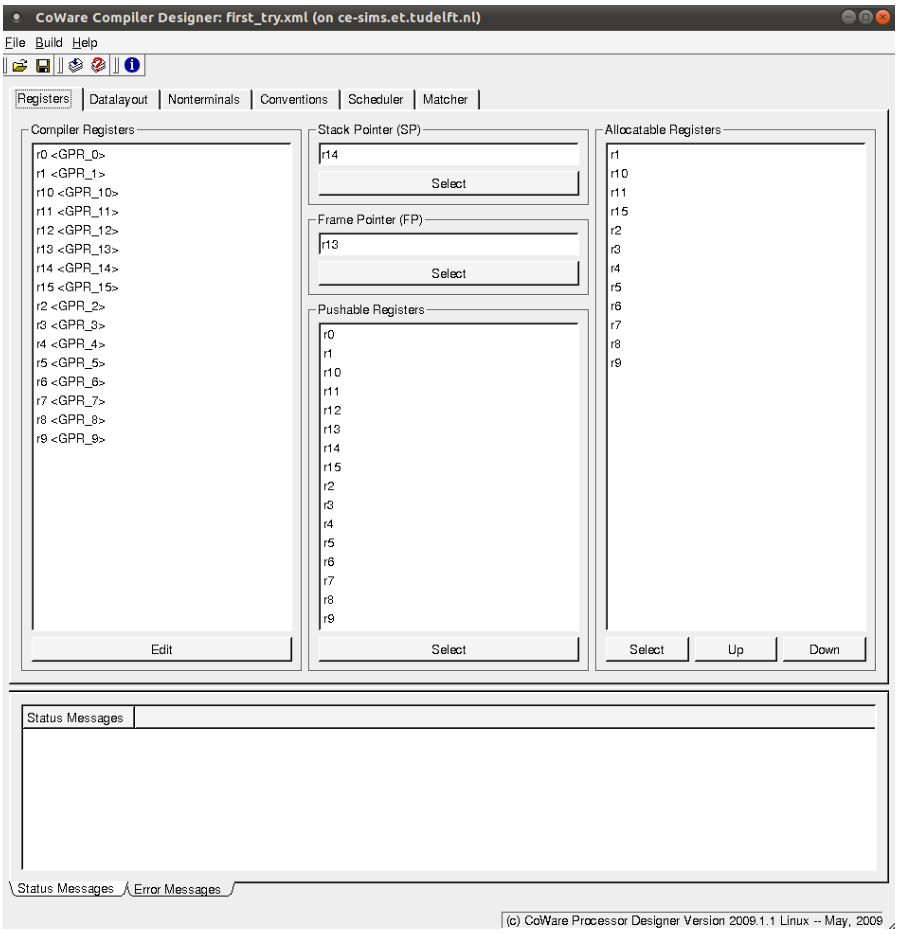
Figure A.5: Memory settings.

- **Matching:** The compiler maps the C-code to assembly instructions. The designer must define the rules that binds C-code to assembly.
- **Register Allocation:** The compiler assigns registers to variables. The designer must specify to the tool the available registers and their purpose.
- **Scheduling:** The compiler puts the code in an an order regarding the architecture latency constraints and optimizations. If the designer wants a scheduler should specify the dependencies and their latency.
- **Emit:** After the compiler completes the above operations produces the assembly file.

Register Allocation: In figure A.6, we show the allocatable registers of our design, which the compiler is free to use for assigning variables and values. From these registers are missing the stack-pointer (r14) the fp (r13) and r12. r12 is not allocatable because we have chosen to use it as a temporary register in our matching rules. The stack- and frame-pointer are used only by the stack. Additionally in figure A.7, are shown the callee-changed registers and the registers that can be used as input (and output) to (from) functions. The registers that are not specified at the callee-changed list are the caller-changed registers.

Matching: As we have already mentioned in section 3.5 the front-end creates and then, provides to the back-end an Internal Representation (IR) of the C program in the form of a tree. The nodes of the tree are called mir (medium-level IR). Then, the back-end matches assembly instructions to the IR representation. The IR representation of a simple piece of C code is shown in figure A.8.

