Neptun - ECC Processor for RFID Tags and Smart Cards

Erich Wenger
wengere@student.ethz.ch
erich.wenger@student.tugraz.at

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Integrated Systems Laboratory
Swiss Federal Institute of Technology
Gloriastrasse 35
CH-8092 Zürich, Switzerland

Graz University of Technology

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master Thesis

Supervisor: Dr. Norbert Felber, ETH Zürich
Dr. Martin Feldhofer, TU Graz
Assessor: Dr. Karl-Christian Posch, TU Graz

May, 2010

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

Erich Wenger

Acknowledgements

First I would like to thank Martin Feldhofer from Graz University of Technology. He was the first contact for all cryptography-related questions. I am looking forward to working with him in the future.

Secondly I would like to thank Norbert Felber and the whole Integrated Systems Laboratory team at the ETH in Zürich. They have been a great support for all my hardware-relevant questions. Thank you for patience and your insight to ASIC design. It was a great pleasure working with this well experienced team. Especially Norbert Felber's constructive criticism was very helpful and highly appreciated. His experience helped me greatly.

Finally I would like to thank my girlfriend and my family for their unwaivering support and understanding over the last years.

Abstract

Radio frequency identification (RFID) tags and smart cards already exist for years. These technologies are used for many different applications. Today, they are on the brink of conversion. The goal is it to combine the advantages of both technologies. The new generation of secure RFID tags should still be cheap and passively powered by an electromagnetic field (which is also used for communication with a reader), but they should also be capable of strong cryptographic algorithms.

The impact of such a product will be huge. Bar-codes will be replaced. Product piracy could be completely stopped using cloning-resistant RFID tags.

In order to achieve that goals, elliptic curve cryptography is used. The elliptic curve digital signature algorithm (ECDSA) requires small keys that result in a very small memory requirement. This makes ECDSA the best suited public-key authentication scheme currently known.

This work presents two different designs. One design called *Neptun* is a full-custom, embedded processor that can be used to evaluate different algorithms and security measures. This is especially important because the complex algorithms can be vulnerable to simple and differential power analysis attacks. *Neptun* is a programmable processor with an custom instruction set that was specially optimized for prime fields. The design also provides a bootloader, an EIA-232 interface, three timers and a parallel I/O. It has been implemented and fabricated as an application-specific integrated circuit (ASIC) using the UMC-L180 technology.

The elliptic curve digital signature generation and verification algorithms have been implemented using a custom built processor simulator. This simulator is capable of parsing assembler source code and generating executables.

The second design is based on *Neptun*. The main difference is that it has been stripped from any unnecessary logic, but no compromise has been made concerning the security of the algorithms. This design is used to show that the presented architecture is viable for RFID tags. With the small area usage of only 14230 gate equivalents and a runtime of 1.65 million cycles, the design is competitive with designated co-processors and the smallest implementation for generating an ECDSA signature for NIST P-192, published so far.

With these results a big step to secure RFID tags has been done.

Keywords: Processor, Radio Frequency Identification, Elliptic Curve Cryptography, Digital Signature, ASIC

Kurzfassung

Radio-Frequenz-Identifikations (RFID) Tags und Chipkarten existieren bereits seit Jahren. Diese Technologien werden für viele verschiedene Anwendungen genutzt. Heute haben sie den Punkt der Zusammenführung erreicht. Das Ziel ist es die Vorteile beider Technologien zu kombinieren. Die neue Generation von sicheren RFID-Tags soll billig und über ein elektromagnetisches Feld versorgt sein (dieses Feld wird auch zur Kommunikation mit einem Lesegerät benutzt). Aber sie sollen auch fähig sein, starke kryptografische Algorithmen zu berechnen.

Die Auswirkungen eines solchen Produkts werden immens sein. Barkodes werden ausgetauscht. Produktpiraterie könnte komplett gestoppt werden durch die Benutzung von kopierresistenten RFID Tags.

Um diese Ziele zu erreichen, wird Elliptische-Kurven Kryptografie verwendet. Der Elliptic Curve Digital Signature Algorithm (ECDSA) verwendet kurze Schlüssellängen. Dies bewirkt eine geringe Speicheranforderung. Damit ist ECDSA das am besten geeignete asymmetrische Authentifizierungsverfahren.

Diese Arbeit präsentiert zwei Designs. Ein Design, mit dem Namen *Neptun* ist ein maßgeschneiderter, eingebetteter Prozessor, der zur Evaluierung verschiedener Algorithmen und Sicherheitsmaßnahmen verwendet werden kann. Dies ist besonders wichtig da die komplexen Algorithmen anfällig für simple und differentielle Leisungsanalyseattacken sind. *Neptun* ist ein programmierbarer Prozessor mit einem maßgeschneiderten Instruction-Set, das speziell für Primfelder optimiert wurde. Das Design beinhaltet auch einen Bootloader, eine EIA-232 Schnittstelle, drei Timer und eine parallele I/O-Einheit. Es wurde implementiert und hergestellt als ein applikationsspezifischer integrierter Schaltkreis (ASIC) unter der Verwendung der UMC-L180 Technologie.

Die Elliptischen Kurven Signatur-Erzeugung und Verifikations-Algorithmen wurden unter Verwendung eines maßgeschneiderten Prozessor-Simulators implementiert. Dieser Simulator ist auch fähig Assembler-Kode zu analysieren und ausführbare Dateien zu erzeugen.

Das zweite Design basiert auf *Neptun*. Der größte Unterschied ist, dass es von jeglicher unnötigen Logik befreit wurde, ohne einen Kompromiss bei der Sicherheit der Algorithmen einzugehen. Dieses Design wird dazu benutzt, dass die aufgezeigte Architektur praktikabel für RFID-Tags ist. Mit einem geringen Platzverbrauch von nur 14230 Gatteräquivalenten und einer Laufzeit von 1,65 Millionen Zyklen ist das Design vergleichbar mit dedizierten Koprozessoren und die kleinste Implementierung zur Erzeugung von ECDSA-Signaturen (unter der Verwendung der NIST P-192 Parameter), die bis jetzt publiziert wurde.

Mit diesen Resultaten wurde ein großer Schritt zu sicheren RFID-Tags gemacht.

Stichwörter: Prozessor, Radio-Frequenz-Identifikation, Elliptische-Kurven Kryptografie, Digitale Signatur, ASIC

Contents

Chapter 1

Introduction

The importance of devices that use a combination of radio frequency identification (RFID) and security technology has grown over the last few years. Several products with security features already exist nowadays. Several of them have already been broken (MIFARE Classic, Keeloq). Their security was mainly based on using custom and secret algorithms. Secure systems should be based on openly available algorithms. Only the key(s) should be kept secret.

The concept behind RFID is that a very small microchip is attached to an antenna. The chip is powered via an electromagnetic field of the reader. The field is also used to communicate with the RFID chip. Because the energy is transferred over a distance, the power consumption for an RFID tag is critical.

Unlike RFID tags, smart cards are systems that use strong cryptographic algorithms. To process those complex algorithms, usually powerful processors are used as smart cards. The power for such cards is usually provided by an active power supply via a galvanic connection. So it has not been very important to keep the power consumption of such cards low.

During the last few years, wireless interfaces have been added to the smart cards. Because of the relatively huge power consumption of smart cards the possible range for that systems is in the area of centimeters.

The idea is to merge the RFID and smart card technologies. The price and range of the new product should be similar to RFID tags, but the security of this product should be as strong as the securtiy of a smart card. To achieve those goals, several restrictions need to be made. One is to highly optimize the intended target tag for a certain kind of security method.

Investigations showed that elliptic curve cryptography is well suited for such applications. It is very resource conservative and based on difficult mathematical problems. The elliptic curve digital signature algorithm is a well established concept for authentication. To be compatible to commonly used standards, standardized NIST curves are used in this design. The set of used domain parameters is called NIST P-192. The advantage of the used prime field is that very fast reductions are possible. As a result, a reasonably low total runtime is achievable.

This thesis presents two different designs. One design, called *Neptun* makes use of a custom build processor and can be used to evaluate different algorithms and security measures. The problem with full-custom ASIC designs is that errors are very expensive. Especially simple and differential power analysis attacks can be used to attack security devices. So the used algorithms need to be carefully evaluated and possible security flaws

need to be fixed. The second design is based on *Neptun*. The main difference is that it is optimized for one cryptographic algorithm. This design is used to show that the presented concept can be used as an area optimized design. The used silicon area usually is proportional to the prices of a chip. Because RFID tags are usually intended to be bulk commodity, the low area requirement is especially important.

The possible applications for such a presented design are immense. It can be used in every environment, where authentication and access control is important. An example are the huge losses in the range of billions that are caused by counterfeit products. Security-enabled RFID tags can be used to identify counterfeit products. This method has the potential to reduce product piracy.

1.1 Outline

This thesis will start with an introduction to smart cards. Chapter 2 gives a short history of chip cards. Especially the applications are important. The presented chip can be used by all of them.

Subsequently this thesis can be split into three parts. The cryptographic algorithms, the hardware for the specialized processor and the discussion of results. Chapter 3 handles an introduction to cryptography. Very general topics like definitions and goals of cryptography as well as symmetric- and public-key cryptography are covered. Naturally those two cryptographic schemes are compared.

Chapter 4 handles digital signatures. The older digital signature algorithm is shortly discussed for comparison with the elliptic curve digital signature algorithm. Both, the signature generation and the signature verification process are discussed in detail. Also the correctness of ECDSA is shown.

To actually understand ECDSA, elliptic curve arithmetic is introduced in Chapter 5. This chapter describes the basic concepts behind elliptic curves as well as point arithmetic. The focus is especially on the performance-relevant point multiplication schemes.

Elliptic curves make use of finite fields. Chapter 6 introduces techniques that are needed to handle prime-field arithmetic on platforms with small data paths. The focus is especially on the multiplication and reduction methods.

Chapter 7 is used as transition from the used algorithms to the ASIC hardware. It describes how a design can be developed, optimized and simulated. Different instruction-set concepts are discussed. Those instruction sets are described in connection with the design flow used for the *Neptun* processor.

By having the idea of a certain design in mind, different designs are discussed in Chapter 8. Size as well as runtime approximations are made.

With the decided design in the back of the mind, a processor is designed in Chapter 9. After the discussion of a general architecture concept and a central processing unit, an arithmetic-logic unit and all its components are described. With some modifications, the processor is made reusable. Also an efficient method for testing is described.

All the previous information has been used to implement a processor, called *Neptun*. Several attributes of this processor are discussed in Chapter 10. After a detailed runtime analysis, the area requirement is discussed and compared to different publicly known designs.

Finally, conclusions are given in Chapter 11. The appendix is used to document various features of the implemented processor in detail.

Chapter 2

Introduction to Chip Cards

Comprehensive material about chip cards can be find in the book [33]. Especially for this thesis relevant chapters are 1, 2, 12, 13 and 14 and are summarized at this place.

The popularity of plastic cards began in the beginning of the 50s in the USA. The cheap plastic PVC made the production of robust and durable cards possible. It was far superior to the cards made from paper or cardboard.

The first full plastic card for payments was offered by Diners Club in the year 1950. It was designated for an exclusive group of people. At first the acceptance of the card was limited to certain hotels and restaurants.

After Visa and MasterCard joined the scene the spread of the plastic money grew increasingly.

Nowadays, the plastic cards makes it possible to pay worldwide without cash money. The owner can pay every time and everywhere without the risk of stolen cash.

In the beginning, the used security measures were very simple. They were based on visual information like an embossed number, an owner and a signature field. This level of tamper resistance was not enough, once the popularity reached a certain point. A first improvement was to add a magnetic stripe on the back of the cards. This added machine-readable data to the cards. This feature reduced the required use of paper but had a flaw. The information on the magnetic stripe could be read, written and deleted by anybody who owned the right equipment. So the magnetic stripe cannot be used for the storage of secret information.

The development of integrated circuits in the 70s and 80s made it possible to integrate a lot of logic on a tiny chips. The big break through was made in 1984. The french PTT made a field trial for phone cards. The tested chip cards were very tamper proof and reliable. The tested chip cards also showed a big flexibility in its applications.

These cards were using simple circuits with security. So they were cheap for production. Also the more complex microprocessor chips were first tested in telecommunications. Because of the positive experience during the use of chip cards in the analog mobile telephone system, chip cards are used for authentication in the GSM network. This was setup in many European countries in 1991. Currently there are more then 600 million user in more than 170 countries.

The development of chip cards in the banking industry was slower. Mainly because the hardware and software for the required mathematical algorithms is very complex. With the introduction of modern cryptography, chip cards became more and more popular. The first country with a nationwide introduction of chip cards for payment was Austria. These chip cards were capable of POS (point of sales) functions, an electronic purse and possible

extensions.

Chip cards present a very important function: electronic signatures. The European parliament released a directive in the year 1999 for the legal foundation of digital signatures.

Apart from the galvanic coupled cards also wireless cards were introduced. Their handling is a lot simpler and more user-friendly. Also the possible fields of applications extended with their introduction.

2.1 Applications

Debit card. This kind of cards makes it possible to pay at the shop directly. Using the debit card, a transaction is started from the owners bank account and the money is transferred to the dealer or service provider.

Electronic purse. Using a terminal of some kind, money can be stored directly on the card. During payment, the balance of the card is decreased and the balance of the second party (usually the dealer) is increased.

This system is fairly complex. The properties are:

Automatic process ability. In order to operate such systems profitably the transactions must be processable by machines.

Transferability. The money should not be bound to a certain medium. It must be transferable via various networks of PCs.

Divisibility. A certain amount of money must be splitable into several parts. This makes it possible to pay exactly the amount that should be paid.

Non central. A transaction of money should be possible when both parties are offline and have no connection to a banking system. Debit or credit cards are usually managed by a central administration.

Monitoring. This requirement is important. There are two cases to consider. In one case there is somebody attacking the system. In the other case there is an error in the system and it is malfuctioning. Both of those procedures must be monitorable or signal some kind of administrator.

Security. The most basic requirement is the protection against forgery. The system would break down immediately if money can be copied or forged.

Anonymity. This means that it should not be possible to map the payments between people. There are two positions to be considered: The operator wants a non-anonymous system, so he can monitor it optimally. The user wants complete anonymity and no traceability.

GSM. The chip card for GSM (Global System for Mobile Communications) telephones is named SIM (subscriber identity module). It represents the identity of a party. Its primary function is to secure the authenticity of a mobile station in a network. Apart from that, a SIM also supports additional functions. It provides the secure execution of programs, the storage of data (phone numbers, short messages and personal adjustments for the mobile phone) and other mobile services.

The main task of the SIM is the authentication in the GSM network. This is a one-way authentication. The network checks the authenticity of the SIM but the

SIM does not check the authenticity of the network. The identification of the SIM works with a, for GSM unique, 8-byte number called the IMSI (international mobile subscriber identity). With this number, the subscriber can be identified worldwide.

UMTS. The Universal Mobile Telecommunication System (UMTS) is declared as the third generation of mobile telecommunication. From the view of the chip card, the biggest difference is the new security module: the USIM (universal subscriber identity module). It uses the ISO/IEC 7816 standards and is backwards compatible with the old norm (the SIM).

Health insurance card. This card provides two basic functions: On the one hand it is an authentication of the patient to the doctor. It can be seen as an electronic health insurance certificate. On the other hand it is a data storage for the computer of the doctor. It can contain billing information and/or a patient history.

Digital signature. A legally binding signature requires two prerequisites: A law that binds all participants legally and a powerful chip card. The card must be capable of numeric operations and the storage of a private key. For that a powerful processor or co-processor is required.

2.1.1 Wireless Applications

There are two kinds of cards that need to be distinct. Although the requirements and the applications for those cards are merging.

The first kind are wireless smart cards. Traditionally smart cards have very powerful embedded processors. These processors are mostly used for various different cryptographic algorithms. Some applications are already mentioned in the previous section. Because these embedded systems are so powerful they usually require a lot of power (in the range of milliwatts). The wireless interface used for those cards is the ISO-14443. Typically the operation range of such a card is some centimeters.

Another type of cards are RFID (Radio Frequency Identification) cards. They are designed to have a wider range of up to several meters. To achieve such distances, the power consumption of such cards must be very low (in terms of microwatts). So the implementations of RFID cards were very limited. Usually they were used to store several bytes of data. Later, some manufacturers implemented symmetric algorithms that can be used to encrypt the connection between a tag and a reader. Many of those algorithms were broken during the last few years.

Other applications are:

Skiing. The type of cryptography used for skiing traditionally is limited. It is more important for this application that the range of the used RFID cards is high. Security is achieved with different methods. Every card contains a unique serial number. All the readers within a skiing area are connected to a central server. At first the serial number is checked for its validity. Because the last position within the skiing area is logged a causality check can be performed. A user cannot be at two distinct places in the skiing area within several seconds. A third check is that pictures are taken of the skiers. The readers are equipped with web cams. So the operator can compare the stored image with the web cam.

Textile industry. Nowadays barcodes are placed on every item sold in a store. The barcodes are unique for a type of product. So the barcode can be used at the

cashier for product identification. This information can be used for billing. A goal of the industry is to replace the barcodes by RFID tags. A product can be uniquely identified per item and per class. This should result in the following changes:

- RFID reader terminals can be placed at the entries of a stock. In this way the number of items is always up to date. This procedure simplifies the required stock management. Pallets of items can be scanned at once.
- RFID readers can be placed at exits of shops. Because every item can be identified uniquely, every item can be checked if it has been paid for. This provides a cheap way to prevent shop lifting. This ability causes problems. The tags can be destroyed or covered in aluminum foil. Then the security system does not work any more.
- The payment at the cashier can be accelerated. The cashier does not have to scan each item manually. A whole basket of products can be read at once. It does not even have to be emptied.
- In the textile industry, a lot of clothes and accessories are forged. A tag, sewed within the clothing can be used to check that it was produced by the declared trade mark. The tag can simply contain a digital signature that was issued by the manufacturer.

Access control. This field of application is huge. It does not only cover the access control to certain areas of a nuclear reactor, secret agencies, office buildings... It also covers active tags used as car keys or remote controlled garage door opener. The currently existing tags are used in a lot of different systems. Usually the systems use some kind of authentication systems. The elliptic curve digital signature algorithm is well suited for the most of those applications (see Chapter 4).

Chapter 3

An Introduction to Cryptography

Cryptography already exists for thousands of years. One very popular and old cipher is the Cesar code. It is a very simple cipher. Every letter in a sentence is rotated by a number of positions. Modern cryptography is based on hard mathematical problems.

This chapter gives a short introduction to cryptography. Just enough to understand the basics behind elliptic curve cryptography, described in the next chapter. For more information about cryptography in general have a look in [30].

3.1 Definition and Goals

Based on [30], cryptography is defined as the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication and data origin authentication.

Let us look on this attributes in detail:

Confidentiality stands for secrecy and privacy. This means that nobody, except the person authorized, should be able to access the unencrypted information.

Data integrity is about the unauthorized change or alteration of data. It must be possible to detect manipulation of data by unauthorized parties. Manipulation can be the insertion, deletion or substitution of data.

Authentication and identification go hand in hand. Two different forms of authentication can be distinguished: Entity authentication is about proving ones authenticity using certain information about oneself to a second party. Origin authentification proves that the provided data/message/information is from a certain sender.

Non-repudiation prevents an entity from denying previous commitments or actions. In the case of a signed contract, this attribute makes sure that a party cannot back out, by denying that the contract was signed.

The goal of cryptography is it to handle those attributes adequate with the available algorithm(s). Every available algorithm should be evaluated. There are various criteria for such an evaluation:

Level of security is usually defined as the number of operations needed to attack the intended objective (e.g. encryption). Because the best method for 'cracking' an algorithm is often discovered over time, the level of security degrades over time.

Most often, the available computational resources for cracking an algorithm are important as well.

Functionality. [30] describes it as the primitives needed to be combined to meet various information security objectives. The most effective primitives for a given objective are determined by the basic properties of the primitives.

Methods of operation. A primitive can have different functionality. This mostly depends on its mode of operation or usage. It also depends on the applied inputs.

Performance refers to the efficiency of the primitive. For an encryption algorithm, this is the number of cycles or the time it takes to finish the encryption.

Ease of implementation refers to the difficulty of practically implementing a primitive. This can vary, depending on the available software or hardware environment.

The importance of those criteria is dependent on the application and the used resources. As an example, let us look at the implemented ECDSA algorithm within this master thesis (using NIST-P192):

Level of security is comparable to a 96-bit Triple-DES (Data Encryption Standard) or a 2048-bit RSA (Rivest-Shamir-Adleman) algorithm.

Functionality. An input message can be signed using a private key. A public key can be used to verify this signature.

Methods of operation. The algorithm can be used for signing a document (message) and for proofing ones authenticity.

Performance. The performance is evaluated with two parameters: The area needed for implementing the algorithm in hardware and the number of cycles it takes to complete one ECDSA signature.

Ease of implementation. In this case it is the time, needed for building the Neptun processor. Approximately five months.

3.2 Symmetric-Key Cryptography

In this cryptography scheme, both parties share the same secret key k, or one that can easily be derived from the other one. The plain message is m. The encoded message is c. An encryption function E_k is used to calculate $c = E_k(m)$. The decryption function D_k is used to derive the plain message from c: $m = D_k(c)$.

Previously, the Cesar cipher has been mentioned. In this case, the key is the number of positions that are shifted in the alphabet.

The biggest concern of a symmetric encryption scheme is to find a key that can be used by both parties. This problem is named as the key distribution problem. A secure channel is needed to exchange a secret key. Only then, symmetric-key encryption can be used (see Figure 3.1)

Symmetric-key encryption schemes can be split into two classes:

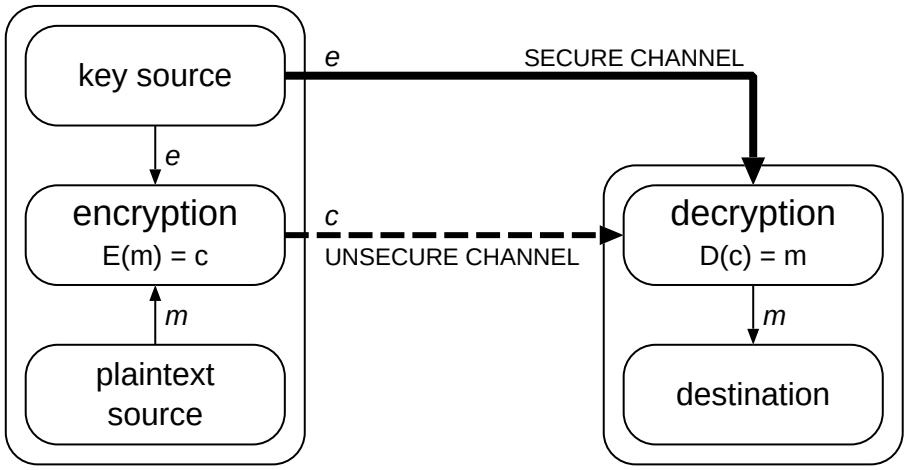

Figure 3.1: Encryption using symmetric cryptography.

Block ciphers are encryption schemes that break a plain text message into blocks of fixed length. Every block is encrypted one by one. A very popular representative of block ciphers is the AES (Advanced Encryption Standard) algorithm. This is an algorithm invented by Joan Daemen and Vincent Rijmen (see [32]).

Stream ciphers can be seen as block ciphers with a block length of one. This method does not need to collect data until a full block length is reached. Every character of the message can be encoded immediately. Famous examples of newly developed stream ciphers are Grain [14] and Trivium [6].

3.2.1 Key-Distribution System

There are three methods for managing keys for symmetric algorithms:

Store them all. Every entity stores a secret key used for communication with any other entity. To be prepared to communicate with $(n-1)$ other entities, $(n-1)$ keys are required. For an RFID tag this is intractable. In total $n^2 - n$ keys need to be stored in this system.

Trusted third party. A third party is introduced. This party stores n keys for n participants. If two participants want to communicate, they first contact the third party using an encrypted connection. The third party generates a key that can be used by both parties. This key can be used for a secure connection between the two participants.

It is very interesting to review this approach for RFID tags. The third party is a central server. An RFID tag can encounter different readers and terminals during its lifetime. The first problem is a political problem because the readers are distributed throughout different companies. Secondly, all the readers must be online all the time, so that the tag can initialize a connection with the server. Thirdly, in the case of a server crash, all the keys can be lost, or the whole system does not work any more.

Public-key cryptography. Very often, the more calculation intensive, public-key cryptography is used for establishing a secure connection. A key exchange protocol is

used so that two parties can agree on a common key. This common key is then used for symmetric cryptography.

3.3 Public-Key Cryptography

Public-key cryptography is also known as asymmetric cryptography. It is asymmetric because the two parties do not share their private key any more. Instead there is a public key, which can be used by any entity to encode a message $c = E_{publickey}(m)$. Encoding a message is a one-way trap function. Using the public key, the message m cannot be recovered. To decode the message, the private key is needed: $m = D_{privatekey}(c)$. This means that the private key is only known to the recipient of the encrypted message c. This scheme is visualized in Figure 3.2.

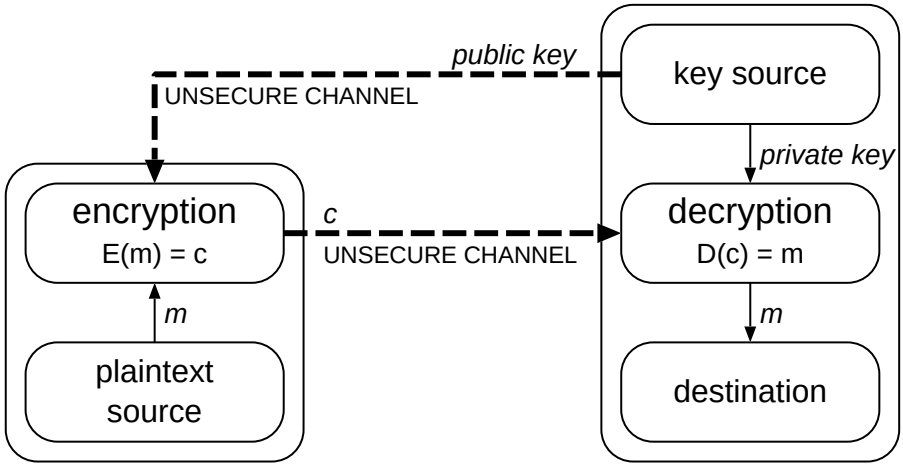

Figure 3.2: Encryption using public-key cryptography.

Note the difference between Figures 3.1 and 3.2. Using public-key cryptography, the public key can be transferred using an insecure channel. The channel for transporting the public key and the channel for the transportation of the message can also be the same.

A physical analog is a post box. Anybody can put something into the box (encrypt a message). Only the post officer with the key for the post box can open it (decrypt a message). The only flaw in this metaphor is that the post officer usually is not the recipient of the mail.

