Low Resource Hardware Implementation of an Edwards Curve Digital Signature Algorithm

Jeremias Peter Kleer
jeremias.kleer@student.tugraz.at

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

Master Thesis

Supervisor: Dipl.-Ing. Dr.techn. Michael Hutter
Supervisor: Dipl.-Ing. Dr.techn. Erich Wenger
Assessor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Karl-Christian Posch

December, 2013

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

EIDESSTATTLICHE ERKLÄRUNG

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

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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

…… ……..
date (signature)

Acknowledgements

I want to thank Karl-Christian Posch, Michael Hutter and Erich Wenger of the Graz University of Technology for their support, their patience and for making this thesis possible.

I also want to thank Angelika Gerstmann and my family for their exceptional patience, support and love.

In addition I want to thank Eva Gerstmann for her time and commitment when correcting my English.

Abstract

Cryptography has become substantial in the digital world. Beside encryption, authentication is a major topic within cryptography. Authentication is important, for instance, in the world wide web where online banking or services of authorities are used.

In the past years, authentication has reached very small devices. They are accessing the Internet and use the same services where authentication is required. Such small devices are often battery powered and have limited resources. In this environment, performing the hard calculations of authentication is a crucial issue and can hardly be performed by a general processor. Here, application specific integrated circuits (ASIC) have the advantage to be tailored to one specific function. ASICs are dedicated hardware modules and perform calculations in a very efficient way. Efficiency in this context is often measured in the area-time product which allows to judge the trade-off between the necessary circuit area against the time the execution of performed task takes.

To perform authentication, several cryptographic systems and protocols are available and are based on different mathematical hard problems. For constrained devices, the elliptic curve discrete logarithm problem is very well suited. Therefore the standardized and approved Elliptic Curve Digital Signature Algorithm (ECDSA) enjoys great popularity. Beside this, Bernstein et al. [8] in 2011 introduced an alternative digital signature scheme: The Edwards-curve Digital Signature Algorithm (EdDSA). On the one hand it has advantages over ECDSA regarding the speed and on the other hand it circumvents security issues of ECDSA.

The key difference of EdDSA and ECDSA is the usage of twisted Edwards-curves rather than elliptic curves in Weierstrass form. These curves have a different defining equation and thus the point addition and point doubling formulas differ. Beside the definition of the general EdDSA the Ed25519-SHA512 was introduced in [8] which has all necessary parameters chosen for an efficient implementation.

In this thesis, a low resources ASIC hardware implementation is presented which is able to perform the scalar multiplication necessary for the Ed25519-SHA512. The implementation supports different word widths and those are evaluated for the most efficient ones. The synthesis on the $0.35\,\mu$m CMOS-process technology of ams AG[1] revealed the following results for different word widths (the area values in the parentheses include the area of the memory): 7.4 (44.4) kGE and $2.04 \cdot 10^6$ cycles for 16 bits, 12 (48.8) kGE and $0.8 \cdot 10^6$ cycles for 32-bit hardware, and 27.9 (66.1) kGE and $0.414 \cdot 10^6$ cycles for 64 bits.

Keywords: EdDSA, Twisted Edwards-Curves, Digital Signature, ASIC, Hardware Implementation, Scalar Multiplication

[1] ams AG ist the new name of the company formerly called austriamicrosystems AG.

Kurzfassung

In der digitalen Welt ist Kryptographie zu einem wesentlichen Bestandteil geworden. Dabei spielt neben der Verschlüsselung insbesondere die Authentifizierung eine wichtige Rolle. Diese ist zum Beispiel im World Wide Web wichtig - speziell etwa beim Internet-Banking oder bei Online-Diensten von Behörden.

Heutzutage wird Authentifizierung auch auf kleinsten Geräten benutzt. Über sie besteht Zugang zum Internet und es ist möglich, auf Angebote zuzugreifen, die eine Authentifizierung erfordern. Diese Geräte laufen oft mit Batterie und bieten daher nur begrenzte Ressourcen. Da die zur Authentifizierung nötigen Berechnungen aufwändig sind, stellen sie hier einen kritischen Aspekt dar - sie können einen universellen Prozessor überfordern. In solchen Fällen empfiehlt sich daher die Benutzung von anwendungsspezifischen integrierten Schaltungen (ASIC). Es handelt sich dabei um speziell abgestimmte Hardware-Module, die auf eine Aufgabe besonders zugeschnitten sind und diese effizient berechnen. Gemessen wird die Effizienz hierbei oft anhand des Flächen-Zeit-Produkts. Dieses ermöglicht die Beurteilung des Kompromisses zwischen einerseits einer raschen Berechnung und andererseits einer kleinen Fläche der Schaltung.

Für die Authentifizierung stehen einige kryptographische Systeme und Protokolle zur Verfügung, die auf unterschiedlichen mathematischen Problemen basieren. Für eingeschränkte Geräte ist der diskrete Logarithmus über elliptische Kurven gut geeignet, der beim Elliptische Kurven Digitalen Signatur Algorithmus (ECDSA) benutzt wird. Daneben hat Bernstein et al. [8] eine Alternative im Jahr 2011 präsentiert - den Edwards-Kurven Digitalen Signatur Algorithmus (EdDSA) - der Geschwindigkeits- und Sicherheitsvorteile bietet.

Der Hauptunterschied zwischen EdDSA und ECDSA ist die Benutzung von Twisted Edwards Kurven anstatt von Kurven in Weierstraß-Form. Diese Kurven haben eine andere Kurvengleichung und deshalb auch abweichende Gleichungen für die Punktaddition und -verdopplung. Neben der allgemeinen Definition von EdDSA wurde in [8] auch Ed25519-SHA512 präsentiert, bei dem die nötigen Parameter in Hinblick auf eine effiziente Implementierung gewählt wurden.

In dieser Arbeit wird eine ressourcenschonende ASIC-Hardware-Umsetzung präsentiert, die für Ed25519-SHA512 die nötige Skalar-Multiplikation berechnen kann. Sie unterstützt mehrere Wortbreiten, die auf ihre Effizienz evaluiert wurden.

Die Synthese mit der 0.35 μm Technologie von ams AG[2] hat zu folgenden Ergebnissen geführt (die Flächen in den Klammern sind inklusive der Fläche für den Speicher): 7.4 (44.4) kGE und $2.04 \cdot 10^6$ Takte bei einer 16-Bit-Architektur, 12 (48.8) kGE und $0.8 \cdot 10^6$ Takte bei 32 Bits, und 27.9 (66.1) kGE und $0.414 \cdot 10^6$ Takte bei 64 Bits.

[2] ams AG ist der neue Name der austriamicrosystems AG.

Stichwörter: EdDSA, Twisted Edwards-Curves, Digitale Signatur, ASIC, Hardware Umsetzung, Skalar Multiplikation

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Authentication is a fundamental process within the society and is required for any authority and administration. In the digital world this process is conveniently solved by public key cryptography (PKC) in which an entity can prove that the information is indeed originated by itself. For example, an entity can sign in on an administration website and fill in a form or receive information that should not be readable for the public. The PKC is based on mathematical hard problems that cannot be reversed or are computationally infeasible. An entity generates a key pair with one private and one public key. It can then use the private key to prove that it is indeed the one.

For PKC the elliptic curve discrete logarithm problem has been identified to be suitable to offer appropriate security for rather small secrets and to therefore be very efficient compared to other mathematical hard problems such as the integer factorization problem or the usual discrete logarithm problem. In this context, the ECDSA is a popular cryptographic system. It is standardized by the American National Standards Institute(ANSI), approved by NIST and even used for high security within governments.

The digital world has evolved and cryptographic applications are not limited to powerful personal computers anymore. Cryptography has become a topic for even the smallest electronic devices. There are mobile phone and even watches with access to the Internet and thus the problem arises that these constrained devices need cryptography. The limited computational capabilities of constrained devices implies thus low resource consumptions and efficiency. Those requirements imply the use of ASICs.

The ECDSA is based on certain elliptic curves and has certain mathematical implications and algorithmic requirements. In 2007, Edwards introduced in [13] a new form of elliptic curves that offer more efficient algorithmic with additionally improved security against attacks. Bernstein et al. in [11], based on Edwards curves, introduced twisted-Edwards curves and further extended the number of elliptic curves that can be used for the efficient algorithms. Bernstein et al. in [8] then introduced the Edwards-Curve Digital Signature Algorithm (EdDSA), a digital signature algorithm which is related to the ECDSA but uses twisted-Edwards curves and has some algorithmic modifications that improve security.

For the rather young and not standardized EdDSA, no hardware implementation as ASIC is known. This thesis' purpose is to implement such a hardware that is suitable for low resource requirements. The focus within this is to optimize the algorithmic and to evaluate the trade off between area and execution time. The Ed25519 signature operations require the Secure-Hash-Algorightm-512 (SHA-512) hash function but this is not considered to be a crucial part and was therefore not included in the implementation.

The signature and verification process are not implemented because this functionality is straightforward and can be easily appended with a SHA-512 module. Therefore the implementation includes all necessary modules and functionality to perform the most important arithmetic ranging from the integer level up to scalar multiplication.

1.1 Outline

In this thesis a historical view, an overview and a general discussion about cryptography will be provided in Chapter 1. Further, symmetric key cryptography will be touched and public key cryptography concepts will be presented.

In Chapter 3, the EdDSA signature algorithm will be discussed. This includes the general requirements, parameters and the algorithms necessary. Then the suggested explicit cryptographic scheme Ed25519 will be discussed and the necessary abstractions and the realization of the presented functionality will be outlined in detail.

The hardware implementation will be presented in Chapter 4. This consists of the used design flow and consideration of a hardware implementation in general. Subsequent to this, the implementation will be presented, starting from a high point of view down to the details. The chapter will end with the discussion about how the implementation was verified for correct calculations.

In chapter 5, the implementation's key facts will be presented and the evaluation of the best configuration will be made. The configurations considered best will then be compared to previous works regarding the key facts.

The conclusion will be presented in Chapter 6 which will end with suggestions of future work.

Chapter 2

Introduction to Cryptography

Cryptography comes into play when secrecy is involved. Secrecy has a long history and so does cryptography. In this chapter we will take a short look at its history. Afterwards, fundamental principles will be discussed and a higher-level view on symmetric-key cryptography and its counterpart public-key cryptography will be provided. Even though symmetric-key cryptography will be discussed, the focus will be on public-key cryptography, where elliptic-curve cryptography and signature systems will be dealt with in more detail.

History and Basic Ideas Cryptography has a long history. Even though he was not the first in history, Julius Caesar is known to have encrypted messages sent to his troops. Gaius Julius Caesar lived in the first century BC and therefore gives an idea about the long history of cryptography. As Simon Singh wrote in [36, Page 14], Caesar used various substitution ciphers. While he was fighting in Gaul, he used the substitution of replacing Roman letters with Greek ones. Further, the substitution of replacing letters with those that are three places further down the alphabet is called the Caesar Cipher. This mentioned cryptographic substitution and other methods until the 20th century were reversible on paper with a pen. In the 20th century a mentionable evolution took place by inventing cryptographic machines. A popular example for these was the Enigma machine, which can be categorized as electro-mechanical rotor cipher machine. It was used in World War II by the Germans. Simon Singh in [36, Page 149] also says that breaking the encryption of the Enigma machine shortened the duration of the war as the secret messages could be read by the Allied. The biggest step in the evolution of cryptography may be the invention of electronics performing cryptographic methods. Since then, reversing the encryption with paper and a pen by "hand" got practically infeasible.

A cryptographic method takes some input and by mapping, or some other mathematical relation, it generates output that is not recognized as the input. For a good cryptographic method, measurements and analysis give no clue about what the actual input was. These measurements might be statistical, counting the occurrences of single letters and groups of letters, mathematical calculations, or simple observations that can be made. In the context of statistical measurements, Shannon 1949 defined in [34, Pages 708-710] diffusion and confusion. He stated that **confusion** is the relation between the simple statistics of the output of a cryptographic method (ciphertext) and the simple description of the used key[1]. This relation should be as complex as possible. In other

[1]The key can be seen as the secret password.

words, each bit[2] of the ciphertext should be as complex as possible related to as many bits of the key as possible. This has the effect that an attacker is prevented to use simple analysis to obtain the secret. **Diffusion** in his definition is the relation of the statistical structure of the message and the output of the cryptographic method. This can be illustrated, for instance, with the redundancy due to the occurrence of long combinations of letters. Analysis of representative texts in a specific language give probabilities of single words' occurrences. This occurrences can be used when the ciphertext is analyzed. But a cryptographic method with good diffusion generates output for which analysis results in useless information. In a wider interpretation, a good diffusion means changing a minimum amount of input leads to a significantly bigger change of the output.

By looking at the history, we can see the evolution from cryptographic scheme relying on keeping the scheme secret, to cryptographic schemes that are publicly available and rely on keeping the key secret. A cryptographic scheme that is not dependent on a secret key, and thus its security only relies on keeping the scheme secret, does not fulfill its purpose reliably. This relies on the simple circumstance that as more entities know the scheme, more possible targets are involved that can be attacked to get information about the scheme. Once the secret is gained by an attacker all others using the scheme are vulnerable. In contrast, a (publicly known) well engineered cryptographic scheme which depends on a secret key serves security more reliable. These cryptographic schemes further give the possibility to change the secret key if wanted and it is not required to engineer a new scheme from scratch.

Since their invention, computers, also have been utilized for cryptography. Therefore all the processed messages have digital representations. Following from this and the fact that computers in general are based on numbers, cryptography nowadays basically deals with mathematical problems. Concluding from the fact that the computation power of computers beat human capabilities by magnitudes, the related mathematical problems have to be hard to solve. Thus, cryptography is concerned with the design and analysis of methods that enable a secure communication in an insecure environment.

For a first look at cryptography, some aspects should be mentioned first. The terms entities, a sender, a receiver and an attacker, can relate to human beings, but not exclusively. As secrecy is a goal in all contexts that in any way deal with private data, an entity might also refer to a computer system, a web browser and its communication counterpart, a web server, or something similar. Further, a message is not required to be a human readable text. This may be a single data packet, when for example a web browser sends a request for a website or the login information, a wireless LAN adapter sends the password to the accesspoint, a car is unlocked via a remote control, or any other digitally generated sequence of symbols.

When thinking about cryptography and a scenario where cryptography is used, there are basically three entities involved. At a message-based view these are a sender, a receiver, and an entity which is not wanted by the sender and receiver to get or manipulate their messages. In literature, concerning cryptography, the two entities communicating are often called Alice and Bob. During an ongoing communication the role of sender and receiver might switch and can be either Alice or Bob. The third entity mentioned before is called Eve. She is assumed to have knowledge about the used cryptographic scheme, considerable computational resources and is able to read and modify all data that is sent over the communication channel. In the following, this convention also applies.

[2]Any data or text that is stored in a computer can be broken into bits. A bit is the smallest unit used in computers and has two states, either one or zero.

Cryptography does not imply confidentiality in every case. As stated in [19, Section 1.1], the list below defines basic goals for schemes, this list, however here is not complete:

- **Confidentiality**: The information transferred between Alice and Bob is not readable for Eve.
- **Data integrity**: Bob can verify that the transferred message received from Alice has not been modified.
- **Authentication**: Bob can verify that the received message has indeed been written by Alice. Eve, in turn, can not create a message that is falsely accepted to have been written by Alice or Bob.
- **Non-repudiation**: A third neutral entity can verify that the very origin of the message created by Alice has indeed been Alice. Thus Alice cannot deny that she created it.

Depending on its purpose, a cryptographic scheme may combine the desired goals or may fulfill other goals such as anonymity for their communication entities or access control for restricting certain resources to specific entities. As an example for a scheme with certain purpose, with outlook on this thesis' topic, downloading a web browser's installation files does not have to be confidential. Nevertheless for security reasons it is desired to verify the origin to avoid installing a Trojan horse or some other type of virus. Here, authentication comes into play as one can verify if the downloaded files have been originated by the trusted developers who signed it.

2.1 Symmetric-key Cryptography

For the following section and subsections, we refer to the *Introduction and Overview* section in [19, Section 1.1]. Symmetric-key cryptography relies on the idea that Alice and Bob share a common secret. This secret enables them to use methods that are publicly available. These public availability however does not harm security as the security is based on a hard to solve mathematical problem and the shared, but secret key. After agreeing on the shared secret, Alice uses the related cryptographic method with this key and the message as input. Afterwards she sends the obtained result to Bob. He uses the counterpart method and the shared key to reverse or verify the operation of Alice.

Figure 2.1 shows the basic scenario for symmetric-key cryptography.

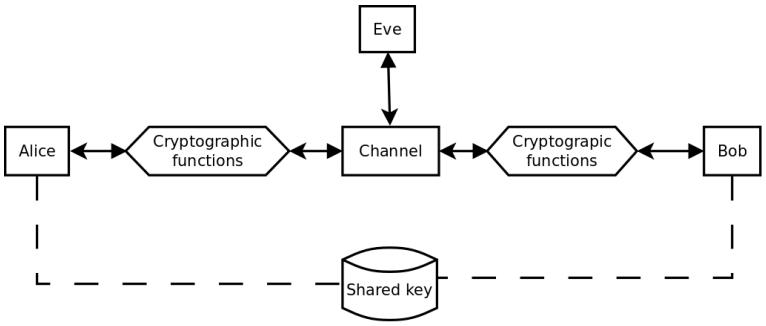

Figure 2.1: Communication in an insecure environment

Depending on the cryptographic goal, the scheme is engineered to fulfill, the above described methods can vary: for example one of the operations can be encryption and the other the counterpart-operation decryption, or signature generation and signature verification. A popular cryptographic scheme based on symmetric-key cryptography is, for instance, the Advanced Encryption Standard(AES) that serve methods for encryption and decryption.

To give a little insight into methods of cryptography, the basic procedures to achieve cryptographic goals will be discussed on a high-level view for two exemplary goals. This will be done respecting the structure of symmetric-key cryptography. For the following description, the shared secret key is assumed to be already accessible. Thus, it must be generated and distributed prior to the actual communication. As the generation procedure is scheme-specific, it will not be discussed in detail. The key distribution, on the other hand, will be discussed in the following paragraph, which focuses on key distribution and management, because this is a significant disadvantage.

Confidentiality As described in short beforehand, confidentiality is achieved by an encryption- and decryption-method. The whole procedure of encryption and decryption with all its details is publicly available. This does not threaten the security since only the knowledge of the key delivers the right results, it merely simplifies the overall process of establishing a secure channel as only the key has to be shared in a secure way.

When using a encryption method, its output should differ if only a single bit within the input changes. This input refers to the message as well as the used key, what has been discussed in the first paragraph of this chapter, regarding confusion and diffusion. The differing bits of the original output compared to the one obtained with the one-bit-flipped-input should ideally be distributed over the whole range of output bits and flipping a certain bit should not be traceable to specific output bits.

Alice uses the encryption method with the shared key and the message as input to obtain the so called ciphertext. This ciphertext is sent over the insecure channel to Bob. After receiving the message he uses the decryption method with the same shared key as used by Alice and the ciphertext as input to obtain the original message. As already mentioned, Eve is able to read the ciphertext. But due to the lack of knowledge of the used key, she will not be able to compute the original message.

Data integrity If data integrity is desired the cryptographic scheme changes in some points compared to confidentiality. After Alice and Bob agreed on a shared key, Alice and Bob use a message authentication code (MAC) algorithm. Alice computes the authentication tag t using the MAC with the shared key and the message as input. The authentication tag is then sent along with the message to Bob. As Bob uses the same MAC algorithm, knows the shared key and received the message, he can compute the authentication tag t' on his own. He uses the message and the shared key as input for the MAC for computing t' and verify if $t = t'$. If the verification succeeds Bob knows that the message is indeed signed with the right shared key and that it is highly probable it was not modified by Eve and indeed sent by Alice.

2.1.1 Key Distribution and Management

Symmetric key cryptography is based on the idea that two entities share a common key. This aspect leads to disadvantages regarding key distributing and key management, which

will be discussed in the following paragraphs. The first discusses the problem of key distribution concerning the need of two entities participating a communication to agree on keying material. The second addresses the problem of managing the keying material in a system that is not limited locally.

2.1.1.1 Key Distribution

Due to the principle of symmetric-key cryptography, the secret key has to be shared between both entities of communication. Sharing the keying material over an insecure channel assuming no one else reads it should not be the basis of a cryptographic system. This would harm the security from the very beginning. The shared key needs to be agreed on in a secure way. This can be assured by physically meeting or a secured channel. Meeting in person is not convenient as soon as the entities involved are spatial separated. The second possibility mentioned could be realized by a trusted courier or an already secured channel to a central distributing facility. Using a trusted courier gets impracticable with increasing distance between the entities. Further, in a scenario with two entities wanting to communicate over a global network such as the Internet, waiting days for the courier to arrive seems far from good. In a closed environment or a locally limited network each entity can initially agree on a key with the central key distributer. Entities then use the secure channel to the distributer to further agree and share keying material when required. But again, in applications concerning the Internet and global communication this is impracticable. For the initial agreeing on the keying material, each entity would have to either physically visit this facility, use a trusted courier or use an already established secure channel. As illustrated before, the first two possibilities have significant drawbacks. Let us assume, for now, that the central key distributer has local branches. For a practicable global system those branches with appropriate keying material would have to be spread all over the world. Further, each of these entities would have to be trusted and secured to ensure no leaking of keying material. This would end up in a huge organization that might never get efficient and convenient for users. Another aspect of the central distributing facility is the growing amount of necessary distribution in operating state. With the large number of nodes participating in the Internet in mind, including not only the server but each computer and mobile device, and assuming each node has separated and secured applications, each needing a secret key, the resulting amount of distribution cannot be handled by a central facility.

2.1.1.2 Key Management

With the increasing number of communication partners, another disadvantage arises. For each of the communication partners, an entity has to store a shared key. This gets impracticable or impossible in applications with limited storage space. This problem can again be handled by a central facility with an initially shared key and thus secured channel. The keys can be managed by the central facility and on demand obtained over the secured channel. As in the previous paragraph, this is impracticable as soon as the number of entities reach just a fraction of the number of nodes in the Internet. Then the management efforts evolve to a serious problem.

Another problem can be imagined in a scenario with a central key management facility. The keying material is shared between at least three entities, both of the end-point entities and the facility. Therefore, this limits the achievable goals of a scheme as non-repudiation is not possible, as [19, Section 1.1] clarifies.

2.2 Public-key Cryptography

Diffie, Hellman and Merkle 1976 in [12] published the concept of a public-key cryptography scheme to handle the disadvantages regarding the key distribution. The concept was engineered to enable security over an insecure channel without the requirement to share a secret previously. Still, the requirement to distribute the public key of an entity in an authentic way has to be met. In contrast to symmetric-key cryptography, this concept is not based on the fact that both entities of the communication share the same key, but each entity has two keys: One for the encryption of messages destined to it, and one for the decryption of messages it received. Due to this, public-key cryptography is also called asymmetric-key cryptography.

The underlying idea is to use a one-way function. This means, given the result of such a function, it is not possible or computational infeasible to reverse the operation without special knowledge. Here, the pair of keys come into play. In the concept, the public key and the message are used as input of the one-way function. Only the knowledge of the private key enables the back transformation to obtain the original message. This requires the public as well as the secret key to be related in a special mathematical way.

Whereas the concept of Diffie, Hellman and Merkle was the first publicly published, a public-key cryptographic system was engineered by Great Britain's Government Communications Headquarters starting at 1969 until 1975, but this was kept secret as Simon Singh in [36, Chapter 5, Page 211] stated. As Simon Singh writes further, Diffie, Hellman and Merkle discovered the concept of public-key cryptography and the need of one-way functions but failed to be the first to publish a working system as they failed to find such a function. But inspired by their concept, this race was won by Rivest, Shamir and Adleman. They found such a one-way function and proposed it 1978. In Subsection 2.2.4 it is discussed and for more detail see the article of Rivest, Shamir and Adleman [30].

After Rivest, Shamir and Adleman's scheme, other public-key cryptography schemes based on different hard to solve mathematical problems were introduced in the following years. Popular mathematical hard problems, in context of cryptography, will be discussed in the following sections.

2.2.1 Concept of Public-key Cryptography

An entity has two keys related to it: the private key and the public key. The private key, as the name already suggests, is kept secret and the public key is distributed to the communication partners in an authentic way. The public and the private key are generated as a pair with a certain mathematical relation. The generation of the keys is computationally easy whereas the reverse computation of the private key with only knowing the public key is computationally extremely difficult. The mentioned mathematical relation results in the following circumstance: The result of a cryptographic method cannot be reversed without knowledge of the counterpart key or the computational effort is higher than trying all possible keys. Therefore, by the use of appropriate long keys, significant security can be assured. This circumstance is referred to as one-way function.

2.2.2 Exemplary Goals

As the central element, the used keys of the cryptographic system changes in contrast to symmetric-key cryptography. The procedures to achieve cryptographic goals are exemplary discussed in the following.

Confidentiality The discussion about achieving confidentiality in public-key cryptography, in contrast to symmetric-key cryptography, starts with obtaining the keys. To encrypt a message destined to Alice, Bob obtains an authentic copy of Alice's public key. He computes the ciphertext with Alice's public key and the message as input for the encrypt method, and sends it to Alice. Alice computes the original message with her private key and the ciphertext as input for the decryption method.

Authentication Authentication is used to either check the authenticity of signed data or an entity.

In general, verifying an entity can be accomplished by a some sort of secret password or by a system of sending a challenge and checking the response. There, the response can only be computed with knowledge of the secret key.

Authentication of data serves the ability to reliably verify that the data was indeed sent by the entity. In turn, this also ensures that the data was not modified as different data causes a different signature. To be able to do this, an entity needs a unique attribute. In public-key cryptography this unique characteristic is directly served by each entities' keys. When Alice wants to write a message to Bob and wants to guarantee him the ability to verify that she indeed wrote it, she signs it. She uses her private key and the message as input for the signature generation function to create a digital signature and sends the signature, along with the message, to Bob. He can verify that the message was written by Alice with her public key. Therefore he checks if the received signature was created by Alice. He further checks if the signature was created for the received message. The verification is only then successful, when both checks succeeded. For public-key cryptography schemes it is characteristic that only the possessor of the private key can create a digital signature which passes the verification method. The signature generation procedure generates different signatures for each message and thus a signature generated for a certain message is only valid for this message.

2.2.3 Mathematical Hard Problems

The mathematical problems which are used in cryptographic schemes have in common that an easy calculated result of the related operation cannot be easily reversed to obtain the original inputs. Such an operation is called one-way function. The mathematical problems differ in the necessary effort to reverse the cryptographic function. This necessary effort is here referred to as hardness. The hardness measurement is based on the best known algorithm to solve the problem. Thus, a new algorithm to solve a particular mathematical problem might lower the attributed hardness.

The most notable mathematical hard problems, due to their use in cryptographic schemes, are the integer factorization problem and the discrete logarithm problem. Based on the discrete logarithm problem, the elliptic curve discrete logarithm problem was developed.

2.2.4 The RSA Cryptographic Scheme

The RSA is a cryptographic scheme published 1978 by Rivest, Shamir and Adleman. For detailed description and further information not covered here see their article [30]. In this section, for simplification the word integer refers to a positive integer.

It was the first public-key cryptography scheme after the concept was published by Diffie, Hellman and Merkle. 30 years after its publication, RSA is still widely used. The

hardness of the integer factorization problem forms the base for the hardness of breaking the RSA. To give an impression of one-way functions and mathematical hard problems, the principle of the integer factorization problem will be discussed here. Further the factorization of an 232 digits long integer will be discussed to get an idea of the necessary effort to reverse a one-way function.

Calculating the result of an integer multiplication is rather simple. Even for rather big numbers with several hundreds of bits, the result can be obtained in fractions of a second as soon as a computer is involved. In contrast to this, the reverse operation, namely the factorization, of a commonly big integer on a single computer takes hundreds of years as discussed in the following.

Integer factorization is the operation to reverse a multiplication and thus to find the multiplicands used. The fundamental theorem of arithmetic, see the reprint of Gauss' Disquisitiones Arithmeticae [18], states that the factorization of an integer is unique. Thus, once the factorization is completed, this is the only factorization possible. To check if a number B_1 is a factor of the integer A_0, the division $A_0/B_1 = A_1$ has to be carried out. If the devision has no remainder, B_1 is a factor. If B_1 is a prime, B_1 does not have to be factorized itself, as primes are only divided by 1 and the prime itself with no remainder. Basically, integer factors cannot be greater than the product. Thus, a strategy to find a factor of a number might be to divide by all these numbers until it delivers no remainder. A division by 1 can be carried out infinitely often with no remainder and serving no practical information about the factors. The division by the number itself does not have a remainder too, the result is 1 and also results in no information about the factors. These divisions are omitted. Further limitations to the factors that are tested can be made: Only numbers up to the square root of the product must be checked as bigger numbers would already be the result of the division by a smaller number than the square root. Every time a factor B_i is found, the search is restarted, but with the result A_i of the division $A_{i-1}/B_i = A_i$ as number to be factorized. Repeating this procedure until all factors are primes delivers the factorization of an integer number. That might not evolve a problem and can be calculated on paper as long as the number to factorize is small enough. Imagine to factorize an integer being the product of primes having hundreds of digits and are of similar length. Although more efficient algorithms than the trivial trying of all possibilities exist, until now, the integer factorization problem has no solution which would harm cryptographic schemes based on it using appropriate big integers. The factorization of the integer described in the following was done using such a more efficient algorithm.

In the cryptoeprint report [24] the factorization of a 768-bit number was reported in 2010. Specifically this number, called RSA-768, has no special structure that may give the possibility to accelerate the computation. Therefore it is categorized as the factorization of a general integer and the best known algorithm was used. The stated computational effort spent would take almost 2 000 years if it was run on a single core 2.2 GHz AMD Opteron processor with 2 GB Random Access Memory (RAM). The actual time spent to obtain the factorization on the specialized hardware was two years. In contrast to this, the effort to compute the RSA-768 integer by multiplication of its factors is about 0.26 microseconds due to a rough estimation. Thus, the hardness of the integer factorization is obvious. For information and details of the number-field sieve-factoring method see the book of Lenstra and Lenstra [27] as this is not discussed here.