Let's describe a bit the figure. The *mirContent* node is created when a load from memory appears in C. In our example we need to load the value of *b* from memory,



CoWare Compiler Designer: first_try.xml (on ce-sims.et.tudelft.nl)

Compiler Registers

```
r0 <GPR_0>
r1 <GPR_1>
r10 <GPR_10>
r11 <GPR_11>
r12 <GPR_12>
r13 <GPR_13>
r14 <GPR_14>
r15 <GPR_15>
r2 <GPR_2>
r3 <GPR_3>
r4 <GPR_4>
r5 <GPR_5>
r6 <GPR_6>
r7 <GPR_7>
r8 <GPR_8>
r9 <GPR_9>
```

Stack Pointer (SP): r14

Frame Pointer (FP): r13

Pushable Registers

```
r0
r1
r10
r11
r12
r13
r14
r15
r2
r3
r4
r5
r6
r7
r8
r9
```

Allocatable Registers

```
r1
r10
r11
r15
r2
r3
r4
r5
r6
r7
r8
r9
```

(c) CoWare Processor Designer Version 2009.1.1 Linux -- May, 2009

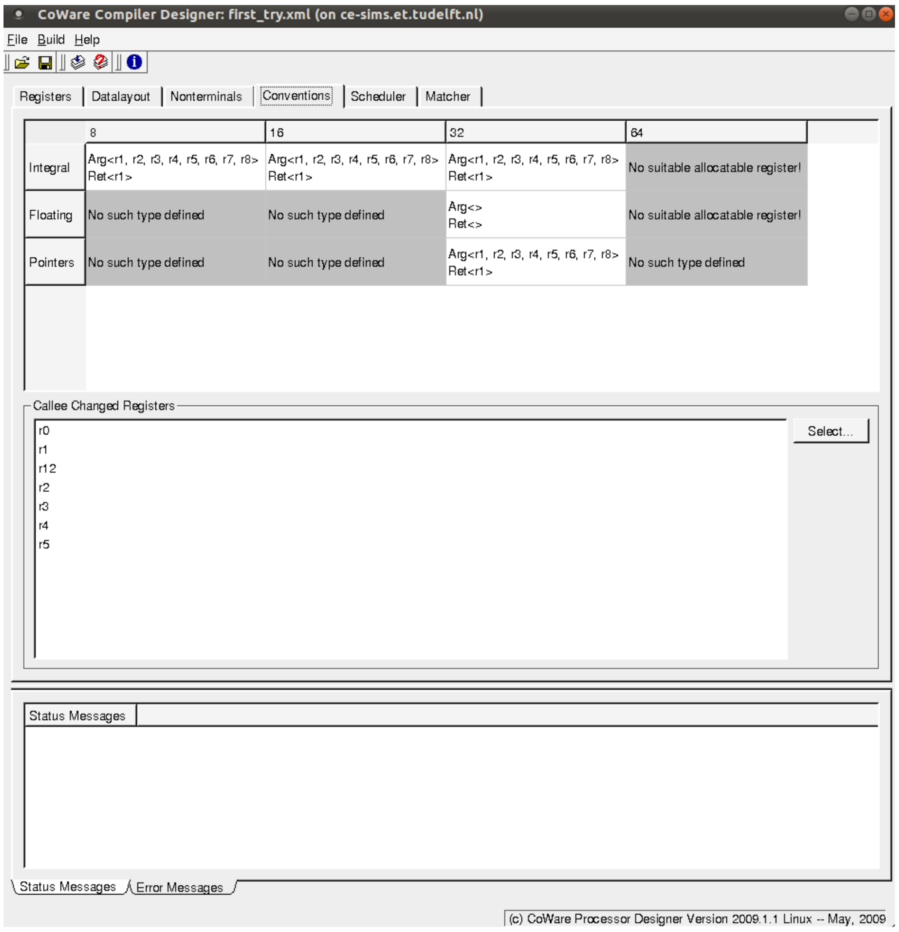
Figure A.6: Allocatable Registers.

thus the *mirContent* appears. Clearly the *mirContent* needs to get as input the memory address of b variable. That is why the mirObjAddr node exists, which feeds the *mirContent* with the *b* address. *MirIntConst* node describes a constant value, which in our example is the *1*. Then, the *mirPlus* describes that there exists an addition. In our case that addition is between a loaded value and a constant. Finally, the *mirAssign* node describes the storing of the addition result to the (right mirObjectAddr node) memory address of *a*.