Usually the term private key is used in connection with public-key cryptography and the term secret key is used in the case of symmetric cryptography. That is because a secret needs to be shared by two parties, but a key, which is only known to oneself, is really private.

3.4 Comparison of Cryptography Schemes

At this point it is interesting to compare the two previously introduced schemes.

Advantages of symmetric-key cryptography:

1. High performance implementations are possible.
2. Short key lengths.

3. Can be used as pseudo-random number generators.

4. Symmetric-key ciphers can be combined. This results in strong product ciphers.

Disadvantages of symmetric-key cryptography:

1. The key must remain secret within all participating parties.

2. A lot of key pairs need to be managed in large networks. Trusted third parties need to be used.

3. Cryptographic practice dictates that the key is changed frequently. If possible it should be even changed within communication sessions.

Advantages of public-key cryptography:

1. The private key must be kept secret by one entity only. (The authenticity of public keys must be guaranteed).

2. Only a functionally trusted third party is required. Functionally trusted means that the third party does not have to store any private keys. In many cases an online connection to a third party is not required at all times.

3. The key pair can remain unchanged for long periods of time (several sessions or years). This also depends on the mode of usage.

4. There are many efficient digital signature mechanisms. They mostly only require small public keys.

5. The total number of required key pairs in a large network is much smaller than in a symmetric-key scenario.

Disadvantages of public-key encryption:

1. In comparison to symmetric-key schemes, the public-key schemes are computationally much more intensive.

2. The required key size is larger compared to symmetric-key encryption methods.

Optimally one should use the advantages of both worlds. Use public-key cryptography for establishing a secure connection but use symmetric-key cryptography for transferring the data.

3.5 Digital Signatures

A fundamental component in cryptography are digital signatures. They are essential for authentication, authorization and non-repudiation. The purpose is to bind an entity to a piece of information. During the process of signing, some secret information held by the signing entity is used to generate a signature for some information.

A perfect application for public-key cryptography are digital signatures. Two processes need to be distinguished:

Signing procedure. The signer creates a signature s for a message m by computing $s = S_{privatekey}(m)$. This results in the pair (m, s).

Verification procedure. Usually the verifier is not the same entity as the signer. As a result, the verification (public) key has to be obtained first. Using this information $u = V_{publickey}(m, s)$ can be calculated. The signature is accepted if u is true. Else the signature is rejected.

3.6 Hash Functions

The book [30] states that a hash function is a computationally efficient function mapping binary strings of arbitrary length to binary strings of some fixed length, called hash-values.

A hash value can be seen as a compact representation of an input string. During this compression, the number of bits (from input to hash) gets reduced. This means that it is theoretically possible to find two input strings that generate the same hash value. This is called a collision. Hash algorithms are designed in a way that it is hardly possible to find a collision. It should be also computationally infeasible to find an input x for a predefined hash-value y so that $h(x) = y$. This is called a pre-image attack.

As it will be seen later, hash algorithms are very important for digital signatures and data integrity checks. During a signature generation, not the whole message is signed but the hash value of the message. So finding a message with the same hash-value as the originally signed message should be computationally infeasible.

To check the validity of a signature the hash function must be publicly known.

Chapter 4

Digital Signature Algorithms

This chapter gives a very short introduction to the Digital Signature Algorithm (DSA). The information from this introduction is used for comparison with the Elliptic Curve Digital Signature Algorithm (ECDSA). This algorithm is described in detail. The information used for this chapter is taken from [12].

4.1 Digital Signature Algorithm

Back in 1991, the U.S. National Institute of Standards and Technology (NIST) proposed the Digital Signature Algorithm (DSA). It was declared by the U.S. Government as Federal Information Processing Standard (FIPS 186) and called Digital Signature Standard (DSS).

DSA is based on Rivest, Shamir and Adleman (RSA). The RSA signature scheme use the fact that $m^{ed} \equiv m \pmod{n}$ for all integers m. A e, d and n must fulfill certain properties. RSA can also be used for generating and verifying signatures, but they have a serious flaw. A signature for a defined message always results in the same signature.

This flaw is taken care of in the DSA Algorithms 1 and 2. The signature algorithm introduces a random ephemeral key. This is a private one-time key. This key is used for the exponentiation that was introduced by Rivest, Shamir and Adleman. This exponentiation is followed by some computations in lines 3 to 5, which results in the signature (r, s).

Algorithm 1 DSA signature generation

Require: Domain parameters (p, q, g), private key x, message m.
Ensure: Signature (r, s).
1: Select $k \in_R [1, q-1]$.
2: Compute $T = g^k \bmod p$.
3: Compute $r = T \bmod q$. If $r = 0$ then go to step 1.
4: Compute $h = H(m)$.
5: Compute $s = k^{-1}(h + xr) \bmod q$. If $s = 0$ then go to step 1.
6: Return(r, s).

This signature is used for the verification in Algorithm 2. This algorithm uses the signature, the message, the public key and the domain parameters. It neither uses the private keys k nor x that are only needed for the signature generation. As a result, the signature can be verified by doing the (previously introduced) exponentiation in line 5.

Algorithm 2 DSA signature verification

Require: Domain parameters (p, q, g), public key y, message m, signature (r, s).
Ensure: Acceptance or rejection of the signature.

1: Verify that r and s are integers in the interval $[1, q-1]$. If any verification fails then return("Reject the signature").
2: Compute $h = H(m)$.
3: Compute $w = s^{-1} \bmod q$.
4: Compute $u_1 = hw \bmod q$ and $u_2 = rw \bmod q$.
5: Compute $T = g^{u_1} y^{u_2} \bmod p$.
6: Compute $r' = T \bmod q$.
7: If $r = r'$ then return("Accept the signature");
Else return("Reject the signature").

4.2 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is built on elliptic curve arithmetic, which is based on finite field arithmetic. Those dependencies are shown in Figure 4.1. In the case of this thesis, prime fields are used. So the 'field arithmetic' and the 'big number and modular arithmetic' blocks can be merged.

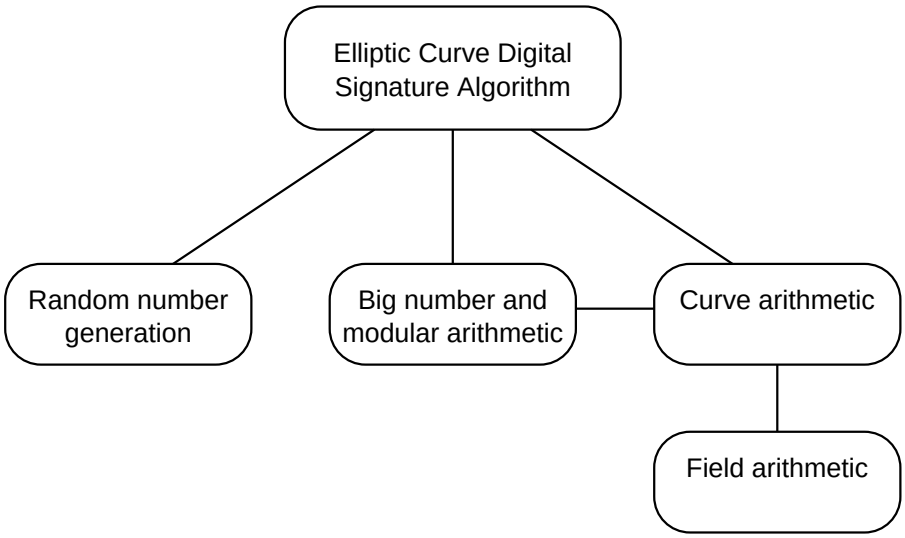

Figure 4.1: ECDSA uses curve arithmetic, which uses finite field arithmetic. Further a random number generator and modular arithmetic is required.

During the life cycle of a signature, several algorithms need to be performed.

1. Generate the domain parameters D. For that a domain parameter generation algorithm should be used (see Chapter 4.2 in [12]). In the case of this thesis, the parameters are already generated and part of the NIST standard FIPS 186-2 [31].

2. Generate a key pair (Q, d) using a key-generation algorithm. The algorithm used for Elliptic Curve Cryptography is shown in Algorithm 3. At first, it selects a random integer as private key. Then a point multiplication is performed with P. P is a previously selected domain parameter. The resulting point Q is used as public key.

3. A signature generation algorithm produces a signature, using the domain parameters D, an input message m and the private key d. It is shown in Algorithm 4.

4. At some point the signature needs to be verified. For that, Algorithm 5 is used. It uses the domain parameters D, the public key Q, the original message m and a signature (r, s). Very important is the fact that the verifier does not need the private key d for its verification. This signature is either accepted or rejected.

Algorithm 3 Key-pair generation

Require: Domain parameters
Ensure: Public key Q, private key d.
1: Select $d \in_R [1, n-1]$.
2: Compute $Q = dP$.
3: Return(Q, d).

Before actually looking at the signature algorithm in detail it is necessary to understand the used parameters.

p	The order of the prime field $\mathbb{F}_p$. This is the underlying field used for all point additions, doublings and multiplications. It is a very important parameter because it greatly influences the performance of the point operations. Because the characteristic of p is neither 2, nor 3, a reduction of the Weierstrass equation can be performed. p is a domain parameter defined by NIST.
a, b	The coefficients of the elliptic curve $y^2 = x^3 + ax + b$ satisfying $rb^2 \equiv a^3(modp)$. In the case of the NIST domain parameters those parameters are predefined. $a = -3$. $0 < b < p$. Because $a = -3$ a field multiplication can be replaced by field additions.
$P = (x, y)$	The base point P is the starting point for each point multiplication, performed during the signature generation algorithm. It is represented with the two coordinates x, y. They are part of the NIST domain parameters.
n	The (prime) order of the base point P. Suppose $Q = \infty$. Now P is added to Q. $Q = Q + P$. This can be done n times, before the identity ∞ is reached again. The parameter n is derived with the selection of P. So it is a NIST domain parameter.
d, Q	d is the private key. Q is the public key. They are generated using Algorithm 3. The public key Q is a point with two coordinates.
r, s	Those two parameters together represent a signature. Separately, they are useless.

There are additional parameters used for other underlying fields. This thesis uses prime fields. So only the for prime field important factors are considered.

4.2.1 Signature Generation

A signature is generated using Algorithm 4. It is using the domain parameters, a private key and a message as inputs. k is selected randomly and used for a point multiplication. k is an ephemeral key. This is a private key that changes for each signature. From the resulting point, only the x-coordinate is needed. The conversion of x_1 to $\bar{x_1}$ is not needed if a prime field $\mathbb{F}_p$ is used. In this case $\bar{x_1} = x_1$ is valid. Because $p > n$, a reduction

needs to be performed. In the following two lines, the second part s of the signature is calculated.

The major calculation is performed within line two. Usually the point multiplication takes more time to calculate than all the other operations together.

Algorithm 4 ECDSA signature generation

Require: Domain parameters, private key d, message m.
Ensure: Signature (r, s).
1: Select $k \in_R [1, n-1]$.
2: Compute $kP = (x_1, y_1)$ and convert x_1 to an integer $\bar{x_1}$.
3: Compute $r = \bar{x_1} \bmod n$. If $r = 0$ then go to step 1.
4: Compute $e = H(m)$.
5: Compute $s = k^{-1}(e + dr) \bmod n$. If $s = 0$ then go to step 1.
6: Return(r, s).

It is interesting to compare the two presented signature Algorithms 1 and 4. A big difference are the used domain parameters and the different method for producing r. The second part of the signature generation algorithms, for generating s, is very similar.

4.2.2 Signature Verification

To verify a given signature, Algorithm 5 is used. Apart from the signature itself, it needs the used domain parameters, the public key and the original message. After the calculation of u_1 and u_2, two point multiplications need to be performed. Because of that, this algorithm uses up to twice the time compared to the signature generation algorithm.

Algorithm 5 ECDSA signature verification

Require: Domain parameters, public key Q, message m, signature (r, s).
Ensure: Acceptance or rejection of the signature.
1: Verify that r and s are integers in the interval $[1, n-1]$. If any verification fails then return("Reject the signature").
2: Compute $e = H(m)$.
3: Compute $w = s^{-1} \bmod n$.
4: Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5: Compute $X = u_1P + u_2Q$.
6: If $X = \infty$ then return("Reject the signature");
7: Convert the x-coordinate x_1 of X to an integer $\bar{x_1}$; compute $v = \bar{x_1} \bmod n$.
8: If $v = r$ then return("Accept the signature");
Else return("Reject the signature").

In comparison with Algorithm 2 the first few lines are similar. Except for a different modulo, u_1 and u_2 are calculated identical.

It is very important to understand that the signature verification works without the private key. Let us assume for now that a signature (r, s) on a message m has been generated by a legitimate signer.

$$
\begin{aligned}
s &\equiv k^{-1}(e + dr) \pmod{n} \\
k &\equiv s^{-1}(e + dr) \\
&\equiv s^{-1}e + s^{-1}rd \\
&\equiv we + wrd \\
&\equiv u_1 + u_2 d \pmod{n}
\end{aligned}
$$

Thus $X = u_1P + u_2Q = (u_1 + u_2d)P = kP$, and so $v = r$ as required.

Chapter 5

Elliptic Curve Cryptography

Elliptic curve arithmetic got popular in the 80's of the last century. That was the time when Neal Koblitz [24] was one of the first to use elliptic curves for cryptography. Since back then the popularity for elliptic curves grew more and more.

In the Austrian 'Bürgerkarte' and many other security-relevant environments, elliptic curves are used nowadays. The question is why they are doing so. The work [11] by Nils Gura et al. can provide an answer. They compared different RSA and elliptic curve algorithms on embedded processors. [12] states that a 224-bit elliptic curve and 2048-bit RSA algorithm have the same level of security. Those two algorithms have been implemented by Nils Gura et al. on an 8-bit embedded processor (ATmega128). In detail, the secp224r1 curve (same as NIST P-224) is used for the elliptic curve arithmetic and the RSA-2048 algorithm is using the public exponent $e = 2^{16} + 1$. The resulting runtime of those algorithms has been within a 15%. The code size of both algorithms is also very similar (RSA is about 30% better). The big difference is in the required size of the data memory. The data memory of the RSA-2048 algorithm has to be more than three times larger than the memory of the 224-bit elliptic curve. This example clearly shows that the elliptic curve algorithms have a clear advantage for embedded devices, although the required algorithms are more complex.

Because the goal of this thesis is to lead the ECDSA algorithm one step closer to RFID devices, some basics are given in this section about elliptic curves. After a short introduction to elliptic curves, important double, add and multiplication algorithms are shown in this chapter.

5.1 Mathematical Basics

The equation behind each elliptic curve is a Weierstrass equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{5.1}$$

In order to ensure that the curve is 'smooth', there is also the discriminant $\Delta \neq 0$ to consider. Δ is the discriminant of E. It is defined with the following equation:

$$\Delta = -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2 d_4 d_6 \tag{5.2}$$

$$d_2 = a_1^2 + 4a_2$$

$$d_4 = 2a_4 + a_1 a_3$$

$$d_6 = a_3^2 + 4a_6$$

$$d_8 = a_1^2 a_6 + 4a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2 \tag{5.3}$$

The smoothness of the elliptic curve is necessary, so that there are no points on the curve with two or more distinct tangent lines.

[12] states that E is defined over K because the coefficients a_1, a_2, a_3, a_4, a_6 of its defining equation are elements of K. Sometimes E/K is written to emphasize that E is defined over K, and K is called the underlying field. Popular choices for those underlying fields are prime fields $\mathbb{F}_p$ and binary extension fields $\mathbb{F}_{2^m}$. For both types of fields, parameters have been recommended by NIST in the FIPS 186-2 standard [31].

A point on the curve has the two coordinates x and y. This point is on the curve only if it satisfies the Weierstrass equation. This can be also written mathematically: L is an extension field of K. The set of L-rational points on E is defined as:

$$E(L) = \{(x, y) \in L \times L : y^2 + a_1 xy + a_3 y - x^3 - a_2 x^2 - a_4 x - a_6 = 0\} \cup \{\infty\} \tag{5.4}$$

∞ is the point at infinity. This point can be seen as identity. It is important for the calculation with points.

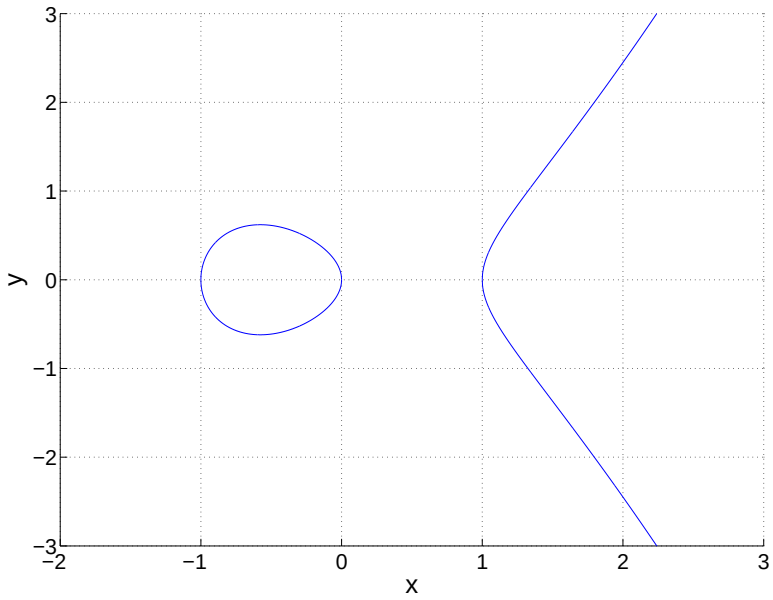

Figure 5.1: Elliptic curve represented over $\mathbb{R}$: $E : y^2 = x^3 - x$.

Figure 5.1 represents an elliptic curve. In this case E is defined over $\mathbb{R}$, or $E/\mathbb{R}$. The points that fulfill $E(\mathbb{R})\backslash\{\infty\}$ are drawn.

A very helpful construct is a transformation. A transformation is performed as an admissible change of variables. Such a transformation can be used to simplify the Weierstrass equation. It depends on some attributes of the underlying field K.

Dependent on the characteristic of K, the equation can be transformed differently. The possible simplifications can be done in the cases $char(K) \neq 2, 3$, $char(K) = 2$ or $char(K) = 3$. Because this thesis uses prime fields $\mathbb{F}_p$ we can concentrate on $char(K) \neq 2, 3$. In this case, the following change of variables can be performed:

$$(x, y) \rightarrow \left(\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1 x}{216} - \frac{a_1^3 + 4a_1 a_2 - 12a_3}{24} \right) \tag{5.5}$$

This transforms E to the curve:

$$y^2 = x^3 + ax + b \tag{5.6}$$

a, b must be in K. The new discriminant can be written as $\Delta = -16(4a^3 + 27b^2)$.

It is helpful to recapitulate the information with an example. The characteristic of $\mathbb{F}_7$ is neither 2, nor 3. The used equation is $E : y^2 = x^3 - x$. In this case $a = 6 \equiv -1 \pmod 7$ and $b = 0$. This is the same curve as in Figure 5.1. The resulting discriminant $\Delta = -16(4a^3 + 27b^2) = -13824 \equiv 64 \equiv 1 \pmod 7 \neq 0$. So the curve is valid and can be used. The resulting points on E are:

$$E(\mathbb{F}_7) = \{\infty, (0,0), (1,0), (4,2), (4,5), (5,1), (5,6), (6,0)\}$$

These points can be represented graphically. In Figure 5.2 the points on $E : y^2 = x^3 - x$ are shown using $\mathbb{F}_7$. In Figure 5.3 the points on $E : y^2 = x^3 - x$ are shown using $\mathbb{F}_{541}$.

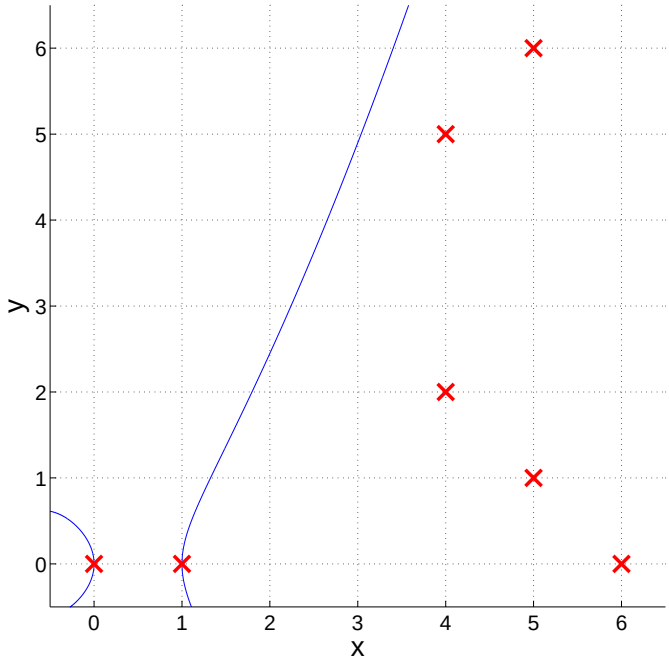

Figure 5.2: Elliptic curve represented over $\mathbb{F}_7$: $E : y^2 = x^3 - x$.

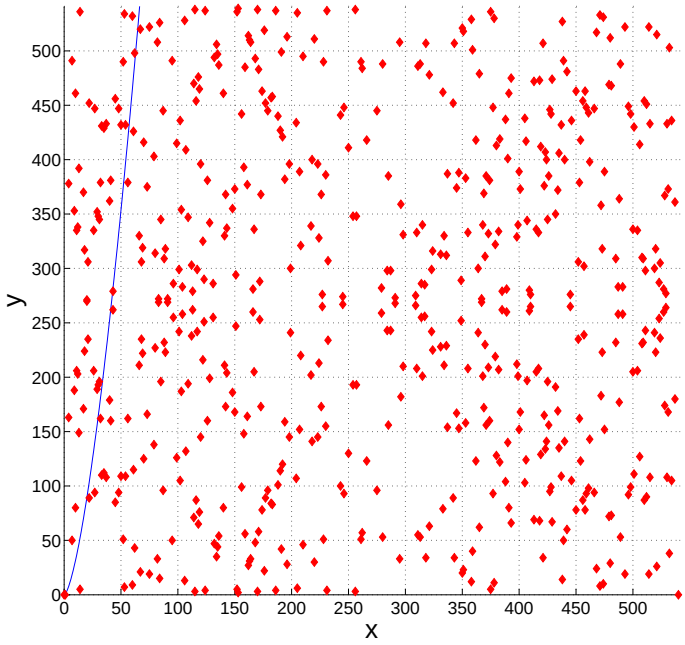

Figure 5.3: Elliptic curve represented over $\mathbb{F}_{541}$: $E : y^2 = x^3 - x$.

5.2 Adding and Doubling

Adding two points can be done geometrically and with the use of formulas. At first, it is explained geometrically.

There are two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ that should be added. Those two points are connected with a line. This straight line is extended until it intersects the elliptic curve at a third point. The point of intersection is mirrored along the x-axis. The resulting point $R = (x_3, y_3)$ is the solution of the point addition. The addition is shown in Figure 5.4.

If the two points P and Q are equal, a doubling algorithm has to be applied instead. To double P, the tangent line at the point P needs to be found. This straight line is extended, until it intersects with the elliptic curve. The intersection is mirrored along the x-axis (similar to the addition). The resulting point is visualized in Figure 5.5.

To use points for calculation, several formulas need to be considered. This definition is taken from [12]. It is required that $E\backslash K$: $y^2 = x^3 + ax + b$, $char(K) \neq 2, 3$:

Identity. $P + \infty = \infty + P = P$ for all $P \in E(K)$.

Negatives. If $P = (x, y) \in E(K)$, then $(x, y) + (x - y) = \infty$. The point $(x, -y)$ is denoted by $-P$ and is called the *negative* of P; note that $-P$ is indeed a point in $E(K)$. Also, $-\infty = \infty$.

Point addition. Let $P = (x_1, y_1) \in E(K)$ and $Q = (x_2, y_2) \in E(K)$, where $P \neq \pm Q$. Then $P + Q = (x_3, y_3)$, where
$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2$ and $y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1$.

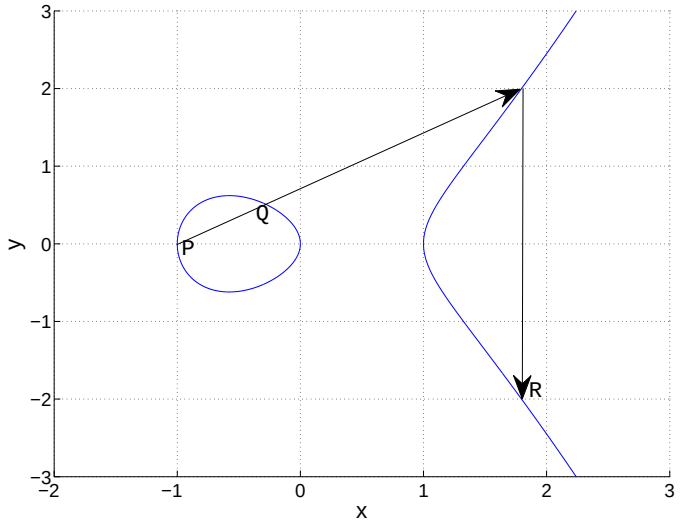

Figure 5.4: Graphical point addition. $R = P + Q$.

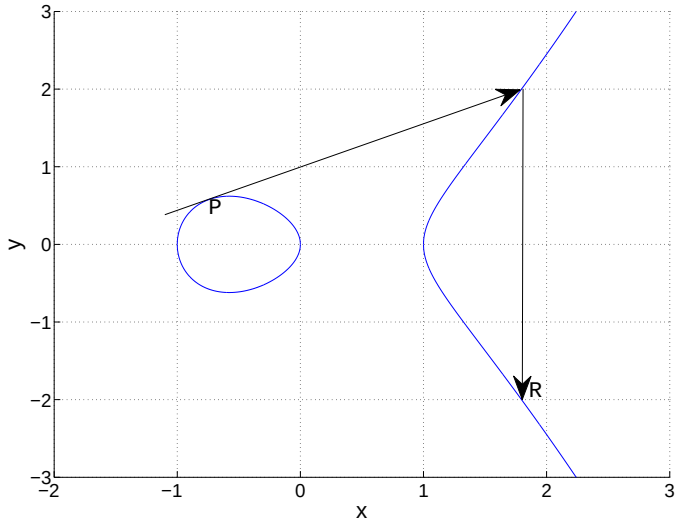

Figure 5.5: Graphical point doubling. $R = 2P$.

Point doubling. Let $P = (x_1, y_1) \in E(K)$, where $P \neq -P$. Then $2P = (x_3, y_3)$, where $x_3 = \left(\frac{3x_1^2+a}{2y_1}\right)^2 - 2x_1$ and $y_3 = \left(\frac{3x_1^2+a}{2y_1}\right)(x_1 - x_3) - y_1$.