The above mentioned estimation with its assumptions is described in the following: The RSA-768 integer is a 768-bit integer with 232 decimal digits. Its factors are reported

as two 384-bit integers. For the multiplication, the product scanning form is assumed as the factors are too wide for commonly used multiplication units in processors. For details of product scanning multiplication see [19, Subsection 2.2.2, Page 32]. Due to this n^2 multiplications of b-bit integer words are needed where n is the number of words the integer is split into. Each multiplication is assumed to take one cycle whereas for the fetching of the operands, the storing of the results and the accumulation after the multiplication, three cycles are added per multiplication. The chosen word width of the multiplication unit is 64 bits because the AMD Opteron has 64-bit multiplication capabilities, as stated in the data sheet [1]. Therefore the 384-bit integers are split into 12 words which leads to $n = 12$ and further $12^2 \cdot 4 = 576$ cycles are estimated. On a 2.2 GHz processor with assumed $2.2 \cdot 10^9$ instructions per second, the whole multiplication will take about 0.26 microseconds. A strong emphasis lies on the fact that this is a rough estimation and the real execution time of the multiplication depends on the possible optimization and the necessary overhead due to the architecture. As this estimation is thought to illustrate the contrast between computing the result of an one-way function and reversing it, this rough estimation is considered sufficient.

2.2.5 Elliptic Curve Discrete Logarithm Problem

The elliptic curve discrete logarithm problem is based on the discrete logarithm, more precisely it is the discrete logarithm with the group of points on elliptic curves over a finite field. As Hankerson, Menezes and Vanstone write in [19, Chapter 1, Page 1], the use of elliptic curves in cryptography was suggested independently by Neal Koblitz [25] and Victor Miller [28]. In the following paragraphs, the basic mathematical relations are discussed, starting with groups, continued with finite fields and the discrete logarithm. Here, elliptic curves over finite fields are developed starting from a geometric point of view. Afterwards the base is built for the discussion about elliptic curve discrete logarithm problem. Finally, curve types and coordinate systems are discussed in short. The mathematics discussed here can be read in *Guide to Elliptic Curve Cryptography* of Hankerson, Menezes and Vanstone [19, Subsections 1.2.3 and Section 2.1], *Combinatorial group theory* of Karrass, Magnus and Solitar [23, Section 1.1] and *Elliptic Curves: Number Theory and Cryptography* of Lawrence Washington [40] for further details and explanations, and more specific references will provided alongside the discussion.

2.2.5.1 Groups And Finite Fields

The term group here is meant in a mathematical sense: A group is defined as a set of elements and an operation that can be carried out. For the group $(G, \cdot)$, G is the set of elements and $\cdot$ is the binary operation. Groups and their properties will be discussed in the following as in [23, Section 1.1]. A group $(G, \cdot)$ is called finite if the number of elements in the set G is finite. The binary operation takes two elements of the set. Since $(G, \cdot)$ is a group, the following four properties are satisfied:

I Given an ordered pair a, b of elements of the set G, a third element c is uniquely determined by $a \cdot b = c$. This is often written as $ab = c$ where the operation's sign is omitted.

II The binary operation $\cdot$ of the group $(G, \cdot)$ is associative: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.

III In the set G an element $e \in G$ exists, such that $e \cdot a = a \cdot e = a$ for all $a \in G$. e is called the *identity* or *unit* and is also written as 1.

IV In the set G, for each $a \in G$ an element $b \in G$ exists, such that $a \cdot b = b \cdot a = 1$. This element is called the inverse[3].

If the following fifth property is satisfied, the group is commutative and is called an abelian group.

V For each $a, b \in G$ of $(G, \cdot)$, $a \cdot b = b \cdot a$ is true.

Suppose we have a finite multiplicative group $(G, \cdot)$ of order q, which means the set of the group has q elements. [19, Page 12] states for an element $g \in G$ a smallest integer t exists, such that $g^t = 1$. Then it is said that t is the order of g. It is stated that for every element $g \in G$ such a t exist. The element $g \in G$ form a cyclic subgroup of $(G, \cdot)$ with the powers of g as the elements of the set and the same binary operation as G. The elements of the group are written as $\langle g \rangle = \{g^0, g^1, ..., g^{t-1}\}$ and the group $(\langle g \rangle, \cdot)$ is called a cyclic subgroup of G generated by g. The same applies to finite additive groups, but then $gt = 0$, which means adding t copies of g, and $\langle g \rangle$ is the set of multiples of g. If an element $g \in G$ exists, such that the order of the element is equal to the order of the group, the element is called a generator of G.

Groups are a part of the abstract algebra and might seem to be a theoretical construct with no benefit. But the integer mathematics can be expressed in groups. In context of the logarithm, the multiplicative group of integers can be replaced by another multiplicative group. If this group is chosen appropriately it results in increased hardness of computing the result of logarithm. Before discussing this in more detail, some more necessary mathematics will be discussed to complete the base of the elliptic curve discrete logarithm problem.

To describe the elliptic curve discrete logarithm problem it is necessary to discuss finite fields. A finite field over the prime p is the triple $(\mathbb{F}_p, +, \cdot)$ with a finite number of elements as discussed in [40, Page 482]. The elements of the set $\mathbb{F}_p$ are the positive integers modulo p: $\mathbb{F}_p = \{0, 1, ..., p-1\}$. A finite field has two operations, with symbols + and $\cdot$, called addition and multiplication. With each of these operations a group is defined and will be discussed in the following:

As discussed in [19, Subsection 1.2.3], the addition operation + together with the set $\mathbb{F}_p$ form the additive group $(\mathbb{F}_p, +)$ which is an abelian group. Thus $a, b, c \in \mathbb{F}_p : a + b = c$ and c is the result of the integer addition modulo p: $c = a + b$ modulo p. Further, 0 is the identity and $-a$ is the inverse of $a \in \mathbb{F}_p$. Thus, given $a \in \mathbb{F}_p$, $a + 0 = a$, and $a + (-a) = 0$. The multiplication operation with all nonzero elements $\mathbb{F}_p^*$ of the same set form a multiplicative group $(\mathbb{F}_p^*, \cdot)$. Just as the additive group, the multiplicative is abelian too. Here $\mathbb{F}_p^*$ is used instead of $\mathbb{F}_p$ as there is no inverse for zero and $\mathbb{F}_p^*$ is the set of positive, nonzero integers modulo p, $\mathbb{F}_p^* = \mathbb{F}_p \backslash \{0\}$. The binary operation is the integer multiplication modulo p: $a, b, c \in \mathbb{F}_p^* : a \cdot b = c$ and thus the result is calculated as $c = a \cdot b$ modulo p. In this group the identity is 1 and the inverse of $a \in \mathbb{F}_p^*$ is written as a^{-1}: given an $a \in \mathbb{F}_p^*$, $a \cdot 1 = a$ and $a \cdot a^{-1} = 1$.

Some more properties apply to finite fields and are listed in the following list. The interested reader might read [19, Section 2.1] for a detailed discussion.

[3] If the group$(G, \cdot)$ is an multiplicative group the inverse of a is written as a^{-1}, in case of an additive group it is written as $-a$.

- For a finite field it is required that the identity elements of the additive and multiplicative group are distinct.
- The multiplication distributes over the addition: Given $(\mathbb{F}_p, +, \cdot)$ then $a, b, c \in \mathbb{F}_p$: $(a + b) \cdot c = a \cdot c + b \cdot c$.
- For each prime p a finite field $(\mathbb{F}_p, +, \cdot)$ exists.
- The addition of an additive inverse might also be written without the addition sign: Given $(\mathbb{F}_p, +, \cdot)$ and $a, b \in \mathbb{F}_p$, then $a + (-b)$ might be written as $a - b$ and referred to as subtraction.
- Similar to addition, the multiplication with the multiplicative inverse $a \cdot b^{-1}$ might be written as a/b and referred to as division.

Beside the mathematical definitions and properties, addition, multiplication and their counterpart operations in finite fields behave like ordinary integer with a subsequent modulo operation on the result.

2.2.5.2 The Discrete Logarithm Problem

The name elliptic curve discrete logarithm problem already indicates its relation to discrete logarithm. But before discussing directly the elliptic curve discrete logarithm problem itself, the discrete logarithm problem will be discussed here in a basic manner. The discrete logarithm is similar to the ordinary logarithm, the solution for x in the equation $a^x = b$, but in contrast operates on a finite cyclic group. The group is assumed to be written as multiplicative group. A discussion about discrete logarithm can be read in [40, Chapter 5] and [19, Section 1.2] for discussion about discrete logarithm systems.

The fact that the discrete logarithm operates on finite cyclic groups leads to some advantage over the ordinary logarithm in the context of computational hardness: The ordinary logarithm allows using the already calculated powers of a to make bigger "jumps". For instance while testing the first 20 powers of a, these are saved. After this, the previous result a^i is not multiplied by a but a^{20} and then tested whether b is still bigger. If this is the case, the time doing 19 multiplications was saved. If not, a^i can be multiplied with a^{10} and tested again, which gives a direct hint if either x is between i and $i + 10$ or between $i + 10$ and $i + 20$. In contrast to this, the finite cyclic group prohibits this shortcut. Take the group of positive integers less than 17 with the group operation defined as the integer multiplication modulo 17, $(\{0, 1, ..., 16\}, \cdot)$. Here $4^2 = 4 \cdot 4 = 16$ whereas $4^3 = 13$ as $64 \bmod 17 = 13$. As in the general case, observing the results of the group operations does not give a clue of how many operations are sill necessary. The alternative strategy left is to calculate the powers of $a : a^i$ with increasing i and comparing the result with b. This is done by calculating the group operation $a^{i-1} \cdot a = a^i$ starting with $a^1 = a$. Thus, this requires $x - 1$ group operations. Therefore, beside the value of x, the hardness of the discrete logarithm problem depends on the used group and the complexity of its operation.

2.2.5.3 Elliptic Curves Over Finite Fields

In this section some aspects of elliptic curves will be discussed. Elliptic curves will be developed from a geometric view of elliptic curves, to a group law and end with elliptic curves defined over finite fields. The discussion in this section refers to Washington's

Elliptic Curves: Number Theory and Cryptography [40] and detailed reference will be provided in the text.

In the context of visualization of elliptic curves, we assume these are defined over real numbers. By doing so, we get more intuitive visualizations. Elliptic curves are plane curves as an elliptic curve in affine coordinates has two coordinates. The name might let one think of shapes with an elliptic form, but it is derived from a mathematical relation. Silverman in [35, Chapter VI, Page 157] writes therefore that elliptic curves " ... are the Riemann surfaces associated to the arc-length integrals of ellipses". Elliptic curves do not look like ellipses in general as shown in Figures 2.2 and 2.3.

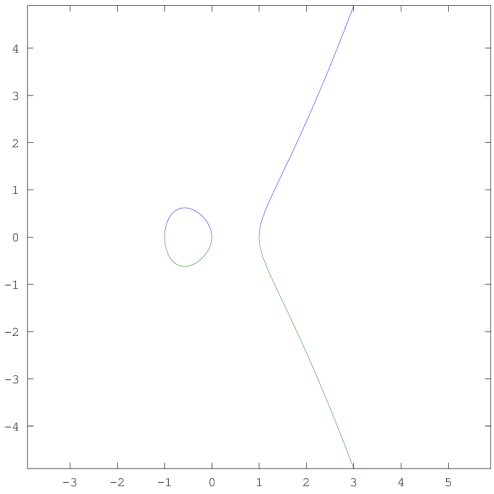

Figure 2.2: Curve $y^2 = x^3 - x$

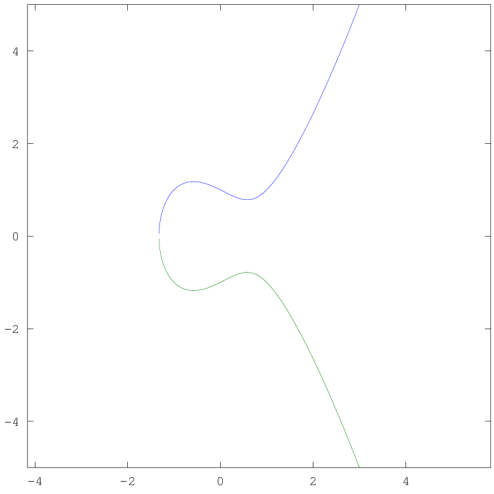

Figure 2.3: Curve $y^2 = x^3 - x + 1$

Points on it have affine[4] coordinates, x and y, and satisfy the Weierstrass equation[5] $y^2 = x^3 + ax + b$, where x and y are the x coordinate and respectively the y coordinate, a and b are constants determined by the used elliptic curve. Further for the elements x, y, a, b in the equation, it has to be specified to which set they belong. As a, b are specified to be elements of the set of a field K, the elliptic curve is said to be an elliptic curve over K. x, y specify the coordinates of a point on the elliptic curve and are of the set of the field L, with $L \supseteq K$. For technical reasons the point at infinity is included in L: $\infty \in L$. This has advantages when a third point is calculated out of two known points: Due to Bezout's theorem an elliptic curve intersects with a line at exactly three points. For fully respecting this theorem some special cases have to be discussed. The intersections can be one of four types and are visualized in Figure 2.4.

The general case, when the line is neither vertical nor a tangent to the curve, is shown by the blue line in Figure 2.4. As long as none of the two points is a tangent and the resulting line is not vertical, the angle and the position of the line has no restrictions. Then it intersects the curve at 3 points.

If the line is geometrically a vertical line, the intersection is defined to be at infinity, see the magenta and the red line in Figure 2.4. For details about the intersection at infinity see [40, Sections 2.1, 2.2 and 2.3].

If the line is a tangent to the curve, see the green and magenta line in Figure 2.4, the point is counted twice and the line intersects the curve at exactly one more point, either at infinity (magenta) or "normal" (green).

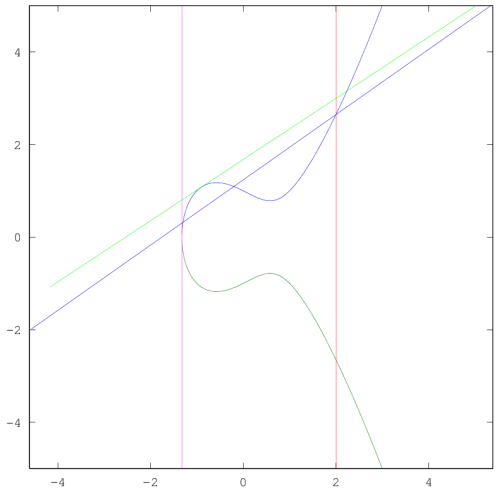

Figure 2.4: Curve $y^2 = x^3 - x + 1$ with 4 different intersections

As a consequence of the y^2 in the equation, an elliptic curve is symmetric to the x axis: By the basic rules of mathematics, for calculating y out of y^2, it must be written as $y = \pm\sqrt{y^2} = \pm\sqrt{x^3 + A \cdot x + B}$. The negative of a point $P_1 = (x_1, y_1)$ is defined to be $-P_1 = (x_1, -y_1)$.

[4]The representation of points can done in other coordinate systems. This is discussed in Subsection 2.2.5.6.

[5]This is the Weierstrass equation for elliptic curves which do not have a characteristic of 2 or 3. A more general equation and discussion for those two cases can be found in [40, Equation 2.1, Page 10].

By examining the intersection with a line we can define a point-addition operation on elliptic curves, $P_1 + P_2 = -P_3$ or $P_1 + P_2 + P_3 = 0$.

1. **General case:** A line through two points, $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, on the curve, intersects the curve at one more point if none of the points is a tangent and the line is not vertical. The line has a slope $s = \frac{y_2 - y_1}{x_2 - x_1}$ and the third point $P_3 = (x_3, y_3)$ can be calculated with $x_3 = s^2 - x_1 - x_2$ and $-y_3 = s \cdot (x_3 - x_1) + y_1$. The mathematical background for this is to find the third root of the equation. This is eased by the knowledge of the other two roots. Details of the mathematics can be found in [40, Section 2.2].

2. **Tangent:** If two points on a curve are very close together, a line through them converge to a tangent of the curve. As a consequence here, if the point is added to itself, the line is a tangent of the curve at this point. The tangent is the first derivation of the equation at this point. Here it is assumed that the line is not vertical. Again, the line intersects the curve at exactly one other point. By implicit differentiation of $\frac{d}{dx}((y)^2) = \frac{d}{dx}(x^3 + A \cdot x + B)$ one gets $2y \cdot \frac{dy}{dx} = 3x^2 + A$ and $\frac{dy}{dx} = \frac{3x^2 + A}{2y}$. As the x, y coordinate of this point is available, the slope s of the line is $s = \frac{dy}{dx} = \frac{3x_1^2 + A}{2y_1}$. When proceeding as in the previous case, but with $P_1 = P_2$, we get $x_3 = s^2 - 2x_1$ and $-y_3 = s(x_3 - x_1) + y_1$.

3. **Vertical case:** If the x coordinates x_1, x_2 are equal, the line through the two points is a vertical line. Here the y coordinates y_1, y_2 are not equal. As mentioned before, ∞ is added to the points on the curve and is said to sit on the top of the y axis. If then a line connecting two points is vertical, the line intersects the curve at ∞, $P_3 = \infty$. By the circumstance that an elliptic curve is symmetric with respect to the x axis, $y_2 = -y_1$, thus $P_2 = -P_1$ and $P_1 - P_1 = \infty$

4. **Vertical tangent:** This is case 2 and 3 combined, $x_1 = x_2$ and $y_1 = y_2 = 0$. First, the line is a tangent as both points have the same coordinates and second, this tangent is vertical as both points' x coordinate are equal. Since the line is a tangent, the point of the intersection is counted twice and as it is a vertical line, the third intersection point is at ∞, $P_3 = \infty$.

Recapitulating the properties of a group as discussed in Subsection 2.2.5.1, the properties I, III and IV of a group and further the property V of an abelian group can be shown to be fulfilled by the point-addition operation with ∞ as the identity element. Showing that property II, the associativity, is fulfilled is rather complicated and would exceed the introductive manner of this section. The interested reader might examine [40, Section 2.4, Pages 20-35], who proves associativity by examining the intersections of lines through three points and handling the different types of intersections of a line and an elliptic curve.

I Two points on the curve can be added by use of the above discussed cases.

III Adding the identity ∞ to a point P_1, $P_1 + \infty = -P_3$, results in $P_3 = P_1$ and is directly obtained from the above discussed "Vertical case".

IV As III, using the "Vertical case" and the circumstance that $P_2 = -P_1$ is a valid point on the curve, $P_1 - P_1 = \infty$.

V A line through two points remains the same line, regardless which of the points is the starting point and the end point, therefore $P_1 + P_2 = P_2 + P_1 = P_3$ holds.

The points on an elliptic curve and the point addition operation form a group as defined before in Subsection 2.2.5.1. The operation is usually called "the addition of points on an elliptic curve". In the later part of this thesis, the multiplication of a point P with a scalar n, often referenced as scalar multiplication or point multiplication, will be done by adding n times the point P.

We discussed the intersection of a line through points on an elliptic curve over real numbers and developed the group structure of points on an elliptic curve with the point addition as its binary operation. We may now continue with elliptic curves over finite fields. A, B, x, y are now elements of this field. This means a point $P_1 = (x_1, y_1)$ has the coordinates as elements of this finite field. Calculations are done within this field and remain valid although the group of real numbers is replaced by a finite field. One may think of layers for the calculation and caution is in some sense necessary, as the operations' signs are similar but do not mean the same operation: $P_1 + P_2$ stands for the point addition of the group of points on an elliptic curve. For calculation this means to proceed as defined beforehand, for instance in the general case, calculating $s = \frac{y_2 - y_1}{x_2 - x_1}$, $x_3 = s^2 - x_1 - x_2$ and $-y_3 = s \cdot (x_3 - x_1) + y_1$. The division, squaring, multiplication, subtraction and addition stated in the formulas of the point addition for s, x_3, y_3 is meant to be done in the field over which the elliptic curve is defined. In case of finite field over $\mathbb{F}_q$ a prime q, regarding s, this results in doing the numerator's finite-field subtraction, the finite-field inversion of the result of the denominator's finite-field subtraction and the finite-field multiplication of both intermediate results. This similarly applies to x_3 and y_3. These finite-field operations, as discussed in Subsection 2.2.5.1, in turn consist of integer operations with additional modulo operations, for instance $y_2 - y_1 = integer\_value(y_2) - integer\_value(y_1) \bmod q$. Note, that the "integer_value" is included to distinguish between the elements of the finite field and integer numbers.

As [40, Chapter 4, Page 95] states, elliptic curves over finite fields have only a finite number of valid points, as the finite field only has a finite number of possible combinations for the coordinates x, y of a point $P = (x, y)$. It is further stated, that the exact number of valid points is difficult to compute for big finite fields, but increases with the number of elements in the field.

Assuming a point P on an elliptic curve over a finite field $E(\mathbb{F}_q)$ and ∞ as the identity element. As discussed in Subsection 2.2.5.3, a t exists and t is the smallest integer such that $t \cdot P = \infty$. In this manner cyclic subgroups generated by elements of order q can be defined.

2.2.5.4 The Discrete Logarithm With Elliptic Curves

After discussing the discrete logarithm in general and elliptic curves over finite fields, we continue with the discrete logarithm with a cyclic subgroup of the group with the set of points on an elliptic curve defined over a finite field over a prime. The name already indicates the involved mathematics. In the following, the necessary steps and the layers of computation are outlined and the main aspects are recapitulated in short to give a rough overview. For a more detailed discussion see the referred sections and the referenced sources.

To proceed top down, we start with the discrete logarithm. This is to find a solution for k in the equation $a^k = b$, with given a, b as we discussed in Subsection 2.2.5.2. As the

structure of computed data permits shortcuts, this is done by computing the powers of one after another and comparing these results with the specified result. The next topic is the group to which the elements a, b belong to, that leads to the cyclic subgroup of the group of points on an elliptic curve with the point addition as the group's binary operation and was discussed in the previous Subsection 2.2.5.3. In this group, copies of the generating point on the elliptic curve are summed up for calculating a multiple of a point, the scalar multiplication. As a consequence of the circumstance that the group operation is addition and not multiplication, the discrete logarithm here computes k out of $a \cdot k = b$. The scalar multiplication, see Subsection 2.2.5.3, in turn, stands for several finite-field operations, as previously stated in Subsection 2.2.5.1. These operations cover the basic usual mathematical operations, addition, subtraction, multiplication, inversion and negation, and are computed as their pendants in usual mathematics with a trailed modulo operation.

Regarding the scalar multiplication, the drafted summing up of copies of a specific point would results in a repeated point addition, therefore calculating kP would need $k - 1$ point addition operations. That can be shortened by a "double and add". This procedure takes P as the initial value of the intermediate result R, starting with the most significant bit being 1 and testing each bit down to the least significant, R is doubled and, according to whether the bit is either set 1 or 0, either P is added to the intermediate or not. For instance we perform the scalar multiplication of a point with 19. The most significant bit set corresponds to $2^4 = 16$, therefore the bits for $2^3 = 8$, $2^2 = 4$, $2^1 = 2$ and $2^0 = 1$ are tested. For illustration the index of R will be increased for each bit tested, starting with $R_0 = P$. The first bit tested, 2^3, is not set 1, so $R_1 = 2 \cdot R_0$. The same applies to 2^2, so $R_2 = 2 \cdot R_3$. The bits for 2^1 and 2^0 are both 1, so the intermediate and afterwards the final result get $R_3 = 2 \cdot R_2 + P$ and $R_4 = 2 \cdot R_3 + P$. Here, instead of 19 additions, the result is obtained after four doubles and two additions.

2.2.5.5 Curve types

The family of elliptic curves is not limited to curves in Weierstrass form. Due to the introductive manner of this section, besides the already mentioned and analyzed Weierstrass curves, twisted Edwards curves will be discussed. As a central topic in this thesis, see the next Chapter 3 for further discussion on twisted Edwards curves.

Different types of curves are expressed by different equations defining such a curve. Those types differ in the way the point addition is done, as discussed in Subsection 2.2.5.3, its formulas are derived from the curve equation. Certain formulas for the point addition of a curve type can have benefits over other types regarding various disciplines. While one type can be the fastest in computation on certain specific architecture and a specific hardware, another type can have attack-resistance advantages over others.

As different types are still part of the family of elliptic curves, it may seem reasonable that some curves of one type may be convertible to other types. This is not true in general, but for instance, as Bernstein, Birkner, Joye, Lange and Peters stated in [11], for every Montgomery curve a twisted Edwards curve can be calculated. As a consequence, it is possible for curves to be converted into another type to take advantage of this type's properties. See Figure 2.5 for the visualization of a twisted Edwards curve and Figure 2.6 for a Montgomery curve. For using such a conversion, one converts the curve equation and the input point for the scalar multiplication. Then the result of the scalar multiplication can be calculated using the new curve's point-addition formulas and then finally the point

is converted back to the original curve.

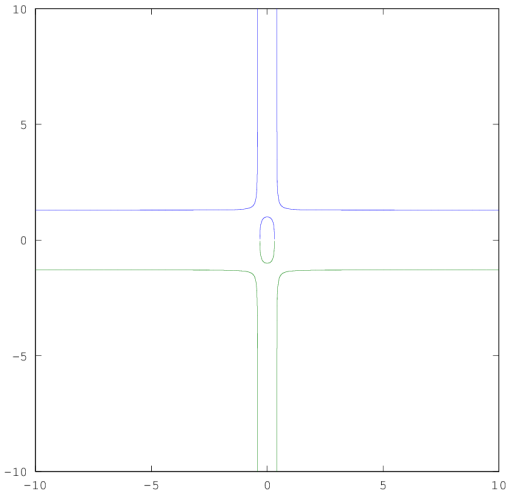

Figure 2.5: Twisted Edwards curve $10x^2 + y^2 = 1 + 6x^2y^2$

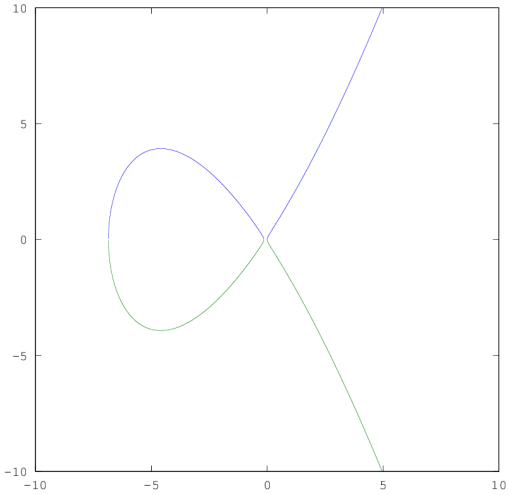

Figure 2.6: Montgomery curve $3y^2 = x^3 + 10x^2 + x$

2.2.5.6 Coordinate systems

As mentioned in Subsection 2.2.5.5, different types of curves can provide certain advantages. Similar to this, coordinate systems allow, for instance, to optimize the speed of calculation for a particular architecture or hardware. The chosen coordinate system also has an influence on the space requirements during calculation as different coordinate systems store additional values. The coordinate system has a direct impact on the point-addition

formulas. Therefore the changes relative to affine coordinates will be discussed. The discussion in this section should be understood in the context of elliptic curves and thus be read as a rough overview. The interested reader can obtain further and more detailed discussion in [19, Section 3.2].

The affine coordinate systems in the context of elliptic curves lets a point on an elliptic curve have an x and a y coordinate. Affine coordinates here are used as reference to compare other coordinate systems with.

Projective coordinates introduce an additional z coordinate and the usage of this can vary. When using the standard projective coordinates, the z coordinate is used to save calculations of inversions. Instead of performing the inversion in every point-addition operation, the z coordinate is used to accumulate the denominators. The only necessary inversion is when calculating back to the affine coordinates. This inversion is rather expensive in the context of the computational effort. The benefit of this depends on the actual architecture and hardware. For example, the hardware implemented with 32 bits word width of this thesis needs 73 384 cycles for doing one inversion in the finite field, in contrast to one finite-field multiplication with 143 cycles. Further hardware implementation details will be presented in Chapter 4.

There are various other coordinate systems with certain benefits and special purposes. Beside the affine and the standard projective coordinates, Jacobian projective coordinates are mentioned to point on the diverse possibilities arising from the coordinate system. Within Jacobian projective coordinates $(X : Y : Z)$ each affine point is represented as $(x, y) = (X/Z^2, Y/Z^3)$. This coordinate systems has an impact on the curve equation. For detailed discussion on this see [19, Section 3.2].

2.3 Security Measurement

The security of a cryptographic scheme is an important property. Assuming a scenario as sketched in Section 2.1, Alice and Bob want to communicate in such a way that Eve cannot read their messages. Further, let's assume Eve has the considerable knowledge of cryptography and computational resources. Eve then chooses the best known procedure to reverse the encryption and to read the decrypted messages. Depending on the cryptographic scheme Alice and Bob use, this takes a specific amount of time. This time depends on the available computational power under Eve's control. More computational power shortens the time, thus time seems not to be a good measurement. The number of operations necessary to reverse the encryption, however, is independent of computational power. Therefore a valid measurement on how secure a scheme is can be the necessary operations with the best known algorithm.

Besides this there is a similar measurement that is based on the necessary amount of work and gives the security of a cryptographic scheme in bits. In National Institute of Standards and Technology's (NIST) recommendation regarding key management [4, Section 5.6], the amount of necessary work to break a cryptographic scheme is compared with the necessary amount of work performing a brute force attack on a symmetric key cryptographic scheme with X-bit key size. The cryptographic scheme is then said to have a security of X bits. The levels of security listed in Revision 3 of NIST's recommendation in [4] are 80[6], 112, 128, 196 and 256. For the security levels 128, 196 and 256 the AES,

[6] In NISTS's Special Publication in [5, Section 1.2, Page 6], it is stated to use 80 bits security until the end of 2013 with acceptance of a certain amount of risk.