The matcher's work is to map the whole IR tree into assembly instructions. This is done through rules, which map patterns of mir nodes to assembly instructions. Such rules are shown also in figures in section 3.5.

Scheduling : In figure A.9 is shown the scheduler tab. We did not occupy with the scheduling part in our work.

We continue by describing how to use the tool. As we have mentioned to the Processor Designer part, before building the compiler, the ISA should already be implemented. So, we assume that the processor and its ISA have been implemented by



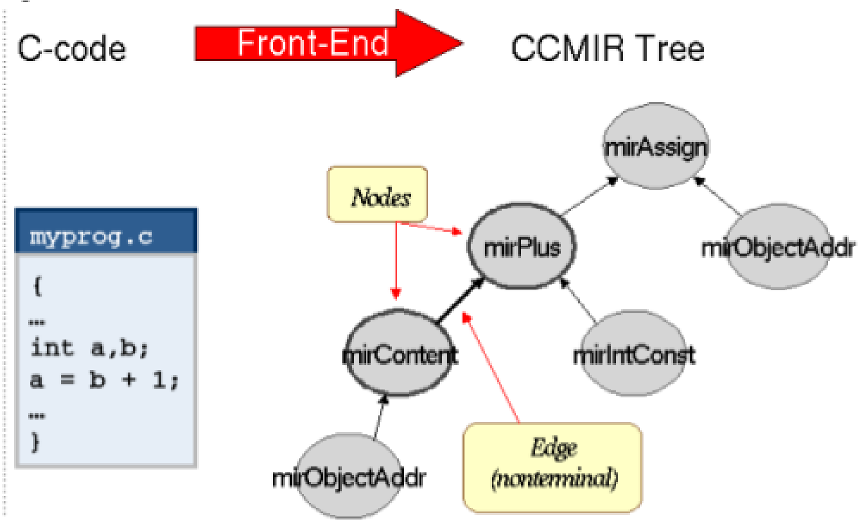
	8	16	32	64
Integral	Arg<r1, r2, r3, r4, r5, r6, r7, r8> Ret<r1>	Arg<r1, r2, r3, r4, r5, r6, r7, r8> Ret<r1>	Arg<r1, r2, r3, r4, r5, r6, r7, r8> Ret<r1>	No suitable allocatable register!
Floating	No such type defined	No such type defined	Arg<> Ret<>	No suitable allocatable register!
Pointers	No such type defined	No such type defined	Arg<r1, r2, r3, r4, r5, r6, r7, r8> Ret<r1>	No such type defined

Callee Changed Registers

```
r0
r1
r12
r2
r3
r4
r5
```

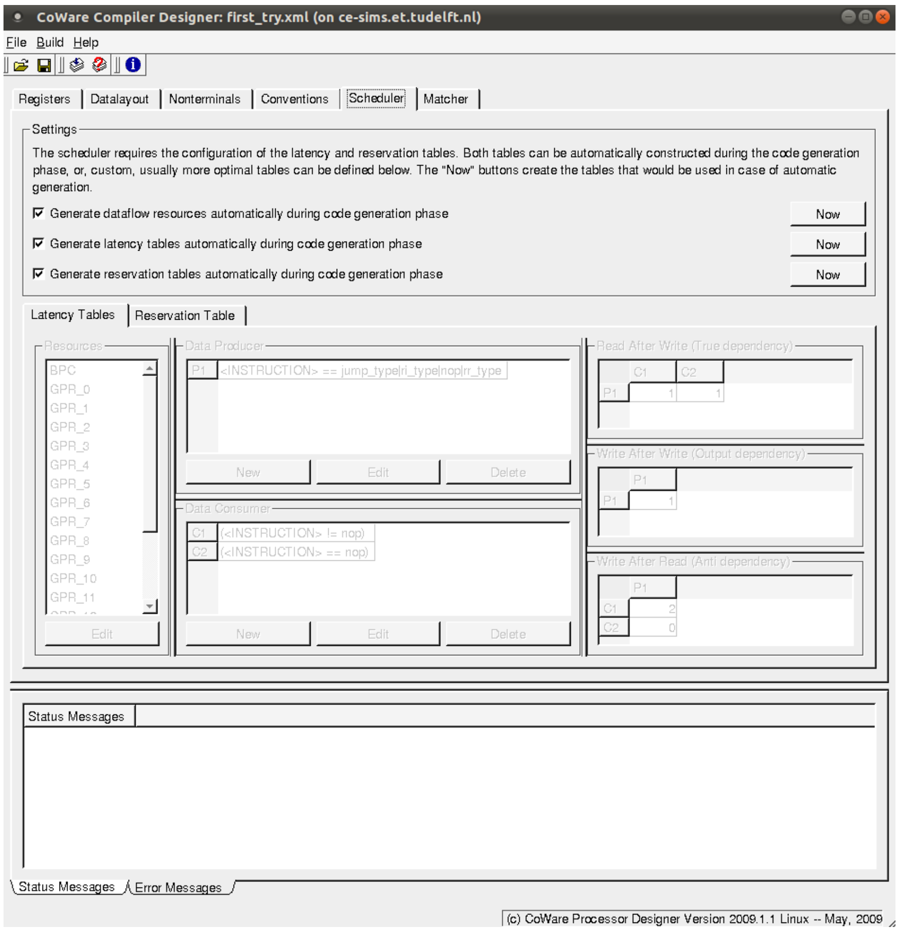
(c) CoWare Processor Designer Version 2009.1.1 Linux -- May, 2009