5.3 Point Multiplication

The point addition and doubling formulas can now be used to derive a multiplication method. A simple multiplication method is displayed in Algorithm 6.

Unfortunately, this multiplication method is not safe from a Simple Power Analysis (SPA) attack. To the understand the problem of such an attack, a short excursion is

Algorithm 6 Right-to-left binary method for point multiplication

Require: $k = (k_{t-1}, ..., k_1, k_0)_2$, $P \in E(\mathbb{F}_q)$.
Ensure: kP.
1: $Q \leftarrow \infty$.
2: **for** i from 0 to $t-1$ **do**
3: If $k_i = 1$ then $Q \leftarrow Q + P$.
4: $P \leftarrow 2P$
5: **end for**
6: Return(Q).

necessary. Let us assume that there is a microprocessor that is used for the calculation of a point multiplication. Such a processor performs every part of the algorithm step by step (or line by line). Dependent on k_i, the processor will either add P to Q or not. This changes the runtime of a single loop cycle. This time can be measured. If k_i is one, the runtime is longer than in the case of k_i being zero. By measuring the power consumption during a point multiplication all bits of k can be recovered. This is a problem. Especially in the case that k is used as temporary private key (an ephemeral key) as it is in Algorithm 5. If k is discovered, the ECDSA signature generation algorithm is not secure any more.

So this vulnerability needs to be taken care of. Algorithm 7 shows a multiplication algorithm which is safe against SPA attacks. It is important to notice at this point that the algorithms is still vulnerable. Depending on the accessed addresses, the power trace changes minimally.

Algorithm 7 SPA safe point multiplication

Require: $k = (k_{t-1}, ..., k_1, k_0)_2$, $k_{t-1} = 1$, $P \in E(\mathbb{F}_q)$.
Ensure: kP.
1: $Q[0] \leftarrow P$.
2: $Q[1] \leftarrow 2P$.
3: **for** i from $t-2$ to 0 **do**
4: $Q[1 \oplus k_i] \leftarrow Q[k_i] + Q[1 \oplus k_i]$
5: $Q[k_i] \leftarrow 2Q[k_i]$
6: **end for**
7: Return(Q).

In order to optimize this algorithm, the succeeding point addition and doubling can be merged into a single function that calculates both results at the same time.

5.4 Comparison of Point Multiplication Algorithms

Because of the long runtime of point multiplication algorithms, a lot of research has been done in optimizing their performance. This overview assumes that $char(K) \neq 2, 3$. So the Weierstrass equation can be written as $E\backslash K$: $y^2 = x^3 + ax + b$. The presented optimizations can be also applied to different underlying fields.

Affine method This is a straightforward implementation of the previously shown formulas. The problem with this implementation is the inverse. Calculating an inverse is computationally much more complex than multiplying or adding two values. Because

of the resulting, excruciating runtime, it should not be considered for implementation.

Standard projection The inverse of the affine method can be avoided by introducing a third coordinate. The point $P = (x, y)$ can be transformed[1] to $P = (X, Y, Z)$ where $(x, y) = (X/Z, Y/Z)$. Firstly, this changes the elliptic curve equation to $Y^2Z = X^3 + aXZ^2 + bZ$. Secondly the addition and doubling formulas need to be changed.

Jacobian projection Similar to the standard projection, a Jacobian projection can be used for transformation. In this case the point P can be represented as $P = (X, Y, Z)$. $P = (x, y) = (X/Z^2, Y/Z^3)$. The new Weierstrass equation is $Y^2 = X^3 + aXZ^3 + bZ^6$. The resulting formulas for *point doubling* $P_3 = 2P_1$ are:

$$X_3 = (3X_1^2 + aZ_1^4)^2 - 8X_1Y_1^2 \tag{5.7}$$
$$Y_3 = (3X_1^2 + aZ_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \tag{5.8}$$
$$Z_3 = 2Y_1Z_1 \tag{5.9}$$

The resulting formulas for a *point addition* $P_3 = P_1 + P_2$ are:

$$X_3 = (Y_2Z_1^3 - Y_1)^2 - (X_2Z_1^2 - X_1)^2(X_1 + X_2Z_1^2) \tag{5.10}$$
$$Y_3 = (Y_2Z_1^3 - Y_1)(X_1(X_2Z_1^2 - X_1)^2 - X_3) - Y_1(X_2Z_1^2 - X_1)^3 \tag{5.11}$$
$$Z_3 = (X_2Z_1^2 - X_1)Z_1 \tag{5.12}$$

Double-and-add Algorithm A lot of computation can be avoided by combining the *point addition* and *point doubling* algorithms. Parts of the formulas do not need to be calculated twice any more. This is specially advantageous in Algorithm 7 in which two points are always added and one of them is always doubled.

Montgomery ladder The Montgomery ladder provides a very special optimization. The key is to keep the difference $Q_1 - Q_0$ constant. Algorithm 7 fulfills this property. Let us look at the example $10100011_b \cdot P$ in table 5.1.

bit	formula		Q_0	Q_1	$Q_1 - Q_0$
1	initialization		P	$2P$	P
0	$Q_1' = Q_0 + Q_1$	$Q_0' = 2Q_0$	$2P$	$3P$	P
1	$Q_0' = Q_1 + Q_0$	$Q_1' = 2Q_1$	$5P$	$6P$	P
0	$Q_1' = Q_0 + Q_1$	$Q_0' = 2Q_0$	$10P$	$11P$	P
0	$Q_1' = Q_0 + Q_1$	$Q_0' = 2Q_0$	$20P$	$21P$	P
0	$Q_1' = Q_0 + Q_1$	$Q_0' = 2Q_0$	$40P$	$41P$	P
1	$Q_0' = Q_1 + Q_0$	$Q_1' = 2Q_1$	$81P$	$82P$	P
1	$Q_0' = Q_1 + Q_0$	$Q_1' = 2Q_1$	$163P$	$164P$	P

Table 5.1: Point multiplication $10100011_b \cdot P$ using Algorithm 7. The result is stored in Q_0.

[1] Lower case letters are used for normal coordinates. Upper case letters are used for projected coordinates.

Izu, Möeller and Takagi [21] presented a Montgomery Ladder. Their formulas already have another improvement built in. They are in a X-only point representation (see next item in this enumeration). The following part is taken from their paper.

Let x_1, x_2 be x-coordinate values of two points P_1, P_2 of an elliptic curve $E : y^2 = x^3 + ax + b$. Then the x-coordinate value x_3 of the sum $P_3 = P_1 + P_2$ is given by

$$x_3 = \frac{2(x_1 + x_2)(x_1x_2 + a) + 4b}{(x_1 - x_2)^2} - x_3' \tag{5.13}$$

where x_3' is the x-coordinate value of $P_3' = P_1 - P_2$. On the other hand, the x-coordinate value of x_4 of the doubled point $P_4 = 2P_1$ is given by

$$x_4 = \frac{(x_1^2 - a)^2 - 8bx_1}{4(x_1^3 + ax_1 + b}. \tag{5.14}$$

X-only point representation An optimization in connection with the Montgomery ladder is to avoid the calculation of the Y-coordinate. Consequently, memory and field operations can be saved. After the last operation the x-coordinates of $kP = (x_1, y_1)$ and $(k + 1)P = (x_2, y_2)$ have been calculated. If needed, the y-coordinate can be recovered using kP and $(k+1)P$. Izu et al. also presented a formula for y-recovery, where $Q_1 = (x_1, y_1), Q_2 = (x_2, y_2)$ and $P = Q_2 - Q_1 = (x, y)$.

$$y_1 = \frac{y^2 + x_1^3 + ax_1 + b - (x - x_1)^2(x_1 + x_2 + x)}{2y} \tag{5.15}$$

Using the previously introduced point projection, this formula works without an inversion.

Common-Z point representation Another optimization is to merge the z-coordinate of the two points used in algorithm 7. The points $Q_1 = (X_1, Y_1, Z_1)$ and $Q_2 = (X_2, Y_2, Z_2)$ can be represented as $Q_1 = (X_1', Y_1', Z') = (X_1Z_2, Y_1Z_2, Z_1Z_2)$ and $Q_2 = (X_2', Y_2', Z') = (X_2Z_1, Y_2Z_1, Z_2Z_1)$. $Z' = Z_1 \cdot Z_2$. Consequently a register can be saved.

Different point multiplication methods are summed up in table 5.2. No distinctions are made between field multiplication and squaring algorithms. Usually a field squaring can be performed faster than a field multiplication. This is not the case for the implementation used in this thesis. Also the field addition and subtraction algorithms are accumulated.

It is possible to create double-and-add implementations for the *affine, standard projective and Jacobian projective* methods. They are not shown in this table. The last three implementations make use of Montgomery ladders. The last two implementations make use of a common-Z point representation.

The algorithm by Izu et al. actually needs 7 registers. The problem is that during multiplications, source registers are used as destination registers (e.g. $A = A \cdot B$). For this operation, an extra register is needed as temporary storage.

An in-place field multiplication method does not support $A = A \cdot B$. The double and add algorithm by Auer does not use such multiplications. That is why hardly any extra memory is required. The cost of the memory reduction comes with the price of extra computation requirement.

Method	Reference	Registers	Calculation
Affine Add	[12]		$1I + 3M$
Affine Double	[12]		$1I + 4M$
			$2I + 7M$
Std. Projective Add	[12]		$14M$
Std. Projective Double	[12]		$10M$
			$24M$
Jacobian Add	[12]		$16M$
Jacobian Double	[12]		$8M$
		10	$24M$
Izu, Möller, Takagi	[21]	7+1	$15M + 23A$
Auer	[4]	7	$18M + 22A$
this work	see App. D	8	$15M + 17A$

Table 5.2: Different multiplication Methods compared. 'I' is an inversion. 'M' is a multiplication. 'A' is an addition.

The author tried to minimize the number of multiplications required by Auer's implementation. The resulting solution uses as many multiplications as Izu et al. An advantage is the minimized number of required additions. The source and destination registers of the multiplications are kept separately.

Because of the memory advantage of Auer's implementation, his algorithm has been used for all subsequent performance evaluations.

Chapter 6

Prime Field Arithmetic

This chapter provides algorithms to deal with prime fields $\mathbb{F}_p$. This is a special kind of field because it is finite. A finite field is also called Galois field.

A field can be seen as a set of elements with some properties. The two most important operations that can be used on a field are an addition and a multiplication. Other operations like a subtraction and a division can be derived from those operations. Very popular fields are the real field $\mathbb{R}$, the complex field $\mathbb{C}$ or the field of rational numbers $\mathbb{Q}$. Those fields are not finite because they define an infinite set of numbers.

For each prime p, there exists a prime fields $\mathbb{F}_p$ with p elements. Integer arithmetic modulo p forms such a prime field $\mathbb{F}_p$. An advantage is that an integer $i \in \mathbb{F}_p$ can be represented with a fixed number of bits because $\mathbb{F}_p$ contains a fixed number of elements. Prime fields are also well suited for cryptographic algorithms such as the ECDSA.

Because prime-field arithmetic is required for the elliptic curve digital signature algorithm it is described in this chapter. Or rather prime field arithmetic is described using an architecture with small word size W.

W usually is a multiple of 8 and is called a word. Because elements of $\mathbb{F}_p$ usually are big integers, several words are needed to store such a big number. $t = \lceil ld(p)/W \rceil$ words are needed to store an integer $i \in \mathbb{F}_p$. In the next sections, a and b are used to present such integers $i \in [0, p-1]$. The memory cells used to store a can be written as $(A[t-1], A[t-2], \cdots, A[1], A[0])$. The rightmost bit and word have the index zero. The index of the leftmost word is $t-1$.

This chapter deals with the simpler addition, subtraction and multiplication operations. Those algorithms result into the majority of the runtime of the ECDSA algorithm. For the inversion (e.g. the Montgomery inversion) algorithm, the reader is encouraged to read the Chapter 2.2.5 in the Guide to Elliptic Curve Cryptography [12].

6.1 Addition and Subtraction

The addition and subtraction algorithms usually are very fast and efficient. They are also a lot easier to implement than the multiplication algorithms, handled in Section 6.2.

One important thing to understand the Algorithm 8 is the carry propagation. When two W-wide words are added, the result has $W+1$ bits. The extra, most significant bit is stored in the carry bit ε. The algorithm first adds a and b. If the sum is larger than or equal to p, p is subtracted from the intermediate result stored in c. In the case of the subtraction, performed within lines 6-9, ε is used as borrow bit.

Algorithm 8 Prime field addition in $\mathbb{F}_p$

Require: Two integers $a, b \in [0, p-1]$ and a modulus p.
Ensure: $c = (a + b) \pmod{p}$.
1: $(\varepsilon, C[0]) \leftarrow A[0] + B[0]$.
2: **for** i from 1 to $t - 1$ **do**
3: $\quad(\varepsilon, C[i]) \leftarrow A[i] + B[i] + \varepsilon$.
4: **end for**
5: **if** $\varepsilon = 1$ or $c \geq p$ **then**
6: $\quad(\varepsilon, C[0]) \leftarrow C[0] - P[0]$.
7: $\quad$**for** i from 1 to $t - 1$ **do**
8: $\quad\quad(\varepsilon, C[i]) \leftarrow C[i] - P[i] - \varepsilon$.
9: $\quad$**end for**
10: **end if**
11: Return(c).

Algorithm 9 shows the subtraction procedure. It is very similar to the addition algorithm. A difference is in line 5. The extra check ($c \geq p$) is not required for a subtraction algorithm, because after the subtraction $c = a - b$, c cannot be larger than p. The algorithm first calculates the difference between a and b. If this difference is negative, p is added to the intermediate result stored in c.

Algorithm 9 Prime field subtraction in $\mathbb{F}_p$

Require: Two integers $a, b \in [0, p-1]$ and a modulus p.
Ensure: $c = (a - b) \pmod{p}$.
1: $(\varepsilon, C[0]) \leftarrow A[0] - B[0]$.
2: **for** i from 1 to $t - 1$ **do**
3: $\quad(\varepsilon, C[i]) \leftarrow A[i] - B[i] - \varepsilon$.
4: **end for**
5: **if** $\varepsilon = 1$ **then**
6: $\quad(\varepsilon, C[0]) \leftarrow C[0] + P[0]$.
7: $\quad$**for** i from 1 to $t - 1$ **do**
8: $\quad\quad(\varepsilon, C[i]) \leftarrow C[i] + P[i] + \varepsilon$.
9: $\quad$**end for**
10: **end if**
11: Return(c).

6.2 Integer Multiplication

When two integers $a, b \in [0, p-1]$ are multiplied, the width of the result is $2t$[1] words. Because this result is a lot larger than p, it needs to be reduced. The reduction of the intermediate result is handled in the next section.

There are two possibilities to multiply two numbers: the operand scanning and the product scanning form.

[1] Remember: t is the number words needed to store p.

In Algorithm 10 the operand scanning form is shown. (UV) is a temporary register that uses 2 words of memory. U is the higher and V is the lower word. The advantage of this algorithm is that every register of a is only read once if its result is buffered. The disadvantage is that c and b need to be read very often.

Line 5 can give the reader some troubles, because a $2W$-bit and two W-bit values are added and the result still is a $2W$-bit value. This can be easily explained by inspecting the worst-case example: $FFFF_h \cdot FFFF_h = FFFE0001_h$. $FFFE0001_h + FFFF_h + FFFF_h = FFFFFFFF_h$. Which can still be represented in a $2W$-bit register.

Algorithm 10 Integer multiplication (operand scanning form)

Require: Integers $a, b \in [0, p-1]$.
Ensure: $c = a \cdot b$.
1: Set $C[i] \leftarrow 0$ for $0 \leq i \leq t-1$.
2: **for** i from 0 to $t-1$ **do**
3: $U \leftarrow 0$.
4: **for** j from 0 to $t-1$ **do**
5: $(UV) \leftarrow C[i+j] + A[i] \cdot B[j] + U$.
6: $C[i+j] \leftarrow V$.
7: **end for**
8: $C[i+t] \leftarrow U$.
9: **end for**
10: Return(c).

The second integer multiplication Algorithm (11) uses a product scanning form. In the case of an available **MULACC** (multiply and accumulate) instruction, this algorithm is faster than the algorithm using operand scanning. This is the reason why the original algorithm from [12] is modified, so an accumulator ACC is used. It has to be $2W + \lceil ld(t) \rceil$ bits wide. So the lower words $ACC[0]$ and $ACC[1]$ have both W bits.

Algorithm 11 Integer multiplication (product scanning form)

Require: Integers $a, b \in [0, p-1]$.
Ensure: $c = a \cdot b$.
1: $ACC \leftarrow 0$
2: **for** k from 0 to $2t-2$ **do**
3: **for** each element of $\{(i,j) | i + j = k, 0 \leq i, j \leq t-1\}$ **do**
4: $ACC \leftarrow ACC + A[i] \cdot B[j]$.
5: **end for**
6: $C[k] \leftarrow ACC[0]$.
7: $ACC \leftarrow ACC \gg W$ $(ACC[0] \leftarrow ACC[1], ACC[1] \leftarrow ACC[2], ACC[2] \leftarrow 0)$.
8: **end for**
9: $C[2t-1] \leftarrow ACC[0]$.
10: Return(c).

In the case of a squaring operation Algorithm 11 can be improved. Adding $A[i] \cdot A[j]$ to the accumulator is equal to adding $A[j] \cdot A[i]$. This can be changed to adding $2 \cdot A[i] \cdot A[j]$ to the accumulator for every $i \neq j$. In the best case, this improves the number of required calculation steps nearly by a factor of two.

6.3 Reduction

A very interesting reduction technique is the method by Barrett. It is described in the book [12]. Because it is of limited usage in this thesis, it is neglected.

The big advantage of the NIST primes, used as modulo, are their special forms. They can be expressed as sum or difference of small numbers and powers of 2:

$$\begin{aligned} p_{192} &= 2^{192} - 2^{64} - 1 \\ p_{224} &= 2^{224} - 2^{96} + 1 \\ p_{256} &= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \\ p_{384} &= 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \\ p_{521} &= 2^{521} - 1 \end{aligned}$$

Obviously all the exponents (except for p_{521}) are multiples of 32. This is specially important for processors with a words size of 32 bits.

But why are those numbers written in this way. Why is p_{192} not written as $FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFF_h$? (Consider that there is an 'E' in this list of 'F'.) The representation with exponents of two can be perfectly used for the reduction of big integers larger than p. Such big integers usually occur as a result of a multiplications.

6.3.1 NIST-P192

This thesis uses the NIST-P192 parameters. So let us investigate the prime p_{192} in more detail. Multiplying the two integers $a, b \in [0, p-1]$ can result in a c with $\lceil ld(c) \rceil = 2 \cdot 192 = 384$ bits. This number can be expressed in the following manner:

$$c = c_5 2^{320} + c_4 2^{256} + c_3 2^{192} + c_2 2^{128} + c_1 2^{64} + c_0 \tag{6.1}$$

This is a base-2^{64} representation of c. Each $c_i \in [0, 2^{64} - 1]$. Using this representation, the succeeding reduction algorithm can be performed swiftly. Higher powers can be reduced:

$$\begin{aligned} 2^{192} &\equiv 2^{64} + 1 \pmod{p} \\ 2^{256} &\equiv 2^{128} + 2^{64} \pmod{p} \\ 2^{320} &\equiv 2^{128} + 2^{64} + 1 \pmod{p} \end{aligned}$$

By applying those reductions to c, the result of a reduction is:

$$\begin{aligned} c \equiv c_2 2^{128} &+ c_5 2^{64} + c_5 \\ + c_4 2^{128} &+ c_4 2^{64} \\ &+ c_3 2^{64} + c_3 \\ + c_2 2^{128} &+ c_1 2^{64} + c_0 \end{aligned}$$

The only inconvenience is that the final result can be larger than p. This can be undone by a series of subtractions $c \leftarrow c - p$, until c is less than p.

By using p_{192} as a prime, an in-place reduction can be done during a multiplication. This can be done in four steps:

1. Use the product scanning method to calculate the higher 192 bits of the multiplication. These are c_5, c_4 and c_3. The destination memory is used as buffer.

2. Reduce c_5, c_4 and c_3 in place. The overflow is stored in a temporary register.

3. Calculate the lower 192 bits of the integer multiplication. They need to be added to the reduced higher 192 bits, which are stored in the destination memory. This can be done in the course of the multiplication, by adding $C[i]$ to the accumulator.

4. At this point, the accumulator is still filled. The accumulator must be added to c by performing a final reduction, until $c < p_{192}$.

The advantage of this method is that it is very fast and hardly uses any temporary memory. The disadvantage is that the following multiplication cannot be performed any more: $c = a \cdot c \pmod{p}$.

6.3.2 NIST-P256

The reduction method used for p_{192} can also be applied to reduce p_{256}. Unfortunately it is more complex. The reduction is summarized in Table 6.1. For instance the higher power 2^{256} can be reduced to $2^{256} \equiv 2^{224} - 2^{192} - 2^{96} + 1 \pmod{p_{256}}$.

A problem with p_{256} is that it cannot be reduced in place as simple as p_{192}.

		c_7 2^{224}	c_6 2^{192}	c_5 2^{160}	c_4 2^{128}	c_3 2^{96}	c_2 2^{64}	c_1 2^{32}	c_0 2^0
c_8	2^{256}	1	-1			-1			1
c_9	2^{288}		-1		-1	-1			1
c_{10}	2^{320}	-1		-1	-1		1	1	
c_{11}	2^{352}	-1		-1		2	1	1	-1
c_{12}	2^{384}	-1			2	2			-1
c_{13}	2^{416}	-1	1	2	2	1	-1	-1	-1
c_{14}	2^{448}		3	2	1		-1	-1	-1
c_{15}	2^{480}	3	2	1		-1	-1	-1	

Table 6.1: Fast reduction using modulo $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

Table 6.1 shows how the upper 256-bit of the multiplication must be added to the lower 256-bit in order to perform a reduction. Right next to c_i is its exponent. So in order to reduce c_{10}, it must be subtracted from c_7, c_5 and c_4 and added to c_2 and c_1. It is important to keep track of the carry bits, when a reduction is performed. By adding c_{10} to c_1 also all subsequent words $(c_2, c_3, \cdots)$ can be affected.

6.4 Montgomery Multiplication

Peter Lawrence Montgomery presented a lot of different methods to improve elliptic curve related operations. One of those improvements can be used to multiply two numbers and reduce those numbers at the same time. This can be done by introducing a third term: r. So if two numbers $a, b \in [0, p-1]$ are multiplied, the result is

$$c = a \cdot b \cdot r^{-1} \pmod{p}. \tag{6.2}$$

There are two requirements for r. $r > p$ and $gcd(r, p) = 1$. Usually p is prime and $r = 2^{Wt}$[2]. So p is odd and r a multiple of 2. So those requirements are fulfilled by default.

This method is of advantage when a lot of multiplications need to be done. Only an initial multiplication with r is needed. $\tilde{a} = a \cdot r \bmod p$. $\tilde{b} = b \cdot r \bmod p$. So a Montgomery multiplication results in

$$\begin{aligned} Mont(\tilde{a}, \tilde{b}) &\equiv \tilde{a} \cdot \tilde{b} \cdot r^{-1} \pmod{p} \\ &\equiv (a \cdot r) \cdot (b \cdot r) \cdot r^{-1} \pmod{p} \\ &\equiv (a \cdot b) \cdot r \pmod{p} \end{aligned} \tag{6.3}$$

The result of $a \cdot b$ is also multiplied with r. To recover the desired result, a multiplication with one is needed:

$$\begin{aligned} Mont(\tilde{a}, 1) &\equiv \tilde{a} \cdot 1 \cdot r^{-1} \pmod{p} \\ &\equiv (a \cdot r) \cdot 1 \cdot r^{-1} \pmod{p} \\ &\equiv a \pmod{p} \end{aligned} \tag{6.4}$$

For the implementation of the Montgomery multiplication, an extra parameter (p') is required. It is defined in $r \cdot r^{-1} - p \cdot p' = 1$. p' can be calculated using the extended Euclidean algorithm. For the Montgomery multiplication in Algorithm 12, only the storage of the lower W bits of p' are required: $p'_0 = p \pmod{2^W}$.

For the ECDSA algorithm, all primes are fixed. So p'_0 can be pre-computed and stored as a constant. Algorithm 12 is known as Finely Integrated Product Scanning Form (FIPS) from the paper [25] by C. Koç, T. Acar and B. Kaliski. It makes use of the multiply-accumulate unit and it works in place.

The advantages of a Montgomery multiplication is that it works for all primes.

[2]Remember: W is the word size of the processor and $t = \lceil ld(p)/W \rceil$.

Algorithm 12 FIPS Montgomery multiplication

Require: Integers $a, b \in [0, p-1]$ and pre-computed p'_0.
Ensure: $c = a \cdot b \cdot r^{-1} \pmod{p}$.

1: $ACC \leftarrow 0$
2: **for** i from 0 to $t-1$ **do**
3: **for** j from 0 to $i-1$ **do**
4: $ACC \leftarrow ACC + A[j] \cdot B[i-j]$.
5: $ACC \leftarrow ACC + C[j] \cdot P[i-j]$.
6: **end for**
7: $ACC \leftarrow ACC + A[i] \cdot B[0]$.
8: $C[i] \leftarrow ACC[0] \cdot p'_0 \pmod{2^W}$.
9: $ACC \leftarrow ACC + C[i] \cdot P[0]$.
10: $ACC \leftarrow ACC \gg W$.
11: **end for**
12: **for** i from t to $2t-1$ **do**
13: **for** j from $i-t+1$ to $t-1$ **do**
14: $ACC \leftarrow ACC + A[j] \cdot B[i-j]$.
15: $ACC \leftarrow ACC + C[j] \cdot P[i-j]$.
16: **end for**
17: $C[i-s] \leftarrow ACC[0]$.
18: $ACC \leftarrow ACC \gg W$.
19: **end for**
20: **if** $ACC[0] \neq 0$ or $c \geq p$ **then**
21: $c \leftarrow c - p$.
22: **end if**
23: Return(c).

Chapter 7

Design Flow for Processor Creation

In many projects, there are certain restrictions to the design. Usually a certain processor is chosen dependent on its properties for the required application. Then the algorithm(s) are implemented on the defined processor. If it is an embedded device, this is usually done in 'C' or 'C++'. During a final optimization phase, several performance relevant 'C' functions are replaced by assembler functions.

This procedure cannot be applied for this thesis. The only requirements have been that a custom processor should be implemented and optimized for the elliptic curve digital signature algorithm. So the initial set of processors is infinite. This problem is comparable with the 'chicken or the egg' causality dilemma. One of both cannot exist without the other.