Security level	Symmetric key algorithm	Integer factorization	Elliptic curve cryptography
128 bits	AES 128	RSA, key size 3 072 bits	ECDSA, key size 256-383 bits
196 bits	AES 196	RSA, key size 7 680 bits	ECDSA, key size 384-511 bits
256 bits	AES 256	RSA, key size 15 360 bits	ECDSA, key size 512+ bits

Table 2.1: Excerpt of NIST's comparison of cryptographic schemes.

with these specified key sizes in bits, is used for comparison.

NIST's recommendation includes comparable security levels for RSA, see discussion about RSA in Subsection 2.2.4. The cryptographic scheme ECDSA, based on the elliptic curve discrete logarithm problem discussed in Subsection 2.2.5, is also included. In the following Chapter 3 the EdDSA scheme will be discussed. This relies on the same mathematical problem as the ECDSA does and is therefore considered to have the same security level. An excerpt of NIST's table can be seen in Table 2.1. For the full table see [4, Page 63] and for a more detailed discussion see the surrounding section [4, Section 5.6].

The table shows that for RSA, in order to achieve the same security as AES, a quite high key size is necessary. For elliptic-curve cryptography the table states smaller key sizes, approximately twice of the security level.

2.4 Summary

We discussed basic ideas of cryptography and discussed aspects of symmetric-key cryptography as well as public-key cryptography. We took a look on elliptic curves and the discrete logarithm and discussed the mathematical backgrounds the elliptic curve discrete logarithm problem implies.

Chapter 3

Edwards-curve Digital Signature Algorithm

In this chapter, we will take a look at the Edwars-curve digital signature algorithm (EdDSA). It was introduced by Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe and Bo-Yin Yang in 2011 [8]. EdDSA is a variant of ElGamal's signature system [14] just as Schnorr's signature system [33], the Digital Signature Algorithm (DSA) and ECDSA. The most obvious difference to other signature systems is that EdDSA uses twisted Edwards curves rather than Weierstrass curves. The use of this type of curves explains the choice of name EdDSA: **Ed**wards-curve **D**igital **S**ignature **A**lgorithm. The authors do not claim any novelty of their modification with respect to other signature systems but emphasize the importance of the selection of a good combination of modification to achieve top performance. EdDSA is a generalization of the introduced and discussed explicit signature system Ed25519-SHA512. EdDSA can be used with other choices of elliptic curves and other parameters.

First, the parameters will be presented in Section 3.1. For a subset of the parameters further discussion will be provided. Second, the operations on the protocol layer will be presented in Section 3.2. Afterwards, security consideration will be made in Section 3.3 and a comparison to the ECDSA will be provided in Section 3.4. Finally, the Ed25519 is discussed in Section 3.5: The set of chosen parameters will be presented and the mathematical backgrounds and techniques involved will analyzed.

3.1 EdDSA Parameters

The authors of EdDSA [8] define various necessary parameters that have to be fixed. After listing these, a limited choice will be discussed in particular.

The parameter $b \geq 10$ is subsequently used to define length in bits of other parameters. This parameter has an influence on the used hash function, the primes, the necessary arithmetic operations and their word width, and the digital signature's length. H stands for the cryptographic hash function with a $2b$-bit output. Hash functions will in short be discussed in Subsection 3.1.2 and will be used for generating the pairs of keys, the signature generation and verification. These operations will be discussed in Section 3.2. The prime q is used for the elliptic curve's underlying finite field $\mathbb{F}_q$. This prime influences the arithmetic, especially the computation of the reduction in the finite field. The $(b-1)$-bit encoding of elements of the finite field $\mathbb{F}_q$ will be used for packing the xy coordinates of a

point. This procedure will be explained in Subsection 3.1.4. The parameter d, used in the curve equation, has to be a non-square element of $\mathbb{F}_q$. There is no solution for a non-square element in $\mathbb{F}_q$ for the equation $d = x^2$. The prime ℓ between 2^{b-4} and 2^{b-3} is used within an additional constraint, which will soon be discussed. It is further used for the signature generation and verification calculations. Base point B is used for scalar multiplication fulfilling $B \neq (0, 1)$ of the set $E = \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q : -x^2 + y^2 = 1 + dx^2y^2\}$. The constraint mentioned above is: When choosing parameters ℓ and B, $\ell B = 0$ must be satisfied. Here ℓB means the ℓth multiple of B. In Subsection 2.2.5.1, cyclic subgroups with a generating element have briefly been discussed. Here B forms such a cyclic subgroup with a set $\langle B\rangle = \{0B, B, ..., (\ell - 1)B\}$

3.1.1 Curve

As already mentioned, a major difference to other digital signature algorithms, e.g., ECDSA is the use of twisted Edwards curves [11]. By a slight modification with respect to the curve equation of Edwards curves (see the article of Edwards [13]), $x^2+y^2 = 1+dx^2y^2$, twisted Edwards equation are defined with the equation $ax^2 + y^2 = 1 + dx^2y^2$. Twisted Edwards curves' coverage of elliptic curves that can be transformed from Weierstrass form is significantly increased. They also cover all Edwards curves. Such a transformation was in short discussed in Subsection 2.2.5.5. These curves can further be used with the fast and unified addition-law (for details see Subsection 3.5.1.3). The term "unified" means that it can be used for addition as well as for doubling. Within the EdDSA, the parameter a of the twisted Edwards curve equation is fixed: $a = -1$.

3.1.2 Hash Function

Hash functions are used to generate a "fingerprint" as output for input data. This fingerprint is also called a "hash". The input does not have to be of a certain length. A hash function has a fixed output length which can be truncated if less output data is required. The NIST published a recommendation of hash functions in [15]. These are of the SHA family with varying hash lengths and therefore a hash function with appropriate length can be chosen out of these.

The hash function maps each possible combination of input bits to a certain output. The output of a hash function changes if a single bit flips and a single flipped bit will propagate over the whole range of output bits. Ideally, when flipping a single bit, the changes in the output are uniformly distributed over the whole range.

In general a hash function computes its output iteratively, where one part of the input I_i of the current iteration i is the output of the previous iteration O_{i-1}. The other part is a fixed-length part of the hash function's input, using all data of input during the iterations. The number of iterations is fixed for a certain hash function. Hashes are also used for verifying the data integrity. Hypothetically, if a hash function would not have a fixed number of iterations, data integrity based on hashes could not be verified.

Assume an input that can be any combination of bits with a fixed length. A hash function has a fixed output length. As the input has more possible combinations than the output, the hash function maps certain inputs to the same output. This is called a hash collision. Although such collisions are unavoidable due to the limitations of the output length, finding such a collision may not be trivial. For a hash function, none of the following tasks should be computable in considerable time because this could be critical for the security of the application:

- For a given hash h, finding an input m such that $hash(m) = h$.
- For a given input m_1, finding another input m_2, such that $hash(m_1) = hash(m_2)$.
- Finding any two different inputs m_1 and m_2, such that $hash(m_1) = hash(m_2)$.

In the context of EdDSA, the hash function is used for several activities: Generating the secret key as well as the public one, calculating the signature and verifying a given signature and its message. Like Schnorr's signature system [33] and in contrast to ElGamal's and some variants of ElGamal's signature system, EdDSA is not vulnerable to attacks by merely finding hash-collision (see the paper [8, Page 131]). This improvement is achieved by hashing not only the actual message, but using additional secret data. This additional data varies depending on the context and is either a part of the hashed secret key or the public key and the base point B multiplied with some scalar. Further discussion about this is provided in Section 3.3.

3.1.3 Primes

A prime is an integer number which has no divider other than 1 and the prime itself, as already discussed in Subsection 2.2.4. Further, discussed in Subsection 2.2.5.1, for each prime q a finite field over this prime $\mathbb{F}_q$ exists.

Some primes enable a faster calculation in a finite field over these certain primes. These are called Mersenne primes and are of the form $M_n = 2^n - 1$. Beside the rare Mersenne primes, also pseudo Mersenne primes exist, which enable fast calculation too. These are of the form $q = 2^m - k$ with $0 < |k| < 2^{\lfloor m/2 \rfloor}$, see the article of Jerome A. Solinas in [37]. Mersenne primes and pseudo Mersenne primes enable fast reduction used in the context of finite-field multiplication: The multiplication of two elements of a finite field $\mathbb{F}_q$ over a pseudo Mersenne prime $q = 2^m - k$ with each element $\forall e \in \mathbb{F}_q : 0 \leq e < q$. Details of the procedure will be discussed in Subsection 3.5.1.2. Calculating the modulo, involving rather big numbers, is expansive in the context of computation time as stated in [19, Subsection 2.2.4]. Here, this can be traded in by some integer operations: multiplications, additions and subtractions. More precisely, there are $\lceil m/n \rceil + \lceil (s+1)/n \rceil$ single-word-integer multiplications necessary, where n is the input width of the multiplication unit and $s = \lceil log_2(k) \rceil$. Further, there are $\lceil (m+n+1)/o \rceil + \lceil m/o \rceil$ single-word-integer additions and $2 \cdot \lceil (m+1)/t \rceil$ single-word-integer subtractions, where o is the input width of the adder unit and t the input width of the subtraction unit. These are the results of the discussion in Subsection 3.5.1.2.

In the context of EdDSA, the primes q and ℓ are used. The have certain requirements as described in the following: The prime q of the finite field for the elliptic curve is required to be $q \equiv 1 \pmod 4$. Each point on the twisted Edwards curve has an x and y coordinate: $\forall x, y \in \mathbb{F}_q : \{0, 1, ..., q-1\}$. If the point is given in projective coordinates this applies to the z-coordinate too. ℓ, in turn, is used for the constraint regarding the parameter base-point B and further when calculating the b-bits $\underline{S}$ of the $2b$-bit signature $(\underline{R}, \underline{S})$: $\underline{S} = little\_endian\_enc(f(secret\_key, public\_key, message) \bmod \ell)$. See Subsection 3.2.1 for details.

Implicitly, as a $b-1$-bit encoding of a coordinate is necessary and an element of $\mathbb{F}_q$ represented by an integer needs $\lceil log_2(q-1) \rceil$ bits, q can have at most $b-1$ bits.

3.1.4 b-Bit Point Encoding

The b-bit point encoding is used for generating a point representation that can be transferred to other systems. Therefore, the encoding must be well defined, otherwise the verification of a signature could not be done, failing to calculate the base point B. The point encoding further compresses a point with x and y coordinates, each coordinate being an element of $\mathbb{F}_q$ and, as discussed in the previous Subsection 3.1.3, needing $b-1$ bits. The compression reduces the length in bits from $2(b-1) = 2b-2$ to b.

Each point in affine coordinates $P \in E = (x, y)$ can be encoded as b-bit string $\underline{P} = \underline{(x, y)}$ with y in the $(b-1)$-bit encoding and x giving the missing bit, which is either 1, if x is negative, or 0 if it is negative. A point P given in this encoding: y can be directly read by taking $(b-1)$ bits, x can easily be recovered by solving $x = \pm\sqrt{(y^2-1)/(dy^2+1)}$ and taking the solution according to the b-th bit.

3.2 Signature Operations

In general, there are two operations regarding a digital signature. On the one hand, generating a signature of a message, called signing a message, on the other hand verifying a received signature for a received message. These two operations will be discussed in this section and algorithms for executing these operations will be presented in pseudo code. For signing a message and afterwards verifying it, a private key and a public key are necessary. These two keys must be generated once and can be reused for future signing operations.

3.2.1 Signature Generation

The signature generation for a given message M is shown below(Algorithm 2) in pseudocode(see [8, Chapter 2]).

Before generating a signature, one has to generate the secret and the public key. The secret key is a b-bit string k and does not have to be generated for each message. Thus a secret key can be used for several signatures. The algorithm for the generation of the keys is also shown below (Algorithm 1). By hashing the secret key k, an integer value a is determined. This is multiplied with the base-point B and b-bit encoded (see Subsection in 3.1.4) into $\underline{A}$. For public key generation only one half of the hashed secret key is used. The rest of the calculated hash $h_b, ..., h_{2b-1}$ is used while generating a signature of a message as additional input for the hash function beside the actual message. This method of hashing not only the message serves additional security as discussed in Section 3.3.

Algorithm 1 Key-pair generation

procedure KEYPAIRGEN(b)
  $k \leftarrow randomNbits(b)$ ▷ $k = b$-bit random value
  $(h_0, h_1, ...h_{2b-1}) \leftarrow H(k)$
  $a \leftarrow 2^{b-2} + \sum_{3 \le i \le b-3} 2^i h_i \in \{2^{b-2}, 2^{b-2}+8, ..., 2^{b-1}-8\}$
  $A \leftarrow a \cdot B$
  $\underline{A} \leftarrow b\_bit\_encoding(A)$
  return $k, h_b, ..., h_{2b-1}, \underline{A}, a$
end procedure

Algorithm 2 Signature generation

procedure SIGGEN($h_b, ..., h_{2b-1}, \underline{A}, a, l, B, M$)
  $r \leftarrow H(h_b, ..., h_{2b-1}, M)$ ▷ $\in \{0, 1, ..., 2^{2b} - 1\}$
  $R \leftarrow rB$ ▷ $\in E$
  $S \leftarrow (r + H(\underline{R}, \underline{A}, M) \cdot a) \bmod l$
  $\underline{R} \leftarrow b\_bit\_encoding(R)$
  $\underline{S} \leftarrow little\_endian\_encoding(S)$
  return $(\underline{R}, \underline{S})$
end procedure

3.2.2 Signature Verification

Given the public key $\underline{A}$, a message M and an alleged signature, the verification process is shown in pseudo-code in Algorithm 3. The message and its signature may be transmitted over an insecure channel. As long as it can be ensured that the public key has been originated by the sender, it can be verified that the intended sender has written and signed the message. The public key $\underline{A}$ of the sender is parsed for $A \in E$, the signature of the message is parsed for $R \in E$ and $S \in \{0, 1, ..., (q-1)\}$. If successful, the equation $8 \cdot S \cdot B = 8 \cdot R + 8 \cdot H(\underline{R}, \underline{A}, M) \cdot A$ is checked and only if it is satisfied the signature is accepted, otherwise it is rejected.

Algorithm 3 Signature verification

procedure SIGVERIFY($(\underline{R}, \underline{S}), \underline{A}$) ▷ Signature $(\underline{R}, \underline{S})$
  try
    $A \leftarrow parse\_for\_point(\underline{A})$
    $R \leftarrow parse\_for\_point(\underline{R})$
    $S \leftarrow parse\_little\_endian(\underline{S})$
    if $(8 \cdot S \cdot B = 8 \cdot R + 8 \cdot H(\underline{R}, \underline{A}, M) \cdot A)$ **then**
      return $ACCEPTED$
    else
      return $REJECTED$
    end if
  catch parseError
    return $REJECTED$
  end try catch
end procedure

3.3 Security Considerations

Beside the general security of elliptic-curve cryptography and the discrete logarithm problem, discussed in Section 2.2, some aspects were taken into account when EdDSA was developed. In this section they will be discussed. The discussion will concern the per-message generated ephemeral key and the input of the hash function within the calculation of the signature. The security level discussed in Section 2.3 is dependent on the chosen parameter. EdDSA is an asymmetric/public-key cryptography scheme and it is of the family of schemes based on the elliptic curve discrete logarithm problem. Therefore the

rule of thumb for elliptic-curve cryptography applies to EdDSA too: its security level is related to the chosen parameter b and is approximately $\frac{b}{2}$ bits.

One substantial security issue is the right implementation of the key generation. EdDSA uses randomness for the secret key generation, but does **not** use randomness to generate the session (ephemeral) key. The session key is calculated deterministically by utilizing the hash function and will be discussed in Section 3.4. The authors of EdDSA in [8, Pages 131-132] state that the secretly, deterministically obtained session key does not threaten the security as it almost has a uniform distribution modulo ℓ, and it is indistinguishable from a truly randomly generated session key. Further, it adds security in some sense since a bad implementation of a random-number generator does not harm the security and ensures a foolproof session key by design. In ECDSA, if the same randomly generated ephemeral key is ever used twice, the secret key can be computed. A prominent example, as [8, Chapter 1] notes, was Sony's ECDSA implementation of code-signing for the PlayStation3 that revealed Sony's long-term secret key. Further, as shown in [29], even the knowledge of a few bits of r for hundreds of signatures lets one gain knowledge of the long-term secret key. EdDSA is therefore resistant by taking the current message into the input of the hash function as well as the secret key when generating $r \leftarrow H(h_b, ..., h_{2b-1}, m)$, thus taking advantage of the distribution of the hash function.

Regarding the hash function, EdDSA further has the property to be resistant against hash collision. In contrast to ECDSA, which hashes the message when calculating the signature, stated in [16] and [21], but like Schnorr's signature system, EdDSA extends the input of the hash function. Schnorr hashes the message together with data calculated from the per-message random session key. This and further details can be examined in his paper in [32, Page 243]. EdDSA includes the public key into the calculation and therefore in [8, Page 132] it is stated that this is an inexpensive countermeasure against simultaneous attacks on multiple keys.

3.4 Differences between EdDSA and ECDSA

ECDSA is widely used and is standardized by ANSI in [2] and NIST approved, see [16], in contrast to EdDSA. They have a few aspects in common, beside its similar name. Both are digital signature algorithm and rely on elliptic-curve cryptography. EdDSA, however, has some modifications compared to ECDSA. These modification are made due to speed optimization and security consideration. They will be summarized in the following. The interested reader might examine the paper [8] which introduced EdDSA and contains detailed comparisons to other signature algorithms, the discussion about ECDSA of Johnson, Menezes and Vanstone in [21], ANSI's standard and the publication of NIST.

The most obvious difference is the usage of twisted Edwards curves in EdDSA over curves with Weierstrass form. As a consequence, the fast and unified addition law can be used for addition as well as for doubling. This in turn is a countermeasure against side channel attacks as addition and doubling cannot be distinguished.

As pointed out in [8], EdDSA is a variant of ElGamal's signature algorithm just as ECDSA. ElGamal in [14, Chapter 3] defined the message m to be in an integer interval $0 \leq m \leq p$ where p is a large prime. He uses m to calculate S of the signature (X, S). ECDSA in this sense replaces m with the output of the hashed message $H(m)$ when calculating S. EdDSA therefore adds further data when calculating the $\underline{S}$ part of the signature: ElGamal's m is replaced with $H(\underline{R}, \underline{A}, m)$, where $\underline{R}$ is the other part of the

generated signature $(\underline{R}, \underline{S})$. As discussed in Section 3.3 this is done to meet security concerns.

The modification aims in a similar direction regarding the session (ephemeral) key. ECDSA defines the session key to be randomly selected within 1 and the large prime n, where n is a domain parameter. EdDSA instead calculates this session key r out of the secret key and the actual message. This causes a security improvement and is discussed in Section 3.3. $r = H(h_b, h_{b+1}, ..., h_{2b-1}, m)$ is therefore calculated, with $h_b, h_{b+1}, ..., h_{2b-1}$ are determined by hashing the secret key, $H(k) = (h_0, h_1, ..., h_{2b-1})$.

3.5 Ed25519-SHA512

EdDSA was introduced by Bernstein, Duif, Lange, Schwabe and Yang in [8] as a digital signature algorithm with no fixed parameters but with constraints concerning the selection of them. In the same paper, they further proposed an "explicit" EdDSA with fixed parameters. In this section, the chosen parameters will be discussed and the possibilities of an implementation of the necessary arithmetic will be shown.

Ed25519-SHA512 (in short Ed25519) is a proposal of an Edwards-curve digital signature algorithm with carefully chosen parameters. The curve chosen is birationally equivalent to Bernstein's Curve25519 from [9]. This Curve25519 is the Montgomery curve $v^2 = u^3 + 486662u^2 + u$ over the same field $\mathbb{F}_q$. When introduced, Curve25519 set new speed records for Diffie-Hellman computations on a Pentium III and does not have structures that allows an attacker to speed-up an attack (see [9, Chapter 3]) by exploiting these structures and reducing the complexity. The following parameters are chosen: The length in bits is chosen to be $b = 256$. For hash-function H, SHA-512 is chosen which has an output of $2b$ bits. The prime q used for the finite field $\mathbb{F}_q$ is $2^{255} - 19$ as it is in Curve25519. The $(b-1)$-bit encoding is usual little-endian encoding of $\{0, 1, ..., 2^{255} - 20\}$. ℓ is chosen to be the prime $2^{252} + 27742317777372353535851937790883648493$, the same as for Curve25519. Parameter d for the elliptic curve is $d = -\frac{121665}{121666} \in \mathbb{F}_q$. The base-point B used for scalar multiplication is $B = (x, 4/5) \in E$ where x is positive (see Subsection 3.1.4).

Parameter d is determined by transforming Curve25519 to a twisted Edwards curve. The curve is thus the Edwards curve $x^2+y^2 = 1+\frac{121665}{121666}x^2y^2$ transformed from Curve25519 using the equivalence $x = \sqrt{486662}u/v$ and $y = (u-1)/(u+1)$ and is further isomorphic to the twisted Edwards curve $-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$ since -1 is a square in $\mathbb{F}_q$ as discussed in [8, Page 130]. As also shown in [8, Page 130] the choice of base point B corresponds to [13] choice $u = 9$.

Regarding the security of Ed25519, [8] states a security level of 2^{128} that is comparable to AES-128 or RSA with approximately 3 000-bit key length. It is further stated that it has no special structures that would comprise the security and the parameters are chosen in a way that does not lower the security. The EdDSA uses keys with 32 bytes length and signatures of 64 bytes length.

3.5.1 The Arithmetic of Ed25519

As defined in Subsections 3.2.1 and 3.2.2, the signing and verification procedure depend on operations on the elliptic curve. These operations are rather complex and can be broken into layers of execution and an illustration of these can be seen in Figure 3.1. Each layer executes one or more operations on the underlying layers. For example, a scalar

multiplication $R = rB$ is calculated when generating the signature $(\underline{R}, \underline{S})$ of a message, with $\underline{R} = point\_encoding_{b-bit}(R)$. Subsequently operations on the underlying elliptic curve are performed, which in turn operate on further underlying layers. In the following paragraphs, the layers of execution are in short mentioned and described "bottom-up". This means starting at the lowest layer up to the topmost layer. Here the lowest layer is meant to be the integer arithmetic. In the context of building hardware as integrated circuits one might think of further layer(s) describing the electric current through the devices. Since engineering digital integrated circuits is commonly done above the layer of calculating the electric current, this will not be considered in this thesis.

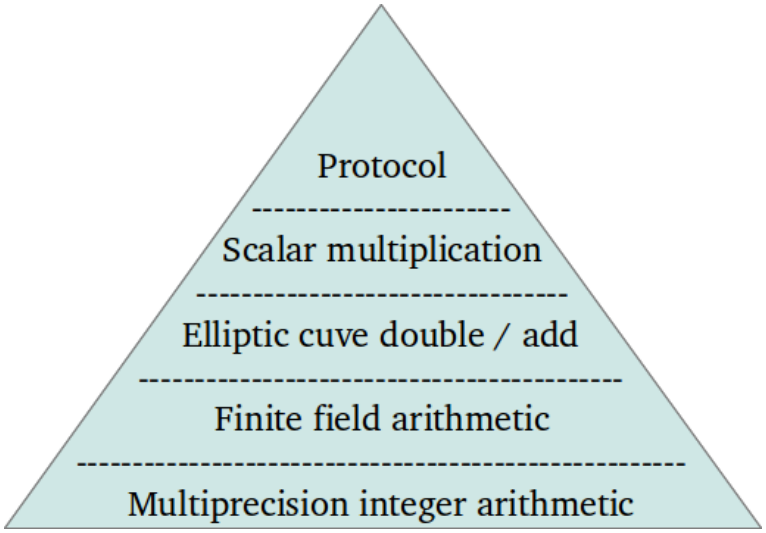

Figure 3.1: Layers of arithmetic

3.5.1.1 Integer Arithmetic

The bottom layer of the used arithmetic in Ed25519 is simple standard integer arithmetic. In the context of Ed25519, operands are commonly $b = 256$ bits wide integers. For most hardware architectures this is too wide to be processed at once. However, this operands can be broken into smaller partitions called words and these words can then be processed one word after each other. Therefore it is not necessary that an arithmetic unit must be able to process 256 bits long operands. The operations needed for the upper layers are addition, subtraction and multiplication. In the following paragraphs these will therefore be discussed in short. References to algorithms which for processes them word by word are provided in the book of Hankerson, Menezes and Vanstone [19, Chapter 2].

Addition and Subtraction. These two operations take n-bit inputs and generate a $n + 1$-bit output. The one additional bit is generated by the operations. It is the most significant bit and in case of addition it is called *carry bit*, when a subtraction is performed, it is commonly called *borrow bit*.

When an integer of w bits length is added or subtracted, with w being greater than the according hardware-unit input-width v, the integer is broken into $\lceil w/v \rceil$ words and called multi-word integer. These words are processed from the least significant upwards and this addition or subtraction is called multiprecision addition or subtraction. Within

the operations on single words, the carry or borrow bit generated is passed on to the next operation on the next more significant word. Concluding, processing w-bit operands with a v-bit hardware unit takes $\lceil w/v \rceil$ operations. Pseudo code of the algorithms performing a multiprecision addition and subtraction, and discussion can be found in [19, Subsection 2.2.1].

Multiplication. In contrast to addition and subtraction the multiplication serves a $2w$-bit result on two w-bit inputs.

If the operands of w bits width are multiplied and $w > v$, where v is the input width of the multiplication unit, the operands are again broken into words and the operation is called multiprecision multiplication. This is similar to the multiprecision addition and subtraction but a bit more complex: Each v-bit word of integer a is multiplied with each v-bit word of integer b. The single $f = \lceil \frac{w}{v} \rceil$ pieces of a, b are indexed, starting with index 0 at the least significant word a_0, up to the most significant one a_{f-1}. Therefore the integer a can be computed out of its pieces: $a = a_0 + (a_1 << v) + (a_2 << 2v) + ... + (a_{f-1} << (f-1))$. The same applies to integer operand b and similar for the result c, but here up to the last piece of the $2f$ words c_i. A single multiplication of two v-bit words a_i and b_j results in a $2v$ bits width. The lower word is added to the intermediate result c_{u+t} and the upper word is added to c_{u+t+1}. Due to the addition to the intermediate result, a carry bit might be generated. This must be either propagate starting at the next higher word or handled somehow else.

The procedure described above does not specify the order with which the words are multiplied. There are two popular algorithm that differ mainly in this aspect. The *operand-scanning form* processes the operands from least significant to most significant word, the *product-scanning form* in contrast calculates one result word after each other starting at the least significant. Both have in common that a multiplier unit is assumed, which has the same input width as the word size and have an output twice as wide as the input. More details and pseudo code for both form can be found in [19, Subsection 2.2.2].

As each word of operand a is multiplied with each word of b, with both operands being of the same size[1], the overall process of multiprecision multiplication includes $\lceil \frac{w}{v} \rceil^2$ single multiplications. As mentioned beforehand, the result of each single word multiplication is added to the intermediate results. For calculating a multiprecision multiplication the mentioned additions regarding the intermediate results may or may not influence the necessary amount of cycles and depends on the used architecture of hardware: While calculating the multiplication of two single words, the result of the previous multiplication, added to the accumulator's value, might be saved back to the accumulator or directly to memory. Thus, such an architecture does not require extra cycles for writing to the memory. Such an architecture is presented in Chapter 4 and was implemented in the context of this thesis.

3.5.1.2 Finite-Field Arithmetic

Finite fields are used for the elliptic-curve cryptography as discussed in Subsection 2.2.5. As mentioned in Subsection 2.2.5.1, for each prime p a finite field exists and this finite field then has p elements. Each finite-field operation with valid elements of the finite field as input always results in a valid element of the finite field. The operations used on this layer are commonly called modular addition, modular subtraction, modular multiplication,

[1] Here the maximum storable size is meant. The width of the actually stored integer may be less.

modular inversion and modular negation. These operations result in integer numbers modulo the prime $q = 2^{255} - 19$: $x \in \mathbb{F}_q : x \in \{0, 1, ..., (q-1)\}$. Within this thesis, the operations on this layer are called, for instance, finite-field addition. Thus, the word *modular* is replaced by *finite field* to clearly show the belonging. Therefore finite-field addition, negation, subtraction, multiplication, inversion and division are based on the underlying integer arithmetic and the fast reduction, and will be discussed in the following. Pseudo code and further details for the algorithms discussed here can be found in the book *Guide to Elliptic Curve Cryptography* in [19, Section 2.2].

Finite-Field Addition. This is based on an integer addition. The integer addition may result in an integer greater than prime q for the finite field. The maximum width of the result is $w + 1$ bits with w being the width of the input operands, if only the width is considered. But due to the fact that the operands are $a, b \in \mathbb{F}_q$ the result $c \in \{0, 1, ..., 2 \cdot (p-1)\}$. As it is solved in the latter for reduction, if the result $c \geq q$ then q is subtracted once and the results is in $c \in \{0, 1, ..., (q-1)\}$.

Finite-Field Negation. Negation in the finite field is done by performing $-a = q - a$. As operand $a \in \mathbb{F}_q$ and therefore $q > a$, the integer subtraction will always result positively, thus $-a \in \mathbb{F}_q$ and testing for a negative integer value is not necessary. As an alternative, and assuming a finite-field subtraction is already available, the negation could be realized by a finite-field subtraction too, calculating $-a = 0 - a$.

Finite-Field Subtraction. As a finite-field addition includes an integer addition, finite-field subtraction includes an integer subtraction. But compared to addition, finite-field subtraction is even simpler: If the result of the integer subtraction is negative, a borrow bit is generated. Thus the result does not have to be checked separately: If a borrow bit is generated q is added. Assuming the availability of a finite-field negation, alternatively the subtraction could be realized by negating the subtrahend and performing a finite-field addition.