Figure A.7: Argument and callee-changed/caller-changed registers.



```
{
...
int a,b;
a = b + 1;
...
}
```

Figure A.8: CCMIR Tree(source from C Compiler Designer Guide).



Settings

The scheduler requires the configuration of the latency and reservation tables. Both tables can be automatically constructed during the code generation phase, or, custom, usually more optimal tables can be defined below. The "Now" buttons create the tables that would be used in case of automatic generation.

- [✓] Generate dataflow resources automatically during code generation phase
- [✓] Generate latency tables automatically during code generation phase
- [✓] Generate reservation tables automatically during code generation phase

Resources: BPC, GPR_0, GPR_1, GPR_2, GPR_3, GPR_4, GPR_5, GPR_6, GPR_7, GPR_8, GPR_9, GPR_10, GPR_11

Data Producer

P1	<INSTRUCTION> == jump_type|ri_type|nop|rr_type

Data Consumer

C1	(<INSTRUCTION> != nop)
C2	(<INSTRUCTION> == nop)

Read After Write (True dependency)

	C1	C2
P1	1	1

Write After Write (Output dependency)

	P1
P1	1

Write After Read (Anti dependency)

	P1
C1	2
C2	0

(c) CoWare Processor Designer Version 2009.1.1 Linux -- May, 2009

Figure A.9: Scheduling tab.

the Processor Designer. From the Processor Designer main window we press the button for opening the Compiler Designer. If we try to build the compiler at that point, the tool will return errors which informs that essential rules are missing (as the stack management rules). Hence, our first work, after the allocating registers, is to specify these essential rules until no error appears. When the compiler is built for the first time the lcc driver and the lisacc compiler are created in the ".../bin" folder of the project. The lcc driver just calls, in the following order: the compiler (lisacc), the assembler (lasm) and the linker (llnk). The next step after the initial compiler construction is to test some C programs. At first we test programs that only consist of the main() function. Such programs are provided together with the tool and are located in folder "/opt/CoWare/V2009.1.1/PD/linux/cosy/lib/compilertrainer/src" in the ce-sims server. The designer after selecting the compiler features he wants to support, he choses and tries to compile the related programs included in there. Usually most of the programs in Level0 folder should be able to compile. When the compiler is not able to compile a certain program returns which rule is missing. In that case the designer should return to the Compiler Designer and add that rule. In the case where the program consists of functions along with the main one, the designer should take care of it. For instance, we did the following:

- We compiled the program by using the *lcc -S program.c* command which created the *program.s* assembly file.

- Then we added in the beginning of the *.text* part of the assembly file (*program.s*) the next assembly instructions:
  li $r1$, $((\_main >> 8)\&0xff)$
  $sftl$ $r1$, 8
  addi $r1$, $(\_main\&0xff)$
  li $r14$, $0x1f$
  $sftl$ $r14$, 8
  addi $r14$, $0xfc$
  jr $r1$, $r0$

  The above code makes the program counter (PC) to jump to the main() function and also initializes the stack-pointer (r14), to the last data memory address.

- It is a good idea to remove the *jr r15*, which appears at the end of the assembly, which returns the PC in the beginning of the instruction memory. If this instructions is left there the program will loop and it will be difficult to be debugged in the Modelsim simulator.