For the Neptun processor this problem has been solved in the four steps:

1. Implement a virtual processor that supports the simplest possible commands.
2. Implement the algorithms using those simple commands.
3. Optimize time critical functions. Those mainly are the field operations. Parallelize instructions if necessary.
4. Implement an instruction set which provides all necessary operations.

In order to apply those steps a software has been written in Java that is used to execute those four points. This program in connection with the design flow is described in the following sections.

7.1 Features of the Simulated Processor

The first step of the design flow is to create a simple processor. This processor can be reused and improved at a later design phase. The initial set of commands was derived from the Thumb [2] and AVR [3] instruction sets. These instruction sets only use a 16-bit command representation and are commonly used for embedded processors with low-power requirements. The focus was to just use as few commands as possible, but also make them as reusable as possible.

This initial instruction set must be modifiable and flexible. So, for the Neptun processor, it has been decided to implement a processor simulator. This simulator should be capable of the following use cases:

- Parse assembler source code.
- Generate some sort of data structure.
- Simulate the code.
- Generate statistics that can be used for optimization.
- Generate executable files.
- Use other programs/libraries to verify the assembler programs.
- Use the simulator to verify the VHDL model.

In the following sub-sections the used data structure and important parts of the processor simulator are described. Especially the used data structure is important for the subsequent design steps.

7.1.1 Data Structure

A short declaration of terms is necessary. A command refers to a type of instruction. An instruction is an instance of a certain command.

Every command is encapsulated in an own class and derived from the common abstract class *GenericCommand*. This ensures certain properties of the derived commands. Every command fulfills the following use-cases:

- Store the parameters (e.g. source/destination register, immediate values, ...) needed for the command.
- Check during creation of an instance if those parameters are valid.
- Execute the command's behavior on the virtual processor.
- Generate the control signals needed for this command execution.
- Generate a 16-bit program word used for the 16-bit instruction set.
- Generate a string representation.

Every instruction is stored within its function. Every function is stored in a program. Only one program can be executed at a time on an instance of the virtual processor. By having this internal data structure the source format of a program does not matter.

A function is created by adding instances of commands. To avoid writing a file parser and editor that supports the appropriate syntax highlighting and checking, the way of writing an assembler program in this thesis is different.

Every assembler function is encapsulated within a Java function (Figure 7.1). It can also contain several assembler functions within a Java function. Usually that is not required. With the call of *Function.Begin()* a new function is started. The parameter is the internal name of the assembler function. The *CommandProvider* is used as an additional abstraction layer. Usually it is named *C*. By entering 'C.' an advanced editor like *Netbeans*

```
/**
 * Calculates BaseA += BaseB mod BaseC
 */
public static void ADD(int length) {
    Function F = Function.Begin("test.modulo");
    CommandProvider C = new CommandProvider(F);

    C.LD(Reg.BaseA, 0);
    C.MOV_LD(Reg.RamOut, Reg.Work0, Reg.BaseB, 0);
    C.ADD_ST(Reg.RamOut, Reg.Work0, Reg.BaseA, 0);
    for (int i = 1; i < length; i++) {
        C.LD(Reg.BaseA, i);
        C.MOV_LD(Reg.RamOut, Reg.Work0, Reg.BaseB, i);
        C.ADDC_ST(Reg.RamOut, Reg.Work0, Reg.BaseA, i);
    }
    C.BRA(BRA.CarrySet, "test.modulo.reduce");

    C.LD(Reg.BaseC, 0);
    C.MOV_LD(Reg.RamOut, Reg.Work0, Reg.BaseA, 0);
    C.CMP(Reg.RamOut, Reg.Work0);
    for (int i = 1; i < length; i++) {
        C.LD(Reg.BaseC, i);
        C.MOV_LD(Reg.RamOut, Reg.Work0, Reg.BaseA, i);
        C.CMPC(Reg.RamOut, Reg.Work0);
    }
    C.BRA(BRA.Lower, "test.modulo.finish");

    F.newLabel("test.modulo.reduce");
    C.LD(Reg.BaseB, 0);
    C.MOV_LD(Reg.RamOut, Reg.Work0, Reg.BaseA, 0);
    C.SUB_ST(Reg.RamOut, Reg.Work0, Reg.BaseA, 0);
    for (int i = 1; i < length; i++) {
        C.LD(Reg.BaseB, i);
        C.MOV_LD(Reg.RamOut, Reg.Work0, Reg.BaseA, i);
        C.SUBC_ST(Reg.RamOut, Reg.Work0, Reg.BaseA, i);
    }

    F.newLabel("test.modulo.finish");
    C.RET();
    Function.End();
}
```

Figure 7.1: This function adds two numbers. A prime is stored relative to *BaseC*.

automatically provides a set of available commands. This editor also highlights syntax errors. Logical errors (a register cannot be used as a certain parameter) are checked during runtime of the simulation. With a call of *Function.End()*, the end of a function is defined.

Labels are added with a call of *F.newLabel()*, where *F* represents the name of the

function variable.

Because Java can be used as a very powerful preprocessor, programming assembler functions is a lot less troublesome than usual. As a result a complex topic like 'Preprocessor Programming', as it exists in *C* and *C++* can be avoided completely.

```
Processor P = new Processor();

testAlgorithm();
Field.multiplication();
Field.addition();
Field.subtraction();

P.program.PrepareExecution();
P.program.Execute();
```

Figure 7.2: This code snippet creates an executable program.

At this point a list of instructions is stored within the *Function* class. The next step is to use a set of functions and create a program. This is shown in Figure 7.2.

Unfortunately, a program is written for a certain processor. This processor has certain attributes. These attributes have an influence to the available instructions (the *Command-Provider* automatically handles some sort of code conversion). A new *program* is generated automatically by creating a new processor. The function calls after the *new Processor()* instruction create new assembler functions as it has been mentioned before. The sequence of those Java function call arranges the assembler functions within the program memory. Usually the point of entry (e.g. after reset) is the first function.

The next step is very important. By calling *PrepareExecution()*, the current set of functions and instructions is compiled. This is done in the following steps:

1. A unique program address is assigned to each command. Because some commands need several program memory entries, consecutive addresses are reserved for the command.

2. All labels are collected. These can be labels that mark the beginning of a function or labels that are used for branching. Each label references an instance of an instruction. So each label automatically knows the address it is referring to.

3. Certain commands make use of those labels (**CALL**, **JMP** and **BRA**). Because labels are stored as strings, every reference needs to be checked with the list of existing labels. If the referenced string is found, the resulting address is stored in the instruction. If the referenced string is not found, an error message appears.

At this point, a program execution can be simulated by calling *Execute()*.

7.1.2 Virtual Processor

A processor consists of registers, memories and peripheral components. A register is represented with a variable. This register can be read and written by every command.

A memory is an array of integers. During a memory access the destination memory or peripheral is determined and its access function is called.

Designing a peripheral is more complex. Its state can change at each cycle. It is important to regard the details so that the Java model is equal to the VHDL model.

The virtual execution of a program can be done in the following steps:

1. Before instruction execution: Calculate the default next program counter
2. Find the instruction at the current program counter address.
3. Execute the command's *Execute* function that simulates its behavior.
4. Update several statistics.
5. After instruction execution: Set the program counter, test all registers for validity, update the states of all peripherals.
6. Print debug output.

Because the main simulation of code is done within the commands, the processor is reduced to a set of registers. Additionally it needs to have some relay properties. In the case of a memory read, the processor's *Read()* function finds the designated module that covers this address and calls its read function.

There are different kinds of debug output available. A straightforward debug output is the current register output. Because of the flexibility of the Java model there are a lot cleverer ways to print debug messages. A debug message can depend on the currently executed function or command and other factors. An example is to print the state of each register (12 memory entries, representing a 192-bit number) during a point multiplication. By comparing the states with the high-level model an error is found more easily.

Something important for the virtual program execution is a stop condition. In the case of this thesis several have been defined:

- A **RET** (return from function) command that finalizes the startup function with no more elements on the stack quits the program execution.
- A cycle counter counts the number of executed program cycles. If a certain bound is reached the program execution stops.
- The execution of an invalid program address results in an error.
- If any assertion fails an error is logged (and printed) and the execution stops. This usually happens if there is some sort of error within the Java model.

7.2 Implementation of the Algorithms

The previously described design of the virtual processor is used to implement the first few algorithms. At this point it is important to keep it simple and start with the easier algorithms. A natural occurring problem is the presence of errors in the initial virtual processor. So it is important to use tests for each algorithm. With those tests not only the algorithms are checked, but also the underlying virtual processor.

It is important to mention that each assembler function must be tested separately and several times. For that it is good to use two kinds of test cases. Those two kinds are

Method	Cycles
Native P-192	997
Optimized P-192	401
Native Montgomery	1580
Optimized Montgomery	651

Table 7.1: Comparison of native and optimized multiplication functions.

separated by their function arguments used for the test execution. The first set consists of minima and maxima of the function parameters. The second set uses random numbers. So each time the function is tested with different numbers. Certain errors can only occur at certain input parameters. A nice example for such a behavior occurred during the design of the P-192 field multiplication. Only after about 100,000 tests of the multiplication function an error occurred. The error was that a carry bit has be handled wrong. This carry flag was zero for the other 99,999 cases and toggled at one very special combination of input parameters.

The initial functions and algorithms have been programmed as simple as possible. The only goal has been to get the system running. Actually, optimizing the implemented time-critical functions is done in the next design phase.

7.3 Parallelizing the Commands

Up to now, algorithms have been implemented using the most basic instructions: **ADD**, **SUB**, **MULACC**, ... This initial instruction set is similar to the Thumb or AVR instruction sets. Unlike the Thumb instruction set, the arguments have not been limited. The only command that is using two cycles is the **LDR** command. All other commands can be executed within one cycle. At this point more complex commands like **POP** and **CALL** are simulated using those more simple commands.

The problem with this initial instruction set is that the resulting functions are very slow. Most parts of the processor are idle all the time. That is quite a big problem if the goal is to compete with co-processor presented in previous papers or the master thesis by Auer in [4].

So the next step has been to parallelize as many time critical operations as possible. Figure 7.3 shows how this idea has been carried out.

There are several steps in the presented optimizations. The first is to split the **LDR** command into an **LD** and **MOVNF** (move without flag update) command. The second optimization is to load a date at each processing cycle and process it as soon as possible. As a result, all commands can access the result of a memory operation directly. Some new commands need to be introduced that load/store data as well as process some data.

Especially optimizing the memory read access is important. During field multiplication a lot more read accesses are needed than write accesses. But also the write accesses can be optimized. Instead of storing a value into a register and storing it in the next cycle, these two operations can be unified.

At this point it is interesting to look at actual performance improvements in Table 7.1. The average speedup achieved with the here presented optimizations is about 2.5.

A further result of this optimization is the reduction of the number of entries in the program memory. This suggests that the size of the synthesized lookup-table (with 76 bits

1	**LD** A1
2	**MOVNF**
3	**LD** B11
4	**MOVNF**
5	**MULACC**
6	**LD** A2
7	**MOVNF**
8	**LD** B10
9	**MOVNF**
10	**MULACC**
11	**LD** A3
12	**MOVNF**
13	**LD** B9
14	**MOVNF**
15	**MULACC**
16	**STR** Acc0
17	**RSACC**

1	**LD** A1		
2	**MOVNF**	**LD** B11	
3	**LD** A2	**MULACC**	
4	**MOVNF**	**LD** B10	
5	**LD** A3	**MULACC**	
6	**MOVNF**	**LD** B9	
7	**MULACC**	**STR** Acc0	**RSACC**

Figure 7.3: Part of a field multiplication algorithm. On the left hand side is the unoptimized code. On the right hand side is the resulting optimized code. **MOVNF** is storing the previously loaded data into a register.

per entry, see 10.3.1) should change accordingly. Unfortunately, the size of the resulted logic stays the same. This is because the entropy of the table stayed the same. In other words, the total number of ones stayed the same, even if the number of zeros decreased.

7.4 Generating a Finite Instruction Set

Up to this point every command has been as flexible as possible. Every bit of the control vector has been modifiable for every instruction. In the sense of performance optimization this is the best thing possible.

Unfortunately for the generation of the reusable processor design (see Section 9.4), the number of possible instructions and their arguments needs to be limited. So the first step is to analyze the used instructions (after the completed optimization process). Therefore the timing critical commands (like **MULACC_LD** and **MOVNF_LD**) need to be kept. For other command combinations that are hardly needed, the optimizations can be undone.

It will be decided in Chapter 8 that the register size is 16 bits. So a load immediate operation needs a 16-bit operand. So there must be a constant representing a load immediate command and 16 bits of data. There are three solutions to this problem.

One is to increase the size of the instruction word to more than 16 bits. Because of the low importance of the load immediate command that is not practicable.

The two other solutions split the command into two cycles. One possibility is to load the higher 8 bits at first and the lower 8 bits during the next cycles. Unfortunately as a result the load immediate instruction cannot be used as jump instruction any more.

The best solution is to store the destination register during the first cycle. During the next cycle, the instruction word is directly stored in the destination register. In this case the load immediate command can also be used as jump command. So the minimum size

of the program word is 16 bits.

To fit all commands within such a small program word several commands need to be limited:

Limit the number of parameters. The shift instructions are hardly used within the algebraic functions. So it is hardly a cutback if the source and the destination register have to be identical.

Limit the type of parameters. Several operations can be done differently. Adding a number to a register can be done in two ways. Either an **ADDI** (add immediate) command can be used or an immediate is loaded to a register and the **ADD** command is used to add the two registers.

In some cases, certain operands for certain commands are not needed. The command **ORI** (or immediate), provided by the AVR instruction set (see [3]), has not been implemented in this thesis.

Limit the bits of an immediate. Naturally, some parameters are automatically limited. A logical shift of a 16-bit register does not need to use more than a $\lceil ld(16) \rceil = 4$-bit parameter. Other parameters like the size of the immediate for an **ADDI** command must be limited.

Limit the bits of a register parameter. A **MULACC_LD** (multiply and accumulate and load data from the memory) needs a 6-bit parameter for the load and two 4-bit parameters for the multiply accumulate (to access all registers). This results in 14 bits used for parameters. For a 16 bit instruction set, this is hardly useable. In the final implementation, the size of the two operands has been reduced to two times two bits.

Limit the type of commands. Initially an **ADD_LD** (add and load) command has been part of the design. Investigations have shown that this command is not needed. An **ADD_ST** (add and store result) command is a lot more useable and improves the performance of the algorithms.

7.4.1 Neptun Instruction Set

At this point it is interesting to take a look at the actually implemented instruction set. Table 7.2 shows the 16-bit instruction set used by the Neptun design. Usually, to describe an instruction set properly, more than 100 pages are needed. In this context, we concentrate on the program-word representation and not on the commands themselves.

The most important rules during an instruction-set construction it that every command must be mutually exclusive. It is not allowed that two distinct commands use the same combination of bits for certain operand combinations.

The properties of the instruction set in Table 7.2 are discussed in the following list:

- *SelOpA* is used to select operand A for the ALU. It is capable to select every input register. For that 4 bits are needed. To make the optimization for the synthesizer easier, the preferred bit indexes for *SelOpA* are 4-7.

- Not every register can be selected as source for operand B. This can be a small disadvantage during the design of a circuit, but the advantage is that *SelOpB* can be represented as three bits. Saving this one bit is important. Otherwise it would not

Name	Description	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	Add	0	0	0	0	Result				SelOpA				0	SelOpB		
ADDC	Add with Carry	0	0	0	0	Result				SelOpA				1	SelOpB		
SUB	Subtract	0	0	0	1	Result				SelOpA				0	SelOpB		
SUBC	Subtract with Carry	0	0	0	1	Result				SelOpA				1	SelOpB		
ADDI	Add Immediate	0	0	1	0	OpA/Res				immediate							
SUBI	Subtract Immediate	0	0	1	1	OpA/Res				immediate							
AND	Logical And	0	1	0	0	Result				SelOpA				0	SelOpB		
OR	Logical Or	0	1	0	0	Result				SelOpA				1	SelOpB		
XOR	Logical Exclusive Or	0	1	0	1	Result				SelOpA				0	SelOpB		
MOVNF	Copy a Register, no flag update	0	1	0	1	Result				SelOpA				1	0	0	0
MVN	Move and Negate	0	1	0	1	Result				SelOpA				1	0	1	0
LDI	Load Immediate	0	1	0	1	Result				X	X	X	X	1	1	1	1
RS	Right Shift	0	1	1	0	Result				SelOpA				0	SelOpB		
LS	Left Shift	0	1	1	0	Result				SelOpA				1	SelOpB		
CMPI	Compare Immediate	0	1	1	1	SelOpA				immediate							
RSI	Right Shift Immediate	1	0	0	0	Res/OpA				0	0	0	0	immediate			
ASRI	Arithmetic Shift Right Immediate	1	0	0	0	Res/OpA				0	0	0	1	immediate			
LSI	Left Shift Immediate	1	0	0	0	Res/OpA				0	0	1	0	immediate			
LDSI	Load Small Immediate	1	0	0	0	Result				1	immediate						
BRA	Branching	1	0	0	1	condition				immediate							
MUL	Multiply	1	0	1	0	0	0	0	0	SelOpA				R	SelOpB		
MULACC	Multiply and Accumulate	1	0	1	0	0	0	1	0	SelOpA				0	SelOpB		
ADDACC	Add to Accumulator	1	0	1	0	0	1	0	0	SelOpA				0	0	0	0
SUBACC	Subtract from Accumulator	1	0	1	0	0	1	0	1	SelOpA				0	0	0	0
RSACC	Right Shift the Accumulator	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
Custom1	ADDI(PC,4,-) & ST(SP,0)	1	0	1	0	1	0	0	1	0	1	1	1	0	1	1	0
Custom2	ADDI(SP,1,SP) & LD(SP,1)	1	0	1	0	1	0	1	0	0	1	1	0	0	0	0	0
Custom3	SUBI(SP,1,SP)	1	0	1	0	1	1	0	0	0	1	1	0	0	0	0	0
MOV_LD	Move RamOut and Load	1	0	1	1	0	0	Base		Result				Mem Rel			
STR	Store Register in Memory	1	0	1	1	0	1	Base		SelOpA				Mem Rel			
ADDACC_LD	Add Accumulator and Load	1	0	1	1	1	0	Base		SelOpA				Mem Rel			
SUBACC_LD	Subtract Acc. And Load	1	0	1	1	1	1	Base		SelOpA				Mem Rel			
ADDACC_ST	Add Acc. and Store Acc0	1	1	0	0	0	R	Base		SelOpA				Mem Rel			
SUBACC_ST	Subtract Acc. And Store Acc0	1	1	0	0	1	R	Base		SelOpA				Mem Rel			
MULACC_LD	Multiply & Acc. & Load	1	1	0	1	0	0	Base		OpA		OpB		Mem Rel			
MULACC_ST	Multiply & Acc. & Store Acc0	1	1	0	1	1	R	Base		OpA		OpB		Mem Rel			
ADD_ST	Add & Store Result	1	1	1	0	0	R	Base		OpA		OpB		Mem Rel			
ADDC_ST	Add with Carry & Store Result	1	1	1	0	1	R	Base		OpA		OpB		Mem Rel			
SUB_ST	Subtract & Store Result	1	1	1	1	0	R	Base		OpA		OpB		Mem Rel			
SUBC_ST	Subtract with C. & Store Result	1	1	1	1	1	R	Base		OpA		OpB		Mem Rel			
CALL	Function Call	Custom1, Custom3, LDI															
CMP	Compare	SUB(OpA,OpB,-)															
CMPC	Compare with Carry	SUBC(OpA,OpB,-)															
JMP	Jump to Address/Label	LDI(PC,imm)															
LD	Load	MOVLD(BaseX,Offset) without Move															
LDR	Load Register from Memory	LD(BaseX,Offset), MOVNF(MemOut,Dest)															
MOV	Copy a Register	Res = OpA | NULL															
POP	Get a Value from the Stack	Custom2, MOVNF(MemOut,Dest)															
PUSH	Store a Value in the Stack	STR(Source,SP,0), Custom3															
RET	Return from Function	POP(PC)															

Table 7.2: Summary of Neptun instruction-set.

have been possible to fit all commands within a 16-bit program word. The default place for *SelOpB* is the bits 0-2.

- Some important commands have the possibility to use immediates. An immediate is an integer that is fixed at program compilation. The way the CPU is designed, an immediate can only be used as operand B. That is why *SelOpB* and the immediates are mutually exclusive. All the immediates are right aligned at bit 0. Certain commands can use larger immediates than other commands. A logical shift left does not need more than a four bit immediate when the word size is 16-bit.

- *Result* represents the destination register of the commands. Similar to the other operands it is placed at the bits 8-11 by default. Something important is that the indices of the result registers and the indices for the operand A selection are coordinated. Some commands (**ADDI**, **SUBI**, logical shift operations) combine the *Result* and the *OperandA* fields. With one field, both parameters are selected.

- The result of a multiplication can only be stored in the Work registers. Either Work0 and Work1 or Work2 and Work3 are used. So only one bit is required for the selection of the destination registers. This is the bit labeled 'R' of the **MUL** command.

- To access the memory for a read or a write operation, a Base register needs to be selected and a 4-bit immediate offset is required. The Base register is selected with the *Base* field. The offset is represented with the *Mem Rel* field.

- The 'R' flag, which is used with the '_ST' commands, is used to shift the result of the accumulator by 16 bits.

- The operands of several parallelized commands needed to be greatly limited. The commands **MULACC_LD**, **MULACC_ST**, **ADD_ST**, **ADDC_ST**, **SUB_ST** and **SUBC_ST** only use two 2-bit parameters. Those commands are usually used in connection with very optimized assembler functions. So using only the Work registers and not being able to access other registers, should not be a problem.

- Some commands do not have their own program word. A compare is the same as a subtraction without storing the result. An (absolute) jump is identical to a load immediate. A return is identical to a **POP** with the program counter as destination register.

- Some commands need to be split into several other commands. A **CALL** is split into three commands. The first stores the return address into the stack (**Custom1**). The second updates the stack pointer (**Custom2**). The third command is a **JMP**. A jump is constructed with a load immediate. So the **LDI** command is used to jump to the function to call.

The presented instruction set has been optimized for the elliptic curve digital signature algorithms. Using it, the signature can be calculated very efficiently.

Chapter 8

Evaluation of Platform

Before actually designing the processor in detail, some basic design decisions need to be made. The most important attribute is the word size of the datapath. This also defines the word size of the multiplier. The larger the multiplier, the less memory accesses are needed. A second attribute is whether a single-port or dual-port memory is used. The advantage of the dual-port memory is that two memory accesses can be done at once.

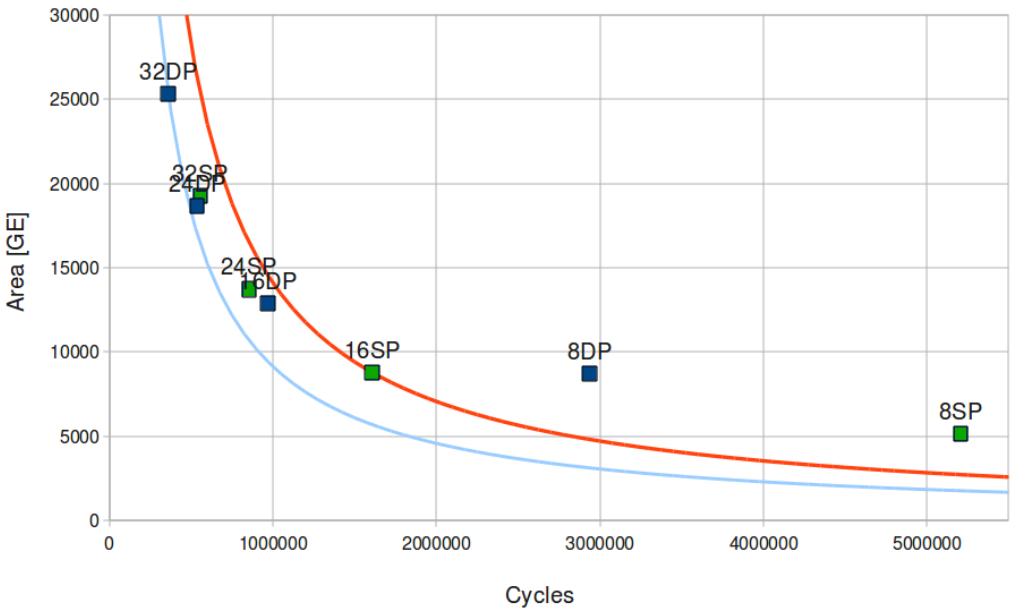

Figure 8.1: The number represents the word size; DP is dual-port; SP is single-port.

In Figure 8.1 only the size of the datapath is included in the area approximation. The size of the control logic (the program memory) is neglected. The two contour lines represent a constant area-cycle product. It has been assumed for this approximation, that the processor is used to calculate a NIST P-192 elliptic curve digital signature.

8.1 Size Approximation

In the following analysis, a 180 nm technology library from UMC (United Microelectronics Corporation) is used. Similar architectures are used for the single and the dual port

versions.

One of the two most important factors is the RAM. In many ECDSA designs (like [4]), this is the major building block. The size of a RAM can be highly reduced by using a macro block. The bit-per-area density is a lot larger using such a macro block. This macro block can be generated using a special tool provided by a manufacturer. For this comparison, the smallest provided, free implementations, by a tool from UMC[1], are used. Nine 192-bit registers are required. So, every RAM in this comparison stores 2048 bits.

The other major building block is the multiplier. It scales quadratically with the word size of the processor. The formula used for its approximation is given in Chapter 9.3.3.

The other components are mostly registers and multiplexers used for the CPU of the processor. For this approximation, a basic model of the datapath has been used. This model has been refined at a later design step and resulted in the model shown in Figure 9.2. This component scales linearly with the word size of the processor. Nevertheless it is the major component for some of the designs shown in Figure 8.2.