Finite-Field Multiplication. For finite-field multiplication the operands $a, b \in \{0, 1, ..., (q-1)\}$ are at first multiplied in usual integer manner. The result c_{int} may be greater than q: $c_{int} \in \{0, 1, ..., (q-1)^2\}$ but it has to be ensured that the result c satisfies $c \in \mathbb{F}_q$. Due to this c_{int} must be reduced and c is obtained. Therefore finite-field multiplication consists of an integer multiplication with a reduction as discussed in the following.

Reduction. Reduction modulo p that has no special form can be an expansive part of a finite-field multiplication as stated in the book *Guide to Elliptic Curve Cryptography* in [19, Subsection 2.2.4]. But here the used prime q is a pseudo-Mersenne prime of the form $q = 2^w - k$. A pseudo-Mersenne prime provides the ability as a Mersenne prime to perform fast reduction. The length w of the binary representation of the used prime is 255 bits. A detailed discussion about reduction and especially fast reduction can be found in the paper of Taschwer in [38]. One "round" of a fast reduction is done as follows: Take the more significant w bits starting at the $w + 1$-th bit, multiply it with 19 and add it to the lower w bits. By doing this twice, an integer result is obtained with maximum width $w + 1$ in bits. Therefore the result may still be greater than q. If this is the

case either once subtracting q or twice subtracting q results in an integer less than q. Therefore, if the first subtraction result is still greater than q, a second subtraction is necessary. The circumstances regarding the bit quantities are discussed in the following. Note that the discussed bit quantities are always assumed to be the maximum possible, whereas the operation counts are assumed to be the minimum possible but as great as necessary to be true for all possible inputs. Further, it is assumed that the multiplier unit's input width and the addition unit's input width are greater than the width of the binary representation of the k part of the prime. Note that in the following the width of an integer and an arithmetic unit's input and output is measured in bits.

1. The result R of a multiplication of two w bits wide integers is $2w$ bits wide.

2. The upper w bits are multiplied with $k = 19$, where $o = \lceil log_2(k) \rceil = 5$. The product will be $w+o$ bits wide because, when doing a multiplication, the widths of the factors are added. The multiplication needs $\lceil w/r \rceil$ single multiplications, with r being the input width of the multiplication unit.

3. This product is added to the lower w bits of R. The sum will be $w + o + 1$ wide because the bigger summand in the addition is $w + o$ bits wide and the addition can generate a carry bit. This multiprecision addition is done with $\lceil (w+o+1)/s \rceil$ single integer additions, where s is the input width of the addition unit. With this step, the first round is complete and the intermediate result is the $w + o + 1$ bits wide integer, the result of the last addition.

4. The second round starts with already smaller integers. Again we take the upper part, above the lower w bits. Here this is only $o+1$ bits. The multiplication with 19 results in a $o + 1 + o$ bits wide product. This multiplication needs only $\lceil (o + 1)/r \rceil$ single multiplications.

5. The second round ends with the addition of the last product to the lower w bits of the result of round one. The addition needs $\lceil w/s \rceil$ single addition steps.

Summarizing the considerations above the reduction of the result of a multiplication of two elements of the finite field, each w bits wide, can be calculated with $\lceil w/r \rceil + \lceil (o+1)/r \rceil$ single-word-integer multiplications, $\lceil (w + o + 1)/s \rceil + \lceil w/s \rceil$ single-word-integer additions and $2 \cdot \lceil (w + 1)/t \rceil$ single-word-integer subtractions. o is the width of the integer k, the input width of multiplication unit is r bits, the input width of the addition unit is s bits and the input width of the subtraction unit is t bits.

Inversion. The most expansive operation in the finite-field arithmetic is inversion. The inversion is necessary for the above layer of elliptic-curve operations and computes R^{-1} of an element R such that $R \cdot R^{-1} = 1 \bmod q$. The inversion of a valid element of the set $1, 2, ..., q - 1$ results in an element of this set.

There are several algorithms available for computing the inversion in a finite fields. The most populars are the *inversion by exponentiation*, the *Montgomery inversion* and the *extended Euclidean algorithm*. To get an idea of how intensive the calculation, is regarding the execution time, the inversion by exponentiation and the extended Euclidean algorithm will be presented in short. Note that the following considerations are made in the context of the finite field of Ed25519.

Inversion by exponentiation obtains the inverse a^{-1} by calculating a^{p-2}. Therefore the intermediate result u gets initialized with a. Then starting at the second most significant bit, down to the least significant bit of $p-2$, executed for each single bit, u is squared and further, if the current bit of $p-2$ is 1, u is multiplied by a. The pseudo code is shown in Algorithm 4. The inversion by exponentiation has a fixed run-time per prime depending on the number of bits that are set in the binary representation of the prime. In the context of (pseudo-) Mersenne primes, the number of bits which are set, may be high. For example the prime used in Ed25519 $p = 2^{255} - 19$ has 252 bits set of its lower 254 bits. Thus one inversion by exponentiation takes 506 finite-field multiplications and is therefore very expansive in terms of run-time.

Algorithm 4 Inversion by exponentiation

```
procedure InvExp(a, p)
    u ← a
    for second_most_significant_bit(p) down-to least_significant_bit(p) do
        u ← u^2
        if current_bit(p) = 1 then
            u ← u · a
        end if
    end for
    return u
end procedure
```

The **extended Euclidean algorithm** is based on the Euclidean algorithm to compute the greatest common divisor(gcd). Computing the gcd is done by iteratively decreasing the operands a, b until one is zero. The other operand is, in turn, the result. The greater operand is decreased: in case $b \geq a : b \leftarrow b \bmod a$, otherwise $a \leftarrow a \bmod b$. Thus, the greater operand is replaced by the remainder of the integer division of the greater one divided by the lower one. By extending the Euclidean algorithm in a special way, it generates the integers x and y in each iteration as by-product, satisfying $ax + by = d$ with $d = gcd(a, b)$. Exploiting this circumstance and further $gcd(a, p) = 1$, where p is the prime of the finite field, in the iteration before one operand equals zero, one of the operands is 1. Due to this and the served x and y: $a^{-1} = x$ is maintained by the satisfied equation $ax + py = 1$.

In the context of computer arithmetics, the binary inversion algorithm based on the extended Euclidean algorithm might be more efficient. It replaces the division by bitwise shifts and subtractions, as due to the binary representation of numbers shifts are far less expansive. The binary inversion algorithm has, like the extended Euclidean algorithm, a variable run-time. The run-time for one inversion is at maximum 510 iterations. In each of the iterations, one operand is shifted left until the operand becomes odd, whereby this is accompanied with one left-shift and a conditional integer addition[2]. One iteration ends with a comparison of the operands and an integer subtraction. For an estimation of run-time see the following paragraph and for detailed explanation of the binary inversion algorithm see [19, Subsection 2.2.5, Page 40].

To give a rough **estimation** of the run-time of inversion by exponentiation and binary inversion algorithm, some assumptions have to be made. These assumptions are believed to reflect the circumstances in usual applications: For an approximation assume a uniform

[2]If the along calculated x (or y) integer is odd, the prime q is added to x (or y).

distribution of zeros and ones for the operands. Further assume that a left-shift, an integer subtraction and a comparison of two operands take as long as an integer addition, checking if a single bit is set (oddness) takes 1 cycle and that the arithmetic unit uses the same word size for the inputs of all of its operations. Also define n as the number of words into which the 255-bit integers are partitioned. As a consequence of the taken assumptions the integer multiplication takes n^2 cycles whereas integer addition takes n. As for the inversion by exponentiation finite-field multiplication is used, this is assumed to take twice as many cycles as integer multiplication does although the reduction-part of the finite-field multiplication performs two multiplications on its own. But due to the small value of the operand, 19, and the according small size of the operand, this special multiplication with 19 can be optimized to finish in a similar time as an addition. However, on a more detailed view, since a reduction is rather complicated according to the number of subsequent operations it contains (two multiplications, two additions and two subtractions), a certain overhead should be calculated. For the common word-width of 32 bits: $n = 8$. Inversion by exponentiation therefore takes $\approx 506 \cdot 2 \cdot 64 + 254 \cdot 1 = 65\,022$ cycles. The binary inversion algorithm takes, due to the assumptions, 255 iterations. Further, in half of the iterations the even integer is shifted left twice. In each iteration two comparison, if the operands are one, 3.5 oddness checks, $2 \cdot 1.5 = 3$ left-shifts, $0.5 \cdot 1.5 = 0.75$ integer additions, one comparison of the operands and one integer subtraction are performed. This takes $\approx 255 \cdot ((2 + 3 + 0.75 + 1 + 1) \cdot 8 + 3.5) = 16\,702.5$ cycles. As one can see, with the previous assumptions, regarding the run-time, the binary inversion algorithm beats inversion by exponentiation by a factor ≈ 3.9 on this word width.

3.5.1.3 Group Operations

Elliptic-curve operations are based on finite-field arithmetic, see discussion in Subsection 3.5.1.2, and executed according to the used curve's addition and doubling formulas, see Subsection 2.2.5.5. These two operations are discussed in Subsection 2.2.5.3. The operands are always points on the curve. Depending on the chosen coordinate representation, see Subection 2.2.5.6, addition and doubling formulas differ, the number of coordinates, a point $P \in E$ has, can vary, as well as the coordinates' meaning[3] of the point. In case of affine coordinates, a point has an x- and a y-coordinate, points in other point representations can have additional coordinates. Choosing from different point representations with different formulas is a trade-off between complexity, the needed time of calculation and the required space needed to store intermediate results as well as coordinates' data. In this section the addition and doubling will be discussed with different point representations.

Elliptic curve's two operations are doubling and addition. The doubling operation requires a single point. It adds the point to itself and results in some other point. The addition operates on two points and results in their sum. Even though doubling and addition may let one think of doubling or adding the coordinates of points, these operations can also be derived from a geometrical representation of points on an elliptic curve as shown in Subection 2.2.5.3. In terms of needed time for computing the result, doubling a point is cheaper than adding two points. When operating on twisted Edwards-curves, the unified addition rule can be utilized and thus the addition rule can be used for doubling as well. This addition rule for points with affine coordinates is $(x_1, y_1) + (x_2, y_2) = (\frac{x_1y_2+x_2y_1}{1+dx_1x_2y_1y_2}, \frac{y_1y_2+x_1x_2}{1-dx_1x_2y_1y_2})$. x_1, y_1, x_2, y_2 are the coordinates of the two input points, d is the parameter as it is in the curve equation. The twisted Edwards-curve

[3]For instance Jacobian coordinates the projective point $(X : Y : Z), Z \neq 0$ corresponds to $(X/Z^2, Y/Z^3)$.

point-addition has no restriction for its inputs as the point addition other curves have, as its denominators $1 \pm dx_1x_2y_1y_2$ are nonzero (see [11, Chapter 6]). There are specialized doubling formulas which are more efficient. This is due to the simple fact that every x_my_n, if $P_1 = P_2$, becomes x_1y_1 and must not be recalculated for x_1y_1, x_1y_2, x_2y_1, x_2y_2.

As discussed in short in Subsection 2.2.5.6, different representations of points on a twisted Edwards curve are possible. At the time this thesis was written, there were addition and doubling formulas available for affine, extended, inverted and projective point representations. Different representations give certain advantages, for instance speed, improvements or security considerations. Using a different coordinate system involves a modification to the curve equation and in turn causes differences in the addition and doubling formulas. The affine formulas, as stated in the paper which introduced the EdDSA [8], are presented above. The extended coordinates store X, Y, Z and T for each point which represents x, y in this way: $x = X/Z$ and $y = Y/Z$. This may lead to the impression of projective coordinates, but there is the additional T, used with the formula $x \cdot y = T/Z$. This representation was introduced by Hisil, Carter, Wong, Dawson in [20] - for further discussion and the explicit formulas please consult this paper. Bernstein and Lange in [6] introduced "inverted Edwards coordinates". These, in contrast to projective coordinates, work with $x = Z/X$ and $y = Z/Y$ and have the advantage of adding two points with saving one multiplication within the finite field. Although the inverted coordinates have a speed advantage, they are not complete anymore, thus separate checking for certain points is necessary, for instance the neutral point as clarified in [6, Page 4]. When twisted Edwards curves were introduced by Bernstein, Birkner, Joye, Lange, Peters in [11], their fast and unified addition formulas where presented and used projective coordinates. As discussed in Subsection 2.2.5.6, projective coordinates save time by reducing the amount of inversions during the computation.

3.5.1.4 Scalar Multiplication

A scalar multiplication multiplies a point on an elliptic curve with a scalar. In the previous Subsection 3.5.1.3 on elliptic curves, point addition and doubling are defined. The stated multiplication is calculated by repeating addition and doubling. Different approaches are available for performing the scalar multiplication. These will be discussed in the following and throughout this section we refer to the book *Guide to Elliptic Curve Cryptography* [19, Subsection 3.3.1] as it discusses all methods. Note that in the following we only consider methods for "unknown point", assuming that the point to be multiplied is not known and no optimizations can therefore be arranged. Multiplying a known point can be sped up by precomputing the multiples of the multiplied point. However, the multiplication of a known point is not considered in the following analysis because the discussed methods can be extended to use the precomputed data anyways. Note that the Montgomery multiplication is not discussed here because it is not applicable due to the fact that it is only possible when Montgomery curves are involved. It only computes the x coordinate of the multiple of the point and recovers the y coordinate from the resulted x coordinate. Only computing the x coordinate is possible due to the special form of curve equation.

Double-and-Add. The first scalar multiplication discussed is the double-and-add algorithm. This serially tests each bit of the scalar and, according to its value, either doubles the intermediate point, or doubles it and afterwards adds the base point to it. The testing of bits can be executed from most significant down to the least significant or in the reverse

order and for pseudo code see [19, Subsection 3.3.1]. The double-and-add can again be implemented with different strategies having different advantages. It is possible to do this with resistance against side-channel-attacks. In the context of scalar multiplication, the information most valuable but threatened is the scalar, with which the base point is multiplied. The leakage of information can take place in form of varying computation time and power consumption[4]. To be resistant, the scalar multiplication can be calculated by always performing the addition after the doubling on the underlying elliptic curve, but according to the current bit of the scalar either discarding the result of the addition or not. Let us assume that the unified addition-law is utilized for double as well. An implementation then can conditionally add the base point but, due to the unified addition-law, power-traces will not give information whether the bit was set or not, as the doublings and the additions are indistinguishable. Assuming the double-and-add algorithm adds the base point only if the bit is set and that about half of the bits of the scalar are set. Then each iteration of the loop will cause a doubling and in half of the iterations the addition of the base point will be carried out. Therefore the run-time can be approximated $\frac{m \cdot A}{2} + m \cdot D$, with A representing one point addition and D representing a doubling.

Non Adjacent Form. By replacing the binary representation of the scalar, as discussed in the previous paragraph, with the non-adjacent form (NAF), the run-time can be reduced. As will be discussed in a moment, adding and subtracting a point takes almost equal time. Instead of only adding points but also subtracting them, the necessary number of operations can be reduced. For the algorithm and detailed description of the NAF of an integer see [19, Subsection 3.3.1]. In the context of elliptic curves, computing the negative of a point is very cheap. For elliptic curves in Weierstrass form, the negative of a point is calculated by taking the negative of the y coordinate. Regarding an elliptic curve over a finite field, this is done by a single finite-field subtraction. In the context of twisted Edwards curves, the negative of a point is calculated by taking the negative of its x coordinate and thus the same consideration concerning the run-time as for elliptic curves in Weierstrass form apply to twisted Edwards-curves. Said this, we specify a point subtraction as the normal point addition but with the negative of a point. Therefore a point subtraction is a point addition with an additional finite-field negation. Due to the complexity of the point addition, the finite-field negation will be, due to the results of the hardware implementation presented in Chapter 4, about 100 times faster. Thus it can be said that the run-time of point subtraction and point addition are almost equal. As discussed in [19, Page 98] an integer scalar can be easily converted from binary representation to the NAF, a signed digit representation. Using the NAF of a scalar, the approximate number of digits being nonzero decreases from $\frac{m}{2}$ to $\frac{m}{3}$ and therefore the number of necessary operations decrease in the same manner: The digits in the NAF of an integer can be either 1, 0 or -1. Therefore, after the doubling, according to the current digit, a point addition, a point subtraction or no operation will be performed. As discussed in [19, Page 98] the NAF of an integer has the property of having at most one digit more than the binary representation and therefore the number of iterations in the loop corresponds up to one possible extra iteration with the number of digits in the binary representation. Using the NAF of a scalar, the run-time in terms of additions (subtractions) and doublings can be approximated with $\frac{m \cdot A}{3} + m \cdot D$. Therefore the number of necessary point additions is significantly decreased compared to the simple *double-and-add* algorithm.

[4]When executing addition only if the current bit is set, the run-time and the power-consumption of the multiplication varies and an attacker might be able to reconstruct the secret key by doing measurements.

Window NAF and Sliding Window. The in the previous paragraph discussed method of NAF can be extended in such a way that each digit may not only have a value of 1, 0 or -1, but also lower and greater values. This method is called window NAF and details can be examined in [19, Subsection 3.3.1]. Each nonzero digit of the scalar is odd and the range of values depends then on the chosen window width $w \geq 2$: Let k be the scalar and k_i the digits in the w-width NAF of the scalar. Then $|k_i| < 2^{w-1}$. In window NAF of a scalar, the number of nonzero digits decrease as the window width w increases. Each nonzero digit in window NAF can lead to a point addition or point subtraction of the intermediate point and a multiple of the point being multiplied. As a consequence, the precomputation of multiples of the multiplied point is necessary. Therefore, with increasing window width, more precomputation and space to store these multiples is necessary. For a more detailed description, pseudo code of the necessary calculation of the window NAF of an integer scalar and pseudo code of the scalar multiplication see [19, Section 3.3].

Out of the window NAF method, a sliding window method can be employed. It uses the above discussed window NAF representation of a scalar integer but parses the scalar in the reverse direction. The parsing is done in such a way that the greatest window with a width $t \leq w$ is searched with the integer $u \leftarrow \{k_i, ..., k_{i-t-1}\}$, $u \bmod 2 = 1$ and k_i are the single digits of the window NAF of the scalar. Therefore a variable-length window slides over the scalar. The sliding window method allows greater multiples of the point and therefore needs more precomputation but reduces in turn the necessary amount of point operations.

As discussed within the comparison of window NAF and the sliding window method in [19, Pages 101 and 102], the better method of those two depends on the available space for storing the precomputed multiples and the costs of precomputation relative to the computation point operation regarding the run-time.

Chapter 4

Hardware Implementation

A central processing unit (CPU) of a personal computer's functionality is in general not limited and can do every kind of computation. In contrast, a hardware implementation, for instance of a cryptographic system, has a very distinct purpose. It is of a very special kind and has no other purpose than doing dedicated computations. Such a hardware implementation therefore is a specialist. Efficiency in this context may have several interpretations and might be the power consumption per finished calculation. For hardware implementation of cryptographic systems the product of the area and the time are often used.

At the beginning of this chapter the goals of the presented hardware implementation will be specified and related work will be discussed. The used design flow, provided by the *Institute for Applied Information Processing and Communications* (IAIK), will be presented. After that, the implemented hardware architecture, components, algorithmic details and the implemented test bench and high-level model will be discussed.

4.1 Goals

The main goal of the hardware implementation presented in this thesis was the low resources consumption for the implementation of a Edwards-curve Digital Signature Algorithm over a prime field.

The Edwards-curve Digital Signature Algorithm (EdDSA) is a non-standard digital signature algorithm. It operates on twisted Edwards-curves and further has modifications relative to the standardized ECDSA which results in improved security and speed. In spite of the advantages of EdDSA over ECDSA, it is rarely used and implemented.

The resource consumption is judged by the area-time product where the area of the hardware implementation is multiplied with the necessary cycles to perform the operation. The area of the synthesized hardware is measured in the technology independent unit *gate equivalent* (GE) after the synthesis step. After synthesis, the hardware is already mapped to standard logic-cells. The technology independence is achieved by dividing the measured metric value of the hardware's area by the area one negated-and (NAND) gate takes. Such a NAND gate is defined to be a NAND gate with two inputs, driver strength one and commonly consists of four transistors. For more details and discussion on this topic see [22, Pages 4 and 37]. Within this work, the time an operation takes is measured in cycles and is thus independent from the actual frequency the hardware runs with.

The chip area increases with the word width of the architecture. A larger word width

results in more transistors within the datapath as it is able to process larger numbers. The number of cycles in contrast usually decreases with increasing word width, because fewer words have to be processed. Let us assume, for example, the integer addition with 256-bit large operands and a word size of 32 and 64 bits. To process the operands one word after another, a 32-bit implementation needs eight cycles. As the word width doubles to 64 bits only half of the time is necessary to perform the same operation. The necessary area, in turn, for 64 bits doubles because processing twice as large inputs require twice as many logic gates.

A secondary goal was to implement the chip in such a way that there is a central definition of an implementation-wide used word size. This enables the evaluation of performance, chip area and efficiency at different word sizes.

4.2 Related Work

There exist numerous elliptic-curve-cryptography implementations for Weierstrass curves. In contrast, implementations with twisted Edwards curves are not very common. On the one hand, this might be caused by the relatively young Edwards curves, introduced in 2007 [13], and the even younger twisted Edwards-curves, introduced in 2011 [8]. On the other hand, this might be caused by the fact, that for cryptography some elliptic curves in Weierstrass form are standardized by ANSI [2], are NIST approved [16] and are used in popular signature algorithm like the ECDSA. Beside a twisted Edwards-curve hardware implementation on an Field-Programmable Gate Array (FPGA) no other hardware implementation is available at the best knowledge of the authors. In Section 5.3 the implementations of elliptic curve in Weierstrass form are compared to the results of this thesis's implementation.

In Baldwin et al. [3] an implementation of a cryptographic system using twisted Edwards-cuves over prime fields was on a FPGA. The implementation was then used to analyze the speed for simple-power-analysis (SPA) resistant as well as non-SPA-resistant algorithms. The implemented elliptic curve processor has capabilities to compute the elliptic curve point addition and point doubling operations on the one hand for Weierstrass curves using Jacobian projective coordinates and on the other hand for twisted Edwards-curves. The implementation and the architecture are also capable of varying the number of arithmetic and logic units (ALU). The algorithms for double-and-add, double-and-add-always were implemented and were analyzed concerning their performance and efficiency for both types of curves. This analysis included the maximum frequency, the area, the time, the power and energy consumption as well as the area-time product. The authors concluded from their results that an SPA resistant twisted Edwards-curve architecture with one ALU and four ALUs give comparable performance to the non SPA resistant Jacobian double-and-add architecture. The non-SPA-resistant twisted Edwards-curve architecture outperforms all Jacobian architectures regarding the time, the area-time product as well as the energy consumption.

Kocabas et al. [26] implemented a binary Edwards-curve hardware targeted to very constrained devices such as passive Radio Frequency Identification (RFID) tags. The implementation uses mixed w coordinate with common Z-coordinates together with the Montgomery ladder. The usage of the w coordinate and the Montgomery ladder for binary Edwards-curves were proposed in [7]. The Montgomery ladder is a efficient way to calculate the scalar multiplication and is applicable where differential point addition and differential point doubling are possible. It has further protection against side-channel

attacks as point addition and point doubling is performed for every bit in the scalar, regardless if the bit is set or not. Together with the Montgomery ladder, the mixed w coordinate simplifies the computation of point additions and doublings and reduce the need of memory to store intermediate results. It is calculated out of the x, y coordinates once at the beginning of the scalar multiplication. After the completion of the scalar multiplication, the x, y coordinates can be restored from the w and Z coordinate. The implementation was verified on an FPGA and synthesis results are reported for the used $GF(2^{163})$ with varying word widths of the architecture.

Chatterjee and Sengupta [10] presented a binary Edwards-curve implementation on an FPGA. It uses the ternary representation of the scalar and was targeted to optimize the speed by an effective use of the FPGA's lookup tables. The ternary representation of the scalar for the scalar multiplication is discussed in Subsection 3.5.1.4. Therefore the scalar multiplication uses the double-and-add algorithm and, due to the ternary representation of the scalar, less point additions are performed. Because of the targeted high speed and the usage of a large number of registers, the implementation uses 240 064 GE.

Satoh and Takano [31] introduced an ASIC hardware implementation of an elliptic curve processor. The supported elliptic curves are of Weierstrass form over prime fields. The architecture is scalable in terms of field size, supports both, binary and prime fields, and can be configured for different word widths. It was targeted on a 0.13 μm-CMOS standard cell library. For scalar multiplication, the NAF form of the scalar was used. For the reduction in the prime field, no special form was exploited. This work is used for comparison in Section 5.3, due to the lack of other comparable implementations with twisted Edwards curves and Edwards curves, although it utilizes Weierstrass curves.

Another elliptic-curve cryptography implementation with Weierstrass curves was introduced by Wolkerstorfer [42]. It is targeted on passively powered RFID tags. Thus the implementation was optimized for low area and low power consumption. The hardware implementation is realized supporting both, binary fields and prime fields. It uses Montgomery multiplication and is capable of calculating the inverse using the extended Euclidean algorithm. Different field sizes were evaluated: 192 bits, 224 bits and 256 bits. The suitability for RFID application was evaluated with 0.35 μm, 180nm and 90nm technologies. The hardware implementation results are used for comparison in Section 5.3.

Wenger and Hutter in [41] compared elliptic-curve-cryptography implementations over binary field and over prime field. The two fields were of comparable size and the implementations were evaluated on scalar-multiplication level as well as at the protocol level. The implementations share the same controller: A processor with Harvard architecture that is optimized for elliptic-curve cryptography. As target technology a 130nm CMOS process technology was used. On the protocol level, to carry out signature generation and signature verification, the binary field implementation forfeited some low resource advantages over the prime field implementation as additional arithmetic was necessary. Their implementation results are presented in Section 5.3. There the prime field implementation is listed on the scalar-multiplication level.

Fürbass and Wolkerstorfer presented an elliptic-curve-cryptography processor [17] targeted for RFID applications. The implementation is able to carry out ECDSA operations on a Weierstrass curve and uses the Montgomery addition ladder for scalar multiplication. It operates with affine coordinates and utilizes the Montgomery multiplication that has an integrated reduction. For the Montgomery multiplication the numbers have to be converted to the Montgomery representation. The 160-bit and 192-bit implementations were

evaluated on different process technologies, 0.35 μm, 250nm and 130nm[1].

4.3 IAIK Design Flow

The IAIK-design-flow is provided by the IAIK and serves as a design flow for Very-Large-Scale Integration (VLSI) development. It enables VLSI development for FPGAs of Altera and Xilinx as well as ASICs based on standard library cells with manifold options and operations. For ASIC development it has standard cell libraries support for ams AG[2] and UMC that seamlessly integrate into the Cadence Design Systems flow. The developer can compile his hardware-description-language (HDL) code, synthesize and place-and-route it. The popular languages Verilog and VHDL are supported as input HDL. After each of the mentioned steps, the output can be simulated using the Cadence's NCSim simulation suite and further one has the option to simply get the textual output in the console or inspect the simulation in a graphical user interface.

In this section the compilation, the synthesis, the place-and-route and the simulation steps will be discussed. The interested reader might examine [22, Subsection 4.2.6] about VHDL and the process of compilation and synthesis.

4.3.1 Compilation, Synthesis and Place-and-Route

The hardware is developed by writing HDL code. This HDL code is hardware independent and, as other programming languages, can be compiled which creates a model that has timing information and can be simulated using simulation software. If the compilation is successful, the next step, synthesis, produces hardware for a certain technology that already includes models of components from the used technology and cell library. The models include accurate timing restrictions and electrical details. The result of the synthesis can be simulated but will be more time consuming due to the detailed models. The next step towards a physical chip is to perform the place-and-route operation. Within this, the components and component groups get placed within the standard cell area and the wires for the signals between them get routed. After the place and route, the chip's layout is finished regarding the electrical details of the chip. As for the compilation and synthesis before, the result or the place-and-route can be simulated and checked to behave still as intended. Additionally, further checks are usually performed that check the layout according to the design rules defined for the target technology and check the layout against the schematic. For a more detailed discussion about place-and-route see [22, Subsection 11.3.6].

4.3.2 Simulation

The simulation is used to inspect, debug and analyze a circuitry. This circuitry can be the result of one of the previously discussed operations with different levels of complexity. The IAIK design-flow utilizes the Cadence's NCSim simulation suite. Because the circuitry usually does not perform any operation without the appropriate stimuli from outside, the simulation further needs a so called test bench that sends a certain sequence of signals to the device-under-test (DUT). The simulation environment therefore provides an interface to access the DUT's signals, simulation parameters and simulation operations.

[1] The authors mentioned 150nm but listed 130nm results.

[2] ams AG is the new name of austriamicrosystems.

The simulation loads the result of either the previous compilation, the synthesis or the place-and-route and starts the test bench. The test bench must be tailored to supply the DUT with the necessary input signals and to read the DUT's output signals in the intended way and interpret them. Thus a test bench is developed to "talk" with one certain DUT. A simulation suite provides functionality of controlling the simulation time. The test bench uses this to apply logic values to the virtual input pins of the DUT at the desired point of time and for a certain duration of simulation time, and watches and measures the DUT's logic values of the output pins during simulation time. The values correspond to the DUT's specific electrical voltage levels but for development of digital integrated circuits this electrical details are unnecessary and abstracted.

The IAIK-design-flow and the utilized Cadence NCSim simulation suite provide support of the programming language TCL for the test bench. The IAIK-design-flow provides the basic functionality for controlling the simulation time, applying stimuli signals to single pins as well as to collections of pins (ports), and reading the levels of single pins and ports. The simulation complexity and the time consumption increase if the result of the synthesis or place-and-route is used. These already include cells and gates with realistic parameters attached, based on the used fabrication process. In case of place-and-route, even parasitics due to the resulting wire lengths are included. The IAIK-design-flow supports console-text output in batch mode as well as the display of the simulated signals in a graphical user interface. The developer can inspect the transitions of the signals over time. In both modes, the simulation is done by executing the test bench. For simulation with the textual mode, the test bench must implement the checking for errors of the results to get an automatic test bench. In contrast to that, with the graphical user, interface the developer can inspect the changes and states of the signals optically.

