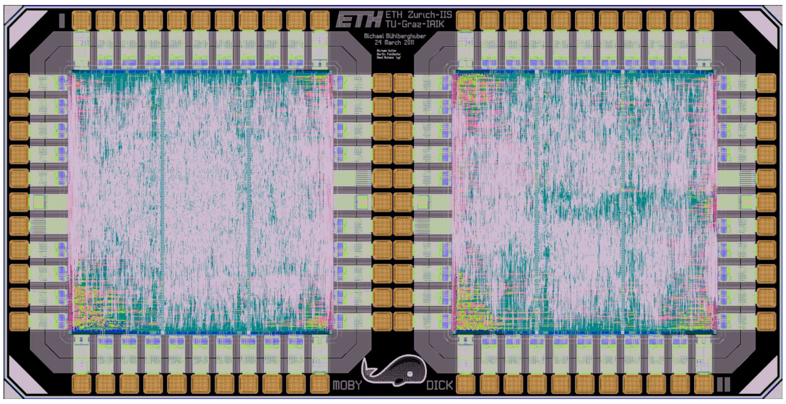
Comparing ECDSA Hardware Implementations based on Binary and Prime Fields

Master Thesis

Michael Mühlberghuber
m.muehlberghuber@student.tugraz.at



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

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

TU Graz
Graz University of Technology

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Supervisors:	Dr. Michael Hutter and Dr. Martin Feldhofer, TU Graz
	Dr. Frank Gurkaynak, ETH Zürich
Assessors:	Dr. Karl-Christian Posch, TU Graz
	Dr. Norbert Felber, ETH Zürich

June, 2011

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 wish to thank, first and foremost, Frank Gurkaynak and Beat Muheim from the Integrated Systems Laboratory at the Swiss Federal Institute of Technology in Zurich for their support during the whole project. They spent a lot of their time helping me to put an idea into practice, especially throughout the backend design of the implemented hardware architecture. I also want to thank Michael Hutter for all the corrections in the final draft of this work. Furthermore I want to thank Martin Feldhofer, because without his help I would have never been able to do this thesis abroad, as well as my assessors at Graz University of Technology and at the Swiss Federal Institute of Technology in Zurich, namely Karl-Christian Posch and Norbert Felber.

And last but certainly not least, I greatly thank my parents and my girlfriend Magdalena for their support throughout my whole study. All of you have been a constant source of support for me during the last months and years.

Abstract

Currently, the most popular and sophisticated digital signature schemes with regard to low-resources are based on elliptic curves. This is due to the fact that Elliptic-Curve Cryptography (ECC) requires a much shorter keylength to ensure the same level of security than traditional public-key cryptosystems like RSA do.

The Elliptic Curve Digital Signature Algorithm (ECDSA) is one of the most popular representatives of elliptic-curve based digital signature protocols and already many different hardware implementations of it exist. Nevertheless, comparing designs from various developers is rather difficult, because there are many different levels of abstraction where the implementations can differ (e.g. elliptic-curve operations, finite-field operations, applied technology). Therefore we used the same design flow to implement two different Application Specific Integrated Circuits (ASICs), both providing digital signature generation and verification according to ECDSA. One of them is based on a binary field and the other one on a prime field. Both chips offer all operations required to generate and verify a digital signature, except the Random Number Generator (RNG) and the hash function.

The two designs have been implemented using the 150 nm technology by the *LFoundry GmbH* and can be clocked with up to 100 MHz. The binary-field based design requires about 42 kcycles for the signature generation and up to 217 kcycles for the verification process. The prime-field based design needs approx. 563 kcycles for the signature generation, whereas the verification requires maximal 564 kcycles.

Our investigations showed that because binary-field arithmetics are more suitable for hardware implementations than prime-field arithmetics, using binary fields as the underlying finite field for ECC is more appropriate. They result in a shorter runtime of the elliptic-curve operations without exceeding a reasonable range with regard to other resource aspects like area and power.

Keywords: Digital Signature, ECDSA, Asymmetric-Key Cryptography, Public-Key Cryptography, Elliptic-Curve Cryptography, Full Precision.

Kurzfassung

Mit dem Algorithmus für digitale Signaturen basierend auf elliptischen Kurven (ECDSA) existiert ein sehr weit verbreiteter Standard zur Generierung und Verifikation von digitalen Signaturen. ECDSA wurde bereits von namhaften Institutionen standardisiert und in der Literatur sind zahlreiche Hardware-Implementierungen von unterschiedlichen Entwicklern zu finden. Vergleiche dieser Implementierungen lassen sich aufgrund der vielen Abstraktions-Ebenen des Algorithmus und den dadurch entstehenden, unterschiedlichen Umsetzungsmöglichkeiten (elliptische Kurven-Operationen, Körper-Arithmetik, verwendete Halbleiter-Technologie, etc.) nur bedingt durchführen. Aus diesem Grund wurde der gleiche *Design-Flow* zur Umsetzung von zwei anwendungsspezifischen integrierten Schaltungen (ASICs), welche die Funktionalität des ECDSA bieten, verwendet. Einer der beiden Chips basiert dabei auf einem Binär-Körper zur Berechnung der elliptischen Kurven-Operationen, wohingegen der andere mit einem Prim-Körper arbeitet. Beide Designs stellen alle, zur Signatur-Generierung und -Verifikation nach ECDSA benötigten Operationen zur Verfügung, mit Ausnahme des Zufallszahlen-Generators und des Hash-Algorithmus.