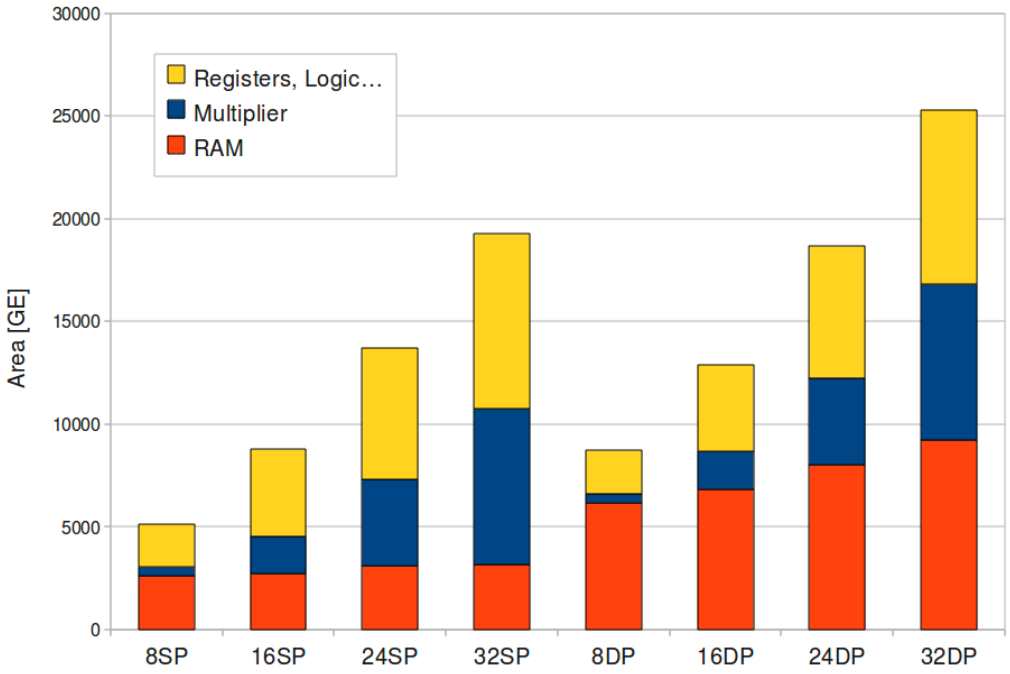

Figure 8.2: This chart shows the size distribution of the various design options.

8.2 Runtime Analysis

The major part of the runtime comes from the scalar multiplications used within the elliptic curve point multiplication. The algorithm by Auer is used for the point multiplication. It uses 3456 multiplications and 4224 additions or subtractions. As a result, the number of field additions, subtractions and multiplications is fixed. The runtime of those field operations is defined by the number of memory accesses they need. The number of words needed to represent a 192-bit number is t.

[1] 'UMC L180$FSA0A$ Memory Compiler: 200901.1.1' was used

In the case of an addition that uses a single-port RAM the memory accesses can be broken into the following pieces: $2t$ reads are required for to read the input registers and t writes are required to store the added intermediate result. Another t reads and t writes are required for the reduction. This results in the total $5t$ memory accesses needed in Table 8.2.

For the multiplication it is assumed that a multiply-accumulate unit is part of the design. So a fast product scanning multiplication algorithm can be used. This algorithm requires t^2 multiplications. Each multiplications needs two operands that must be read. That makes $2t^2$ read accesses in total. The other parameters are gathered using a sample implementation of a NIST P-192 field multiplications.

Table 8.2 shows the formulas used for the runtime approximation. The resulting numbers are shown in Table 8.1.

The inversion algorithm as well as any other algorithms used in an elliptic curve point multiplication are neglected for this approximation.

	t	Mult.	Addition	Total Cycles
8 SP	24	1360	120	5207040
16 SP	12	392	60	1608192
24 SP	8	198	40	853248
32 SP	6	124	30	555264
8 DP	24	732	96	2935296
16 DP	12	222	48	969984
24 DP	8	116	32	536064
32 DP	6	75	24	360576

Table 8.1: Resulting runtimes of the various implementations. The total runtime is $3456MUL + 4224ADD$. It is assumed that $C_{Add} = C_{Subtract}$ and $C_{Square} = C_{Multiply}$ with C being the number of cycles needed by the algorithm.

	Write	Read	Total Memory Accesses
Addition SP	$2t$	$3t$	$5t$
Addition DP	$2t$	$2t$	$4t$
Multiplication SP	$4t + t/3$	$2t^2 + 4t + t/3$	$2t^2 + 8t + 2t/3$
Multiplication DP	$4t + t/3$	$t^2 + 2t + t/6$	$t^2 + 6t + t/2$

Table 8.2: Table 8.1 has been generated using the formulas presented here.

8.3 Design Decision

The decision of the design was made using two guidelines: As fast as necessary and as small as possible. As a result, the 8-bit versions need to many cycles. So, the smallest design, which will be described in the following chapters, is a design with a 16-bit datapath and a single-port RAM.

Chapter 9

Processor

In the past years, many different implementations of elliptic curve processors and co-processors have been made. Some of them (like [1], [29] and [9]) concentrated on speeding up the elliptic curve operations. Others had the goal to minimize the total chip area required for their implementations.

A very interesting co-processor with small chip area has been proposed by Auer in [4]. The design consists of a dual-port memory a datapath and control logic. The dual-port memory is the largest component. It is made from a synthesized array of registers. The components used in the datapath are a multiply accumulate unit and several logic elements (adder, and, or, xor, ...). Most of those components are also used in processors.

Many smart card implementations have embedded processors. So his design would be used in addition to a processor.

So the idea was to use the components of Auer's design and build a dedicated processor. This processor should require a small chip area and in terms of speed, it should be able to compete with dedicated co-processor designs.

In this chapter all necessary elements for such a processor are described.

9.1 Architecture

The processor uses a standard Harvard architecture. That means that the program and the data memory are separated. Figure 9.1 shows the basic design of the processor.

The program memory is a table with 76 bits per entry. Each of those entries is used as control signals for the processor. It should be noted that these control signals are directly stored in program memory. Currently, the most space-efficient way of implementing the program memory is to let the synthesizer generate a look-up table using logic elements. For a more detailed comparison of program-memory implementations, see Section 10.3.1.

The data memory is split into different memory regions. The most important component is the RAM. It is used as working storage for the calculation of an elliptic curve signature. But this algorithm also needs constants. Those are stored in a designated ROM or look-up table. The third part of the memory region is used for any peripheral I/O module. Because their registers are memory mapped, no special commands for accessing a special I/O bus are required.

The used memory bus can also be used to upgrade the processor with some memory-mapped logic. Another way of extending the design is to adapt the datapath of the CPU. This approach is not recommended.

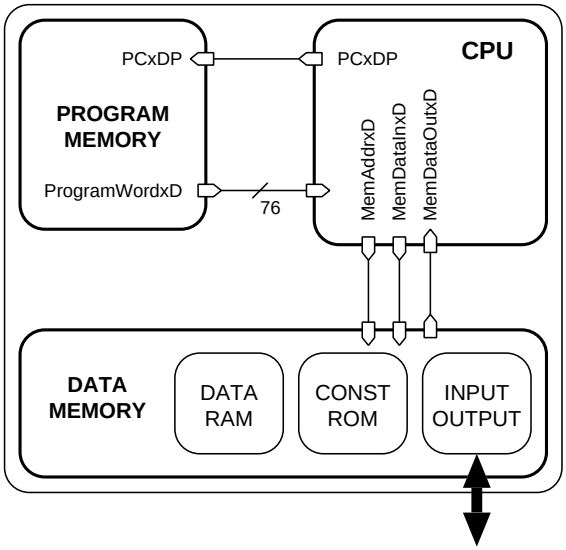

Figure 9.1: The program counter is used as index for the program memory. The resulting data is used as control vector for the processor. The memory read/write control signals are embedded within this control vector. Any I/O is placed within the data memory.

9.2 CPU

A central processing unit (CPU) is the primary element carrying out computer functions. Usually it has four very important tasks:

- It **fetches** a new instruction from the program memory. In this case, this is done by applying the program counter to the program memory.
- The program word gathered from the program memory is **decoded**. This means that dependent on an opcode, it is decided what kind of operation to perform. In the design showed in Figure 9.1 the program word is directly used as control vector for the CPU.
- The main task is to **execute** the designated operation. In this step, the different parts of the CPU are connected. As a result the desired operation is executed.

  In this phase, the two operands for the ALU are selected (see Figure 9.2) and the needed units (see Figure 9.3) are activated. At the end of this phase, new data is available for storage at the output of the ALU.
- The final step is to **write back** the new data. The designation can be any register or a memory address.

A CPU is a vast and complex machine. The following section tries to describe its parts, starting with the registers:

Program counter A program counter (PC) needs to be included in every processor. It marks the current position of the execution in the program memory. Usually the PC is always incremented by one. There are a few operations (Jump, Call, Branch, ...) that can modify the PC differently. In many implementations, dedicated hardware

is used to perform those operations. In this processor design, the new value for the PC is calculated using the ALU.

Stack pointer During execution of programs a stack can be used to store return addresses, function arguments or temporary values. The two basic operations are **PUSH** and **POP**. The **PUSH** command stores a value at the current position of the stack and decrements the stack pointer by one. The **POP** command increments the stack first and loads the current value from the stack.

For the elliptic curve algorithms, the stack is not very critical. That is why no special logic is connected to the stack pointer. Basically the SP is equal to the base pointers and can also be used as base pointer.

Base pointer The base pointer registers (BaseA - BaseC) are used for memory address generation. Field additions and multiplications need two arguments and a destination. 192-bit values can be represented by 12 memory entries (each 16 bits wide). Because $\lceil ld(12) \rceil = 4$, the possible offset for indirect memory accesses is 4 bits.

Accumulator A very important part of the whole design is the multiply accumulate unit embedded within the ALU. The result of a multiplication has 32 bits. For a NIST P-192 field multiplication 12 of those results are summed up, 36 bits are needed at least. This results in three accumulator registers (Acc0-2). Acc0 stores the least significant word. It is defined that every register is 16 bits in size. So a 48-bit accumulator is used. Other possible operations are: right shift the accumulator register by 16 bits and add/subtract a 16-bit value to/from the result.

State In this register, all status flags of the processor are stored. These status flags are: Carry, Zero, Overflow and Negative. Bit 4 is Negative xor Overflow. Bit 5 is the negated version of the Carry flag. For Subtractions, it can be thought of a Borrow flag.
This register cannot be written directly. Certain bits can be set/cleared by using **ADD** or **SUB** commands.

Work registers All the previously described registers have a special purpose. From the requirement of the ECDSA, four work registers (Work0-3) have been defined. These are the only registers that can be used as ALU operand A and B. The result of a multiplication can be stored within them.

Why four registers? One register is used to store the second operand of a **MULACC** command. Further two registers are needed as temporary storage during the field multiplication. So at least three registers are needed during a field multiplication. Nevertheless, using the four registers as cache greatly improved the runtime of the field multiplication at a small price.

Memory result There is one more 16-bit word. It is not a register but it can be used similarly as operand for the ALU. The *MemDataOutxD* data signal contains the result of the last memory access. It is updated by any new memory access.

This is a very important part of the design! There are two commands to read data from the memory. The **LDR** (Load to Register) command loads data from the memory (e.g. Data RAM) and stores it in a register. It needs two cycles for this operation. The **LD** (Load) command is a simplification of the **LDR** command.

It simply loads data from the memory. The data can be accessed and processed directly. This modification reduces the time needed for a memory operation from two cycles to one cycle.

The size of some of those registers could be reduced in some future designs. Not every register needs all 16 bits (PC, SP, Base, Acc2). For simplicity of the algorithm design, they are all 16 bits in the described implementation.

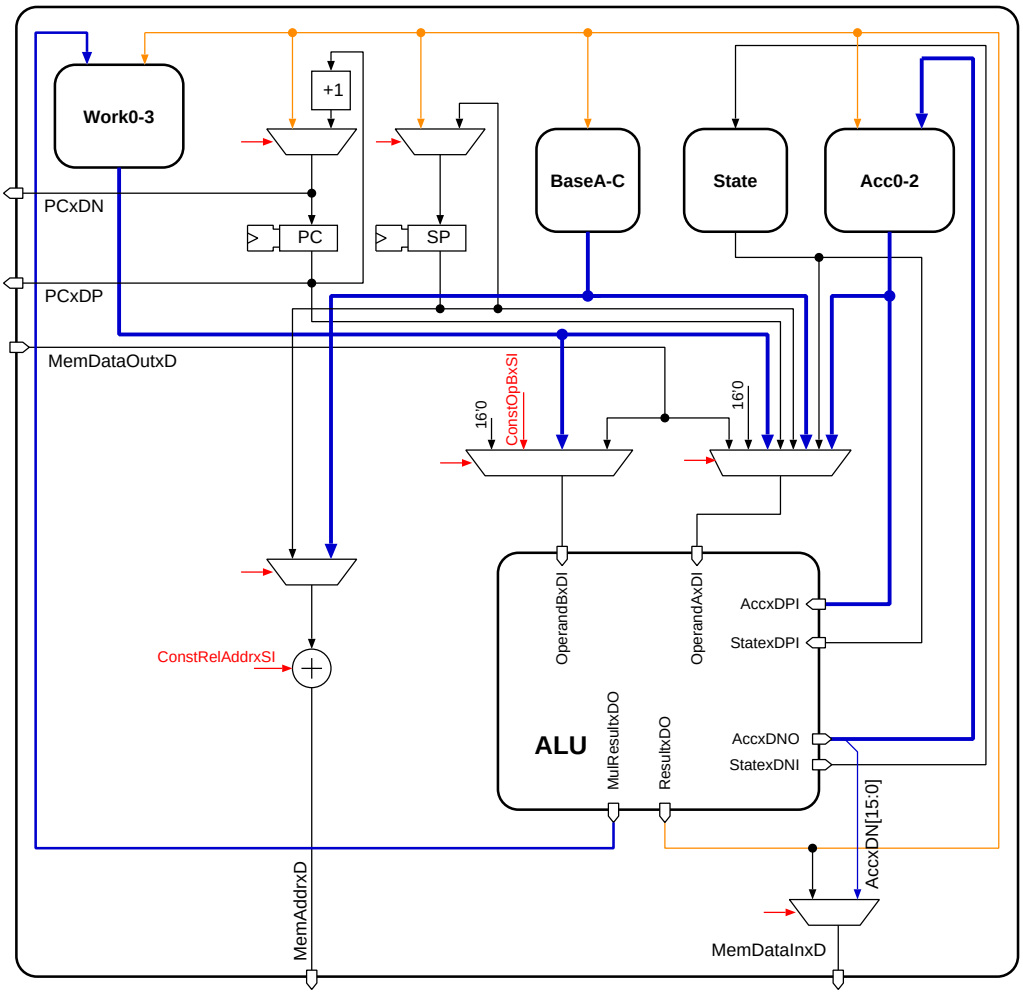

Figure 9.2: Design of the CPU. Two registers are selected with the two multiplexers and handled within the ALU. The results are stored within the memory or a register.

Apart from the registers the following facts are important to know about the design in Figure 9.2:

- Every register can be used as operand A. Additionally the memory result and zero can be used as operand A.
- Operand B is used for the work registers and a constant. This 16-bit constant is stored within the control vector. The load immediate (**LDI**) uses this 16-bit constants for the register modification. The memory result can also be used as operand B.

- For writing values to the memory, the *MemDataInxD* signal is used. There are two sources for this signal. The first source is the result of any ALU operation. The second is the next value for the Acc0 register. This speeds up the multiplication algorithm. The advantage of this design is that the result of an operation can be stored directly in the memory. This reduces the number of memory operations. The disadvantage is that it increases the critical-path delay. Especially the access time for the memory-mapped I/O is added to the time of the multiply-accumulate unit, which is always part of the critical path.

  Because the clocking frequencies of smart cards are very low, the advantage overweights the disadvantage. A solution for the timing problem is to either use operand A as *MemDataInxD* signal (bad), to not use the *AccxDN* signal as a source or to put a register between the *MemDataInxD* signal and the Memory Mapped I/O (better).

- The shifter for the accumulator is within the block labeled **Acc0-2**.

The most important logic element of the CPU is the ALU. This is described in the next section.

9.3 ALU

The two major inputs of the ALU are the operands A & B. In every cycle, they are added, ORed, XORed, ANDed and shifted. One of these intermediate results is selected and passed on to *ResultxDO*. It is defined that the subtrahend is always *OperandBxDI*, because this operand B is invertible.

The other important part of the ALU is the multiply-accumulate unit. The signal *EnMulxSI* is a good example for the use of operand isolation. Especially because the multiplier is the largest logic block in the whole design, this is a big energy safer. The 32-bit result of the multiplier can be used directly or added to the accumulator. The investigation of the P-256 multiplication shows the need to subtract values from the accumulator unit. The logic XOR is used to invert *OperandAxD*. As a result *OperandAxD* can be added or subtracted without the use of extra logic elements.

In the multiplication algorithms, the accumulator needs to be shifted by 16 bits. This is not done within the ALU. It is part of the register logic in the CPU.

9.3.1 Adder

A central component of each processor is its adder. Adding two values is simple. More tricky is the carry propagation. The handling of the carry flag is especially important, for handling big (e.g. 192-bit) numbers with a smaller (e.g. 16-bit) adder

An addition is defined as $(C, R) = A + B$. An addition with carry propagation is defined as $(C, R) = A + B + C_{old}$. It is very helpful to use an example.

Let us assume, we have a 2-bit adder unit. We want to add two 4-bit numbers. A is fixed to 1010b. B is varied. B_0 are the lower two bits of B. On a processor, this would be calculated with the following two operations:

ADD 10b, B_0
ADDC 10b, B_1
This example is shown in Table 9.1.

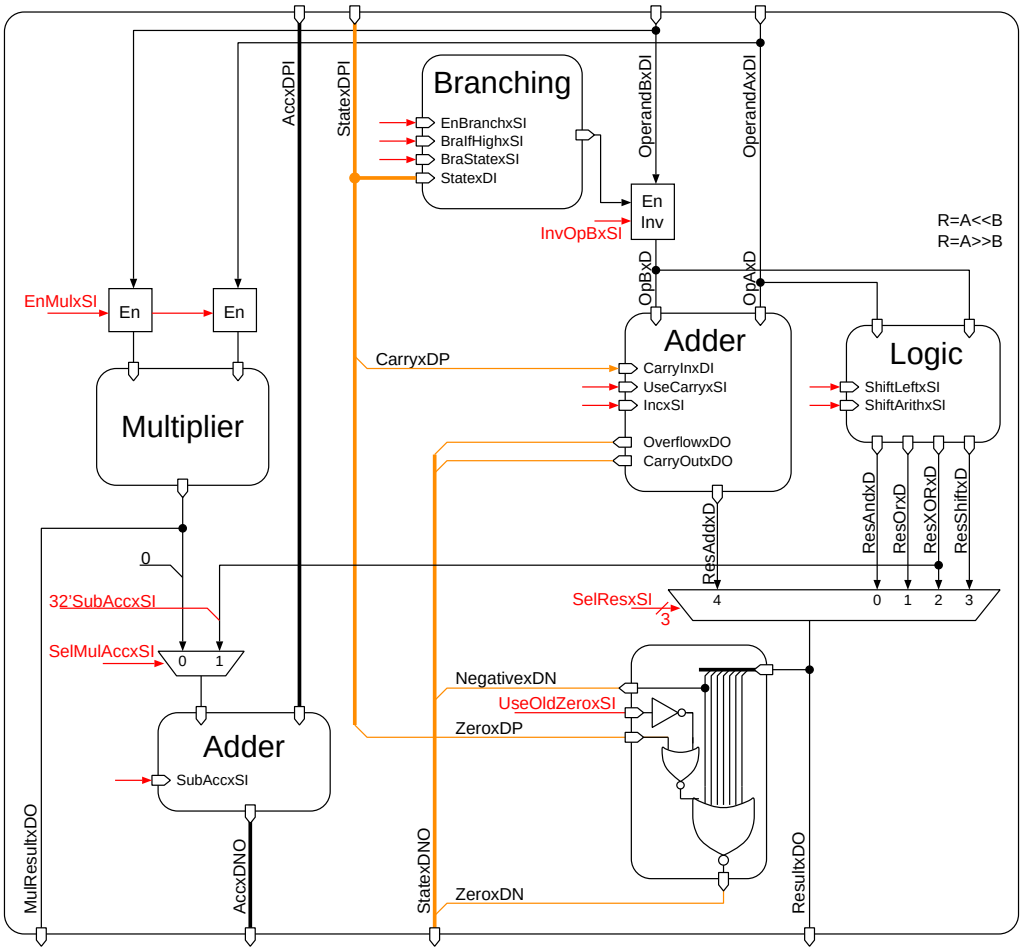

Figure 9.3: ALU that is capable of adding, subtracting, multiplying, accumulating, logic functions and branching.

The carry propagation for an addition is very well defined. But it is different with the carry propagation of a subtraction. There are two major interpretation of the carry flag for a subtraction.

One way is the reinterpret the carry flag as a borrow flag. A **subtract with borrow** operation is defined as $A - B - C$.

An other interpretation uses the advantage of two's complement representation, where $-B = not(B) + 1$ and $A - B$ can be computed as $A + not(B) + 1$.

Let us take a look at another example in Table 9.2. A **SUB** is followed by a **SUBC** command. The $-B = not(B) + 1$ representation is used. So the subtraction is defined as $(C, R) = A + not(B) + 1$ and subtraction with carry is defined as $(C, R) = A + not(B) + C_{old}$. Recognize that by inverting the carry flag, the borrow flag, mentioned above, is obtained. The most significant bit (MSB) of the result is the inverted carry flag.

Another very important operation is the comparison operation. It basically is the same as a subtraction but the result is not stored. For a comparison of two unsigned integers, the zero and the carry flag are needed. The zero flag is 1 if the result of the operation is equal to zero. The behavior of the zero flag after a **ADDC** or **SUBC** is a bit different. It is equal to 1 iff the old zero flag is 1 and the new result is equal to zero.

A	B	C	ADD	C	ADDC	Result
1010	0000	0	10	0	10	01010
1010	0001	0	11	0	10	01011
1010	0010	1	00	0	11	01100
1010	0011	1	01	0	11	01101
1010	0100	0	10	0	11	01110
1010	0101	0	11	0	11	01111
1010	0110	1	00	1	00	10000
1010	0111	1	01	1	00	10001

Table 9.1: Carry propagation in an addition.

A	B	not(B)	Z	C	SUB	Z	C	SUBC	Result
0101	0000	1111	0	1	01	0	1	01	00101
0101	0001	1110	1	1	00	0	1	01	00100
0101	0010	1101	0	0	11	0	1	00	00011
0101	0011	1100	0	0	10	0	1	00	00010
0101	0100	1011	0	1	01	0	1	00	00001
0101	0101	1010	1	1	00	1	1	00	00000
0101	0110	1001	0	0	11	0	0	11	11111
0101	0111	1000	0	0	10	0	0	11	11110
0101	1000	0111	0	1	01	0	0	11	11101
0101	1001	0110	1	1	00	0	0	11	11100
0101	1010	0101	0	0	11	0	0	10	11011
0101	1011	0100	0	0	10	0	0	10	11010
0101	1100	0011	0	1	01	0	0	10	11001
0101	1101	0010	1	1	00	0	0	10	11000
0101	1110	0001	0	0	11	0	0	01	10111
0101	1111	0000	0	0	10	0	0	01	10110

Table 9.2: An example for a subtraction of two values.

So if the difference is zero, the zero flag is 1 and operand A is equal to operand B. Additionally by using the carry flag it can be checked in which relation the two compared numbers are. See also Section 9.3.4 about branching.

The adder unit in Figure 9.4 supports adding and subtracting 4-bit integers. For subtracting, the $-B = not(B) + 1$ representation is used. The inverter for operand B is not shown. Table 9.3 shows the control signals during the **ADD**, **ADDC**, **SUB** and **SUBC** operations.

9.3.2 Barrel Shifter

Figure 9.5 shows a barrel shifter. This shifter is capable of the following operations: Left Shift, Right Shift, Arithmetic Right Shift.

Many other processors provide additional functionality, like rotating and storing the overflow in the carry flag. This functionality has been omitted in order to reduce the complexity and size of the design. These operations are mainly used during SHA-1 calculation

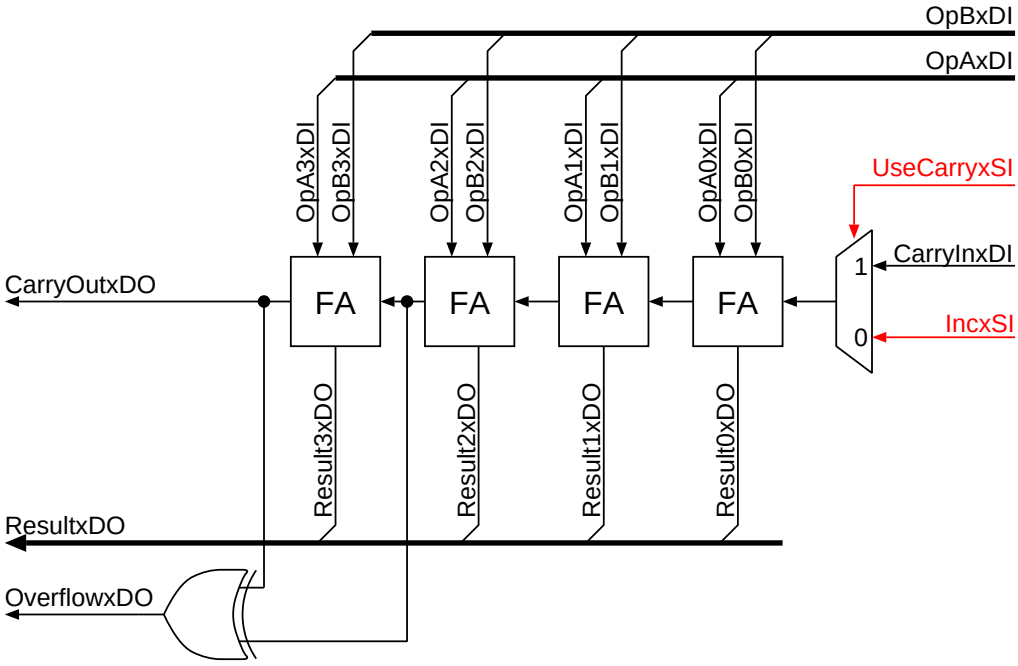

Figure 9.4: This adder unit is capable of adding and subtracting two integers (under the assumption that OperandB is invertible).

Operation	UseCarryxSI	IncxSI	InvOpBxSI
ADD	0	0	0
ADDC	1	0	0
SUB	0	1	1
SUBC	1	0	1

Table 9.3: This truth table needs to be fulfilled by the instructions. *InvOpBxSI* is used to invert Operand B.

and this algorithm is not very time critical. However, these use cases can be imitated with a combination of Left and Right Shifts.

By using the multiplier, the Barrel Shifter can be removed completely. As it is commonly known, a left shift with i is the same as a multiplication with 2^i. The advantage of a multiplication with 2^i is that it results in two shifts at once. The upper half contains a right shift with $N - i$ bits. N is the word size. The lower half of the multiplication result is equal to a left shift by i bits. If these two shifted results are ORed, the result is a rotation of the original value. Unfortunately, an Arithmetic Right Shift cannot be modeled that easily.

The Area S required by an N-bit barrel shifter can be approximated easily. $\log_2 N$ levels of N multiplexer are required to shift right and left.

$$A_{total} = A_{MUX3} \cdot N \cdot \log_2 N \tag{9.1}$$

If it should be possible to rotate, several multiplexer with three inputs need to be replaced by four input multiplexer.