Details on the test bench are presented in Subsection 4.9.2 and the required data generation is presented in the Subsection 4.9.1.

4.4 The Architecture

The architecture of a hardware implementation of a complex system like a cryptographic system, strongly advises the usage of abstraction. This is meant in such a way that certain groups of similar functionality are formed and each of these groups becomes a submodule. Further abstraction in terms of arithmetic levels, necessary for elliptic-curve cryptography, will be discussed in the following. Whereas an implementation not necessarily requires the abstraction and encapsulation of functionality in submodules, the contrary approach, a flat collection of transistors, got impracticable long ago, as Kaeslin in [22, Subsection 4.2.1] states.

The hardware implementation presented here was made with a separated controlpath, datapath and memory as illustrated in Figure 4.1. The figure shows the main modules and the signals between the modules. The signals ware not visible in all detail because some of the control signals are bundled or have a minor relevance. Note, that the size of the modules is not representative as the modules area consumption anyhow depends on the used word width. In the illustration the dependence of the bus access on the busy signal of the controlpath was illustrated by the multiplexers located above the memory unit which are controlled by the busy signal. The bus interface has some address checking implemented and sends control/start signals to the controlpath in case of write access by the bus to the dedicated command addresses. The controlpath in turn is responsible for the control signals to the datapath and the memory.

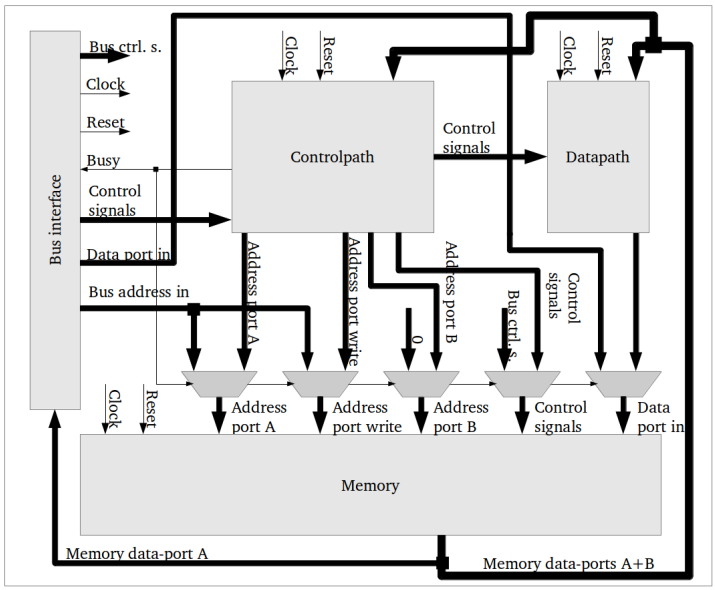

Figure 4.1: Illustration of the architecture

The interface of the implementation has an address port, a data-in and a data-out ports with the same width as discussed in following Section 4.5. Beside that, there are the usual clock and asynchronous reset input signals, a busy signal, a chip-select and write-enable signal. The interface was chosen to be simple but it enables direct reading from the memory and writing into it. The implementation's targeted purpose is to calculate the scalar multiplication on the twisted Edwards-curve of the Ed25519, although all the arithmetic operations can be started via the memory mapped interface. This will be discussed in Subsection 4.5.4.

The implementation of the sub modules will be discussed in detail in the following Sections 4.6 - 4.8. In this section a coarse overview is provided.

The controlpath holds the logic and further is responsible for the correct execution of sequence of operations. It is in control of the memory and the datapath. The arithmetic levels are illustrated in Figure 3.1. Each of these levels represent some operations and each of those operations' logic is contained in the controlpath.

The datapath consist or simple integer arithmetic and copy functionality. In the datapath are multiplexers which use the input control-signals to direct the input data-signal to the appropriate functionality and multiplexers that direct the resulting signals to the output port.

This implementation has a centrally defined word width m which is used throughout the design. Thus the operations and sub modules take integers of m bits length, for instance the multiplier as well as the adder does. The data signals, the internal as well as those used for the interface to the outside, are of m bits length. The word width will be discussed in the following Subsection 4.4.1.

The chip is designed to be synchronous, which means there is a global clock throughout the implementation, and every memory element and state transition within the chip can change its state only dependent on this clock, as Kaeslin defines in [22, Subsection 5.2.1]. Therefore it is possible to be either dependent on the rising or the falling edge of the

clock. This hardware implementation is designed to use the rising edge of the clock. An illustration of one cycle is shown in Figure 4.2. Kaeslin [22, Section 5.5] states the advantage of a synchronous design over an asynchronous one. Timing problems simply do not arise by the methodology of synchronous design and the usage of modern design-automation flows and standard-cell libraries. For this implementation no clock gating is

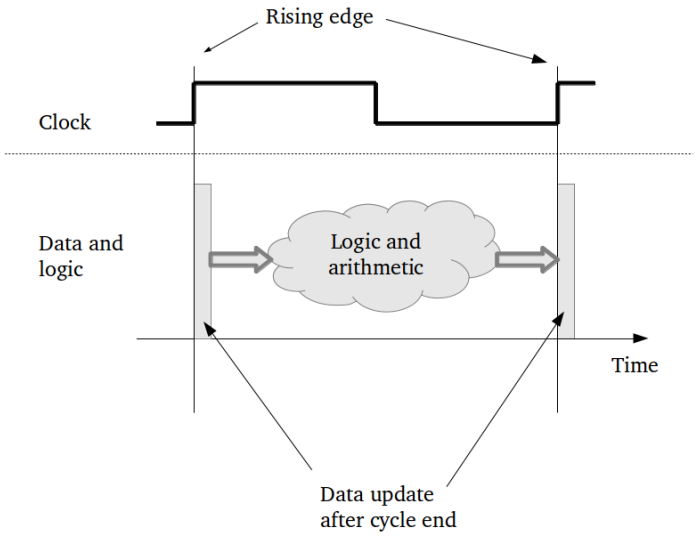

Figure 4.2: Illustration of a synchronous design

used. This is a technique to reduce the power consumption by disabling the clock transition for specific parts and modules that are not used.

4.4.1 Word-Width Consideration and Limitations

The word width m is centrally defined and can be chosen at compile time. The implementation is down to the last nook dependent on the central definition and scales everything relative to it. This regards the size of the ports of the modules as well as of the interface, the width of the used signals, the internal structures and it also applies to every operation that is implemented in such a way that this operation processes each word of the operand. When choosing the word width, three restrictions have to be respected and will be discussed in the following.

The first one is that a word width has an upper limit of 256 bits. This is caused by the fact that the operands have at maximum 256 bits. In general, a greater word width reduces the necessary cycles for multiprecision operations as the number of cycles depends on the quotient of $n = \frac{256}{m}$: m is the used word width and resulting n is the number of words each register has. A word width greater than 256 bits only increases the necessary area of the hardware implementation without a benefit. Then, only unused bits would be added and thus this would cause unnecessary big arithmetic units. A greater value is not considered reasonable and is therefore not supported by the implementation.

The second restriction is that the word width must be a power of two and is based on three design decisions. Firstly, the word width is defined to be equal to the address width and restrictions to the address width also limit the word width. This has the background

to minimize the internally stored addresses: Smaller addresses need less bits when stored. Therefore the memory-address block is always[3] located at the very beginning starting with the address zero. The implementation's properties need a fixed location in the address space because in these properties essential information is stored. Information such as the amount of addressable memory and the addresses of the command addresses that is crucial for any device accessing by bus. As the memory reserves the lower end of the address space, another fixed address is induced by linking the address width to the word width. With this linkage, there is a fixed end of the address space that can be used for the properties. Secondly, there must be no gap within the whole address block used by the memory: Internally the addresses width is minimum width possible to address all words in the memory. Gaps within this block would result in a larger address block and would unnecessarily increase the size of any address storage. Thirdly, within the memory are blocks that are strongly related. These are the registers which consist of one or more words. The sum of the words' widths of a register is 256 bits. In case of a word width smaller than 256 bits, the address of each memory word can be split into a register part and a word part. As a consequence of this and that there must not be a gap between the last word of one register and the first word of the next register, the number of words within a register must be a power of two. Other amounts of words would generate a gap. For instance if the register would be split into 7 words, the first word address would be '000' in usual binary representation, the second one would be '001' and so on up to the seventh with '110'. In this case the word address '111' would not be used as the next register has a different register address and its word address starts at '000'.

The third restrictions is that the word width must be at least 16 bits, which is based on some preliminary design decisions: The first decision is that the address width is equal to the used word width. The second decision is that a direct memory-interface is used. The third is that all the operations of the implementation can be started by bus access and that the static properties can be read. The first decisions link the address width and the word width and thus restrictions for the address limit the word width too. Therefore, the word width has a lower limit that is influenced by the three types of addresses: The sum of the necessary addresses must be expressible with an integer which is at maximum as wide as the word width used in the implementation. The number of words in the memory decreases with a higher word width whereas the number of addresses for starting the operations and reading the properties are fixed numbers. In detail there are 16 register in the RAM and three Read-Only Memory (ROM) entries necessary with each 256 bits wide and thus $19 \cdot 2^8$ bits of memory are used and split into words with a specified size. In case of eight bits word-width a register would be split in 32 words and the necessary word-address width for this is five bits. The address width, by decision, is the same as the word width and therefore three bits would be left for the register address. But, for distinguishing between the 19 registers[4] also five bits would be necessary and thus eight bits are no applicable value for the word width. The next greater candidate for the word width is 16 as it is the next power of two. In case of 16 bits, the word-address width is four and by the linkage of word width and address width, the rest of the address width can address 4 096 registers. Therefore 16 is the first and smallest power of two that fulfills the requirements. Regarding the next powers of two, the necessary word-address width further decreases whereas the addressable words increases.

[3] The size of the memory-address block varies with different word widths.

[4] Here the 17 addresses for the commands and 15 addresses for the special properties are not mentioned as eight bits are already to small for the memory addresses.

As a consequence of the above three restrictions and limitations the word width can have either a value of 16, 32, 64, 128 and 256 bits. Each of these word widths were tested for correct computation and an evaluation of their efficiency will be presented in Section 5.2.

4.5 The Interface

As Kaeslin in [22, Subsection 4.2.1] states, modularity within an architecture and abstraction of functionality is an important design principal for modern hardware designs. Modularity then implies the use of interfaces to enable the outside to access the inside functionality and data. An interface of a hardware implementation in general offers the "outer world" the possibility to control the behavior of the hardware and to get information of it. In the following, the interface of the presented hardware implementation will be discussed. This includes the used signals and their characteristics, how a bus access is performed and further the accessible data. This includes the implementation's properties as well as the functionality, which operations are available and which parameters must be set.

The interface to a bus which connects devices. These devices may be of diverse types but they have the interface for the bus they are connected to in common. The bus specifies the procedure on how a communication between two devices takes place and thus how data is transferred between them. This hardware implementation's interface was not destined to a particular known bus and only has the necessary signals to communicate with the test bench. In the following, these communications will be presented.

4.5.1 General Considerations and Characteristics

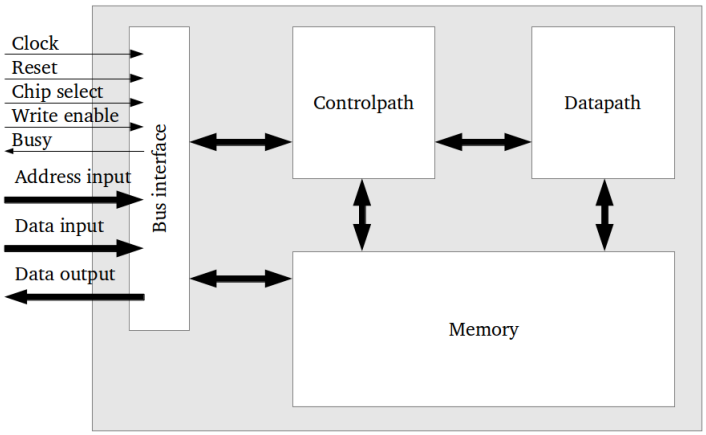

Figure 4.3: Illustration of the architecture including the interface

This section presents and discusses the signals and their purpose, characteristics and details, and how to access the different parts of the implementation.

The interface of the implementation which is connected to the outside controller enables direct memory access. The implemented operations can be started by write accesses to the addresses they are mapped to. By reading certain addresses properties of the

implementation can be obtained. An illustration of the interface of the implementation is presented in Figure 4.3. There are single bit signals for the outside communication: the usual clock and asynchronous reset signals, a chip-select, a write-enable and a busy signal. Further the interface has an address-input, a data-input and a data-output port, which all share the same width as the word width used throughout the implementation as discussed in the previous Subsection 4.4.1. Beside the data output, there is a second output signal which gives information whether or not an operation is started and running.

The clock is generated outside and is delivered through the interface. It is used as global clock for the implementation. This implementation is synchronous and sensitive to the rising edges of the clock. Synchronous means that every part of the chip is dependent on the clock as it is discussed in Section 4.4.

The asynchronous reset input resets all inner states of the implementation and is propagated into all sub modules. It ensures a well determined starting point at the next clock cycle after the reset signal ended. The reset of the implementation is active high. For further information about reset signals in general and especially asynchronous reset-signals see Kaeslin's discussion in [22, Pages 210, 296 and 303].

A bus access is done by applying an address to the address input, applying the chip select and specifying the direction, either read or write, and accordingly applying an input at the data-input port when writing. An access can have three purposes. The most obvious is to access the chips internal memory, with either a read operation or a write operation on the internal memory. Such accesses are discussed in Subsection 4.5.2. Secondly, there are certain addresses which are used to start a computation on the chip. If these addresses are used for any write operation, the according command is started at the next rising edge. The starting procedure and the available commands will be discussed in Subsection 4.5.4. Thirdly, there are some special addresses that can be read for obtaining constants from the implementation such as the number of registers, the word width or similar properties. These accesses and the available properties are discussed in Subsection 4.5.3. According to the above mentioned three different types of addresses, the address space is divided into blocks. The actual explicit addresses included in those blocks and further the number of addresses vary with different word widths. The address space and the location of the blocks is illustrated in Figure 4.4. The address space is greater than the actual usable space, determined by the memory, the special addresses and the command addresses. Each row in the illustration specifies the 256 bits wide portion called register. Each register is further split into n words, each m bits wide, where m is the word width used implementation-wide and $n = \frac{256}{m}$. Each word has its own unique address and is discussed in Subsection 4.4.1. The curled lines should illustrate that the number of words per row depends on the used word width.

The usable memory is located at the lowest addresses starting at address zero. The command addresses that are used for starting the different functions of the implementation are located immediately after the last address used by the memory and leave no gap of unused addresses between. The special properties addresses are located at the very end of the address space and thus leave a gap between them and the commands' addresses. As the number of used addresses by the memory varies according to the used word width, the explicit addresses of the single commands vary too. The similar apply to the special addresses: due to the varying end of the address space, depending on the value of the word width, these addresses vary too. But in contrast to the commands' addresses the special properties' addresses can be calculated by the knowledge of the word width. To obtain the addresses of the commands the special properties deliver the necessary information

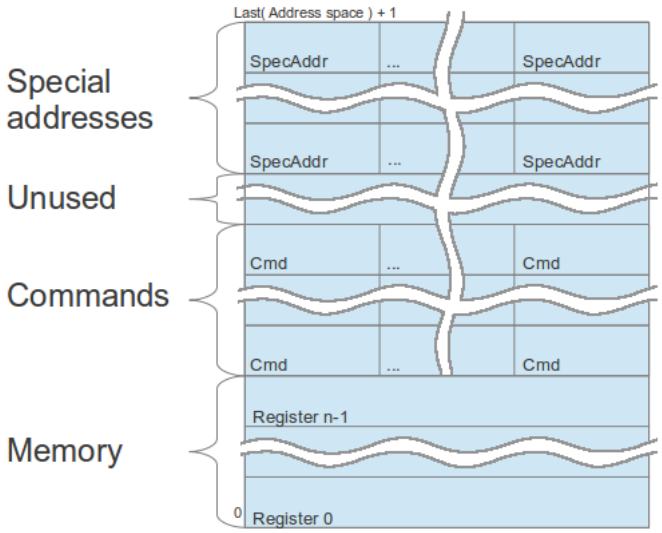

Figure 4.4: Schematic illustration of the address space

to calculate them. The address of each property, command and memory word of the implementation is calculated at compile time depending on the word width. The only exception of course is the first word in the first register as this is always zero.

For the following discussion it first has to be clarified that the memory must have an interface with data ports that have the implementation-wide used word width. The reason for this is that all calculations within the implementation are done with this word width and therefore need the operands with this word width from the memory. Further the interface of the chip is used for direct memory access. The width of the address, the data-input port and the data-output port of the interface are all of the same width, which is the implementation-wide used word width. This has three main causes: the area, simplicity and continuity. Using another port width than the implementation's word width would increase the necessary area of the implementation as well as it would increase the complexity. This is because then an additional logic and wiring for accessing the memory by the bus would be necessary as the memory must have an interface with the implementation's word width. Assuming that the chip's interface uses another word width, there are different ways to handle this. For instance it would be possible to leave the memory's interface untouched and to implement additional logic in the bus interface that stores the data in a buffer until, for instance, the whole 256 bits of the operand are transferred. Another possibility might be to implement additional ports in the memory module and to directly connect this to the outside interface without buffering. Further, a possibility would be to provide an output-data port that has double the width of the implementation's word width. In this case, less modification would be necessary as the memory already has two read ports for fetching both operands of calculations. If the data-input port should be of the double width too, this would have a higher impact on the additional efforts necessary as the memory currently has only one write port. As a consequence either the memory would need a seconds write port or the write access would span two cycles. Each of these possibilities have in common that they will increase the implementation area and the complexity of the logic necessary for accessing the memory.

The third reason for the usage of the implementation's word width for the bus-interface ports is continuity. The word width throughout the implementation is pinned to a single specified word width. Using another word width for the interface interferes with this continuity.

The operands used in the finite-field operation, the elliptic-curve operations and the scalar multiplication are 256 bits wide. For a word width less than 256 bits, the width of the operands is wider than the data input of the implementation. Then, the operands cannot be transferred from the outside to the implementation within a single cycle. Therefore some options are discussed in the following. An option would be to implement a separate function in the implementation's bus interface that manages the storage of the data in memory at the appropriate location. For instance, a write to a certain address with a value of zero means that in the following cycles the operands for the integer addition are transfered. The interface, in turn, takes the data from the inputs, knows where to put the single words in memory and thus enables the correct calculation of the addition. This would be convenient for the accessing device but, in turn, needs extra logic within the chip. The actually chosen option is to enable direct memory-access to the internal memory from outside. Therefore it is assumed that the data is written at the appropriate addresses in the memory before the actual computation starts. The outside controller transfers the data words sequentially by writing into memory by bus access. The direct memory does not threaten the computation in terms of critical internally used variables. Only big data is stored in this memory, including coordinates of points and the scalar used for the scalar multiplication and therefore the arithmetic cannot be harmed by any wrongly addressed write. It also enables the debugging during development in a way that the test bench has convenient functionality to read and write the specific addresses in the memory of the implementation by bus access. On the one hand this functionality in the test bench reduces the implementation's area and on the other hand the programming in a high level language can be accomplished in a more convenient way than it can be done in hardware.

4.5.2 Bus Access

The interface to the test bench is implemented as a so called bus slave. Thus it only answers to bus request and therefore just reacts. In the following the procedures for reading and writing to the chip over the bus will be discussed.

The interface of the implementation has some signals for controlling the access. It has an address port for specifying the destination. This regards reading and writing into memory, reading properties such as the used word width or the number of registers, and starting an operation. When accessing the implementation by bus, the active-high chip-select signal has to be '1' and the active-high write-enable signal determines if either reading ('0') or writing ('1') is desired. According to the write-enable signal, the data input or data output will be involved in the operation. There is a busy signal, active high, which gets '1' at the first rising edge after a command is started. Starting at this point of time, until the computation is finished, the input at address, data input and data output is ignored. Each bus access is finished in the next cycle without any handshake-procedure or other signals involved. As long as no command is started and the busy signal is '0', each cycle can be used to either write or read. If a read operation is carried out by accessing the, the resulting data will be stable at the data output of the chip from the next rising edge for one cycle. Assuming a read operation at cycle i: the desired address to read must be stable at this point of time within cycle i when the clock has the next rising edge,

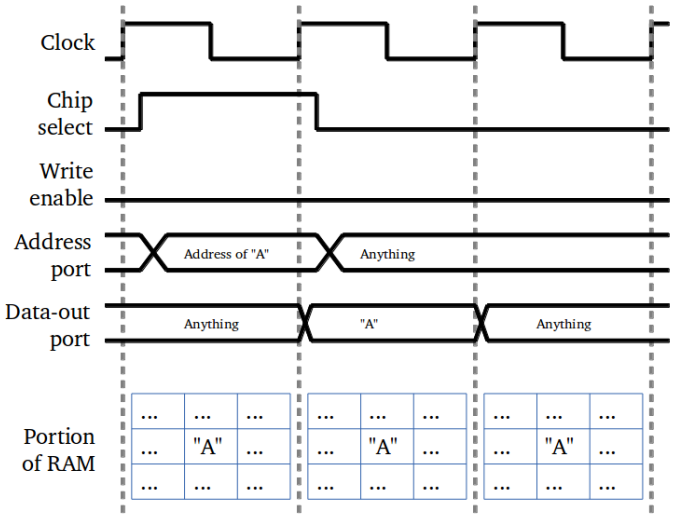

Figure 4.5: Illustration of a read bus-access from the RAM

immediately after the rising edge and for the whole duration of cycle $i + 1$, the result is stable at the data output. In the following, two illustrations of bus accesses are included and will be discussed. There, the signals are drawn over time and the addressed word's value in RAM is shown per cycle. The time is split by vertical gray dashed lines into clock cycles and as the architecture is synchronous, the logic calculates the new values within the cycles and the memory elements change their value at the cycle borders. The single pin signals are illustrated as a waveform that are either low or high and the transitions from these states as vertical lines. A multiple-pins signal (port) is illustrated with two horizontal lines and a textual description of the value between these lines. A change of the port's value is drawn with two crossing lines. It is, however, not intended to specify the duration of time the change of the value takes but just to illustrate that the value changes and takes a specific amount of time depending on the electrical details of the hardware. An illustration of a read bus-access is shown in Figure 4.5. There, the value "A" of a word in the RAM is read and stable at the data-output port during the next clock cycle after the bus access. The bus access starts anytime within a clock cycle but respecting the basic timing constraints. Coarsely, these constraints regard a certain short time before and after the clock's rising or falling edge, depending on what the hardware implementation is sensitive for. Further details about the mentioned timing constraints can be found in [22, Subsection 4.3.6]. The illustration the input signals change at the same point of time even though it is only necessary that all the relevant signals are set up until the end of the desired clock cycle.

The illustration in Figure 4.6 shows a write bus-access to the RAM. The same statements and constraints as for the read bus-access apply here. The differences of the write bus-access with respect to the read bus-access are the set high write-enable signal and the clock cycle the data transfer takes place. The data only needs to be stable at the data input for at most a single cycle, depending on the point of time relative to the rising edge of the clock, more precisely until the corresponding clock edge, but again respecting the time constraints as mentioned for the read bus-access. The data is processed or written

right after the next rising edge. In case of a bus access with a memory address, the data is written at the desired location at the rising edge of the clock, and in case of a command address, the operation is started at the rising edge.

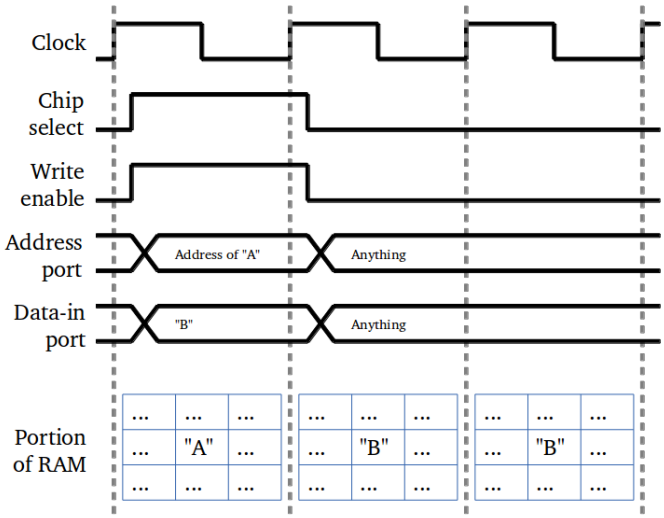

Figure 4.6: Illustration of a write bus-access into the RAM

4.5.3 Special Addresses

The implementation has some read-only properties that can be read by bus access. These serve information about the used implementation details that may vary according to the used word width. A write operation on these addresses will simply have no impact. The addresses space spanned by the used address width is organized in blocks and the discussed read-only properties are located at the very end at the highest addresses. Details for this address space separation are discussed at the beginning of this Section 4.5. How the properties' value can be read will be presented in Section 4.5.2.

During development, the supported word widths were tested and the test bench was implemented in such a way that, by specifying the used word width, the addresses of the read-only properties where calculated. By reading the values of these properties, the addresses of the commands were calculated as well as those of the single words of the registers for accessing the memory. This was implemented to ease the switching between the different word widths supported. In case of any switching, the addresses change and therefore did not have to be adapted to the according word width by hand. This had the advantage, during development, that changes in the number of used registers did not require the modification and maintenance of any addresses within the test bench.

The readable properties of the implementation will be discussed in the following. The first properties discussed will be the addresses and offsets necessary to define the operands used within operations. These values are necessary to load the addresses that will be used for the operation as their operands' location is variable. The operation of loading addresses and their purpose is discussed in Subsection 4.5.4.1. Further properties that can be read are the word-part's width of the addresses in memory and the minimum

width of the register part of the addresses due to the number of memory registers used. These values vary depending on the word width and are therefore used to calculate the destination addresses when writing into memory by bus. An important property is the start of the address block where the command addresses are located. These command addresses depend on the number of registers in memory and the used word width. By knowledge of the start address of this block, the explicit addresses can be calculated. Each operation that can be started has a unique command address, which will be discussed in the following Subsection 4.5.4. The number of register in the memory is also readable as property. Whereas not all registers are writable and their values are constant, this value serves the upper limit for accessing the memory. Which registers in the memory are writable will be discussed in Section 4.8. When using the chip for scalar multiplication, the registers in memory where the operands are stored have a fixed position. This regards both, where the inputs have to be written to and where the result is stored. The operands for the scalar multiplication are the point to multiply, the initial point's value and the scalar the point is multiplied with. General considerations about the scalar multiplication are listed in Subsection 3.5.1.4 and some details of the implementation presented here will be discussed in Subsection 4.6.2. Another property is the used implementation's word width. Whereas this is redundant information as this must be known to be able to read this property and a simulation error would occur with a not matching word word width, anyhow, it was used as verification during development as this information was checked against the specified one.

4.5.4 Command Addresses

The chip presented here can perform the necessary arithmetic up to the scalar multiplication for the twisted Edwards curve used within the Ed25519. Each of the implemented operations has its own unique address. The command addresses form a block without gaps as discussed in Section 4.5. The operations used within the computation of the scalar multiplication can be started separately by simple bus access as described in Subsection 4.5.2. By performing such a write bus-access, the operation gets started. Immediately after the start, the busy signal of the interface gets active as long as the computation lasts. During this duration any bus access is ignored and the operation can only be stopped by the asynchronous reset signal that results in a full reset. The arithmetics can be seen as broken up into layers of arithmetics as presented in Subsection 3.5.1 and further the operations available are listed below, grouped by their abstraction layer and ordered from the integer up to the scalar multiplication. Prior to the presentation of the operations, the procedure to specify the operands is presented. The discussion about the command addresses also include the necessary operands. The corresponding addresses must be loaded into the controlpath and the data must be transferred to memory before the operation is started. For the single operations, some restrictions must be respected regarding the chosen operand locations in memory. Due to the need of additional space for the intermediate results, some operations use fixed registers in memory to store these intermediates. Therefore those locations in memory are not allowed to be used as input operands. Those restrictions are noted when relevant. While an operations is performed, other operations may be used subsequently and their restrictions regarding the forbidden memory locations of the subsequent operations are inherited. For instance the finite-field multiplication uses the integer multiplication and the reduction and thus the restrictions of those operations also apply to the finite-field multiplication.

4.5.4.1 Loading Addresses of Operands

The loading of the addresses that are used for the operands can be seen as helping operation and is therefore even below integer arithmetic. Most operations that can be started have no fixed address to load the operands from. Specifically this is true for all operations from integer up to elliptic-curve cryptography regarding the abstraction levels of arithmetics. Thus, all the operations below the scalar multiplication have operands variably located in memory. Therefore, if an operation should be started, the addresses of the operands must already be well defined. If they are not, there will be simply no meaningful result as the addresses previously used will be reused again. When the operations with variable located operands are called internally, the locations within the memory were either fixed while development or depend on the operands and are passed to the subsequent operation.

The operands are loaded into the controlpath's internal storage. At the start of the next operation the stored addresses are used to perform the calculation. Loading the operands address is done by writing to a fixed position in the memory. Nevertheless it is not guaranteed that the width of the integer is equal to the word width and that this integer is aligned to the word borders within a register. As long as the implementation can assemble the addresses within a single register, this will be done: The implementation may be modified in the future and as a consequence the addresses for loading may not fit in a single register any more. The explicit addresses and the eventually necessary offsets within the words and the memory must be determined by reading the special properties as described in Subsection 4.5.3.