Beide Chips wurden unter Verwendung der 150 nm Technologie des Halbleiterherstellers *LFoundry GmbH* gefertigt und können mit bis zu 100 MHz Taktfrequenz betrieben werden. Das Binär-Körper basierte Design benötigt ungefähr 42 kcycles zur Generierung einer digitalen Signatur sowie maximal 217 kcycles für die Verifikation. Die Prim-Körper basierte Implementierung erstellt eine digitale Signatur in rund 563 kcycles und benötigt bis zu 564 kcycles um ein solche zu verifizieren.

Unsere Untersuchungen zeigen, dass sich aufgrund der Tatsache, dass Binär-Körper-Operationen einfacher und effizienter in Hardware umzusetzen sind als Prim-Körper-Operationen, Binär-Körper besser für die Hardware-Implementierung von elliptischen Kurven basierter Designs eigenen. Das Binär-Körper basierte Design resultiert in kürzeren Laufzeiten für die elliptischen Kurven-Operationen ohne dabei in Bezug auf Fläche und verbrauchte Leistung negativ aufzufallen.

Stichwörter: Digitale Signatur, Elliptic Curve Digital Signature Algorithm (ECDSA), Asymmetrische Kryptographie, Public-Key Kryptographie, Elliptische Kurven-Kryptographie.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the *analog world*, authentication of a person is mostly accomplished by the use of a written signature or any biometric identification, which can be uniquely associated with humans (e.g. fingerprints, iris recognition or DNA profiling). During the last years and centuries the importance of the *digital world* has increased dramatically, hence nowadays a digital signature is at least as important as its analog counterpart, if not even more important.

Authentication in a *digital communication process* is not restricted to humans only, but also includes entities like computers, ATMs, or smartcards. Because these *participants* are getting smaller and smaller and must be ready to run without large supply units (or even without supply units, just being powered passively), hardware solutions for an authentication process are targeted, which require low resources with regard to area, runtime, and power. One state of the art technique to generate and verify a digital signature, which satisfies these needs and which has been treated within this work, is the Elliptic Curve Digital Signature Algorithm (ECDSA). It is based on point operations, performed on top of a special type of cubic curves, also known as *elliptic curves*. The definition of such an elliptic curve includes an underlying finite field, which is required to execute the curve operations on top of it. Currently the two most commonly used finite fields for elliptic-curve operations are binary fields and prime fields. Comparing ECDSA designs from various developers based on different finite fields is quite tricky, because there are many abstraction levels (i.e. elliptic-curve operations, finite-field arithmetic, applied technology) where the designs can differ.

In this thesis we implemented two ECDSA Application Specific Integrated Circuits (ASICs), both working at the full bit width of their operands (i.e. *full precision*) with the aid of the equivalent Electronic Design Automation (EDA) tools and the same manufacturing technology. The first design uses an elliptic-curve arithmetic working on top of a binary field, whereas the second one is based on an underlying prime field. The definitions for the shapes of the elliptic curves as well as the definitions for the underlying finite fields have been taken from two standards provided by the National Institute for Standards and Technology (NIST), namely the B-163 and P-192. These standards provide sophisticated parameters, well suited for cryptography purposes like digital signature generation and verification. Moreover these two standards make use of the narrowest operands among the standards provided by the NIST and are therefore most suitable for low-resource hardware designs. Furthermore, the two chips provide both parts of an authentication process, i.e. the signature generation and the signature verification. In order not to be vulnerable to timing attacks and simple power attacks with regard to the signature generation, the signing process has been implemented in such a way that it can be performed within a constant runtime, always performing the same operations, independent of its input. Communication of the chips with their environment has been realized using a standardized System on Chip (SoC) interface with an 8-bit wide data bus.

Previous works on hardware implementations of ECDSA typically use an elliptic-curve arithmetic based on a single type of finite fields. Moreover, sometimes the digital signature protocol is implemented only partially, e.g. providing just the elliptic-curve operations. We provide two ECDSA ASIC designs, offering the full functionality required for a mutual authentication process, one based on a binary field and the other one based on a prime field.

1.1. Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces the topic of cryptography, starting with some historical background, followed by the major goals of a cryptosystem. It also covers the differences on symmetric and asymmetric cryptosystems as well as the most popular mathematically hard problems used within public-key schemes, including the Elliptic Curve Discrete Logarithm Problem (ECDLP).