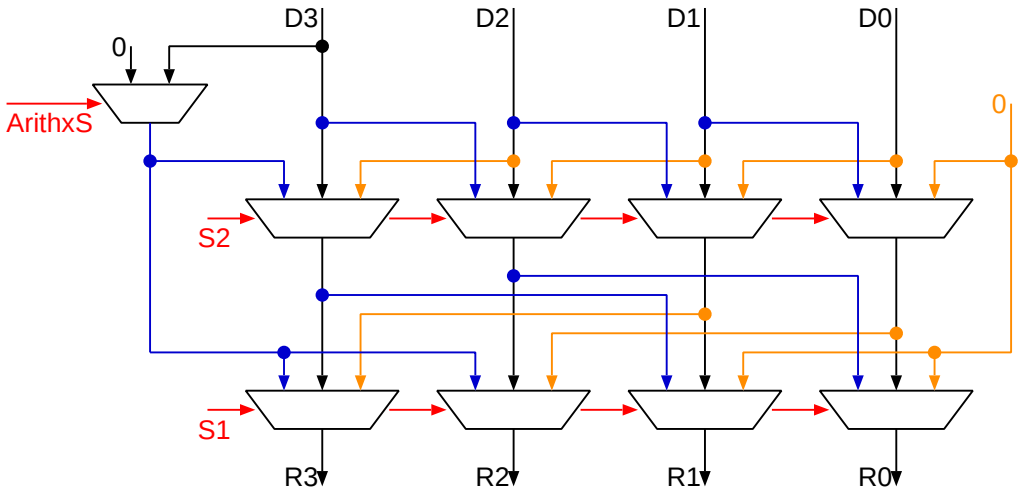

Figure 9.5: This block diagram shows an implementation of a barrel shifter.

9.3.3 Multiply Accumulate

The resulting output delay of a multiplier is by far larger than the delay of any of the previously presented modules. A multiplier usually consists of $N-1$ adders (N is the word size of the input operands). Each of these adders is N bits wide. By using logical ANDs as one bit multipliers, the adders are used to sum up all of those intermediate values. As a result, the total area requirement A is

$$A_{total} = A_{AND} \cdot N^2 + A_{FA} \cdot (N^2 - 2N) + A_{HA} \cdot N \tag{9.2}$$

The here presented formula is valid if carry-safe adders are used to add up the intermediate values. The advantage of this kind of adder is that it is very area efficient. The disadvantage is its speed. To speed up the multiplier, a different addition scheme should be used. Reto Zimmermann gives a very good overview of different addition schemes in [35]. Nowadays the synthesizer can choose a multiplier automatically, dependent on the required latency.

An accumulator is an adder placed after the multiplier. It is used to quickly sum up results generated by the multiplier. In order to sum up M results of a N-bit multiplier, a $(2N + \lceil \log_2 M \rceil)$ adder is required.

9.3.4 Branching

Status Flags

Using Table 9.2, some rules for comparing unsigned integer can be established.

$B < A$ Z xor $C = 1$ *
$B \leq A$ $C = 1$
$B = A$ $Z = 1$
$B \neq A$ $Z = 0$
$B \geq A$ Z xor $C = 0$ *
$B > A$ $C = 0$

Some comparisons (marked with *) does not need to be implemented. You only have to switch order of the parameters A and B, to test those cases.

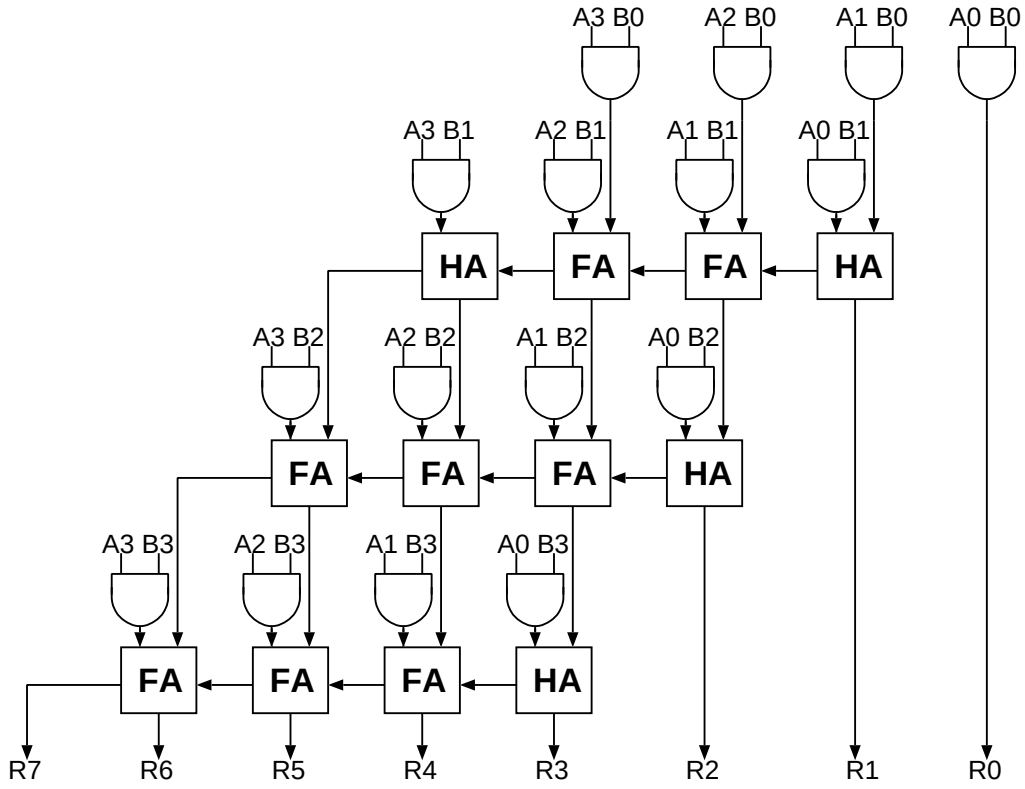

Figure 9.6: The ANDs are used as one-bit multipliers. The adders are used to sum up the intermediate results.

Usually, a processor also provides functionality to compare signed values. In order to do that, a combination of the negative and the overflow flags are needed.

Executing a Branch

Figure 9.7 shows a very efficient implementation of a branching logic. OperandA is the program counter. OperandB provides a relative jump address. The result is a new value for the program counter.

There are two possible cases:

Branch condition is false In this case OperandB is forced to zero. The result is $PC+1$. The one is added, by setting the increment input of the adder to one.

Branch condition is true OperandB is represented as a two's complement number. As a result $PC + OperandB + 1$ can also result into a jump to a lower address.

The beauty of this implementation is that the jump is always executed in exactly one cycle. This can only be achieved, because there is no pipeline used in the whole design.

9.4 Reusable Processor Design

Elliptic curve cryptography is a complex topic. Many of the used algorithms are still under research. Every year new attacks get published that make certain algorithms vulnerable

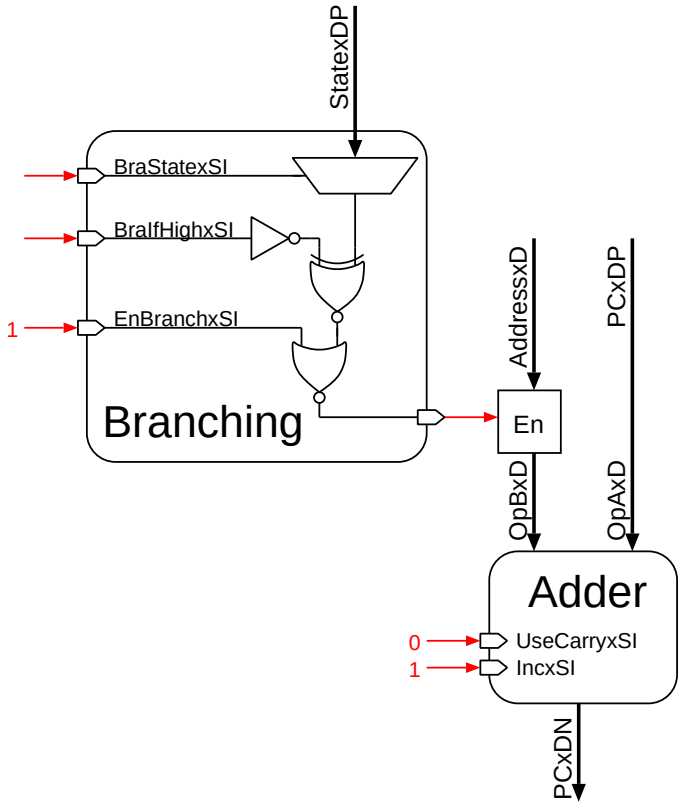

Figure 9.7: Dependent on the branch logic, either 1 or $OperandB + 1$ is added to the program counter.

and many implementations unusable. So there is a good chance that an ECC design that is good now, cannot be used in several months or years.

A good example is Hein's master thesis sommarized in the paper [13]. All the algorithms are correct but an attack by Hutter et al. in [16] has made his design vulnerable. The conclusion in the case of this master thesis has been to make the manufactured chip reusable. With the goal of a platform that can be used to evaluate different algorithms the following changes has been made to the design:

- The constants and program memory are replaced by RAMs. The input of the program counter register is used as program RAM index.
- A 16-bit instruction set is used.
- A bootloader to initialize and test the RAMs has been added.
- Different peripherals (e.g. I/O interfaces) have been added.

Ideally, the program memory should be replaced by a Flash or an EEPROM. Unfortunately neither of these non-volatile memories are available in the used 180 nm technology from UMC. One possibility is to add an external memory which is used for the design. The drawbacks of this solution are the resulting disastrous performance and big possibility to make errors. Such a design should be tested on a FPGA first.

So it has been decided to use a RAM as program memory. This RAM must be initialized. There are several ways to do this. A bootloader is used in this thesis. This

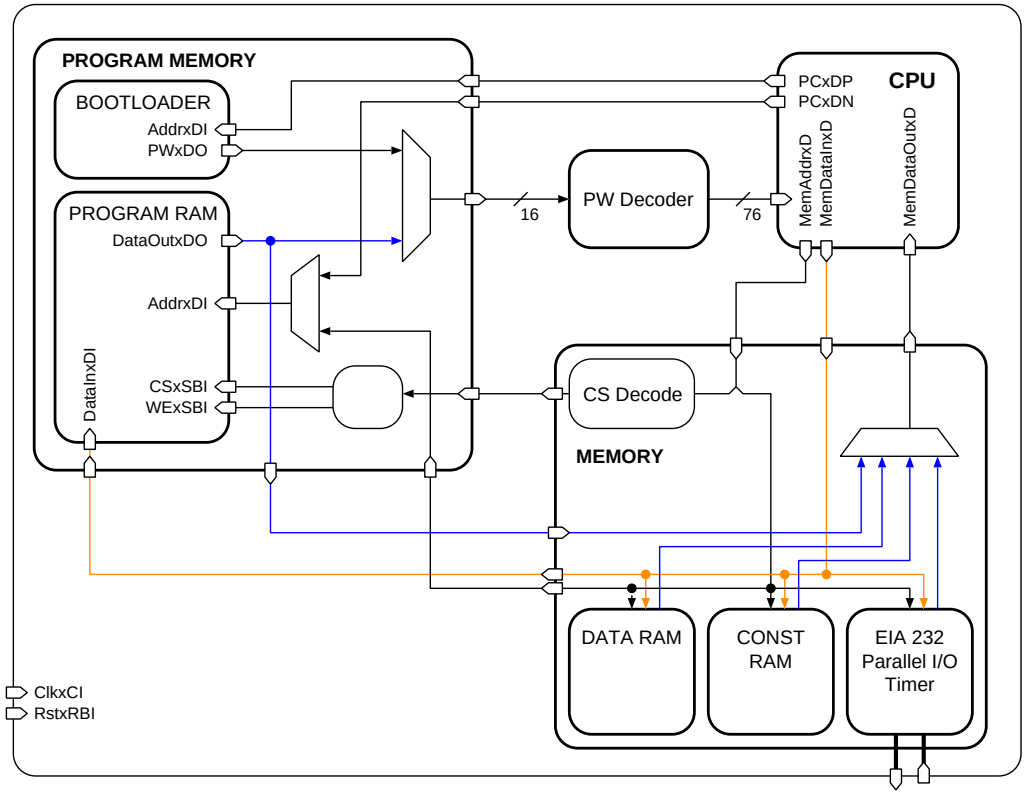

Figure 9.8: This block diagram represents the manufactured chip *Neptun*.

bootloader is a fixed program, stored as a look-up table. The bootloader is started after the power-up of the processor. It uses two possible sources (a EIA-232 interface and a SPI) to initialize the RAMs.

Storing a 76-bit control vector in the program memory is not possible any more. The place provided by a 'MiniAsic' setup is to small for such a huge program RAM. A program RAM with a width of 76 bits and about 3500 entries (needed by the signature algorithm) is too large for the MiniAsic setup available. To use the available space efficiently, a 16-bit instruction set is used.

The further enhancements of the design are described in the following sections.

9.4.1 Instruction Set

The size of each Program Memory Entry has been reduced from 76 to 16 bits. This procedure is also described in Section 7.4.

This is possible because most control signals are unused within certain instructions. As an example: During an addition the direction and result of a logic shift is not important at all. The most important part of this process is that the speed of the algorithms stays the same. Nearly as important is the re-usability of the instruction set.

Also the used instruction set is shown in Section 7.4. For now, let us take a look at a summary of the 40 instructions:

- 22 Arithmetic Instructions (Add, Subtract, Multiply)
- 12 Instructions with Memory Access

- 11 'Parallel' Instructions

9.4.2 Test Strategy

The testing of the chip is split into two phases. In the first phase, the chip is tested using a scan chain. All test vectors used in this phase can be generated automatically. These test vectors can be used by a chip tester to test all registers and logic of the design. During this phase the RAMs are 'block isolated'. This means that the inputs of the RAMs are artificially made observable and the outputs are made controllable. The RAMs itself must be tested separately.

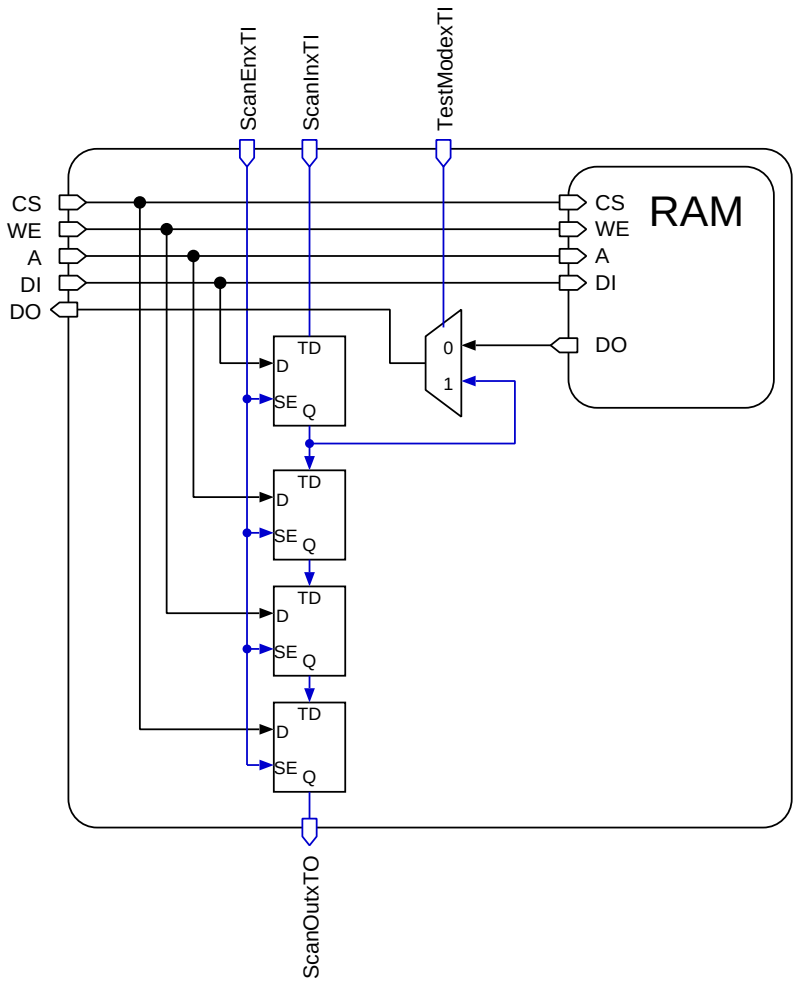

Figure 9.9: Block-isolation of a one-bit RAM.

Block isolation is best described by looking at Figure 9.9. During normal operation *TestModexTI* is 0. The inputs are directly connected to the RAM and the output 'DO' is passed by the multiplexer.

During Testing, *TestModexTI* is 1. In this case the added registers are important. If *ScanEnxTI* is 0, the input values are loaded into the scanable registers. If *ScanEnxTI* is 1, the state stored in any register is forwarded to the next register in the scan chain.

Because *TestModexTI* is 1, the output of the block is controlled by the contents of the

data input registers. Because these registers are modifiable by the scan chain, the output is controllable.

One drawback of block isolation is the area overhead. Additionally the scan registers change the state at every cycle. It is recommended to use a method like operand isolation or clock gating. This would reduce the power consumption during normal operation. In the final implementation of Neptun, an AND gate is placed at the inputs of the registers. As second input, the *TestModexTI* signal is used.

For RAM testing the bootloader is used. By sending a certain command to the bootloader, the RAMs are tested automatically. At first, every entry of the RAM is written with a certain value. Secondly all stored values are read and checked. This is done three times. Two predefined constants (AAAAh, 5555h) are used for the first two test cycles. For the third test cycle, a 'running one' is used. This means that the test pattern changes for each RAM address. The initial pattern is 0001h, followed by 0002h, 0004h, 0008h, ... The previous pattern is always rotated by one bit.

9.4.3 Bootloader

The bootloader is the startup point of every boot process. The detailed description is given in the attachment. For now, let us look at the use cases:

RAM testing. By sending a certain command, the RAMs are tested. The answer message tells the user, if this self test was successful.

RAM initialization. Before actually executing a program, the RAMs need to be initialized. This can be done in two ways. One is to send a certain message via the RS232 control interface. An other way is to configure the bootloader to read an SPI flash and store its contents in the RAMs.

Starting the program. This can be done by a jump or a call to the designated program address. In the case of a call, the bootloader is prepared for the case that the program returns.

9.4.4 I/O Interfaces

The most important rule for memory mapped I/O is to separate control, status and data register. Memory mapping means that (nearly) all registers used in the peripherals are accessible via a unique memory address. Once more, a detailed description is given in the Appendix.

Parallel I/O. This simplest peripheral module connects 32 registers directly to the pins of the chip. 16 pins are used as inputs. 16 pins are used as outputs. Some of the pins have a predefined use, defined by the bootloader.

EIA-232. This asynchronous serial interface only provides a receive and a transmit pin. The most important features are the double buffered receive and transmit bytes and the freely configurable clock divider[1].

[1] Actually modifying the clock signal raises many problems. As an example, it complicates the testing of the chip. Instead the clock signal is counted, and every N cycles, the logic is enabled.

Timer/Counter. In order to be prepared for future protocols (e.g. ISO-14443), three timers can be used. These timer are capable of using an input trigger signal. Further, they can output a signal based on the comparison of the counter register with two comparator registers. Because of that, they can also generate pulse width modulated (PWM) signals.

Chapter 10

Results

The previous chapters do not only describe a theoretical construct of a processor. The processor has been implemented and is able to compute the elliptic curve digital signature generation and verification algorithms. In Section 10.1 the runtime of the used algorithms is investigated. It is important to know that the used algorithms are currently 'state of the art'. For this runtime analysis the Neptun processor is used. This very flexible processor is discussed in Section 10.2.

Section 10.3 discusses the area requirement of two stripped implementations. Only the most important parts of the processor are used in those implementation. Any I/O or unused memory cells are removed from the design.

Because no design is perfect in terms of area and runtime requirements, the results can still be improved (see Section 10.4) and should be compared with other designs (see Section 10.5).

10.1 Algorithm Analysis

All algorithms analyzed in this section use the 16-bit instruction set. Please keep in mind that using the 76-bit control vector look-up table results in a faster execution time. This is because several operations in the 16-bit instruction set are limited and multiple cycles are used to perform them.

The algorithms described in the next two sections are designed so that the size (number of code lines) is minimal and the runtime is as fast as possible.

10.1.1 Signature Generation

For the elliptic curve signature generation, introduced in Algorithm 4, a lot of different algorithms are needed. Some statistics about their runtime is given in Table 10.1. The number of function calls can be seen as importance factor of the algorithms.

Those algorithms are also described in here:

P192 field algorithms. These are the most optimized algorithms. The execution time is hardly improvable. The multiplication algorithm's original runtime is 997 cycles. After the introduction of the parallelized instruction set it is 401 cycles. By using the work registers as caches and consequently avoid memory accesses, the runtime of the algorithm has been reduced to 328 cycles.

Utility functions. These (specially optimized) functions are mainly used by the high-level functions and the *Montgomery inversion* algorithm.

Montgomery multiplication. This function has been optimized for speed. Because of its minor importance, it should be reprogrammed to reduce its size. This is especially important because of the large program memory synthetization result.

Montgomery inversion. This algorithm makes use of the utility functions. That is the reason for its small size and its fairly small execution time. The biggest part of the inversion algorithm is spent in the utility functions. That is the reason for its minor influence in Table 10.1 and its 8.8% of the execution time in Figure 10.1.

The algorithm is used twice: Once for the recovery of the point coordinate x which is standard projected and a second time for the inversion of k.

SHA-1 algorithm. This algorithm is used for the calculation of the SHA-1 hash function.

Point multiplication. The main job this algorithm is to call the P-192 field algorithms in the correct order. The main source for its long execution time are the **CALL** commands.

ECDSA signature generation. This function is rather small and simple. Its main duty is to call all the other algorithms.

	Function Calls	Code Lines	Cycles	
P192.Multiplication	3438	328	1127664	67.97%
P192.Add	2865	64	183360	11.05%
P192.Subtract	1337	65	86905	5.24%
PointOperation.Multiplication	1	388	64222	3.87%
Utilities.Div2	535	99	52965	3.19%
Montgomery.Inversion	2	215	39811	2.40%
Utilities.ADD	810	38	30780	1.86%
SHA1.Block	2	371	27510	1.66%
Utilities.Copy	769	26	19994	1.21%
Utilities.CMP	278	38	10564	0.64%
Utilities.SUB	275	38	10450	0.63%
Montgomery.Multiplication	6	656	3900	0.24%
Utilities.CopyExt	1	12	324	0.02%
SHA1.FinalBlock	1	60	207	0.01%
ECDSA.Sign	1	122	108	0.01%
Utilities.Clear	7	15	105	0.01%
SHA1.SetIndex	3	18	50	0.00%
SHA1.Init	1	32	32	0.00%
SUM	10332	2585	1658951	100.00%

Table 10.1: Implemented functions used for generating a signature. 'Function Calls' is the number of times, the function is called.

Most of the instructions displayed in Table 10.2 are rather unimportant. They are needed within the algorithms but rather unimportant in terms of execution time. The top five commands have an affiliation with the memory. They either load or store data. The

	Cycles	Used	Total Cycles	
MULACC_LD	1	386	458772	27.65%
MOV_LD	1	421	360666	21.74%
LD	1	177	284790	17.17%
ADDC_ST	1	57	154452	9.31%
ADDACC_ST	1	14	48132	2.90%
CALL	4	93	41324	2.49%
LDR	2	116	38846	2.34%
MULACC_ST	1	22	37884	2.28%
SUBC_ST	1	33	32439	1.96%
MOVNF	1	87	21704	1.31%
RET	2	19	20662	1.25%
LDSI	1	135	20425	1.23%
STR	1	129	18949	1.14%
ADDC	1	16	17540	1.06%
ADD_ST	1	6	16854	1.02%
ADDACC_LD	1	4	13752	0.83%
BRA	1	31	10731	0.65%
CMPC	1	22	8965	0.54%
RSI	1	25	8852	0.53%
LSI	1	26	7864	0.47%
OR	1	25	7559	0.46%
SUBI	1	10	3770	0.23%
MOV	1	38	3153	0.19%
SUB_ST	1	3	2949	0.18%
JMP	2	14	2940	0.18%
ADD	1	24	2125	0.13%
CMP	1	5	1754	0.11%
AND	1	17	1697	0.10%
POP	2	9	1544	0.09%
PUSH	2	9	1544	0.09%
LDI	2	35	1460	0.09%
ADDI	1	16	1284	0.08%
CMPI	1	16	1141	0.07%
XOR	1	15	1091	0.07%
SUB	1	6	578	0.03%
RS	1	2	192	0.01%
ASRI	1	1	191	0.01%
MULACC	1	24	144	0.01%
MVN	1	2	80	0.00%
RSACC	1	14	78	0.00%
MUL	1	12	72	0.00%
LS	1	1	2	0.00%
SUM		2117	1658951	100.00%

Table 10.2: Commands used for generating a signature. The total number of commands does not represent the total number of program memory entries. E.g. **CALL** needs four entries per instance.

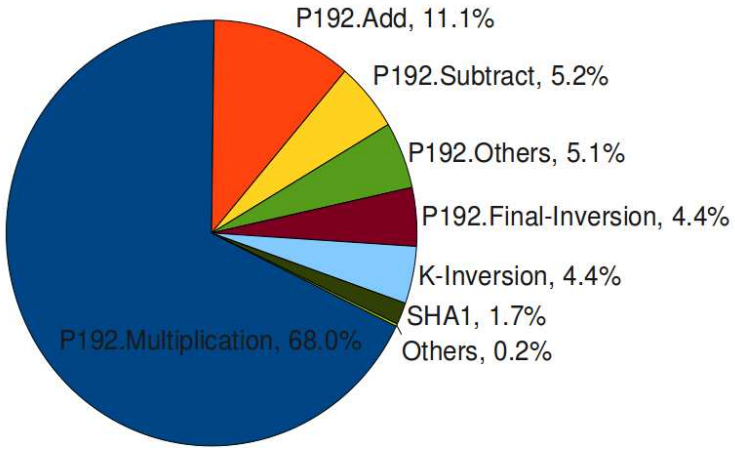

Figure 10.1: The total execution time of 1658951 cycles split into its main sources. It is obvious that the multiplication still (even after all the optimizations) is the main source for the long execution time.

MOV_LD command is used for storing the previously loaded data into a register and loading some new data.

By investigating this table, the biggest overhead of the implemented processor versus a dedicated co-processor design becomes obvious. The **CALL** and **RET** commands waste 61986 cycles. This is an overhead, you would not have in a dedicated hardware.

The **LDR** command is a sign for unoptimized code. 38846 cycles are still subject for optimization.