After the *a.out* execution program is created, we then continue as we described in the Program-Designer part.

Possible problems while building the compiler are following:

- Missing rules

- Mistake in the mapped assembly of a rule

- Mistake in the processor itself

Bibliography

[1] *Misty1 RFC, http://tools.ietf.org/html/rfc2994*, November 2000.

[2] *Modern Compiler Design*, Wilet, 2000.

[3] *SiMS project, http://sims.et.tudelft.nl*, September 2005.

[4] CoWare, *C Compiler Design Guide.*

[5] CoWare, *Processor Design Guide.*

[6] C.Strydis, *"Suitable cache organizations for a novel biomedical implant processor"*, The 26th IEEE International Conference on Computer Design, Lake Tahoe, California, USA (2008).

[7] C.Strydis and G. N. Gaydadjiev, *"The Case for a Generic Implant Processor"*, 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC 08), pp. 3186-3191, Vancouver, Canada (2008).

[8] C.Strydis, G.N. Gaydadjiev, and S.Vassiliadis, *"A New Digital Architecture For Reliable, Ultra-Low-Power Systems"*, Proceedings ProRISC 2006, November 2006, pp. 350–355.

[9] C.Strydis, G.N.Gaydadjiev, and S.Vassiliadis, *"An extensive survey of microelectronic implants"*, ACES (2006).

[10] Dhara Dave, *"Automated Implant-Processor Design"*, August 2010.

[11] D. Halperin et al, *"Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses"*, IEEE Symposium on Security and Privacy (2008).

[12] D.Halperin et al, *"Security and privacy for Implantable Medical Devices"*, IEEE (2008).

[13] E. Freudenthal et al, *"Practical Techniques for limiting disclosure of RF-equipped medical devices"*, IEEE (2007).

[14] G. Contreras et al, *XTREM: A Power Simulator for the Intel XScale Core*, LCTES04 (2004).

[15] H. Chae et al, *"Maximalist Cryptography and Computation on the WISP UHF RFID Tag"*.

[16] Kasper B. Rasmussen et al, *"Proximity-based Access Control for Implantable Medical Devices "*, CCS'09 (2009).

[17] S.K.S. Gupta et al, *"Criticality Aware Access Control Model for Pervasive Applications"*, IEEE (2006).

[18] Sriram Cherukuri et al, *"BioSec: A Biometric Base Approach for Securing Communication in Wireless Networks of Biosensors Implanted in the human body"*, IEEE (2003).

[19] T. Denning et al, *"Absence makes the heart grow fonder: new directions for implantable medical device security"*, (2008).

[20] J. Hennessy and D. Paterson, *Computer architecture book*, D. Penrose, 1990.

[21] Z. Herczeg and D. Schmidt, *Energy simulation of embedded XScale systems with XEEMU*, Journal of Embedded Computing - PATMOS 2007 selected papers on low power electronics (2009).

[22] Behrooz Parhami, *Computer arithmetic algorithm and hardware designs*, Oxford University Press, 2000.

[23] D. P. Riemens, C. Strydis, and G.N. Gaydadjiev, *"Exploring suitable adder designs for biomedical implants"*, November 2010.

[24] C. Strydis, *Implantable microelectronic devices, pp. 171*, July 2005.

[25] C. Strydis and G. N. Gaydadjiev, *"Profiling of Lossless-Compression Algorithms for a Novel Biomedical-Implant Architecture"*, The 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis, pp. 109-114, Atlanta, Georgia, USA, October 2008 (2008).

[26] C. Strydis, G.N.Gaydadjiev, and S.Vassiliadis, *"A generic digital architecture and compiler for implantable devices"*, ACES (2005).

[27] C. Strydis, C. Kachris, and G. N. Gaydadjiev, *"ImpBench: A novel benchmark suite for biomedical, microelectronic implants"*, The 26th IEEE International Conference on Computer Design, Lake Tahoe, California, USA (2008).

[28] C. Strydis, D. Zhu, and G. N. Gaydadjiev, *"Profiling of Symmetric-Encryption Algorithms for a Novel Biomedical-Implant Architecture"*, CF'08 (2008).

[29] J.C.A. van der Lubbe, *"Basic Methods of Cryptography"*, VSSD, 1998.