At maximum there are three parameters regarding the location in memory of the operands for the operation: Two addresses for the input operands, however not all operations use both addresses, and one address for the output where the result will be stored in. For instance the integer multiplication need a forth register and in such a special case a fixed register is used. Therefore this register must not be used as any of the other addresses. In the following it is assumed that, when two operands are used, operand "a" is the one on the left side of the operator and "b" is on the right side. Exemplary assume a subtraction $a - b$: here a is on the left side and b is on the right side. Beside the two input parameters a and b there is a third address that can be loaded into the controlpath and which determines the register where the result is stored in and is called "w". Therefore assume a subtraction with prior loaded addresses and data transferred to the memory at the corresponding locations. The inputs a, b have been transferred to registers and the result will be stored at w. The subtraction will be then performed calculating $w \leftarrow a - b$.

4.5.4.2 Integer Arithmetic

For this layer of abstraction command addresses for addition, subtraction and multiplication are defined. These three operations use both loadable read addresses to specify the input operands and the write address to specify the location the result will be stored. In contrast to addition and subtraction, the multiplication additionally uses a forth register to store the upper 256 bits of result. This is always the same fixed register in memory and therefore this register must not be used for any of the operands and the lower 256-bit register of the result.

4.5.4.3 Finite Field

The implementation has command addresses for these listed finite-field operations: addition, negation, subtraction, multiplication, inversion and division. These operation except the negation and inversion, take two operands. The two exceptions take only one operand. As previously stated for operations with two operands, operand a is fixed to be the operand on the left side and b on the right side. For single operand operations, operand a of the loadable addresses is used to specify the memory location for the operand. Due to the relation to finite-field operations, the fast reduction is listed here. It uses integer operations to calculate the result of any finite-field multiplication as discussed in Subsection 3.5.1.2. The result is written into the location in memory specified by w. Due to the length of the intermediate result, the register used by the normal integer multiplication and the optimized multiplication, three registers must not be used. Finite-field addition uses one fixed register for the intermediate result. The finite-field subtraction and negation in contrast do not need and reserves any additional register. The finite-field multiplication has one directly reserved register but also indirectly reserve the register for the reduction, the multiplication and the optimized multiplication as this operations are called while execution. The finite-field inversion uses the finite-field multiplication and needs a separate register to store intermediate results. Therefore it indirectly has the restrictions from finite-field multiplication of usable registers for its operands. Relative to the inversion, the finite-field division has one more reserved register for the intermediate result and inherits the restrictions from the inversion.

4.5.4.4 Group Arithmetic and Scalar Multiplication

For the elliptic-curve operations the doubling and addition two points on the elliptic curve are necessary. Each of these points has the three coordinates x, y and z and therefore it is not possible to specify the single coordinates of each point independently as only two input addresses are accessible via the load operation. Therefore this is solved by specifying the addresses of the registers where the x coordinates of both points are stored and the corresponding y and z coordinates are fixed to be at the following for the y coordinate and the one after the following for the z coordinate. The elliptic-curve operations, doubling and addition, call the underlying finite-field operation several times and therefore inherit almost all restrictions from all finite-field operations. The only exception is the one register reserved for the finite-field division as this operation is not used while computation but is necessary for computing the affine coordinates from the projective ones that are used throughout the implementation. The operations need five additional registers to store the intermediate results.

As a consequence of all the restrictions, regarding the reserved registers, there are eight registers left that can be used as inputs where for each point three consecutive registers are necessary and therefore the possible combination are limited.

The scalar multiplication, in turn, uses only fixed registers and allows no freedom in placing the operands. It expects the base point and the initial point's value to be transferred to specific registers. The scalar with which the base point is multiplied with is also expected to be transferred to a certain register.

4.6 The Controlpath

The controlpath of an implementation is responsible for the control of the sequence of operations. It is in control of the memory and thus responsible that the data is available for computation when necessary. The controlpath computes the addresses and signals them to the memory within the cycle before the data is needed. The controlpath also controls the operations of the datapath using the control signals. It switches between functionality, enable and disabled modules, and routes the signals in the right direction by controlling multiplexers. As the controlpath contains all the logic, it is natural that its complexity increases with the complexity and number of the implemented operations. In case of elliptic-curve cryptography there are several levels of abstraction necessary. All operations, except the integer arithmetic operations, are certain sequences of other operations.

The controlpath is implemented using a finite state machine and is described in Subsection 4.6.1. In this context each operation is represented as a state and is further split into single steps of execution. As a consequence of the composed operations, it may be necessary to switch to another operation during execution of one operation and afterwards continue at the previous operation at the next step. This is done via saving the currently used addresses, the current state and step before another operation is started and will be discussed in a paragraph in Subsection 4.6.1. The operations are implemented in such a way that there can be a no "loop" as this would harm the successful execution of the operations. A loop in this context is meant as follows: For instance assume an operation A that uses operation B. A loop would be if B in turn uses A or uses another operation that uses A. The implemented context-switching behavior has one separate entry for each operation. A loop then would cause the overwriting of the information whose operation started the operation A in the first instance and thus would cause an infinite loop.

4.6.1 State Machine

The purpose of the controlpath is to manage the memory and the datapath. It calculates and sends the addresses of the necessary data of the next cycle to the memory and sets the signals to the datapath whose basic operation should be done within it. The addresses that are necessary depend on the currently performed operation and the step of the operation. As stated in [22, Section 1.7] the controlpath can be implemented using a finite state machine, a stored program that is executed using a program counter and microcode instructions, or a combination of these techniques. In a microcode the operation and the operands are encoded. In the following, the techniques will be discussed in short. For further details see the discussion of Kaeslin in [22, Section 2.2] where the suitability and decision hints are discussed in detail. Details regarding the controlpath in general will be presented and in the following subsection implementation details will be presented as well.

A controlpath can be implemented using a stored program. During execution, the program counter is used to determine the current position in the program. One instruction of the program after the other is read and executed. One entry in the program determines which instruction to execute and specifies the involved operands. Such an entry is called microcode. The current microcode is fetched, interpreted and executed, and afterwards the program counter is modified accordingly. In case of data dependent branches, the program counter is set to the desired jump destination whereas in the other cases it is increased and points on the next instruction that is executed in the next cycle. Architectures with

program memory not necessarily require, that its entries have a fixed length and therefore may vary. Additionally, these architectures with program memory, program counter and microcoded instructions are considered general purpose. The sequence of operations, the used data and thus the result depends on the program in the program memory. Therefore, any algorithm can be executed on these architectures that fits into the available resources. Processors in personal computers are of this type of architecture.

A finite-state machine (FSM) has a finite set of states. It can only be in one state at a time but can change its state to one of its other states. The change of the state is called a state transition and such a transition can only take place at the border between two cycles when the clock signal has the edge the implementation is sensitive to. A FSM remains in its state until a state transition is triggered. For instance a state will run a fixed number of cycles until the result is ready and thus consists of a fixed number of steps. Within the steps, the addresses of the data's memory location that are used, for example, are increased until all words are processed. A state within a FSM may thus be the addition of two very long integers that are processed one word after another starting from least significant up to the most significant.

As Kaeslin in [22, Subsection 2.2.1] discussed, a FSM is suited for algorithm and sequences of operations that do not overly depend on the actual data. Dependence on data is meant is such a way that the values of the data influence which operation is performed and thus result in branches and different sequences of operations according to the data. It is further stated that an implementation with a general purpose processor with program memory and microcoded instructions is better suited for this kind of purpose. FSM architectures have advantages over general purpose processors. The complexity of the datapath, the necessary operations and memory requirements are known in detail and thus the architecture can be tailored to meet these requirements. It will therefore be more efficient and smaller implementing a FSM with some counters.

The controlpath was implemented using a finite-state machine. Each operation on the arithmetic's layers has its own state with one exception. This exception regards the integer addition and subtraction which share the same state but differ in the internally saved parameters. Further, there are some transfer operations that have an additional state. They are used implementation internally, are thus not accessible by bus access and share the same state but the transfers operation's detail again depend on some specific parameters. As already mentioned for the integer addition, integer subtraction and the transfer operation, some of the states need an additional storage in the controlpath. The storage is used for parameter and algorithmic variables. These storages of the single states only need at most just a few bits and are not accessible by bus access.

State Transitions. A FSM can have multiple states but can always be only in one state. It can change its state and this is called a state transition. The hardware implementation presented in this thesis may change its state during calculation of one operation or by a stimulus through a bus access.

The FSM of the implementation presented in this thesis has a certain set of possible states. Each of these operates with one or two operands that reside in the memory and calculate a result that is again stored in the memory. In each step within a state, certain operations of the datapath are signaled to be executed and the datapath in turn needs data from the memory. The datapath itself only executes basic tasks with the delivered data and the controlpath has to signal the memory which data of the memory is read and/or written. Therefore the controlpath was implemented to have three registers that

the memory's address inputs are connected to. During state transitions these registers are used for transferring the address-parameter to the next state. The design of calculation on different layers of arithmetic abstraction, executing operations on the underlying layer and afterwards jumping back to the "caller" was implemented in such a way which reminds of a context switch in computer software. If a state transition is triggered, the control-path automatically saves the current context and when this state in turn has finished its computation, the implemented automatism takes care of restoring the context of the caller. The context saving and restoring here regards the current status, the step within it, and the parameter addresses of the state. The sequence of operations and the probably necessary state transitions do never build a loop as discussed previously in this section. Note, that in here a loop is not considered to be as in programming languages, a *do-while loop* or some similar. Assume observing the states the FSM has during a scalar multiplication and drawing a graph of the states of the FSM, the single states are the nodes, the state transitions are the edges and every state transition results in a new edge and a new node. The graph has the scalar multiplication as its root and every elliptic-curve addition and doubling is a node that is connected to the scalar multiplication. Each of these additions and doublings has further finite-field-operations nodes connected to it and this is continued until integer operations at the end. After finishing an operation the edge is gone backwards. Therefore having no loop results in the circumstance that on every path from scalar multiplication to an integer operation no operation is listed twice. When implementing context saving, the available size of space to save the current context to is a crucial property as if it is too small, data will be lost and will end in an error. Within this implementation the, necessary storage space has a hard upper bound because no loop is possible. This upper bound is simply the number of possible states multiplied with the size of one context save. This circumstance led to the idea of assigning each state a fixed unique number and to use this as index in the context saving array where each element holds three addresses, the calling state and its step. As mentioned, some operations need further storage to save algorithmic variables but again, as no loop is ever generated within any path in the graph, each state is finished before it is called again and initializes its variables in the first cycle.

4.6.2 Details of Operations

The preliminary description of the necessary operations and arithmetics in Subsection 3.5.1 can be extended with the some details. These notable details of the implementation presented in this thesis are discussed in this section.

The implemented integer addition and subtraction operations behave like unsigned integer operations known from software programming-languages like *C* or *C++* and are discussed in Subsection 3.5.1.1. If the addition results in a number that does not fit into 256 bits an overrun happens. Then the 257-th bit gets cut off and internally stored as the carry bit. The similar applies for subtraction: If the result gets less than zero, then a borrow bit is generated and stored. The result of $a-b$ is then $\forall a, b | 0 \leq a < b < 2^{256}, a-b = 2^{256}+a-b$.

The multiplication of the integer arithmetic is performed using the product-scanning form as discussed in Subsection 3.5.1.1 and again behaves like the unsigned integer multiplication of software programming-languages even though here no overrun will be generated and the upper 256 bits will be saved. No overrun can occur because of the operands which are limited to 256 bits and the 512-bit result fits into the two registers used. The product-scanning form of the multiplication was implemented using an accumulator. The

listing of this form was taken from [19, Subsection 2.1.2] and slightly modified as presented in Algorithm 5. The modification regards the used accumulator that is here width optimized in contrast to the template product-scanning algorithm. Here, the inputs a, b are multiple words long and their single m-bit words get accessed by $a[x]$ and $b[y]$ respectively, where x, y are the indices ranging from the least significant word 0 to the most significant $n-1$. In the same manner but with a different number of words, the accumulator r and the result c are accessed: $r[z]$, $c[w]$. The accumulator r is of size $2m + \lceil log_2(256/m) \rceil$

Algorithm 5 Product-scanning form of a multiprecision multiplication with an accumulator

```
procedure MUL( a, b )                          ▷ a, b are the operands
    r ← 0
    for k = 0 to 2t − 2 do
        for each element of i, j|i + j = k, 0 ≤ i, j ≤ t − 1 do
            r ← r + a[i] · b[j]
        end for
        c[k] ← r[0]
        r ← r >> m                             ▷ Shift right by m bits
    end for
    c[2t − 1] ← r[0]
    return c                                   ▷ Return the result c
end procedure
```

because at most $256/m$ chunks of $2m$-bit integers are added and can at most result into $\lceil log_2(256/m) \rceil$ extra bits before one word is transferred to c and the value of r is shifted right. Note that the accumulator must be chosen wider if the amount of shifted out bits is less than the extra $\lceil log_2(256/m) \rceil$ bits necessary for the carry bits of the additions when summing up the single products in the accumulator.

The reduction is used for finite-field multiplication. There, the two operands are multiplied in integer manner and afterwards reduced to be less than the finite field's prime. The operands are elements of the finite field and thus are integers in the range $\{0, 1, ..., p-1\}$, where p is the prime of the finite field. Due to the special form of the used prime, a fast reduction technique can be used. It consists of some sub steps, including a multiplication of the upper 255 bits with 19 (a five-bit integer). Although performing a usual multiprecision integer-multiplication is possible, this multiplication can be replaced by a specialized multiplication that takes the very limited width of the operand into account. This specialized multiplication is tailored to multiply a 256-bit integer with a small constant: it is basically implemented as the multiprecision multiplication in the product scanning form. For each word in the result it multiplies all words of the two operands whose added indices are equal the index of the destination word and sums up those products. But here the information of the number of words the small operand contains is available and thus used to skip multiplications that anyway result in zero will. For detailed information see the discussion in Subsection 3.5.1.2.

The elliptic-curve point-doubling was implemented using the dedicated doubling formulas. The elliptic-curve addition and doubling were implemented using five 256-bit register storing intermediate results. The explicit formulas for doubling and addition were taken from [11, Chapter 6] and are listed in Algorithm 6 and Algorithm 7. There, all the operations are meant to be performed within the finite field. The listed formulas

Algorithm 6 Twisted-Edwards addition

procedure TWISTEDADD($P_1 = (X_1, Y_1, Z_1), P_2 = (X_2, Y_2, Z_2), a, d$)
$A \leftarrow Z_1 \cdot Z_2$
$B \leftarrow A^2$
$C \leftarrow X_1 \cdot X_2$
$D \leftarrow Y_1 \cdot Y_2$
$E \leftarrow d \cdot C \cdot D$
$F \leftarrow B - E$
$G \leftarrow B + E$
$X_3 \leftarrow A \cdot F \cdot ((X_1 + Y_1) \cdot (X_2 + Y_2) - C - D)$
$Y_3 \leftarrow A \cdot G \cdot (D - a \cdot C)$
$Z_3 \leftarrow F \cdot G$
return $P_3 = (X_3, Y_3, Z_3)$
end procedure

Algorithm 7 Twisted-Edwards doubling

procedure TWISTEDDBL($P_1 = (X_1, Y_1, Z_1), a$)
$B \leftarrow (X_1 + Y_1)^2$
$C \leftarrow X_1^2$
$D \leftarrow Y_1^2$
$E \leftarrow aC$
$F \leftarrow E + D$
$H \leftarrow Z_1^2$
$J \leftarrow F - 2H$
$X_3 \leftarrow (B - C - D) \cdot J$
$Y_3 \leftarrow F \cdot (E - D)$
$Z_3 \leftarrow F \cdot J$
return $P_3 = (X_3, Y_3, Z_3)$
end procedure

need seven registers for the intermediate variables. Due to the used curve, its parameter $a = -1$ influences the formulas in such a way that the multiplication is replaced with a finite-field negation. To reduce the number of necessary registers of the implementation, these formulas were slightly modified whereas the results are equal to the original formulas. As the implementation does not have a separate storage for the resulting third point, one of the points' registers which is used for the inputs, will be used for the destination registers too. Therefore, it had to be ensured that no input is overwritten before its value is used and thus all lines in the procedure that list one of the coordinates as input must be processed before this input is overwritten. To find out how to reduce the number of necessary registers, the dependencies between the variables has to be analyzed. When a variable is not used anymore it can be used for the other calculations. As soon as the input-dependent lines are processed, the destination registers can be used to store other intermediate results before they get their final value. In the implemented twisted-Edwards addition, two variables relative to the original formulas were saved by reusing the register B in line $G \leftarrow B + E$ as the variable B is not used after this point of execution and resulting in $B \leftarrow B + E$. Further, by using the Z_3 destination register for intermediate results and replacing B with Z_3, one register less is used. The modified formulas are listed in Algorithm 8 By doing similar replacements and modifications for the doubling formulas, the number of necessary registers were reduced from seven to five. Here the line $H \leftarrow Z_1^2$ is moved after the $D \leftarrow Y_1^2$ line and thus after the H-line all values of the inputs are used and the destination registers can be used without threatening the inputs. The variable E gets replaced with $-C$ in all lines as the parameter a is minus one and thus only six registers are used for intermediate results. The variable B is only used once when computing X_3 and thus by putting the line $X_3 \leftarrow B - C$ after the H-line and modifying the final X_3 line to $X_3 \leftarrow (X_3 - D) \cdot J$ the variable B can be used for F. Therefore F can be replaced by B and thus one register less is used. By moving the Y_3 line to end behind the Z_3 line and as it is not dependent on J it can be used for storing the result of J as used for X_3 and Z_3. Here a third variable less is used for the computation of a twisted-Edwards doubling. Because the twisted-Edwards-curve addition needs five, the free fifth register while calculating the twisted-Edwards doubling is used during computation for intermediate results. The resulting formulas are listed in Algorithm 9. Some of

Algorithm 8 Modified twisted-Edwards addition

procedure TWISTEDADDMODIFIED($P_1 = (X_1, Y_1, Z_1), P_2 = (X_2, Y_2, Z_2), a, d$)
- $A \leftarrow Z_1 \cdot Z_2$
- $Z_3 \leftarrow A^2$
- $C \leftarrow X_1 \cdot X_2$
- $D \leftarrow Y_1 \cdot Y_2$
- $E \leftarrow d \cdot C \cdot D$
- $F \leftarrow Z_3 - E$
- $Z_3 \leftarrow Z_3 + E$
- $X_3 \leftarrow A \cdot F \cdot ((X_1 + Y_1) \cdot (X_2 + Y_2) - C - D)$
- $Y_3 \leftarrow A \cdot Z_3 \cdot (D - a \cdot C)$
- $Z_3 \leftarrow F \cdot Z_3$
- **return** $P_3 = (X_3, Y_3, Z_3)$

end procedure

the formulas in the addition Algorithm 8 and Algorithm 9 are still concatenations of more

Algorithm 9 Modified twisted-Edwards doubling

procedure TWISTEDDBLMODIFIED($P_1 = (X_1, Y_1, Z_1), a$)
$B \leftarrow (X_1 + Y_1)^2$
$C \leftarrow X_1^2$
$D \leftarrow Y_1^2$
$H \leftarrow Z_1^2$
$X_3 \leftarrow B - C$
$B \leftarrow D - C$
$Y_3 \leftarrow B - 2H$
$X_3 \leftarrow (X_3 - D) \cdot Y_3$
$Z_3 \leftarrow B \cdot Y_3$
$Y_3 \leftarrow B \cdot (-C - D)$
return $P_3 = (X_3, Y_3, Z_3)$
end procedure

than one finite field operation and are split into single operation in the implementation.

Different strategies for implementing the scalar multiplication were discussed in Subsection 3.5.1.4. The implementation presented in this thesis uses double-and-add. As presented preliminary at first in each iteration the point Q is doubled, the current bit in the scalar is read and if this bit is set, the base point B is added to Q.

4.7 The Datapath

The datapath consists of the data-processing units of the hardware. It takes the inputs and calculates the results due to the signals received from the controlpath. As stated in [22, Section 1.7] the datapath does not only operate on data input but may also have some buffers and accumulators. Further it is stated that the operations in a datapath often include arithmetic and logic operations but can also include the functionality of switching and routing.

In Figure 4.7 the datapath is shown including the three basic modules with their control signals. Please note the only extra signal that crosses the main data-flow direction from the top to the bottom. This is the carry-/borrow- bit signal that can be used for a conditional copy operation after a finished addition or subtraction. There can always be only one module that directs its data to the memory and according to the architecture of the memory with one write port there would not be a purpose of a second result.

In the following the included operations in the implemented datapath will be presented in short. These operations are the integer arithmetic operations addition, subtraction and multiplication, and the copy functions. All the operations in the upper abstraction levels are certain combinations of these few operations.

4.7.1 Copy Operation

The copy functionality in general takes the input data received from the memory, may or may not modify it and sets data signals to the data output directed to the memory. Beside the very basic one-to-one copy, the implementation also has a conditional copy, a copy function that can set the most significant bit zero and one that sets a whole word zero. The conditional copy selects the data directed to the output depending on a condition:

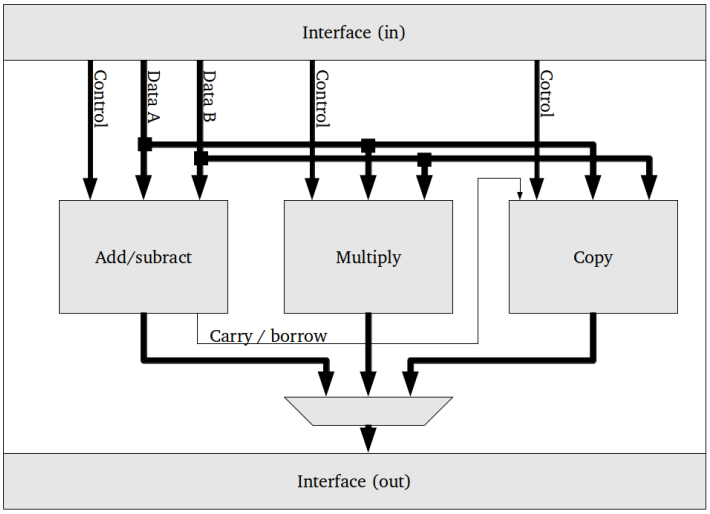

Figure 4.7: Illustration of the datapath with its modules and signals

Dependent on the set condition bit, either data from port A or port B of the memory interface is directed back to the output port to the memory. This conditional copy further has an automatism implemented which stores the generated carry and borrow bit of the addition and respectively the subtraction. The automatism prepares a conditional copy dependent on the result of the operation and thus can be used directly after one of these operations. In Figure 4.8 an illustration of the different modes of the copy operations is shown. There, the different modes are signaled by the controlpath and controls which data

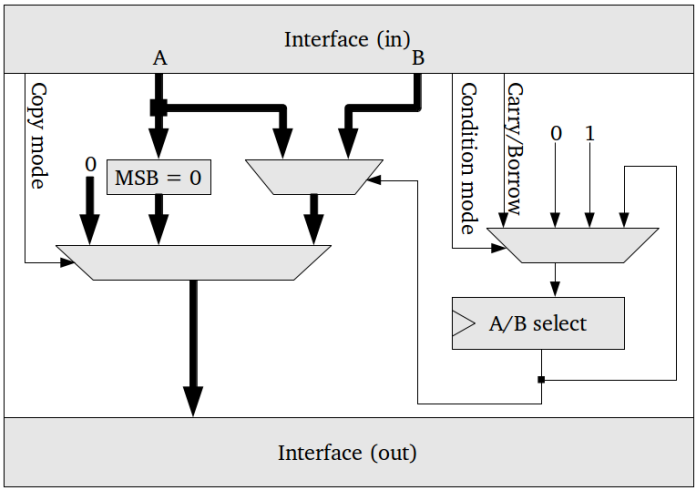

Figure 4.8: The copy operation's modes

is signaled to the RAM. On the right the condition register is shown with the different modes.

4.7.2 Addition and Subtraction

Addition and subtraction are two very basic operations on which all the other operations depend on. In this implementation the width of the operands are basically 256 bits wide but the addition and subtraction unit's word width is in general less than this. As discussed in Subsections 3.5.1.1 and 4.6.2, addition and subtraction are implemented as multiprecision operations which process one word after another, starting from the least significant up to the most significant. The utilized operations in the datapath do not care about which word is currently processed and always do the same job. They generate a carry and respectively a borrow bit automatically which is used the next time the addition or subtraction is carried out and the bit was not reset by a controlpath signal. The addition is implemented using by extending the inputs by one bit and after addition returning this highest bit as carry and the rest as the regular result. The subtraction is performed by extending the operand A with an additional '1' bit, subtracting operand B of this and toggling the additional most significant bit in the result. By doing so, the most significant bit, representing the borrow bit, is set '1' if operand B was bigger and set '0' otherwise. An illustration of the described module can be seen in Figure 4.9. On the left the mode

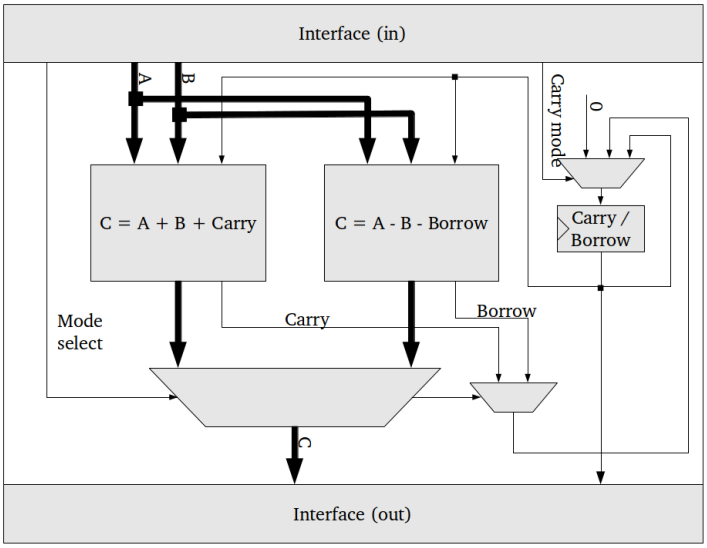

Figure 4.9: The addition and subtraction module in the datapath

selects the desired output and on the right the generated carry and borrow bit register with the update and clear functionality is shown.

4.7.3 Multiplication

The integer-multiplication functionality in the datapath is used for usual multiprecision multiplication and the special optimized multiprecision multiplication with 19 used in the fast reduction. It consists of a dedicated multiplier with the implementation-wide used word width and an accumulator necessary for the multiprecision multiplication. The multiplication functionality in the datapath has several signals to control the behavior of the datapath due to the complexity of the multiprecision multiplication presented in the Subsections 3.5.1.1 and 4.6.2. Beside the chip-select for the multiplication, there are

signals to determine the operation and signals to specify the operands. The operation can either clear the accumulator, multiply and accumulate, multiply, accumulate and shift the least significant word out or shift the least significant word out. Due to the multiplication with 19, signals to specify the operands are necessary. For this multiplication mode the signals specify whether operand A is the data received from input A or the concatenation of the data from input B and the most significant bit of input A. The signals also specify whether operand B is the data from input B or the integer 19. In the illustration in Figure

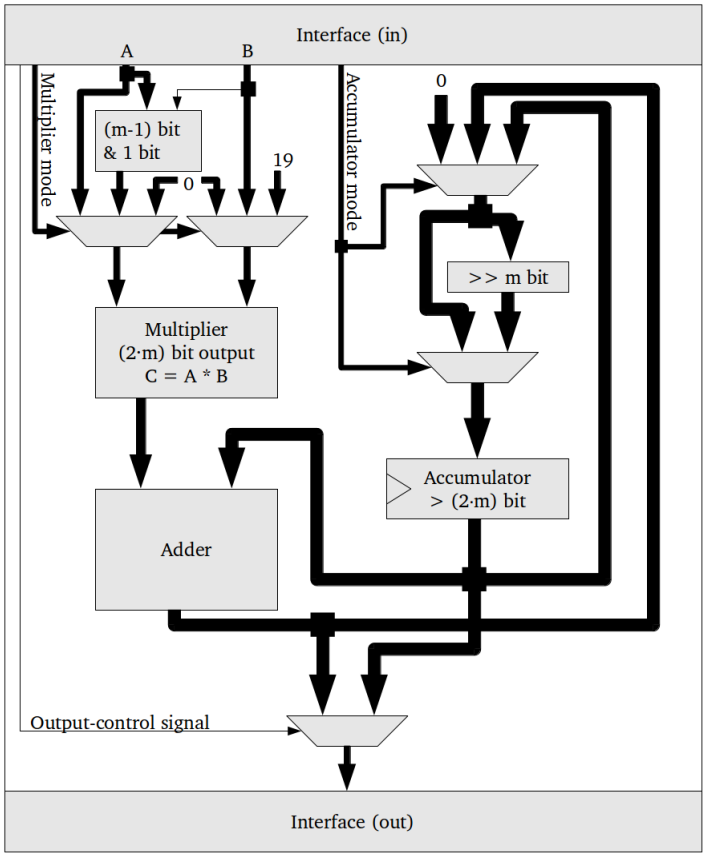

Figure 4.10: The multiplication module with the multiplier and the accumulator

4.10 the multiplier, the adder and the register of the accumulator are shown. The different line widths are used to point the different widths of the signals out. Due to the different modes the multiplication supports, the module consists of some multiplexers which are used to switch the single data that is actually used. The controlpath sets the signals that control the accumulator and the output mode. Due to according signals, the accumulator is either updated with the actual multiplier result added to the previous accumulator value, the previous accumulator value or zero and additionally the value can be shifted right by one word width. Note that the adder of the multiplication unit is a separate one and not the adder of the addition unit.