A limitation of the 16-bit instruction set versus the 76-bit control signals is the limited use of shift commands. It is not possible to store the result in a register which is not the source register. Consequently a **MOVNF** (Move without update of the status flags) command needs to be used beforehand. This is a waste of 21704 execution cycles.

10.1.2 Signature Verification

Similar to the signature generation algorithm, presented in the previous section, a statistic can be made for the functions used during the elliptic curve signature verification. This statistic is given in Table 10.3. The elliptic curve signature verification algorithm is shown in Algorithm 5.

Extending the implemented signature generation algorithm is not very hard. With a few extra functions, a signature verification algorithm is implemented. The changes are:

PointOperation.MultiplicationExtended This multiplication method can perform a point multiplication with a random base point. Similar to the *PointOperation.Multiplication* function, this function uses a Montgomery ladder which only uses x-coordinates. The biggest extension is the manual, initial point doubling. For this algorithm, the x and y-coordinates of the point are needed. The actual increase in runtime is marginal.

PointOperation.YRecovery Because the multiplication schemes use an x, z representation of the points, an YRecovery function is necessary. For an universal Point Addition, the two points must be in a (x, y, z)-representation.

	Function Calls	Code Lines	Cycles	
P192.Multiplication	6843	328	2244504	71.72%
P192.Add	5686	64	363904	11.63%
P192.Subtract	2663	65	173095	5.53%
PointOperation.Multiplication	1	388	63900	2.04%
PointOperation.Mult.Extended	1	485	63166	2.02%
Utilities.Div2	540	99	53460	1.71%
Montgomery.Inversion	2	215	40067	1.28%
Utilities.Copy	1531	26	39806	1.27%
Utilities.ADD	816	38	31008	0.99%
SHA1.Block	2	371	27510	0.88%
Utilities.CMP	280	38	10640	0.34%
Utilities.SUB	277	38	10526	0.34%
Utilities.InitRam	1	13	3572	0.11%
Montgomery.Multiplication	5	656	3256	0.10%
Utilities.CopyExt	1	12	324	0.01%
PointOperation.YRecovery	2	150	300	0.01%
ECDSA.Verify	1	296	292	0.01%
SHA1.FinalBlock	1	60	207	0.01%
Utilities.Clear	7	15	105	0.00%
SHA1.SetIndex	3	18	50	0.00%
SHA1.Init	1	32	32	0.00%
SUM	18664	3407	3129724	100.00%

Table 10.3: Functions used for a Signature Verification.

Point addition. This algorithm is part of the Verification Algorithm. Because of the standard projection of the input points, this is done very fast.

Montgomery inverse. Because of the use of projected points throughout the algorithm, only two inversion algorithms are necessary. One inversion is used for the calculation of s^{-1} and one to reverse the standard projection.

93.8% of the resulting total runtime is caused by the two point multiplications. Further 4.7% are caused by the inversion algorithms. The other 1.5% are distributed on the other parts of the algorithm.

10.2 Area Analysis of Neptun

The chip used for the analysis of the runtime is called Neptun. Compared to a design, which is optimized for a minimal area requirement, Neptun is different. This chip, which is currently in production, has a lot more features and is more reusable:

- The CPU is freely programmable.
- The evaluation of different algorithms is possible.
- Different set of constants can be used.
- The RAMs are sized in a manner so that larger NIST fields can be implemented and evaluated.
- The bootloader can be used to monitor the RAMs after a program is executed. The bootloader also provides a default interface for an operator.
- A serial I/O is used for the communication with a host PC.
- Three timers can be used to simulate any protocol
- 16 inputs and 16 outputs can be programmed.

The relative area requirements for Neptun are shown in Table 10.4. In order to generate this summary, the synthesis results are used. The area requirements for pads and power rings are ignored.

The largest parts of the design are the RAMs. They need more than 85% of the available chip area. The CPU covers only less than 7% of the design. It needs an area of 6089 gate equivalents. The bootloader is a synthesized 16-bit look-up table with 649 entries. The rest of the design is filled with peripherals. The smallest peripheral is the parallel I/O. It consists of two 16-bit registers.

The layout of Neptun is shown in Figure 10.2. A big part of the required area is used by the pads and the power rings. Whereas the outer power rings were already predefined it was possible to configure the inner power rings manually. The outer rings are two rings for the core supply voltage with two additional rings on the metal layers underneath. A net of supply lines has been placed on top of the program RAM because initial power simulations have shown a very large voltage drop at the bottom of the RAM. That is caused by the relatively large power requirement of the power RAM. The actual synthesized logic is placed between the RAM macros. To decrease the resistance of the horizontal power lines vertical bars are placed. They are connected to the power rings of the RAM macros.

	Area [μm^2]	Area [GE]	
Program RAM	559173.44	59648.98	67.70%
Constant RAM	74168.17	7911.78	8.98%
Data RAM	74168.17	7911.78	8.98%
CPU	57084.98	6089.45	6.91%
CPU (ALU)	30876.04	3293.66	3.74%
Bootloader	13130.42	1400.67	1.59%
EIA 232	12739.84	1359	1.54%
Timer 0	8152.59	869.67	0.99%
Timer 1	8152.59	869.67	0.99%
Timer 2	8127.59	867	0.98%
Instruction Decoder	3518.53	375.33	0.43%
Parallel I/O	2840.44	303	0.34%
Total	825974.56	88109.59	100.00%

Table 10.4: The area requirements of the Neptun ECC Processor.

10.3 Area Analysis of Low-Area Designs

There are two use cases to consider for the presented processor. The *Neptun* processor is designed for flexibility and can be used to evaluate algorithms. The area requirement is not of any concern. The second use case is a specialized implementation which is designated for an RFID tag or smart card. A design for such a device should have a low power consumption and be as cheap as possible. The price usually is proportional to the area requirement of a design. So it is important to keep the area requirement as low as possible.

The two designs presented here have been removed from any extra logic or flexibility. They are just used to generate or verify a signature. They have been synthesized but no power simulation has been performed.

Part	Signing	Verifying	Difference
Execution Time [Cycles]:			
Cycles:	1658951	3129724	+88.7%
Number of Entries:			
RAM	112	152	+35.7%
Program Memory	2487	3171	+27.5%
Constants	105	151	+43.8%
Area Requirement [Gate Equivalents]:			
Program Memory	5861	6991	+19.3%
CPU	5112	5112	-
RAM	2553	2988	+17.0%
Other	704	850	+20.7%
Total	14230	15941	+12.0%

Table 10.5: Comparison of two processor implementations. One is used for signature generation, the other one for verifying a generated signature.

After investigating Table 10.5 it becomes obvious that the verification algorithm needs more resources than the signature generation algorithm. But in terms of area, the overhead is only 12%. Because the CPU is the same in both designs, the only parts with larger

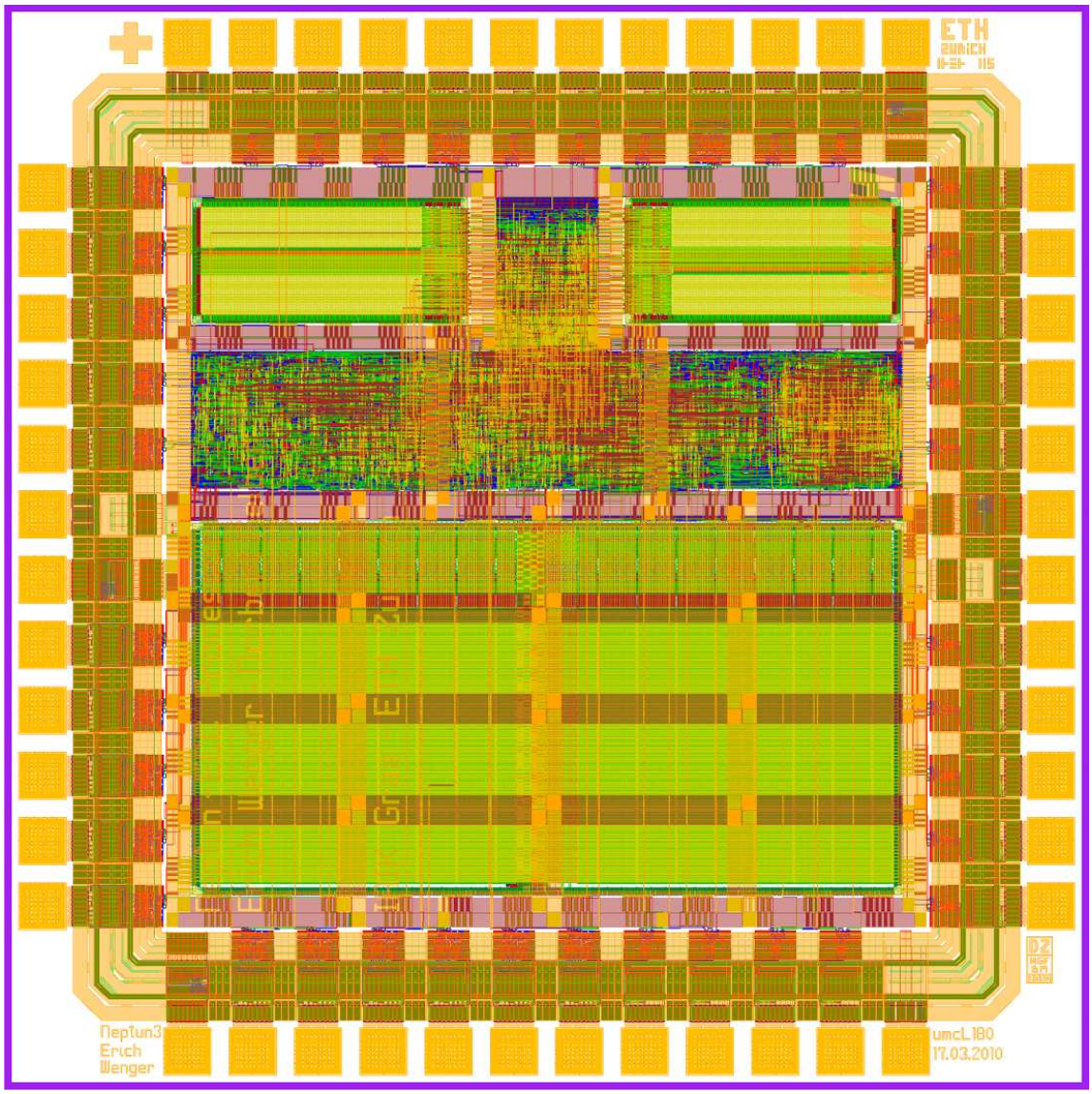

Figure 10.2: The final layout of the chip: Neptun.

requirements are the memories.

The largest part of the designs is the program memory. That is why it has to be inspected in detail. This is done in the next section.

10.3.1 Program Memory Implementation

The program memory is used to store all instructions. The program counter is applied as index for this collection of instructions. After some delay (which is secondary), the program memory delivers an instruction word that is executed by the CPU. It is the largest component of the processor. That is a good reason to investigate in different implementations:

76-bit synchronized look-up table. Suppose the the program has 3000 entries. Then the task of the synthesizer is it to optimize this huge table. A first approximation

with 0.1 Gate Equivalent per bit (22800 GE in total), seems to be terribly wrong. Figure 10.4 shows that only 0.03 Gate Equivalents per bit are necessary.

16-bit look-up table with instruction decoder. The table also needs more entries, because certain commands (**LDI**, **CALL**, **JMP**, ...) need more cycles in the compressed instruction set. The total size of the table can be reduced from 76x3100 bits (=235kbits) to 16x3400 (=54kbit). Unfortunately, the total size required for this implementation increased. To use a table with less than 16-bit entries is not possible. That is the minimum size of an instruction word, required by the **LDI** command.

16-bit ROM with instruction decoder. The idea is to replace the synthesized 16-bit look-up table with a more space efficient ROM. Obviously the sizes of the available ROMs are too large. So it was not even necessary to perform such a synthetization. The size of the program word decoder is 345 Gate Equivalents.

There is also a fourth implementation. It is a 76-bit ROM. Unfortunately, this ROM is huge. So it is not considered for this comparison.

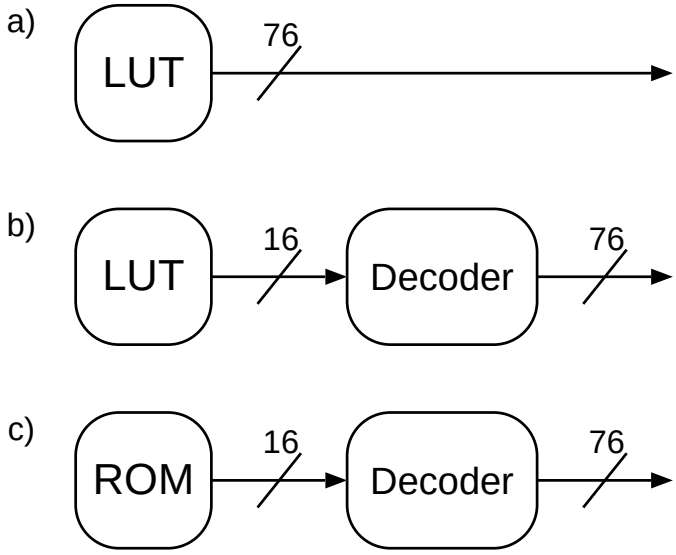

Figure 10.3: Different implementations of the program memory. a) 76-bit synthesized look-up table. b) 16-bit synthesized look-up table with instruction decoder. c) 16-bit ROM with instruction decoder.

Figure 10.4 shows the results of two different programs. The smaller program is the signature and the larger program (more entries) the verification algorithm. For comparison reasons, the sizes of the smallest free 16-bit ROMs[1] are given.

Obviously, the smallest implementation is the 76-bit synchronized look-up table. This implementation is used as program memory for the smallest possible implementation.

[1]The smallest free ROM generated by 'UMC L180$FSA0A$ Memory Compiler: 200901.1.1' is used.

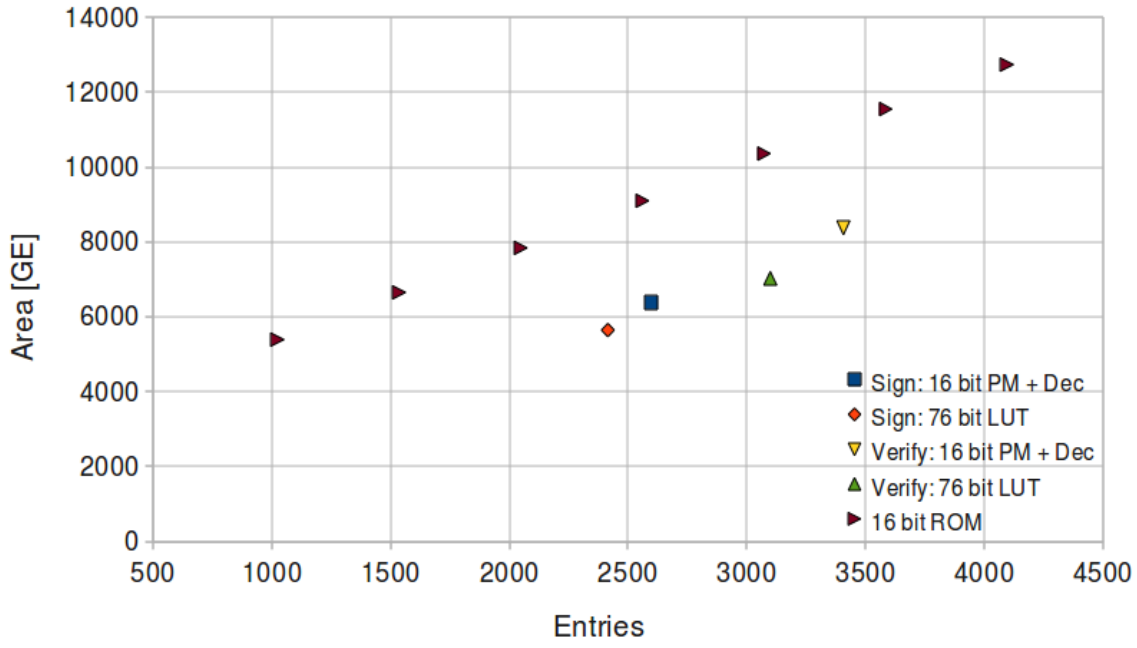

Figure 10.4: Different synthetization results compared. The ROM implementation is without the instruction decoder. The number of entries differ because various commands (e.g. **LDI**) need extra entries in the 16-bit version.

10.4 Possible Improvement

Even though months are spent implementing and optimizing the algorithms, it is always possible to make them 'better' and trim them a bit.

- Examine all implemented algorithms and optimize them according to their importance. Especially focus on large function with a low number of total cycles (e.g. *Montgomery.Multiply*. This should result in a reduction of necessary program memory entries.
- Currently, every multi-cycle instruction (e.g. **CALL**) is split up into several program words. By modifying the instruction decoder less program memory entries are needed. This should reduce the size of the program memory.
- For multiplication algorithms, a more powerful **MULACC** command is useful. A **MULACC** that automatically in/decrements a base register and executes this sequence several times should decrease the number of entries in the program memory.
- In order to reduce the size of the ALU, the 16-bit shifter can be rationalized. The multiplier can be used as a shifter instead.
- The size of several registers (Base*, Acc2) can be decreased.
- Unify the two separate Point Multiplication algorithms used within the ECDSA Signature Verification Algorithm. This reduces the number of necessary program memory entries for the signature verification design.

- Use a larger RAM and a faster point-multiplication scheme. This will speedup the execution time of the point multiplication algorithm and result in a better Area-Cycle product.
- Implement a squaring algorithm. Hankerson et al. suggests in [12] that the squaring algorithm is up to twice as fast as a multiplication. For that, a change in hardware is needed.

10.5 Comparison with other Work

Before it is possible to compare this work with work of predecessors we need to introduce some distinctions. They are all area optimized ECDSA designs. Some of them calculate a signature using binary extension fields $GF(2^m)$. The big advantage of those designs are the simpler field-multiplication schemes because binary extension fields do not have the need of carry propagation.

On most smart cards, a processor is part of the design. Most of the designs by competitors are co-processor. These processors are used to speed up the ECDSA calculation. The Neptun processor already is a processor. There is no need for a co-processor or any additional processor. This is an advantage, because smart cards usually contain processors. This bigger picture (the (co-)processors in a smart card) should be considered for evaluation. A summary of many previous designs is given in Table 10.6. It is important to note that some of those designs do not calculate an ECDSA signature. Some of them are only used to do point multiplications.

	Area [GE]	Cycles [kCycles]	ECC Curve	VLSI technology	Processor
Kumar 2006 [26]	15094	430	B-163	AMI C35	NO
Hein 2008 [13]	13685	306	B-163	UMC L180	NO
Yong Ki Lee 2008 [27]	12506	276	B-163	UMC L130	YES
Leinweber 2009 [28]	8756	191	B-163	IBM L130	YES
Auer 2009 [4]	24750	1031	P-192	AMS C35	NO
Fürbass 2007 [8]	23656	500	P-192	AMS C35	NO
Neptun 2010	**14230**	**1659**	**P-192**	**UMC L180**	**YES**

Table 10.6: Comparison of implementation with related work.

Obviously, in terms of speed the Neptun processor cannot compete with the $GF(2^{163})$ implementations. In terms of area, Neptun is comparable with the three oldest $GF(2^{163})$ implementations. Leinweber [28], currently has the smallest elliptic curve implementation.

Let us take a look at the other designs using $GF(p_{192})$. Auer [4] is using a multiply-accumulate unit with a dual-port RAM. Because his memory unit has two ports it could be nearly twice as fast. Actually it is 1.6 times faster. The design by Fürbass is a lot faster but it also is 70% larger than the presented design in this thesis.

As previously mentioned, the most important factor for RFID designs is the area requirement. By focusing on this requirement, the presented design can compete with all the other designs. Leinwebers design only is 40% smaller than the stripped Neptun design.

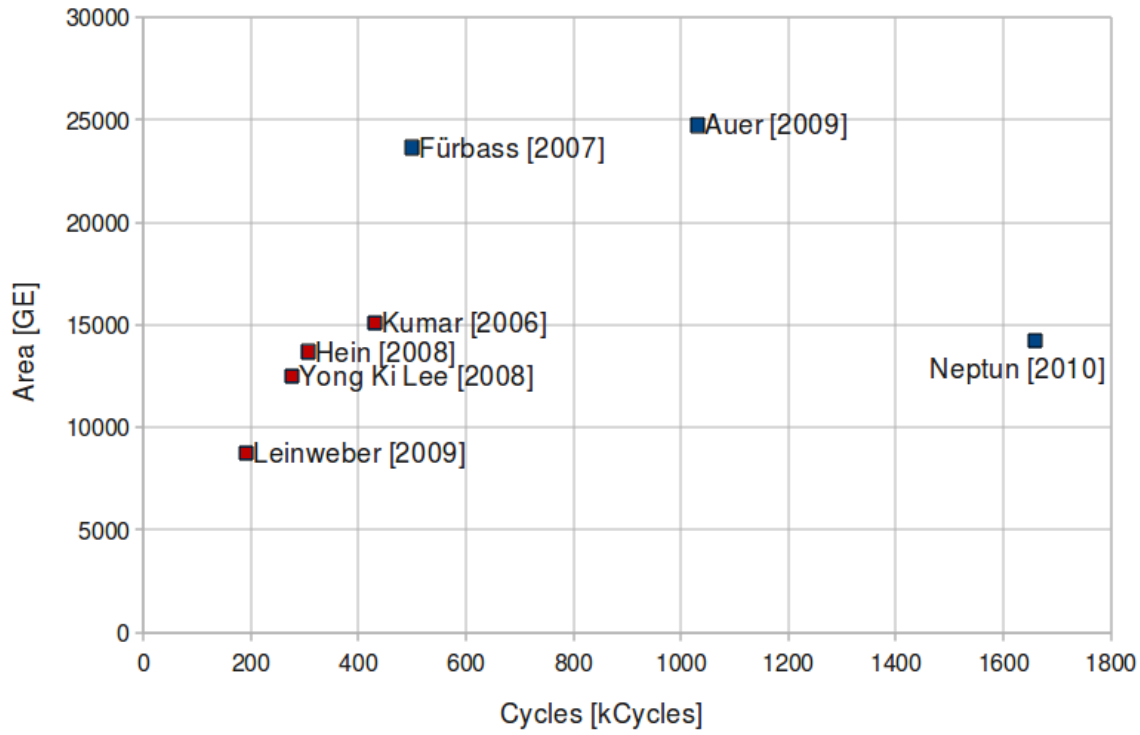

Figure 10.5: Area vs. runtime of various implementations. The red squares use $GF(2^{163})$, the blue squares $GF(p_{192})$.

Chapter 11

Conclusions

This master thesis had the goal to create an ASIC design that is capable of calculating an elliptic curve digital signature algorithm. Investigations showed that it is more efficient to design a dedicated processor. Although writing a processor that is specialized for efficient prime-field operations is a complex objective, it has been successfully finished. The final processor architecture is called *Neptun*. It uses an efficient Harvard architecture and an optimized 16-bit datapath.

But not only the a processor has been implemented. Also the elliptic curve digital signature generation and verification functions for a NIST P-192 curve. The signature algorithm takes 1.65 million cycles to generate a new signature. By synthesizing a stripped design, the area requirement of the design is only 14230 gate equivalents. This makes it to our knowledge to the smallest implementation for generating an elliptic curve signature for NIST P-192, published so far. By modifying the used program, a signature can also be verified. Only a small increase in area requirement is needed. This can be done in 3.1 million cycles. Previous implementations only concentrated on calculating a scalar point multiplication or just the signature algorithm. The signature verification feature extends the possible applications for this processor.

It is also possible to compare the implementation with previous implementations using embedded processors as platform. An optimized implementation using an 8-bit AVR processor, doing a NIST P-192 elliptic curve point multiplication used 9.9 million cycles (see [10]). An other implementation, using a 32-bit TM1300 processor by Trimedia is using 3 million cycles (see [15]).

A big advantage of the processor is that it comes with a simulator and an assembler. These tools can be used to easily write new programs for the processor and extend its capabilities. Because this tool is written in Java it is platform independent and modular. It can be easily extended to test new instructions or generate extra user-defined statistics. With the existing framework it is easily possible to extend the existing algorithms. It would be easily possible to implement an elliptic curve signature algorithm that uses NIST P-256 parameters. Also a symmetric-key algorithm, such as AES can be implemented, using the boolean logic of the processor. Admittedly the initial performance would not be breathtaking but the processor is easily extendable with special features needed for AES operations.

Because of the complexity of the implemented algorithms it is necessary to evaluate their vulnerability to simple and differential power analysis attacks. This can be done with *Neptun*. This processor brings a bunch of interesting features. It is programmable, provides a powerful and flexible 16-bit instruction set and also reusable peripherals. The

implemented timers can be used to simulate any kind of protocol. So the processor is already prepared for complex protocols like ISO-14443 or ISO-7816 (see [17, 18, 19]).

Neptun can even be used as a stand-alone processor. The only disadvantage is that no non-volatile memory is available in the used UMC 190L technology. So the program and constants need to be stored in an external flash, which is connected via SPI. Because the private key also has to be placed in this flash, the private key is easily discovered by monitoring the SPI lines. In this sense, the processor is not yet ready for production. But that was never a goal of this project.

To be ready for production, an efficient test strategy is needed. This is especially important for low-cost devices that should be produced in large quantities (in term of millions). Because this processor has a small area footprint it can be classified as a low-cost design.

The architecture used for this project is well documented with block diagrams (see Figures 9.2 and 9.3) and designed modularly. This makes it possible to easily reduce the size of the implementation. Possible approaches and ideas are listed in Section 10.4.

The results of the implementations, resulted by this thesis are summarized in Table 11.1.

Design	Area		Runtime	Technology
	[μm^2]	[GE]	[kCycles]	
Neptun processor	825974	88109		UMC L180
Signature design	133397	14230	1659	UMC L180
Verification design	149437	15941	3130	UMC L180

Table 11.1: Summary of synthesis results.

After reading all this praise about the *Neptun* processor, one question might arise: What is the future of the *Neptun* processor?

- Firstly, the chip will be tested. All the simulation results will be verified on the real-world chip.
- A printed circuit board to evaluate different attributes of the chip will be made.
- Algorithms will be evaluated. Different SPA and DPA attacks will be performed to proof the security of the implemented algorithms.
- The design will be further optimized. 14230 gate equivalents are not yet a lower bound for the design.
- Further algorithms such as AES will be implemented. The ALU will be extended to support substitution box operations within one cycle.
- The ISO-14443 and ISO-7816 protocols will be implemented.
- If there is a commercial application for the processor, it is useful to have a 'C' compiler. Maybe a compiler infrastructure such as LLVM will be used.
- Maybe the processor will be bought by a big semiconductor company and used for all future RFID tags. If the interested reader is a member of such a company, please contact the author.

- Most importantly, the *Neptun* processor will be reused for different future designs of the author.

It can be said without remorse that the initial goal was not only reached but exceeded.

Appendix A

Abbreviations

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
CPU	Central Processor Unit
DES	Data Encryption Standard
DPA	Differential Power Analysis
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
EIA	Electronics Industries Association
FIPS	Federal Information Processing Standard
GE	Gate Equivalent
GSM	Global System for Mobile Communications
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
NIST	National Institute of Standards and Technology
PC	Program Counter
GFN	Quad Flat No Leads Package
RAM	Random Access Memory
RFID	Radio Frequency Identification
RISC	Reduced Instruction Set Computing
ROM	Read-Only Memory
RSA	Rivest-Shamir-Adleman
SIM	Subscriber Identity Module
SP	Stack Pointer
SPA	Simple Power Analysis
SPI	Serial Peripheral Interface
UMC	United Microelectronics Corporation
UMTS	Universal Mobile Telecommunications System
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Appendix B

Processor

B.1 Features

- High Performance 16 bit Microcontroller
- RISC Architecture
  - 40 Instructions - Most Single Clock Cycle Execution
  - 12 Registers
- Volatile Program, Data and Constant Memories
  - 10K Bytes Program Memory (5120x16)
  - 1K Byte Data Memory (512x16)
  - 1K Byte Constant Memory (512x16)
- Peripheral Features
  - One Full Duplex EIA-232
  - Three 16 bit Timer/Counter with Output Compare and Input Capture
  - 32 Parallel Programmable I/O (16 Inputs, 16 Outputs)
- I/O and Package
  - QFN 56 Package
  - 4 Pad Power Supply Pins
  - 4 Core Power Supply Pins
  - 40 I/O Pins
- Supply Voltages
  - 3.3V Pad Supply Voltage
  - 1.8V Core Supply Voltage
- Operating Speed: Up to 55.55MHz

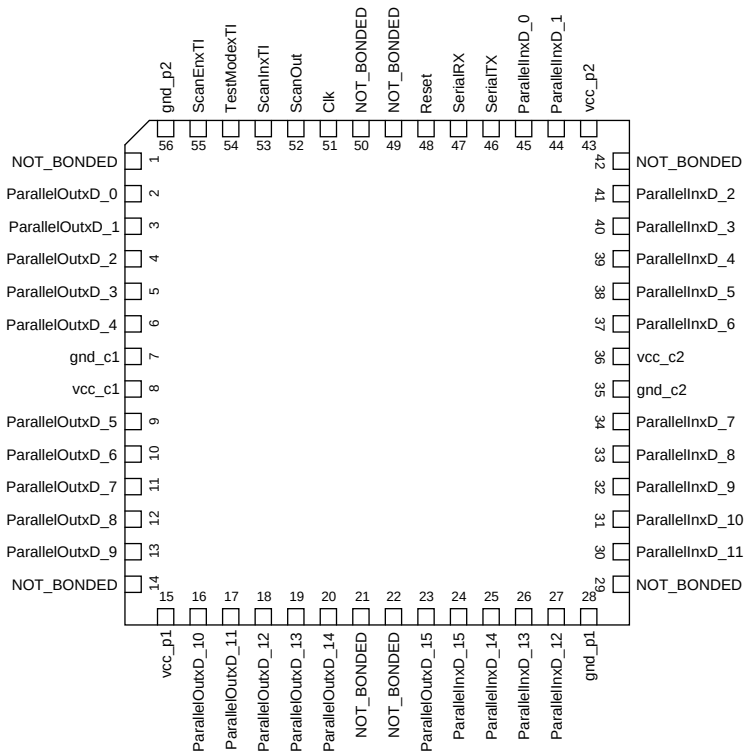

Figure B.1: Pinout of Neptun

B.2 Pin Configuration

B.2.1 Pin Description

Pin(s)	Description
vcc_c1, vcc_c2 gnd_c1, gnd_c2	Core Voltage Supply Pins. Recommended: 1.8V
vcc_p1, vcc_p2 gnd_p1, gnd_p2	Pad Voltage Supply Pins. Recommended: 1.8V
SerialRX, SerialTX	Receive and transmit line of EIA-232
Clk Reset	Clock Input System Reset
ParallelOutxD	Parallel Programmable Outputs
ParallelOutxD_9	Reconfigurable as Timer0 Output Compare
ParallelOutxD_10	Reconfigurable as Timer1 Output Compare
ParallelOutxD_11	Reconfigurable as Timer2 Output Compare
ParallelOutxD_12	Low after Reset, High in Bootloader, Low in Application
ParallelOutxD_13	High after Reset, Bootloader: SPI Chip Select
ParallelOutxD_14	Low after Reset, Bootloader: SPI Data Output
ParallelOutxD_15	Low after Reset, Bootloader: SPI Clock $\leq$ Clk/30
ParallelInxD	Parallel Programmable Inputs
ParallelInxD_9	Also used as Timer0 Trigger
ParallelInxD_10	Also used as Timer1 Trigger
ParallelInxD_11	Also used as Timer2 Trigger
ParallelInxD_12,13	Bootloader: EIA-232 Frequency Divider: 00 ... 16MHz / 138 $\rightarrow$ 115200 bps 01 ... 16MHz / 416 $\rightarrow$ 38400 bps 01 ... 48MHz / 416 $\rightarrow$ 115200 bps 10 ... 13.56MHz / 117 $\rightarrow$ 115200bps 11 ... divider: 4 $\rightarrow$ speed up simulation (ParallelInxD_13 is the most significant bit)
ParallelInxD_14	If Set, the Bootloader reads the data from an attached SPI EEPROM and initializes the internal RAM. After that, the application is started.
ParallelInxD_15	Bootloader: SPI Data Input
ScanEnxTI	Input, Enables the Scan Chain, Used for testing.
TestModexTI	Input, High during Scan Chain testing, Low during normal Operation
ScanInxTI	Input of the first register in the scan chain
ScanOutxTO	Output of the last register in the scan chain

B.3 Memory Mapping

The memory is divided into four sections. They are distinct by the two most significant bits. See table B.1.

The Data Memory, Constant Memory and the Memory Mapped Registers can be read or written anytime. There are some special considerations with the Bootloader and the Program Memory. The Bootloader is not accessible via Memory Accesses. It is used fore

00	0000h - 0289h	Bootloader
00	0000h - 0200h	Data Memory
01	4000h - 4200h	Constant Memory
10	8000h - 9400h	Program Memory
11	C000h - FFFFh	Memory Mapped Registers

Table B.1: Address regions of internal memories. The first column represents the two most significant bits of the addresses.

code execution only.

During Bootloader execution, the Program Memory can be read or written like the other memories. When a program is executed from the Program Memory, the Program Memory can neither be read, nor written. In the rare case that this is necessary to access it anyways, the following two bootloader functions can be used: *Bootloader.ReadProgramMemory* and *Bootloader.WriteProgramMemory*.

B.4 Bootloader

The bootloader supports two major operating modes:

1. Read SPI Flash and store in RAMs
2. Operating Interface via EIA-232

These operating modes are selected with the use of pin *ParallelInxD_14*. A feature, both modes have in common is to set the *ParallelInxD_12* pin, during bootloader operation.

B.4.1 SPI Flash

As reference, the read protocol of two SPI Flashs by two different manufacturers has been considered . The command 03h is used, followed by three 00h bytes. After this short sequence, 12288 Bytes are read. The first byte are the 8 higher bits of address 0000h. The second byte are the lower 8 bits of address 0000h. The third byte are the 8 higher bits of address 0001h. And so on...

ref datasheets

The three memories are represented in this order: Data RAM (1024 bytes), Constant RAM (1024 bytes) and Program RAM (10240 bytes).

The bootloader cannot distinct if the read cycles has been successful or not. Anyways the program is started using a **CALL** to address 8000h (the beginning of the program memory). It is the same behavior as starting the program, using a 'P' Serial Message.

In the case that the program returns, the *ParallelInxD_14* is ignored and the Serial Interface is reinitialized.

B.4.2 Serial Interface

After the reset of the device and a cleared *ParallelInxD_14* pin, the serial interface is initialized. Because the processor can be operated at different clock rates, the bootloader supports different clock dividers. Immediately after reset, the pins *ParallelInxD_12 & 13* are read. The clock divider for the EIA-232 interface is set accordingly.

Every character sent to the chip is immediately answered (except in the case of an error). The Serial interface supports different commands. They are distinct by the first byte. The command byte:

'T' Test the memories. Each memory is tested 3 times. Twice with constants (5555h and AAAAh) and once with a running 1. If a test is finished successfully, a '+' is answered. Else a '-' sign. In total, there must be nine '+': '+++++++++'.

'S' For writing data, a modification of Motorola S-Records are used. Originally, the address is a byte address. Here, the two address bytes are used as a word address. Sample: `S11380005C0FC0808886B583888EB584580F16911E`.
Every record starts with a 'S'. Followed by a '1', signalling two address bytes are used. Followed by two ASCII digits, representing a byte count in hexadecimal. 'Count' bytes are following. The next two bytes are the designated write address (`8000`). The data bytes are following. The first byte is always the high byte. The second byte is the low byte. Both parts of the word must be written at once. The final byte (`1E`) is a checksum. In the case of an error at any point, '-' is sent to the host.

'R','K','M' These commands are used to read the Memories. 'R' reads the Data RAM. 'K' reads the Constant RAM. 'M' reads the Program RAM. The representation of the data is similar to the S-Record used for writing. The 'S' is replaced by 'R', 'K' or 'M', respectively. Sample: `R11300007BCBA2A12E39A1F09DA89802449BEF12AC`.

'J' Orders the processor to execute address 8000h. It is not expected that the processor returns.

'P' Call the program memory address 8000h using a **CALL** command. In the case, the program returns, the Bootloader reinitializes the Serial Interface.

'I' Initializes the RAMs with zeros.

Appendix C

I/O Interfaces

This chapter concentrates on the memory mapped interface of the peripheral functions and not on how they actually work.

Table C.1 shows a summary of all memory mapped registers.

C.1 EIA-232

Control Register Write this register after the Clock Divider Register is configured.

bit	name	description
0	CONTROL_RX_ENABLE	set to 1 to enable receiver
1	CONTROL_TX_ENABLE	set to 1 to enable transmitter

Status Register This is a read only registers. All flags are reset automatically.

bit	name	description
0	STATUS_RX_DATA_READY	is 1 if new data is ready to read
1	STATUS_RX_FRAME_ERROR	is 1 if stop bit was not logic one
2	STATUS_RX_DATA_ERROR	is 1 if buffer overwritten before read
3	STATUS_RX_RECEIVING	is 1 if currently receiving new data
4	STATUS_TX_TRANSMITTING	is 1 if currently transmitting
5	STATUS_TX_BUFFER_EMPTY	is 1 if transmit buffer is empty

Transmit Buffer Write the lower 8 bits to initialize a byte transfer. Write only, when STATUS_TX_BUFFER_EMPTY is 1.

Transmit Shift Register Shows, how many bits have already been transmitted.

Transmit Clock Counter This 16 bit counter is used to divide the processor clock. It is initialized with the 'Clock Divider Register' and counts down. New bit is transmitted, when zero is reached.

Receive Buffer Contains the latest received full byte. (lower 8 bits) After read, the STATUS_RX_DATA_READY is cleared.

Receive Shift Register Displays the current receiving bits.

Receive Clock Counter This 16 bit counter is used to divide the processor clock. Is initialized with the 'Clock Divider Register' and counts down. New data is read at $(divider/2)$. This provides a 10% bitrate tolerance.

Address	Peripheral	Read/Write	Name
C000	EIA-232	R/W	Control Register
C001	EIA-232	R	Status Register
C002	EIA-232	R/W	Transmit Buffer
C003	EIA-232	R	Transmit Shift Register
C004	EIA-232	R	Transmit Clock Counter
C005	EIA-232	R	Receive Buffer
C006	EIA-232	R	Receive Shift Register
C007	EIA-232	R	Receive Clock Counter
C008	EIA-232	R/W	Clock Divider Register
C040	Parallel I/O	R/W	Output Register, 2000h after reset
C041	Parallel I/O	R	Input Register
C080	Timer0	R/W	Control Register
C081	Timer0	R	Status Register
C082	Timer0	R/W	Counter Register
C083	Timer0	R/W	Compare Register A
C084	Timer0	R/W	compare Register B
C0C0	Timer1	R/W	Control Register
C0C1	Timer1	R	Status Register
C0C2	Timer1	R/W	Counter Register
C0C3	Timer1	R/W	Compare Register A
C0C4	Timer1	R/W	compare Register B
C100	Timer2	R/W	Control Register
C101	Timer2	R	Status Register
C102	Timer2	R/W	Counter Register
C103	Timer2	R/W	Compare Register A
C104	Timer2	R/W	compare Register B

Table C.1: Summary of memory mapped registers

Clock Divider Register The system clock frequency divided by $(divider + 1)$ results in the used communication bitrate.

C.2 Timer

All three timers are built in the same way:

Control Register Write this register, after the compare registers are initialized. In default, the counter is not counting. Restart overrules Stop. External Trigger overrules Compare Event. Writing 'force bits' overrules everything.

bit	name	description
0	Enable	set to 1 to enable timer
1	InvertExternalTrigger	set to 1 to invert the external trigger
2	TriggerResetOnPosedge	set to 1 to reset counter when posedge detected on trigger signal
3	TriggerCountWhenHigh	if 1 counter counts when trigger signal is high
4-5	CompareAEventSelect	if 'Counter Register' = 'Compare Register A' then: '00' - do nothing '01' - set output '10' - clear output '11' - toggle output
6-7	CompareBEventSelect	if 'Counter Register' = 'Compare Register B' then: '00' - do nothing '01' - set output '10' - clear output '11' - toggle output
8	CompareBSTOP	set to 1 to stop counting when 'Counter Register' = 'Compare Register B'
9	CompareBRESTART	set to 1 to restart counting (from zero) when 'Counter Register' = 'Compare Register A'
10	OverrideOutput	set to 1 to override the ParallelIO Output
12	StartCounting	set to 1 to force the counter to start
13	StopCounting	set to 1 to force the counter to stop
14	ResetCounter	set to 1 to reset the counter value

Status Register Shows the current operating condition of the processor.

bit	name	description
0	STATE_Counting	if 1 counter is counting
1	STATE_OutputCompare	is 1 if OutputCompare is 1

Counter Register Represents the counter register.

Compare Register A Used for an output compare event.

Compare Register B Used for an output compare event and stop or restart event.

C.3 Parallel I/O

Output Register Every bit represents the state of a parallel output pin. The reset value is 2000h. The SPI Chip Select line is immediately set after reset.

Input Register Directly represents the registered state of the parallel input pins.

Appendix D

Montgomery Ladder Implementation

The implementation presented here makes use of the formulas presented by Izu, Takagi, ...:

$$x_3 = \frac{2(x_1 + x_2)(x_1 x_2 + a) + 4b}{(x_1 - x_2)^2} - x \tag{D.1}$$

$$x_4 = \frac{(x_1^2 - a)^2 - 8x_1 b}{4(x_1^3 + ax_1 + b)} \tag{D.2}$$

These formulas used here can be transformed using a common-Z standard projection.

$$X_3 = [2(X_1 + X_2)(X_1 X_2 - 3Z^2) + 4bZ^3 - XZ(X_1 - X_2)^2]4(X_1^3 - 3X_1 Z^2 + bZ^3) \tag{D.3}$$

$$X_4 = [(X_1^2 + 3Z^2)^2 - 8X_1 Z^3 b](X_1 - X_2)^2 \tag{D.4}$$

$$Z' = Z(X_1 - X_2)^2 4(X_1^3 - 3X_1 Z^2 + bZ^3) \tag{D.5}$$

Table D.1 shows the resulting algorithm. $Q_3 = Q_1 + Q_2$. $Q_4 = 2Q_1$. X_3 and X_4 are the X-coordinates of Q_3 and Q_4. x (without index) is the X-coordinate of $P = Q_2 - Q_1$.

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Init	X_1	X_2	Z	XXX	XXX	XXX	XXX	XXX
$T_3 = T_2 * T_2$			read	write	XXX	XXX	XXX	XXX
$T_4 = T_3 + T_3$				read	write	XXX	XXX	XXX
$T_4 = T_3 + T_4$				read	write	XXX	XXX	XXX
$T_5 = T_2 * T_3$			read	read		write	XXX	XXX
$T_3 = b * T_5$				write		read	XXX	XXX
$T_5 = T_0 * T_1$	read	read				write	XXX	XXX
$T_5 = T_5 - T_4$					read	write	XXX	XXX
$T_6 = T_0 + T_1$	read	read					write	XXX
$T_7 = T_6 * T_5$						read	read	write
$T_1 = T_0 - T_1$	read	write				XXX	XXX	
$T_5 = T_0 * T_0$	read					write	XXX	
$T_6 = T_5 - T_4$					read	read	write	
$T_4 = T_4 + T_5$					write	read		
$T_5 = T_0 * T_6$	read					write	read	
$T_5 = T_5 + T_3$				read		write	XXX	
$T_3 = T_3 + T_3$				write			XXX	
$T_6 = T_0 * T_3$	read			read			write	
$T_0 = T_4 * T_4$	write				read			
$T_3 = T_3 + T_7$				write	XXX			read
$T_4 = T_6 + T_6$					write		read	XXX
$T_4 = T_4 + T_4$					write		XXX	XXX
$T_0 = T_0 - T_4$	write				read		XXX	XXX
$T_4 = T_1 * T_1$		read			write		XXX	XXX
$T_1 = T_2 * T_4$		write	read		read		XXX	XXX
$T_2 = T_4 * T_0$	read		X_4		read		XXX	XXX
$T_0 = T_5 + T_5$	write				XXX	read	XXX	XXX
$T_3 = T_3 + T_3$				write	XXX	XXX	XXX	XXX
$T_4 = x * T_1$		read			write	XXX	XXX	XXX
$T_3 = T_3 - T_4$				write	read	XXX	XXX	XXX
$T_0 = T_0 + T_0$	write				XXX	XXX	XXX	XXX
$T_4 = T_3 * T_0$	read			read	X_3	XXX	XXX	XXX
$T_3 = T_0 * T_1$	read	read		Z'		XXX	XXX	XXX
Result	XXX	XXX	X_4	Z'	X_3	XXX	XXX	XXX

Table D.1: A very fast double and add algorithm.

Appendix E

Original Assignment

The next two pages show the original assignment for this thesis.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Master Thesis at the Department of Information Technology and Electrical Engineering

Autumn Term 2009

Erich Wenger

Secure Circuits for RFIDs and Smart Cards with Low Power Consumption and Small Size

Advisors: Norbert Felber, ETZ J84, 044 632 5242, felber@iis.ee.ethz.ch
Luca Henzen, ETZ J72.2 044 632 6686, henzen@iis.ee.ethz.ch
Christoph Roth ETZ J89 044 632 7647, rothc@iis.ee.ethz.ch

Advisor TUG: Martin Feldhofer, martin.feldhofer@iaik.tugraz.at

Date of Issue: Wednesday, 16-October-2009
Deadline: Tuesday, 16-April-2010

Introduction

An ASIC implementation of digital circuits for RFID tags and smart cards is the goal of this Master Thesis. They have to be optimized for low energy consumption, for low silicon area, and for high DPA resilience.

System Description

The system specification is one of the first tasks of this project. The core knowledge of TU Graz in cryptography and RFID, and of the Integrated Systems Laboratory in VLSI design shall be used to get to a realistic specification.

Tasks

The main task of the Master Thesis is the development of the ASIC. For the specified functionality, the design space has to be explored in order to get a series of near-optimal solutions concerning energy consumption. At least one item thereof has to be chosen for integration in silicon. It has then to be optimized for DPA resilience. After placement and routing and careful verification, the fabrication output for tape-out is to be prepared. For efficient ASIC design, the lecture *VLSI II: Entwurf von hochintegrierten Schaltungen* has to be followed. The book *Digital Integrated Circuit Design* by H. Kaeslin, on which this lecture is based, contains also the topics of the lecture *VLSI I: Architektur von hochintegrierten Schaltungen* (held in Spring Semesters) which are prerequisites for a successful chip design.

A more detailed task description is presented in the document *Master Thesis Erich Wenger* of the TU Graz.

Organization

The specification for the result of this thesis comes from IAIK of TU Graz. Martin Feldhofer plays a consulting role in this project. Questions concerning the specifications are to be discussed mainly with him. The focus of support from the IIS-ETHZ side is the VLSI implementation of the ASIC.

Due to this organization, it is important to keep both IIS and IAIK informed on the state of the project. While this happens with IIS on the regular weekly meetings. IAIK should be informed, e.g. by short e-mails following these meetings, and by exchanging important documents whenever actual. In case of questions and problems, we expect the Master Student to contact the advisors and/or co-advisors any time.

Report

Document your investigations and the results in a written report. Include also attempts that were not successful. This report must be formulated such that it is understandable not only by specialists, but by any microelectronics engineer. Program and HDL code can be included on a CD.

After the conclusion of the work, the results are to be presented in a talk of 20 minutes duration at ETHZ and also at TU Graz.

Zürich, 14-October-2009 N. Felber Prof. W. Fichtner

Die Arbeit wird ohne Rückgabe der Schlüssel nicht akzeptiert!

Bibliography

[1] K. Ananyi, H. Alrimeih, and D. Rakhmatov. Flexible hardware processor for elliptic curve cryptography over nist prime fields. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(8):1099–1112, 2009.

[2] ARM Corporation. 16-bit Thumb Instruction Set. Available online at `http://infocenter.arm.com/help/topic/com.arm.doc.qrc0006e/QRC0006_UAL16.pdf`, May 2010.

[3] Atmel Corporation. 8-bit AVR Instruction Set. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, May 2008.

[4] A. Auer. Scaling hardware for electronic signatures to a minimum. Master's thesis, TU Graz, October 2008.

[5] E. Brier and M. Joye. Weierstraß Elliptic Curves and Side-Channel Attacks. In D. Naccache and P. Paillier, editors, *Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002, Paris, France, February 12-14, 2002, Proceedings*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer, 2002.

[6] C. D. Canniére and B. Preneel. TRIVIUM Specifications. eSTREAM, ECRYPT Stream Cipher Project (`http://www.ecrypt.eu.org/stream`), Report 2005/030, April 2005.

[7] M. Feldhofer and J. Wolkerstorfer. Strong Crypto for RFID Tags - Comparison of Low-Power Hardware Implementations. In *IEEE International Symposium on Circuits and Systems (ISCAS 2007), New Orleans, USA, May 27-30, 2007, Proceedings*, pages 1839–1842. IEEE, May 2007.

[8] F. Fürbass and J. Wolkerstorfer. ECC Processor with Low Die Size for RFID Applications. In *Proceedings of 2007 IEEE International Symposium on Circuits and Systems*. IEEE, IEEE, May 2007.

[9] T. Güneysu and C. Paar. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154, pages 62–78, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 925–943, 2004.

[11] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, 6th International Workshop, Cambridge, MA, USA, August 11-13, 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2004.

[12] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[13] D. Hein, J. Wolkerstorfer, , and N. Felber. ECC is Ready for RFID - A Proof in Silicon. In *Workshop on RFID Security 2008 (RFIDsec08)*, July 2008.

[14] M. Hell, T. Johansson, and W. Meier. Grain - A Stream Cipher for Constrained Environments. eSTREAM, ECRYPT Stream Cipher Project (`http://www.ecrypt.eu.org/stream`), Report 2005/010, 2006. Revised version.

[15] Y. Hu, Q. Li, and C. Kuo. Efficient implementation of elliptic curve cryptography (ECC) on VLIW-micro-architecture media processor. In *2004 IEEE International Conference on Multimedia and Expo, 2004. ICME'04*, volume 2, 2004.

[16] M. Hutter, M. Medwed, D. Hein, and J. Wolkerstorfer. Attacking ECDSA-Enabled RFID Devices. In M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud, editors, *Applied Cryptography and Network Security - ACNS 2009, 7th International Conference, Paris-Rocquencourt, France, June 2-5, 2009, Proceedings*, volume 5536, pages 519–534. Springer, May 2009.

[17] International Organisation for Standardization (ISO). ISO/IEC 7816-4: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 4: Interindustry commands for interchange. Available online at `http://www.iso.org`, 1995.

[18] International Organisation for Standardization (ISO). ISO/IEC 7816-3: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 3: Electronic signals and transmission protocols. Available online at `http://www.iso.org`, September 1997.

[19] International Organization for Standardization (ISO). ISO/IEC 14443: Identification Cards - Contactless Integrated Circuit(s) Cards - Proximity Cards, 2000.

[20] K. Itoh, T. Izu, and M. Takenaka. A practical countermeasure against address-bit differential power analysis. *Cryptographic Hardware and Embedded Systems-CHES 2003*, pages 382–396, 2003.

[21] T. Izu, B. Möller, and T. Takagi. Improved Elliptic Curve Multiplication Methods Resistant against Side Channel Attacks. In A. Menezes and P. Sarkar, editors, *INDOCRYPT*, volume 2551 of *Lecture Notes in Computer Science*, pages 296–313. Springer, 2002.

[22] T. Izu and T. Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In *Public Key Cryptography*, pages 371–374. Springer, 2002.

[23] H. Kaeslin. *Digital Integrated Circuit Design – From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, 2008. ISBN 978-0-521-88267-5.

[24] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[25] Ç. K. Koç, T. Acar, and B. S. K. Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[26] S. S. Kumar and C. Paar. Are standards compliant Elliptic Curve Cryptosystems feasible on RFID? In *Workshop on RFID Security 2006 (RFIDSec06), July 12-14, Graz, Austria*, 2006.

[27] Y. K. Lee, K. Sakiyama, L. Batina, and I. Verbauwhede. Elliptic-Curve-Based Security Processor for RFID. *IEEE Transactions on Computers*, 57(11):1514–1527, November 2008.

[28] L. Leinweber, C. Papachristou, and F. Wolff. Efficient Architectures for Elliptic Curve Cryptography Processors for RFID. 2009.

[29] C. McIvor, M. McLoone, and J. McCanny. Hardware Elliptic Curve Cryptographic Processor Over $rmGF(p)$. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 53(9):1946–1957, 2006.

[30] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[31] National Institute of Standards and Technology (NIST). FIPS-186-2: Digital Signature Standard (DSS), January 2000. Available online at `http://www.itl.nist.gov/fipspubs/`.

[32] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.

[33] W. Rankl and W. Effing. *Handbuch der Chipkarten*. Carl Hanser Verlag, 4th edition, September 2002. ISBN 3-446-22036-4.

[34] E. Savas and C. Koc. The Montgomery modular inverse-revisited. *IEEE Transactions on Computers*, 49(7):763–766, 2000.

[35] R. Zimmermann. Computer arithmetic: Principles, architectures and vlsi design. Technical report, March 1999.