4.8 The Memory

The memory in the implementation is on the one hand, responsible for storing operands, results and intermediate results. Therefore a random access memory is used. The memory is also used to store constants that are relevant for the Ed25519 computations in the read-only memory. Both types are accessible through the memory interface, however the ROM is not writable. In every cycle one word can be written and two words can be read at once and therefore has one write-address- and one write-data-signal input and two read-address-signal inputs and two read-data-signal outputs. Both types of memories are organized in registers of 256 bits width. Each of these registers is split into words of the same size as the implementation-wide word width, each of these words has a separate address and can be accessed using this address. The number of words n each register consists of is the result of $n = \frac{256}{m}$, where m is the implementation-wide used word width. Therefore this number depends on the used word width. As discussed in Subsection 4.4.1 the word width is restricted to a power of two. Due to this, the address of the first word of a register is the last word's address of the previous register plus one. The address block of the RAM is located at the very beginning at zero. The address block of the ROM is located right after the one of the RAM, leaving no gap between. An address has a register part and may have a word part. In case of 256 bits word width, there is only one word used per register and an address has no separate word-part. In the other cases an address can be split into them at bit borders. Due to the restriction regarding the word width to be a power of two, there are no gaps with unused addresses in the address space the memory spans.

4.8.1 Random Access Memory

The random access memory allows read and write operations and thus it is natural that the values stored in the RAM change while the chip is running.

The RAM consists of 16 registers of 256 bits width and thus 4 096 bits. Six of these are used for storing the base point of the twisted-Edwards curve and the intermediate point during the calculation of the scalar multiplication and after finishing the result point. For the scalar multiplication one further register is needed for storing the scalar. The twisted-Edwards addition and doubling need five register for the storage of the intermediate values. The four remaining registers are necessary for the multiprecision multiplication, the optimized multiprecision multiplication, finite-field operations (addition and multiplication share one reserved register), and the fast reduction. The RAM in this implementation is instantiated simply by defining the signal as an array of words. Each of these words in turn is of VHDL's unsigned type with the implementation-wide used word width. This is considered natural as it reflects the circumstances of the architecture with words of the defined word width. Although it can be the case that there is no word-part within an address, when a 256 bits word width is used, each address always has a register part. In the RAM's logic the index of the word within the array is calculated with the register part of an address multiplied with the number of words per register. In case of 256 bits word width, the number of words per register is 1 and the index is then equal to the register index.

4.8.2 Read-Only Memory

For this type of memory only read access is possible and the values stored in it do not change during execution time. In this implementation the ROM is used to store the prime $2^{255} - 19$, the number related to the prime and used for the inversion by exponentiation $2^{255} - 21$ and the d parameter of the used twisted-Edwards curve in Ed25519 $-\frac{12665}{12666}$. These values are calculated during compile time using the built-in VHDL capabilities of the *unsigned* data-type. For the calculation of the d parameter by VHDL it was necessary that some finite-field operations were implemented. These operations are realized again using the capabilities of the *unsigned* data-type. The access to the three ROM entries where implemented using simple range checking of the address and subtracting the first address of the RAM entry off the address. The three constants that are necessary result in 768 bits of ROM.

4.9 Verification by Software

Like software, hardware should be verified in order to prove that the implemented code behaves as intended. As the name indicates, hardware has a direct connection to physics and therefore the verification of hardware is at least as computationally hard as software, which has a pure digital environment. In this section, the simulation and verification of hardware will be discussed and afterwards the used techniques and details will be presented. The interested reader might examine Kaeslin's discussion about functional verification [22, Chapter 3] that relates to the efforts made for simulation and verification of the presented hardware implementation. Beside the functional verification, Kaeslin in [22, Chapter 12] discusses verification techniques to check the implementation against errors within the synthesis. These checks are covered by the IAIK-design-flow and are discussed in short in Section 4.3.

Software is executed in a pure ideal, digital environment. Hardware simulations take complex models into account and is therefore considered to be a computational intensive task. Software in general can be executed, tested and the functionality verified on the target platform in seconds. The verification of hardware implementation can be done by simulations, running programmable hardware or by building the specific hardware and test it in realistic environment. These techniques have in common that they imply more effort than necessary for verification of comparable software.

Simulations can be carried out within minutes and the result can be inspected as discussed in Subsection 4.3.2. The problem with this is that it is just a simulation and can only be as accurate as the used models. Hardware, written in some hardware-description language, has several intermediate stages until the final layout of the chip is reached. The single stages towards the final result are of increasing complexity and contain more detailed models of the physics. A simulation with less details can be carried out in acceptable time and covering more test cases. More detailed models of the simulated hardware can be used to verify specific critical functionality and based on this, the developer can make evident statements. Beside the bare simulation time, the time to reach a level of complexity is the sum of the preceding stages.

The second mentioned possibility, namely to use programmable hardware, is an option where some abstracted functionality is tested. It can hardly be used for a detailed simulation of a design where the exact behavior of single gates or exact timing are subject of interest because of the substantial difference between those two systems: On the one

side the hardware implementation targeted to get a chip and on the other side a programmable chip which has fixed logics and gets a new routing between the single blocks. Due to the naturally resulting differing locations of the blocks, the routing of wires is substantial different and thus the signals' propagation time is. Additional to the limited simulation capabilities of this hardware, buying the necessary equipment can already be a serious barrier. For detailed information about *field-programmable logic* see [22, Subsection 1.2.3].

The third possibility of producing the hardware directly will be impracticable and no option in the general case. This takes a significant amount of time and is to costly in most cases to verify functionality at an early stage of development.

As discussed in this paragraph, simulation can be used to verify HDL code without extra costs and was thus used for the hardware implementation presented here. Prior the actual implementation of the hardware, a highlevel model was created and parallel to the hardware a test bench was developed. The highlevel model together with the test bench were used to verify the functionality of the implementation on an automatic level and will be discussed in the following sub sections.

4.9.1 Highlevel Model

A highlevel model (HLM) is a software program that is used to simulate the hardware. It behaves as the modeled system regarding the algorithm at a certain abstraction level. The included details depend on the chosen abstraction level. The HLM provides the ability to evaluate and debug algorithmic details on an abstracted level. It is further used to generate test data which consists of input data and the corresponding outputs. The DUT can then be tested for correct calculations when the inputs were applied and the calculation returns the result.

The HLM presented here was implemented before the actual hardware was. On the one hand it was used to get an idea of the hardware architecture and its components and verify the used algorithms. On the other hand it was utilized to produce verification data for the simulation of the hardware implementation. The correctness of the highlevel model was verified using the C reference-implementation and is presented in Subsection 4.9.1.2.

The HLM was written in the programming language Java. As discussed in Subsection 3.5.1 the arithmetic consists of multiple abstracted layers and some arithmetic operations within these layers. It has a separate class for each of these operations. As it is for the hardware implementation, the model has a centrally defined word width. The modeled operations are implemented in such a way that the operations split the operands into words and process them word by word as it is done in hardware. For modeling the operands the *java.math.BigInteger* class was used because it supports the basic arithmetic operations and can handle integer numbers much wider than the at least necessary 256 bits. The operations itself were realized processing not the whole 256 bits at once but word by word although the *BigInteger* supports all the necessary operations. This was done in order to get more detailed insight before the actual hardware was implemented and thus not to struggle with algorithmic details in the more complicated hardware-description-language environment. For the integer operations, the algorithms from [19, Subsections 2.2.1 and 2.2.2] were implemented. The multiprecision multiplication was tested with both prominent variants, the product-scanning and operand-scanning form, because at this time the one to be used within the hardware implementation had not been chosen yet.

Although the *BigInteger* class has a built-in modulo operation which could be used to calculate the results for the finite-field operations, the fast reduction was implemented using the same algorithm as latterly used for the hardware itself and discussed in Subsection 3.5.1.2.

The elliptic-curve operations in the HLM are based on the same formulas for twisted-Edwards curves of [11, Section X] as used for the hardware implementation but not including the optimizations discussed in Subsection 4.6.2. By using the original formulas, the HLM needs seven 256-bit intermediate-result variables and can be used to verify that the optimizations lead to a correct algorithm.

The HLM's scalar multiplication was implemented using the double-and-add method. For each bit, a twisted-Edwards-curve doubling was performed and, according to the bit's value, either a twisted-Edwards-curve addition with the base point is carried out or the this addition is skipped.

4.9.1.1 Design-Flow Integration

The IAIK-design-flow provides functionality based on central makefile. For using the HLM within the design flow, a console based program is necessary which takes its parameter by command line.

The highlevel model was built in such a way as to take parameters from the command line that specify the word width to use internally and for which of the modeled operations test data should be generated. With the parametrization by command line, the HLM integrates in the IAIK-design-flow. The design flow checks for test data before a simulation is started and invokes the Java HLM which in turn outputs to the console. This output contains debug and status messages as well as the test data with a predefined pattern and is written on the appropriate file by the design flow. The design flow catches the HLM's output and extracts only lines with the defined pattern in it. The pattern is "TCL", but the position is not limited to only line starts. The HLM prints lines like "TCL_add 1 5 6" to the console, the design flow does not filter it out and writes it on the simulation input file. The design flow takes this input file and tries to execute it. Therefore the test bench in this example is expected to have a function defined and named "TCL_add" taking three parameters. According to the currently selected tests in the makefile different test data gets generated.

In the following Subsection 4.9.2 the test bench is presented which takes the generated test data.

4.9.1.2 Verification of the Highlevel Model

The highlevel model is used to verify the functionality of the hardware implementation. Therefore it delivers the input data and the expected result. This expected result was calculated within the highlevel model with a model of the hardware implementation. The highlevel is not the first node in the chain to verify the hardware implementation and therefore its functionality must be checked for correctness.

The highlevel model is written in Java and has convenient capabilities to calculate twisted-Edwards curve operations. To verify that the implemented highlevel model indeed calculates the operations correctly the EdDSA C-reference-implementation of Bernstein, Duif, Lange, Schwabe, Yang available in the SUPERCOP[5] toolkit was used. At the time

[5]The website of the SUPERCOP benchmarking-toolkit is `http://bench.cr.yp.to/supercop.html`.

of the implementation of the highlevel model the latest available version[6] of the toolkit was used. The SUPERCOP toolkit has been developed for measuring the performance of cryptographic software like hash function and secret- as well as public-key cryptography systems.

The C reference-implementation was used to verify the results of the scalar multiplication implemented in the highlevel model Java. It was extended to take the command-line arguments, execute either the twisted Edwards curve point addition, doubling or the scalar multiplication with the desired input. Therefore the algorithmic was not modified but only the capabilities of specifying the inputs and printing the result hexadecimally coded to the *standard out* of the console application. The reference implementation was used as binary, compiled with the GNU Compiler Collection (GCC) C-compiler[7].

The extended command-line-parsing capabilities enable to switch between different operations and modes. The testable operations are the twisted-Edwards curve operations point addition and point doubling as well as the scalar multiplication. Additionally the reference implementation's used base point can be printed to output and enabled the verification whether the calculated base point in the highlevel model is correct. The procedure of verification of the operations was as follows: Firstly the highlevel model executes the reference implementation with some parameters specifying the functionality to test and some parameter that control the input data. The high level model captures the generated output that includes the input parameters passed to the reference implementation's arithmetic and the calculated result. These values are hexadecimally coded and are parsed by the highlevel model. Lastly the highlevel calculates the operation on its own with the captured input parameters and compares this with the captured result.

4.9.2 Test Bench

The test bench's purpose is to interact with the DUT, simulate stimuli and read the DUT's output. It must be tailored for the DUT to be able to interact with it. To enable the timing of critical interaction with the DUT, the test bench needs either control of the simulation time or at least knowledge of it to act appropriately.

The test bench implemented for this hardware implementation is written in TCL. The very basic functionality of simulation time controlling, and reading and writing single as well as collections of pins is provided by the IAIK-design-flow.

4.9.2.1 Hexadecimal-String Manipulation

The test bench for the hardware implementation, on the base of the TCL functions provided by the IAIK-design-flow, is equipped with hexadecimal-number functionality. This was necessary due to the limitations of integer variables within TCL 8.4 as therein integers are limited to be 32 bits wide. The implemented hexadecimal functionality includes creation, arithmetic, bitwise, manipulation and analysis operations. The value of the hexadecimal numbers are represented and stored as strings. The implemented functions parse and process the strings character wise but are implemented carefully to not mess up with single bits where necessary.

The test bench has functionality to create some special hexadecimal numbers: Firstly the important conversion from usual integer to hexadecimal-number string. Secondly

[6]The used version is available at `http://hyperelliptic.org/ebats/supercop-20120225.tar.bz2`.
[7]Available for several platforms at the website `http://gcc.gnu.org/`.

hexadecimal numbers with all bits being set can be created and the number of ones is specified by parameter. Finally a function was implemented that delivers a hexadecimal number that is filled with ones in a specified range: from least significant up to a specified bit no bit is set, then up to the second specified border all bits are set. The arithmetic operations that are implemented are the addition and the subtraction. There are two versions of the operations - one that operates on two hexadecimal numbers and the other which operates on an integer and a hexadecimal number. The processing of the string is done character wise and is like the previously discussed multiprecision addition and subtraction, see Subsection 3.5.1.1, with a word width of four bits. The bitwise operations are *and*, *or*, *exclusive or* and *not*. Again, the single characters are processed one after another. In the case that two hexadecimal numbers are processed that differ in width, the shorter hexadecimal number's missing bits are assumed zero. For the *not* operation, the width in bits of the number must be specified since the hexadecimal number has no predefined upper limit for its width as the usual integer variables but it is only limited by the physical memory of the simulation-running computer. Beside the integer and bitwise operations, other manipulation functions are implemented. These include left and right shift operation, concatenation and cutting operation. As analysis functions a simple compare function is included, which of the two operands is greater, as well as difference visualization function.

4.9.2.2 Stimuli, Readouts and Checks

The test bench includes functionality for controlling the simulation time, interaction with the device-under-test and hexadecimal number operations. As soon as these are available, functions on a higher abstraction level can be realized. The bus access in general is discussed in Subsection 4.5.2 but in the following, the bus-access functions and the functions for the start of operations on the chips are presented from the test bench's point of view.

The lowest level in the test-bench-to-chip interaction hierarchy is represented by functions to set and read pins and is available by the basic functionality of the IAIK-design-flow. The next higher is to perform bus access with reading and writing to addresses. Therefore signals have to be sent in a certain combination and sequence. The bus-access *read*-function as well as the *write*-function starts with setting the chip's *chip-select* signal active and the corresponding value to the *write-enable* signal. Further, the desired address has to be applied to the address-input port and if the operation is *writing* into the chip, the data has to be applied to the data-input port. If it is a reading access, the result is at the chip's output during the next clock cycle. Each bus access always takes one cycle, with the exception that the result of a read bus-access is available during the next cycle, but multiple consecutive can be performed leaving no unused cycle in between. Thus it is possible to perform any sequences of n bus accesses, *read* and *write*, in n cycles, as long as no operation on the chip is started. The whole process of performing an operation on the chip is a concatenation of bus accesses, waiting until the busy signal gets inactive and further bus accesses to reading the result out of the memory. The necessary parameters regarding the right location of the operands depend on the single operations. Because the operands for the operations are always 256 bits wide, the test bench was extended with functionality to read and write 256 bits wide integers by consecutive bus accesses, each transferring a single word. Except for the scalar multiplication, the addresses of the operands have no fixed location and are discussed in Subsection 4.5.4.1. The addresses have to be specified by writing the register numbers to a fixed location and afterwards start the *load-addresses*

operation by another write to the corresponding command address. This is the same for all operations except the values of the transfers and therefore this sequence of bus accesses a TCL function is provided too. At this point of functionality, a testing environment can be conveniently implemented. The test bench is implemented in such a way that it has some high-level test-functions which are called by recorded and filtered output of the Java high-level model. These test-functions take care of loading the addresses, transferring the data into memory, starting the operations, waiting for its finish, reading out the result and comparing the read with the expected one.

Chapter 5

Results

In this chapter, the results for the hardware implementation will be presented. First, the key facts, the used tools and technology, and the figures of merit to judge the synthesized hardware will be summarized and presented. A more detailed discussion about those aspects can be found in Chapter 4. Second, the results of different word widths of this work's implementation are listed and analyzed. Finally, the results that were judged best will be compared to previous work.

5.1 Synthesis Aspects

The judgment of the efficiency in context of low resource hardware-implementations will be done by using the necessary area after the synthesis and the time to execute a scalar multiplication. These parameters substantially depend on the used word width. The hardware implementation was implemented to support different word widths: 16, 32, 64, 128 and 256. The data processing modules, the adder and the multiplier, are instantiated using the word width. The syntheses were performed with the different word widths supported, although the multiply-accumulate module consumes a huge amount of area when 64 bits and above were used. Therefore the results for 64 bit word width is already considered questionable in context of low resource hardware-implementations and results above 64 bits are added for completeness as they are basically supported by the implementation.

For the synthesis, the IAIK design flow was used. This utilizes the Cadence Encounter RTL compiler. The ams hitkit 4.0 with the 0.35 μm C35B4 technology was used for synthesis and was integrated into the design flow as discussed in Section 4.3. The C35B4 is a Complementary Metal Oxide Semiconductor (CMOS) technology with four metal layers.

The area is measured in μm^2 by the synthesis tool but is presented here in gate equivalents (GE) as this represents a technology independent measurement for the area. A gate equivalent is the area of one NAND gate (see Section 4.1). The time the operations take to finish is measured in cycles and the listed values were measured during the simulations. For simulations, the Cadence NCSim suite was utilized. For the comparison of efficiency, the area-time product was used and is calculated by multiplying the area (GE) with the time (cycles). For measuring the area, the time and the maximum frequency of the hardware implementation, it was synthesized with one megahertz. This frequency is low enough to obtain a synthesis result with no extra optimization from the synthesis tool. As a result, the synthesis delivers the critical path. The critical path is the longest

Table 5.1: Synthesis results for different word widths

Word width [bit]	$\mathbf{A}_{total}$ [GE]	**T** [cycles]	**AT-prod.** [MGE]	$\mathbf{A}_{no-mem}$ [GE]	$\mathbf{A}_{no-mem}$**T-prod.** [MGE]	$\mathbf{F}_{max}$ [MHz]
16	44 440	2 042 625	90 774	7 404	15 124	21.083
32	52 887	801 537	42 391	16 085	12 893	14.315
32 (corrected)	48 841	801 537	39 147	12 039	9 650	14.315
64	66 087	414 465	27 391	27 880	11 555	8.414
128	126 985	279 297	35 467	88 417	24 695	4.911
256	370 783	221 441	82 106	331 122	73 324	2.528

chain of transistors the data must traverse within one cycle of the clock. The data needs a certain time to get through the critical path and out of this time the maximum frequency was calculated. After this, the hardware implementation was synthesized again with a frequency a little lower than the maximum frequency. The synthesis result of this run was then simulated again to verify correct results. The results were not used for the synthesis results presented because the synthesis tool already applied optimizations noticeable for the area and critical path values and thus would complicate the comparison of different word widths.

5.2 Varying Word Width

The hardware implementation supports different word widths and was synthesized with all of them. The syntheses were run with a cycle frequency of one megahertz. In the following the results will be presented.

Results Overview. In Table 5.1, the key parameters of the synthesis are presented. This consists of the area in gate equivalents, the time of one scalar multiplication, the area-time product (AT-product) and the maximum frequency of the hardware implementation. The time in cycles was measured with a scalar consisting of all ones. Although this is not a typical value, it is the worst case and is therefore considered as representative. These cycle counts are therefore the cycle counts if the double-and-add-always scalar-multiplication would have been used. The area is listed twice: once the total area of the hardware implementation and the second time the total area minus the area of the memory module. On the one hand this is done to point out the word-width dependence for the arithmetic and logic. On the other hand there were no efforts made to optimize or lower the necessary area of the memory because the focus was on implementing the arithmetic and logic. The listed area-time products have the unit MGE, which stands for mega gate equivalents. Thus, the area-time products' unit remain the area's unit. Note, that the second line for 32 bits that is titled "32 (corrected)" has a lower area consumption than the normal 32 bits' value. This includes the mean value for the controlpath area of the 16-bit and 64-bit version. The reason for this is an untraceable doubling of the necessary area for the controlpath and will be discussed in the context of the single modules' areas in the following.

In Table 5.1, the area-time products for the total area, including the memory, states

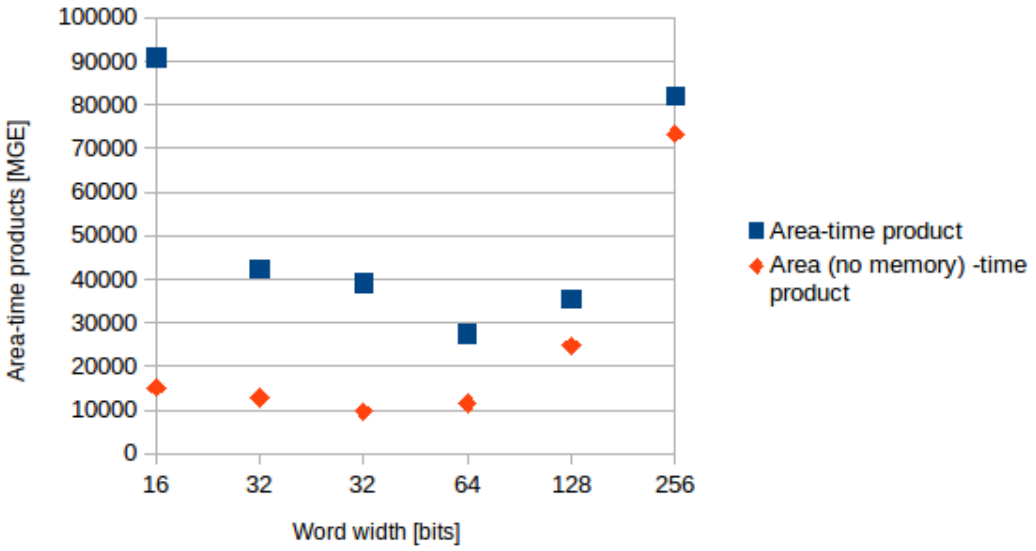

Figure 5.1: The area-time products for different word widths

the best values for the 64-bit hardware, followed by the 128-bit and 32-bit versions. In these values, the shorter time of higher word widths weights more than the lower area. This is because the area of the memory stays almost constant and thus the lower area of the arithmetic and logic relative to the total area weights less. Consecutively, the area-time products, which do not take the area for the memory into account, were considered to emphasize the tradeoff between area and time. Again, the 64 bits have the lowest value, followed by the 32 bits and the 16 bits. These observations are illustrated in Figure 5.1. The maximum frequency decreases with the factors, from the lower word width to the next greater and starting at 16 bits, 0.679, 0.5878, 0.5836 and 0.5148.

In Figure 5.1 the AT-products over the word widths are shown. It contains two data rows, once the area of the memory is included and once it is excluded. When it is included the cycle counts have a greater influence in the AT-product. Then the differences between the data points are higher and deliver the 64-bit architecture most efficient. When it is excluded, the tradeoff between area and time is considered to be reflected better as the area of the memory is at a high level. Thus, here the cycle-count decrease is weighted the same as the area decrease.

Areas of Chip Components The hardware implementation consists of certain blocks and builds a hierarchical tree. This hierarchy is represented in Table 5.2 with different indentions and the area of one block consists of the sum of its sub blocks that are below it in the hierarchy. The areas are presented in Table 5.2. The necessary area for the memory needs more than the half of the total area for the lower word widths 16, 32 and 64 bits. But at the higher word widths, the multiplier becomes becomes the more dominant part. The necessary area for the control path remains almost constant with only the 32-bit architecture having a considerable greater value. That is unexpected, because only internal memory elements are sized according to the minimum width necessary to hold the number of words and small supporting logic for loading the addresses of the

Table 5.2: The synthesized areas of the single components in gate equivalents

Module	**16 bits**	**32 bits**	**32 bits, corrected**	**64 bits**	**128 bits**	**256 bits**
Memory	37 036	36 802	36 802	38 207	38 568	39 661
Controlpath	4 509	8 386	4 340	4 171	4 221	4 097
Datapath	2 385	6 918	6 918	22 700	82 769	324 987
Mult.-accumulate	1 756	5 739	5 739	20 437	78 201	315 981
Add./sub.	189	371	371	735	1 462	2 918
Total	44 440	52 887	48 841	66 087	126 985	370 783

operands vary. The analysis of this circumstance revealed that at this point, the synthesis tool drastically synthesized more logic gates. Further analysis of the detailed logic gates showed that the synthesis tool took many more gates compared to the 16-bit and 64-bit versions: 2.5 times more "AND into NOR", four times more multiplexer, twice as many inverters, more than twice as many NAND and NOR gates, five times more "OR into NAND" and twice as many clock inverters. Due to this observation, a corrected 32-bit version is listed. The only difference to the original 32-bit synthesized result is that the area of the controlpath is the mean value of the controlpaths' areas of the 16-bit and 64-bit synthesized results. The mean was chosen because it is considered the most reasonable value. The datapath increases in size with increasing word widths. The multiply-and-accumulate module increases with the factors 3.2674 from 16 to 32 bits, 3.5613 from 32 to 64 bits, 3.8264 from 64 to 128 and 4.0406 from 128 to 256 bits. Therefore, if only the multiplier's area was considered, lower word widths would be preferable. The adder's area in contrast increases almost linearly with factors ranging between 1.96 and 1.99 when the word width is doubled.

Beside the synthesized area, the different word widths used for synthesis influence the number of cycles too. In Table 5.3, the amount of cycles necessary to calculate the single operations are listed. The operations include the scalar multiplication, the two elliptic curve operations on the twisted-Edwards curve, the fast reduction for the prime of the underlying finite field and the multiprecision addition and multiplication. For the scalar multiplication, the worst case for the scalar was tested with all bits in the scalar being set. Then, in each iteration of the double-and-add algorithm, the twisted-Edwards-curve addition had to be performed. In this table, the necessary count of cycles for the scalar multiplication is less than the half when doubling the word-width from 16 bits to 32 bits: 39.241%. From 32 bits to 64 bits the cycle-count decrease is not that high and the 64-bit's cycle-count is 51.71% of the 32 bits. For the word-width, steps from 64 to 128 bits and 128 to 256 the speedup further decreases: the 128-bit version's cycle count is 67.39% of the 64-bit version's and the 256 bit version's is 79.285% of the 128-bit one's cycle count. The decreasing speed up is caused by the necessary overhead when switching the states in the finite-state machine and the different word-width-dependent run-times of the operations.

These results of Table 5.3 are illustrated in Figure 5.2: For the single operations, all cycle counts are divided by the maximum value for that operation. In all cases, the 16-bit architecture has the highest cycle count. Therefore, for instance, the cycle counts for the scalar multiplication are divided by the 16-bit architecture's value. For a better visualization of the speedups of the different word widths, the cycle counts for the single

Table 5.3: The amount of cycles for the single operations

Operation	**16 bits**	**32 bits**	**64 bits**	**128 bits**	**256 bits**
Scalar multiplication	2 042 625	801 537	414 465	279 297	221 441
Twisted-Edwards addition	4 965	1 933	993	667	528
Twisted-Edwards doubling	3 011	1 195	623	421	334
Finite-field inversion	194 832	73 384	36 948	24 802	19 741
Multiprecision multiplication	258	66	18	6	3
Multiprecision addition	17	9	5	3	2
Fast reduction	122	74	50	38	31

operations where connected by lines. The graph has a logarithmic y scale to respect the word-width doubling on the x scale. In the diagram, the slopes of the lines become less steep for increasing word widths, as discussed. The speedup by increasing the word width for scalar multiplication, the finite-field inversion, the addition and the doubling on the twisted-Edwards curve are very similar. This is considered to be based on the fact that these operations consist of many subsequent operations and therefore share the same average speedup. The multiprecision multiplication on integers has the steepest slope for the word width doubling from 16 to 32 bits as it is natural because of the quadratic influence of the number of words.

The comparison of different word widths' area-time products lead to the result that the 32-bit architecture is best suitable for the scalar multiplication. For very constrained conditions with the low area footprint, it is considered to be more important that another metric is used, one in which the lower area has a greater impact. Then the 16 bits may be preferable.

5.3 Comparison to Previous Work

In this section, the hardware implementation results of this thesis are compared to related work. To the authors' best knowledge, there is no previous ASIC hardware implementation on **twisted Edwards-curves** over a prime field of a comparable size. Further, there is no previous work on an ASIC hardware implementation on **Edwards curves** over a prime field of comparable size too. There is an FPGA implementation for twisted-Edwards curve in [3]. There is also Edwards curve FPGA implementations in [39]. Nevertheless hardware implementations on ASIC and FPGA are not comparable. For comparison, four ASIC hardware implementations of elliptic curves over prime fields are presented. Two of these are of a comparable field size and the other two are of smaller size. Although the used elliptic curves are in Weierstrass form, due to the lack of comparable candidates using Edwards curves or twisted Edwards curves, those are used. Further discussion about related work is given in Section 4.2.

In Table 5.4, the synthesis results of the implementations of [31], [42], [41], and [17] are compared to the best results of this thesis. For a better comparison, the total areas excluding the areas for the memories are used because of the significant differences of gate equivalents per memory bit. The synthesis results of this work ends up with 9 to 9.6 GE per memory bit and is much greater than for instance the 2.96 GE per bit of [31]. The time is measured in number of cycles per scalar multiplication for the worst case with all

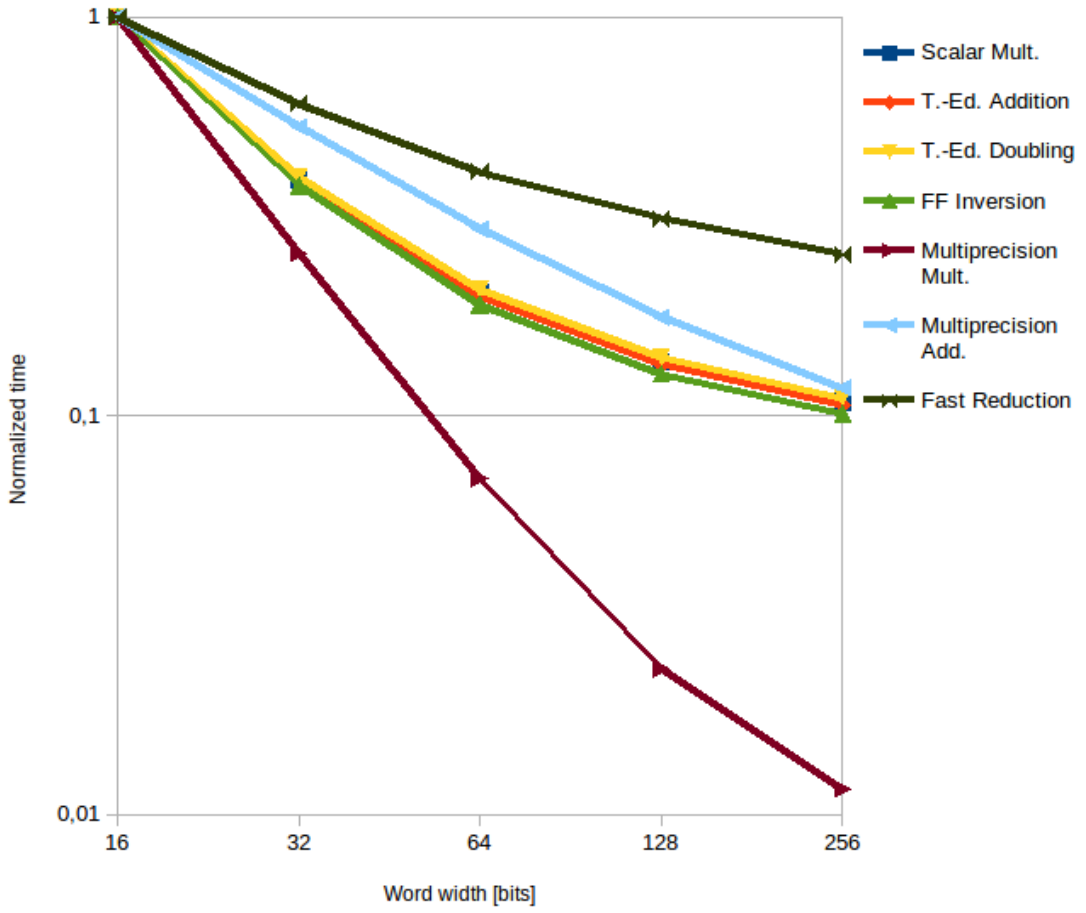

Figure 5.2: The normalized execution times of the different operations

bits of the scalar are set.

The one-bit difference between this work's synthesis results and those listed in the upper part of the table, 256 bits versus 255 bits, is considered negligible as stated in [41, Chapter 3]. In the synthesized area result of [31], the area of the binary-field controller is excluded because for comparison with this work this is not relevant. The hardware implementation of [31] supports different word widths. For comparison the one with the best area-time product was used. The implementation of Wolkerstorfer in [42] has the capability of dual field computations. On the one hand, it can operate on elliptic curves over prime fields, and on the other hand over binary fields. It is stated that this capability has no great impact on the area. The implementation can be used with different field sizes resulting in different synthesis results. For a direct comparison the well suitable 256-bit architecture is used. The results of the other widths are listed too. For the results of this thesis, the 32-bit, 64-bit and 32-bit with corrected controlpath area are considered. The results of Wenger et al. [41], Fürbass et al. [17] and the two other results of Wolkerstorfer [42] are for smaller field sizes.

For the following analysis, if not stated explicitly, this work's corrected 32-bit synthesis

Table 5.4: The area, time and area-time product

	Field size [bits]	**Area** [GE]	**Time** [cycles]	**AT-prod.** [MGE]
Satoh et al. [31], 32 bits	256	39 920	880 000	35 130
Wolkerstorfer [42]	256	14 807	1 175 500	17 406
This work, 32 bits	255	16 085	801 537	12 893
This work, 32 bits, corrected	255	12 039	801 537	9 650
This work, 64 bits	255	27 880	414 465	11 555
Fürbass et al. [17]	192	13 183	502 000	6 618
Wenger et al. [41]	192	8 535	1 312 616	11 203
Wolkerstorfer [42]	192	11 773	677 500	7 976
Wolkerstorfer [42]	224	13 423	904 900	12 147

result is used for comparison. The listed synthesized area in Table 5.4 of [31] is at high level and even higher than the 64-bit ASIC hardware architecture of this thesis. The time of [31] for carrying out a scalar multiplication is roughly at the level of the 32-bit architecture of this thesis implementation. In context of the area-time product the implementation results of this thesis are about 2.7 to 3.6 times better than the one of [31]. The three results of [42] give a hint of the influence of different field sizes on the area and the cycle count and further are of comparable efficiency. The area and the cycle count of the 256-bit results of [42] is a little higher and has an AT-product about 1.8 times higher. The other two results' AT-products of [42] with the smaller field sizes are about 0.8 times and 1.25 times this work's results. The result's AT-product of [17] is about 0.685 times the one of this work, but operates on a smaller field. Its area is only 9% higher but the cycle count is lower and about 0.626 times the one of this work. On the one hand, the area result of [41] is smaller and about 0.71 times the area of this work's result. On the other hand it takes about 1.64 times more cycles. Although it operates on a smaller field, its AT-product is a little higher and about 1.16 times of this work's result.

Chapter 6

Conclusion

Cryptography has reached the very constrained electric devices such as mobile phones and embedded systems. To be able to perform cryptographic calculation on these devices, it is crucial to be efficient and to meet low resource requirements. ASICs deliver in this context the optimal solutions as they are very distinct and flexible. They can be tailored to fulfill exactly the requirements and are therefore in a superior position over general purpose processors.

The Ed25519 is an Edwards-curve Digital Signature Algorithm (EdDSA) with fixed parameters. It has good properties for low resource implementations and is therefore a considerable alternative to the standardized Elliptic Curve Digital Signature Algorithm (ECDSA).

This thesis' purpose was to implement a low resource hardware to perform the arithmetic necessary for the explicit cryptographic scheme Ed25519 as defined in Chapter 2. The low resource quality is evaluated by the area-time product that pose as figure of merit. It factors in both critical properties of area and execution time.

This thesis can be split into two main parts. First, the analysis of the Ed25519: The cryptographic surroundings were discussed in and the mathematical backgrounds were developed in Chapter 2 from a very basic level up to considerations in context of the implementation. Its specifications were discussed, security considerations were made and the differences to the ECDSA were analyzed in Chapter 3. Finally, the different approaches and algorithms available for implementations on the single abstraction levels were discussed in Chapter 3. The second part of this thesis is the low resource hardware-implementation, its presentation, analysis and discussion. In Chapter 4, related work and the work flow utilized were presented. This includes a rough view on hardware implementations in general and the stages on the way to the final chip layout. The different word widths supported by the implementation and their impact were discussed. Then the details of the single modules were presented: Starting with the chip's interface that is used to communicate to the outside and transfer data input and output, over to the controlpath that manages the operations and their sequences. Subsequently, to the datapath that performs the operations, and finally the memory. At the end of the chapter, the test bench and the highlevel model were discussed, and how the hardware implementation was verified for correct calculations. Afterwards, in Chapter 5, the synthesized results of the developed hardware implementation were evaluated for the one with the lowest resource consumption. This was judged by area-time product that evaluates the tradeoff between the consumed chip area and the cycle count to perform the operation. Then, these synthesis results were compared to hardware implementations of related works which were

measured with the same metric.

The hardware implementation presented in this thesis is able to perform a scalar multiplication in projective coordinates with an arbitrary point on the used curve. It has functionality to carry out an inversion in the finite field with the inversion-by-exponentiation algorithm. The inversion is necessary to transform the obtained projective point on the curve back to affine coordinates.

The analysis of the synthesis results revealed that the area of the necessary memory has a big impact on the results: For the 16-bit architecture it consumes 82.3 % of the total area, 75.4 % for 32 bits and 57.8 % for 64 bits. The area-time product with the total area and the time necessary for one scalar multiplication is then the lowest for the 64-bit architecture. When excluding the area of the memory from the calculations, the 32-bit architecture has the lowest resource consumption. Those two architectures take a total area of 48.8 gate equivalents (GE) for the 32 bits and 66.1 GE for 64 bits, and perform a scalar multiplication in 801.5 kilo cycles (kCycles) and 414.5 kCycles respectively. With the used 0.35 μm CMOS technology, the synthesis outputs the areas of 2.667 mm^2 for 32 bits and 3.601 mm^2 for 64 bits. The time that one scalar multiplication takes on this architectures, run with the maximum frequencies, is 55.99 milliseconds for 32 bits and 49.26 milliseconds for 64 bits.

In this work, a hardware implementation was presented, that carries out a scalar multiplication for the EdDSA with a reasonable security of about 128 bits. The achieved resource requirements implies that it can be integrated into constrained devices with only a few square millimeters of necessary area. It is able to calculate the computationally intensive part of the key generation, signature generation and signature verification in fractions of a second. The EdDSA allows a straight forward implementation due to the unified and complete elliptic-curve point-addition formula. Because of this completeness it is secure against simple power analysis and as a result of the unified formula, no additional checks for exceptional points is necessary. By only a slightly modified protocol relative to the ECDSA, the EdDSA gains additional security improvements.

6.1 Future Work

The presented hardware implementation was tailored for the Ed25519-SHA512 and can carry out the scalar multiplication and the finite field inversion. Due to this specialization there arise further tasks and questions.

Based on this thesis' hardware implementation, the completion of the signature generation and process would be very interesting. That might raise the value of the implementation from a theoretical level to a practically relevant module.

A considered worthy future work is to bring the hardware implementation to a certain platform of a constrained device. There, the evaluation of the necessary modification and resulting resource requirements is of interest, especially the power consumption.

This thesis' attention was to develop the arithmetic and logic for a low resource hardware implementation and thus the used memory was instantiated in a very straight forward array of flip flops. Therefore, this leaves open the field for optimization regarding the memory as it is done in related works.

When developed, some design decisions were made. It is considered to be of value to break up these and to generate a flexible parameterizable hardware that can be reused and synthesized by specifying these parameters. In this context, the circumstance that it was developed for one certain twisted Edwards-curve with a finite field over one certain

prime comes into play. This has a deep impact on the developed arithmetic: This prime is a so-called pseudo Mersenne prime and enables the use of fast reduction. This suggests to modify the hardware implementation in such a way that it can compute the fast reduction by simply specifying the used prime in an abstract way.

Another considered worthy modification is to enable a configurable multiplier. This should be instantiated independently from the other modules' word width and thus enables the exploration of the optimal word width of the implementation in more detail.

Appendix A

Definitions

A.1 Abbreviations

AES	Advanced Encryption Standard
ALU	Arithmetic and Logic Unit
ANSI	American National Standards Institute
ASIC	Application specific integrated circuit
AT-Product	Area-Time Product
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DSA	Digital Signature Algorithm
DUT	Device Under Test
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edward-curve Digital Signature Algorithm
FF	Finite Field
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GE, kGE, MGE	Gate Equivalent, 10^3 Gate Equivalents, 10^6 Gate Equivalents
HDL	Hardware-Description Language
IAIK	Institute for Applied Information Processing and Communications
kCycles	Kilo (1 000) Cycles
MAC	Message Authentication Code
NAND	Not-And
NAF	Non-Adjacent Form
NIST	National Institute of Standards and Technology
PKC	Public Key Cryptography
RAM	Random Access Memory
RFID	Radio Frequency Identification
ROM	Read-Only Memory
SPA	Simple-Power-Analysis
SHA, SHA512	Secure Hash Algorithm, Secure Hash Algorithm 512 bits
VLSI	Very-Large-Scale Integration

Appendix B

Technical Specifications

The technical specifications of the hardware implementation with different word widths.

16 Bits

Process Technology: 0.35 μm CMOS, standard cell library

Area: 2.426 mm^2

Maximum frequency: 21,083 MHz

Time for one scalar multiplication: 0.09688 s

Functionality:
- Scalar multiplication for Ed25519-SHA512
- Finite-field Inversion

32 Bits

Process Technology: 0.35 μm CMOS, standard cell library

Area: 2.667 mm^2

Maximum frequency: 14,315 MHz

Time for one scalar multiplication: 0.05599 s

Functionality:
- Scalar multiplication for Ed25519-SHA512
- Finite-field Inversion

64 Bits

Process Technology: 0.35 μm CMOS, standard cell library

Area: 3.608 mm^2

Maximum frequency: 8,414 MHz

Time for one scalar multiplication: 0.04926 s

Functionality:
- Scalar multiplication for Ed25519-SHA512
- Finite-field Inversion

Appendix C

Final Chip Layouts

The place-and-route results for 16 bits, 32 bits and 64 bits will be presented. For the results in the Figures C.1, C.2 and C.3 a frequency of 1 MHz was chosen.

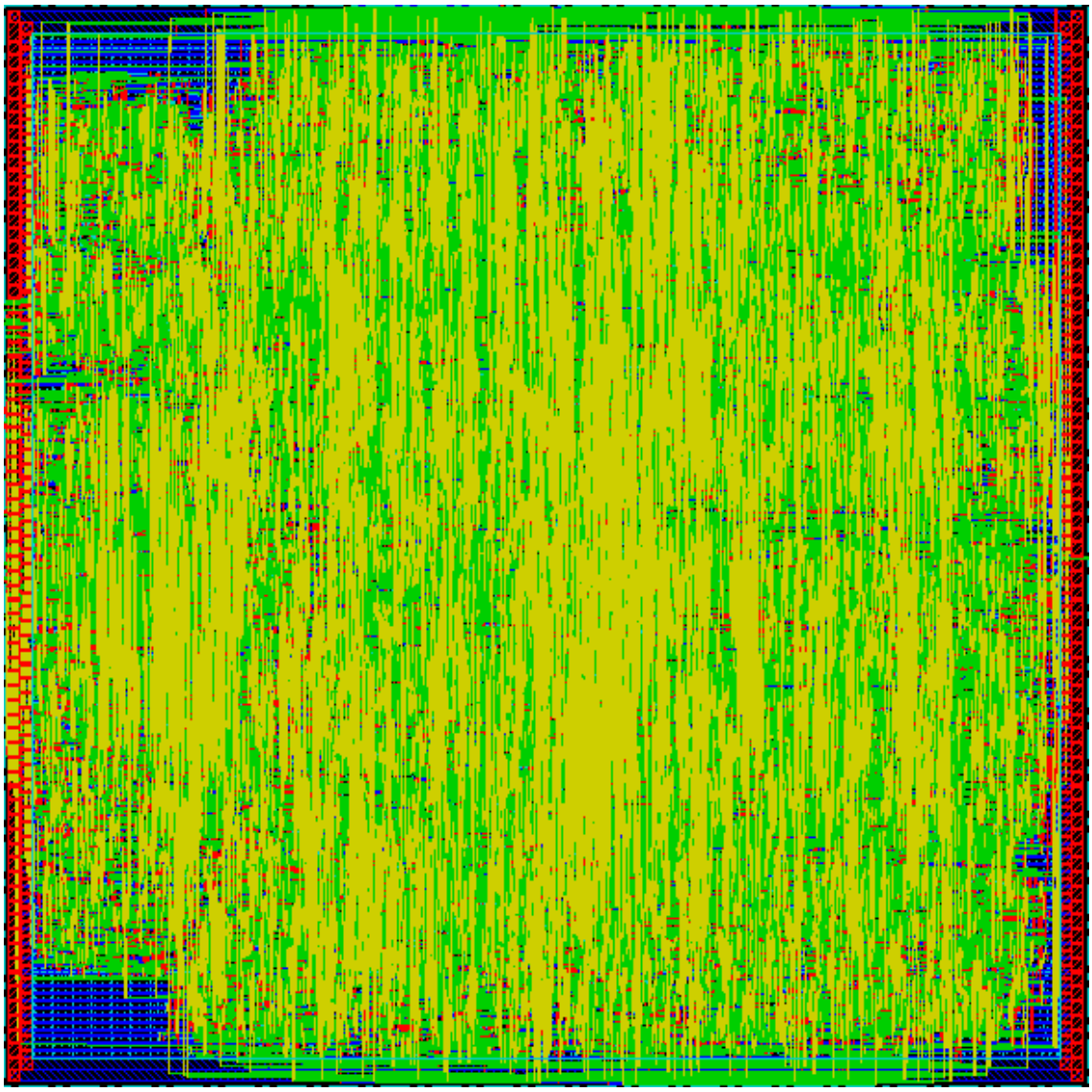

Figure C.1: Place-and-route results of the 16-bit architecture

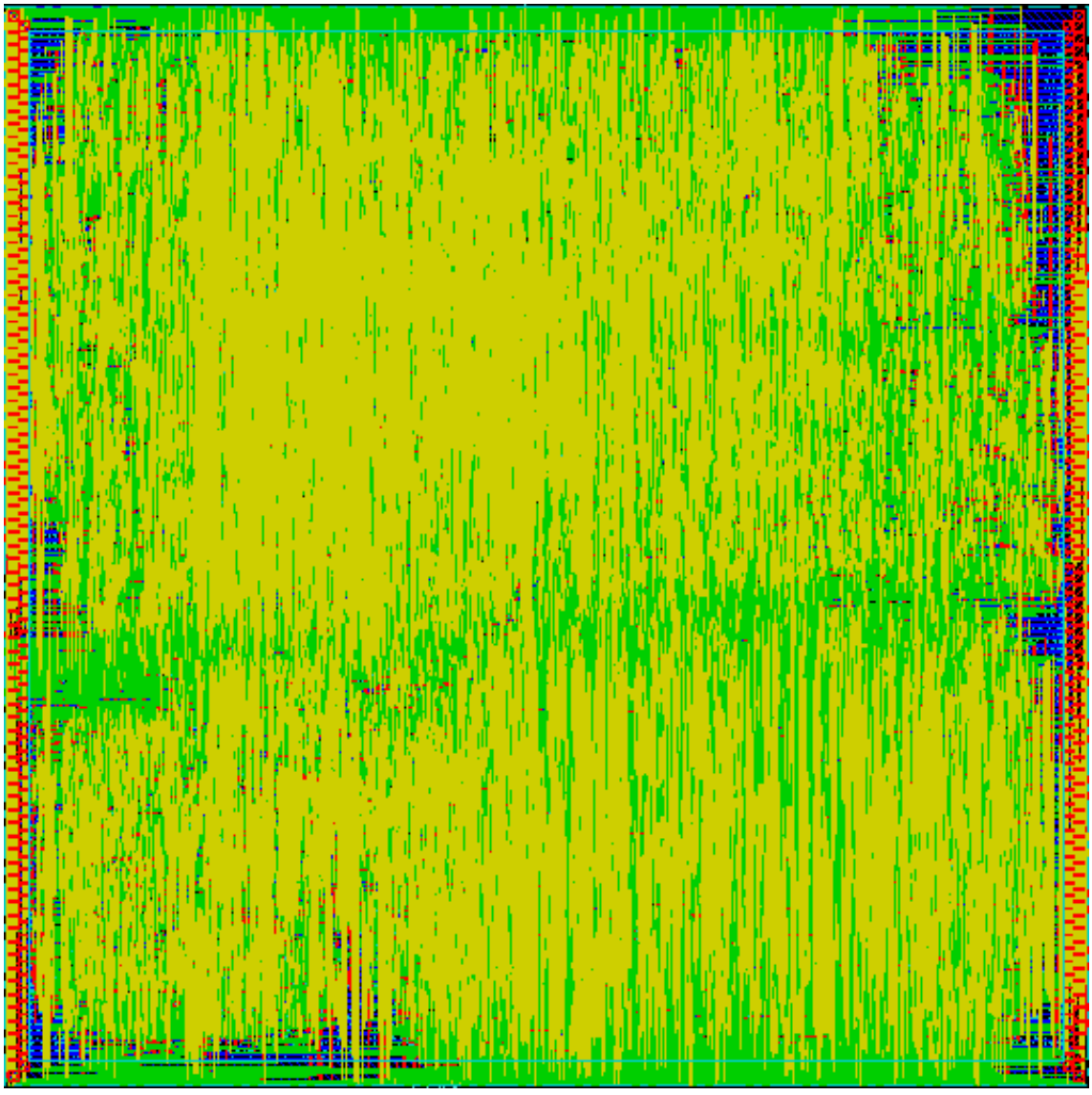

Figure C.2: Place-and-route results of the 32-bit architecture

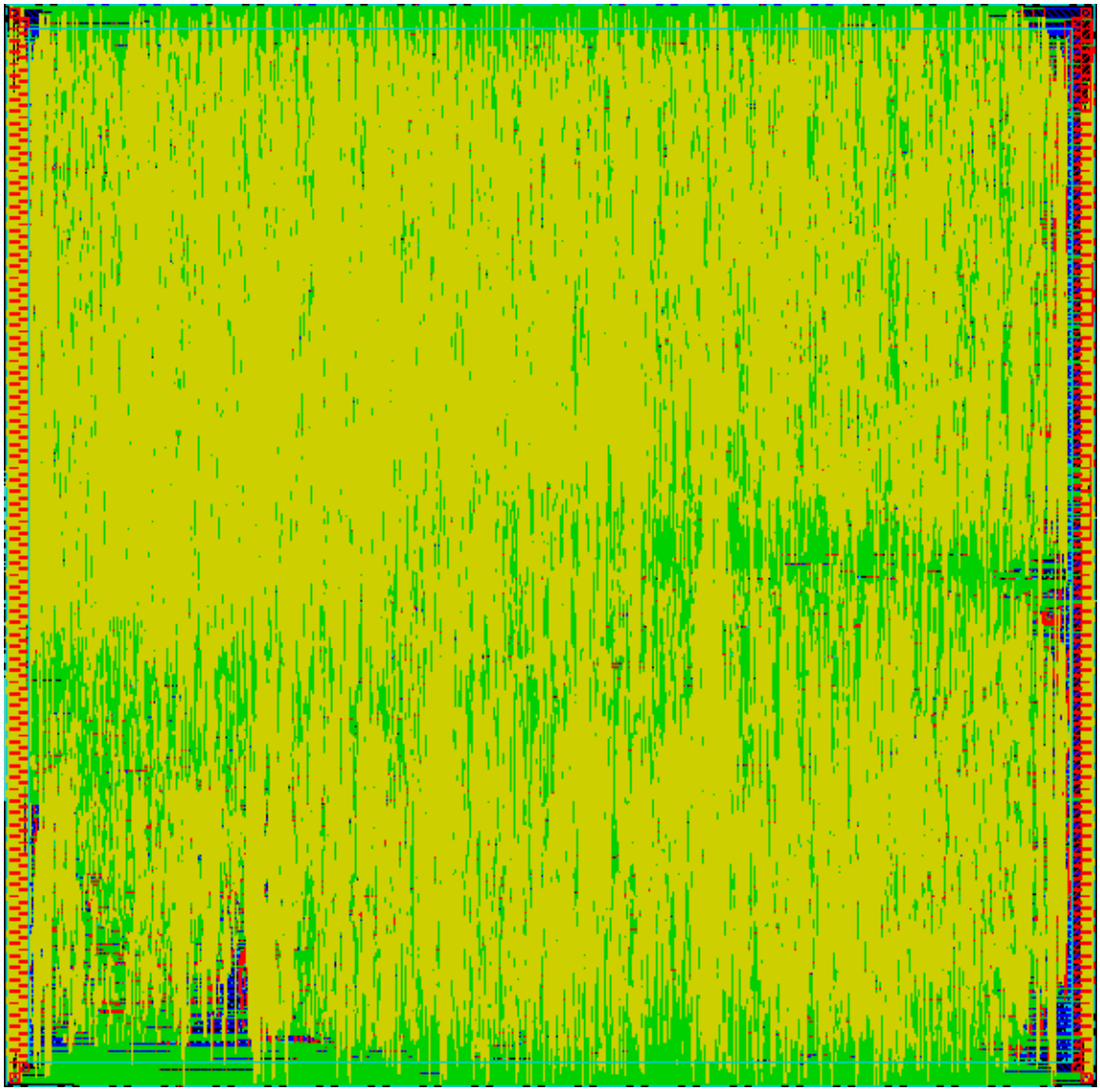

Figure C.3: Place-and-route results of the 64-bit architecture

Bibliography

[1] *Family 10h AMD OpteronTM Processor Product Data Sheet.* `http://support.amd.com/us/Embedded_TechDocs/40036.pdf`.

[2] A. N. S. I. A. B. Association. Ansi x9.62:2005: Public key cryptography for the financial services industry, the elliptic curve digital signature algorithm (ecdsa), 2005.

[3] B. Baldwin, R. Moloney, A. Byrne, G. McGuire, and W. P. Marnane. A hardware analysis of twisted edwards curves for an elliptic curve cryptosystem. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 355–361. Springer, 2009.

[4] E. Barker, W. Barker, W. Burr, W. Polk, M. Smid, P. D. Gallagher, and U. S. For. Nist special publication 800-57 recommendation for key management – part 1: General, 2012.

[5] E. Barker and A. Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication*, 800:131A, 2011.

[6] D. J. Bernstein and T. Lange. Inverted edwards coordinates. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 20–27. Springer, 2007.

[7] D. J. Bernstein, T. Lange, and R. R. Farashahi. Binary edwards curves. In *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 244–265. Springer, 2008.

[8] Bernstein, Daniel and Duif, Niels and Lange, Tanja and Schwabe, Peter and Yang, Bo-Yin. High-Speed High-Security Signatures. In Preneel, Bart and Takagi, Tsuyoshi, editor, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-23951-9_9.

[9] Bernstein, DanielJ. Curve25519: New Diffie-Hellman Speed Records. In Yung, Moti and Dodis, Yevgeniy and Kiayias, Aggelos and Malkin, Tal, editor, *Public Key Cryptography - PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Berlin Heidelberg, 2006.

[10] A. Chatterjee and I. Sengupta. Fpga implementation of binary edwards curve usingternary representation. In *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*, pages 73–78. ACM, 2011.

[11] Daniel J. Bernstein and Peter Birkner and Marc Joye and Tanja Lange and Christiane Peters. Twisted Edwards Curves. Cryptology ePrint Archive, Report 2008/013, 2008. `http://eprint.iacr.org/`.

[12] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644 – 654, nov 1976.

[13] Edwards, Harold M. A normal form for elliptic curves. *Bull. Amer. Math. Soc. (N.S.)*, 44(3):393–422 (electronic), 2007.

[14] El Gamal, Taher. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[15] P. FIPS. 180-4. *Digital signature standard (DSS)*, 2012.

[16] P. FIPS. 186-4. *Digital signature standard (DSS)*, 2013.

[17] F. Fürbass and J. Wolkerstorfer. Ecc processor with low die size for rfid applications. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1835–1838. IEEE, 2007.

[18] C. F. Gauss. Disquisitiones arithmeticae, 1801. english translation by arthur a. clarke, 1986.

[19] Hankerson, Darrel and Menezes, Alfred J. and Vanstone, Scott. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[20] H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted edwards curves revisited. In *Advances in Cryptology-ASIACRYPT 2008*, pages 326–343. Springer, 2008.

[21] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63, 2001.

[22] H. Kaeslin. *Digital integrated circuit design: from VLSI architectures to CMOS fabrication*. Cambridge University Press, 2008.

[23] A. Karrass, W. Magnus, and D. Solitar. Combinatorial group theory. *Presentations of Groups in Terms of Generators and Relations*, 1976.

[24] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit rsa modulus. Cryptology ePrint Archive, Report 2010/006, 2010. `http://eprint.iacr.org/`.

[25] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[26] U. Kocabas, J. Fan, and I. Verbauwhede. Implementation of binary edwards curves for very-constrained devices. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 185–191. IEEE, 2010.

[27] A. Lenstra, J. Lenstra, H.W., M. Manasse, and J. Pollard. The number field sieve. In A. K. Lenstra and H. W. Lenstra, editors, *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 11–42. Springer Berlin Heidelberg, 1993.

[28] V. S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology—CRYPTO'85 Proceedings*, pages 417–426. Springer, 1986.

[29] Nguyen, Phong Q. and Shparlinski, Igor E. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Designs, Codes and Cryptography*, 30:201–217, 2003. 10.1023/A:1025436905711.

[30] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[31] A. Satoh and K. Takano. A scalable dual-field elliptic curve cryptographic processor. *Computers, IEEE Transactions on*, 52(4):449–460, 2003.

[32] C. Schnorr. Efficient identification and signatures for smart cards. In G. Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer New York, 1990.

[33] Schnorr, Claus-Peter. Efficient Identification and Signatures for Smart Cards. In *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '89, pages 239–252, London, UK, UK, 1990. Springer-Verlag.

[34] C. E. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal, Vol 28, pp. 656–715*, 1949.

[35] J. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics, 106. Springer-Verlag New York, 2009.

[36] S. Singh. *The Code Book: How to Make It, Break It, Hack It, Crack it*. Delacorte Press, 2002.

[37] J. Solinas. Pseudo-mersenne prime. In H. van Tilborg and S. Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 992–992. Springer US, 2011.

[38] M. Taschwer. Modular multiplication using special prime moduli. In P. Horster, editor, *Kommunikationssicherheit im Zeichen des Internet*, DuD-Fachbeiträge, pages 346–371. Vieweg+Teubner Verlag, 2001.

[39] J. Wang and X. Wang. Dual-form elliptic curves simple hardware implementation. In *Intelligent Computing Technology*, pages 520–527. Springer, 2012.

[40] L. C. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Discrete Mathematics and Its Applications. Taylor & Francis, 2008.

[41] E. Wenger and M. Hutter. Exploring the design space of prime field vs. binary field ecc-hardware implementations. In *Information Security Technology for Applications*, pages 256–271. Springer, 2012.

[42] J. Wolkerstorfer. Is elliptic-curve cryptography suitable to secure rfid tags. In *Workshop on RFID and Lightweight Cryptography, Graz-August*, 2005.