Because elliptic-curve arithmetic requires an underlying finite field to perform the curve operations, the third Chapter starts with a brief introduction on number theory. Next, the actual definition of an elliptic curve and the additive group which can be defined on such a curve are presented. Also the group operation, namely the chord-and-tangent rule will be treated in more detail throughout this chapter. Due to the fact that different point representation types play a major role in elliptic-curve arithmetics, they will be considered afterwards. Since the elliptic-curve point multiplication forms the core of Elliptic-Curve Cryptography (ECC), it is described extensively using an example at the end of Chapter 3.

During Chapter 4, the required finite-field arithmetic for binary fields and prime fields and appropriate algorithms to perform the field operations, suitable for hardware designs, are considered. The covered operations are addition, subtraction, multiplication, squaring and division.

ECDSA is introduced in Chapter 5. The two most time-consuming operations during the signature generation and the signature-verification process are the elliptic-curve point multiplication and the multiple-point multiplication. The fifth chapter presents different approaches to perform these operations with regard to the required runtime.

Throughout Chapter 6, the implementations of the two ECDSA designs are described.

This is done by going through all the steps which are required during the development of a hardware design. First, the basic architecture is presented, followed by some information on the high-level model. Next, properties of the Hardware Description Language (HDL) model are given and thereafter the designs of the binary and prime-field arithmetic are covered. At the end of the chapter, the chip interfaces as well as the verification of the HDL model are considered.

In Chapter 7, the results of the two implementations are presented. This includes information about their timings, area and power consumption as well as the maximal possible frequency. Finally, during the last chapter, i.e. Chapter 8, a conclusion is drawn and some future work is suggested.

Chapter 2

Yet Another Cryptography Introduction

This chapter will give an overview on cryptography in general. It starts with a brief historical introduction, followed by the central goals of a cryptographic system. Afterwards, the two main types of cryptography, namely symmetric and asymmetric cryptography, will be explained. The focus of the thesis lies on elliptic-curve cryptography which is an asymmetric-key technique that is especially suitable for resource-constrained devices. The chapter will close with a timeline, including the most relevant events with regard to asymmetric cryptography.

2.1. Historical

Cryptography has been of great interest for humans for more than 2000 years. Already a few centuries B.C. the Greeks invented the first cryptography tools. The *Scytale*, for example, consists of a simple stick with a certain diameter, which was only known to the two parties who wanted to communicate with each other in a "secure way", and a strip of leather or parchment. In order to "encrypt" a message, the sender wrapped the leather around the stick and wrote the message on it. Then a courier had to bring the strip of leather to the second party, which "decrypted" the message by wrapping the strip of leather around its own stick with the same diameter and was then able to read the message. Only those parties who knew the correct diameter of the stick were able to read the message easily. Figure 2.1 shows a sketch of such a Scytale.

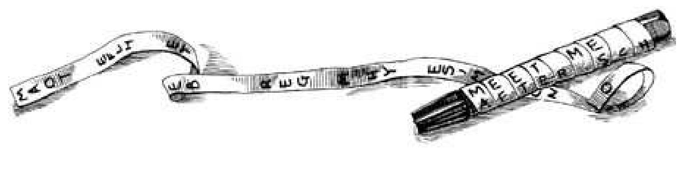


Figure 2.1.: Scytale[1].

[1]Source: `http://www.apprendre-en-ligne.net/crypto/menu/index.html`

The main fields of application for cryptography tools in those days had been the army and diplomatic institutions. Till this day, these two sectors still invest the most money to push cryptography improvements. Although the Scytale can not be considered to be "secure" with regard to state of the art cryptography, it already made use of one of the most important principles in cryptography:

Definition 2.1.1 (Kerckhoffs' Principle)**.** *The security of a cryptosystem should only be based on the secrecy of the key being used. Hence also the applied algorithm should always be made public.*

In the case of the Scytale the "secret key" was the diameter of the sticks being used by the communicating parties.

2.2. Cryptography Goals

Beside the Kerckhoffs' Principle, there exist other main goals which should always be kept in mind when designing or implementing a cryptosystem:

Confidentiality: Only the parties which are designated to participate in a communication should be able to read the messages. Unauthorized parties shouldn't be able to read them.

Integrity: As soon as the messages being sent within a cryptosystem get modified (e.g. by an adversary), the participating parties should be able to recognize the modification.

Entity Authentication: When parties communicate with each other within a cryptosystem, they have to convince the other participants of their identity.

Data Authentication: When a party receives a message from another party, the receiver must be able to verify that the message indeed originates from the supposed sender.

Non-Repudiation: As soon as a party sends a message to another party, the sender is not able to repudiate that the sent message does not origins from him.

In order to fulfill the aforementioned cryptography goals, two main types of cryptography have been developed during the last years and centuries and will be described in the following.

2.3. Symmetric-Key Cryptography

The Scytale, mentioned in Section 2.1, is a simple example for the so called *symmetric-cryptography schemes*, which are based on a private secret key, only known to the entities participating in the communication. This type of cryptography is therefor often called *secret-key cryptography*. Sharing the secret key in a group of entities requires a key distribution using a secure channel. The subsequent communication is then encrypted

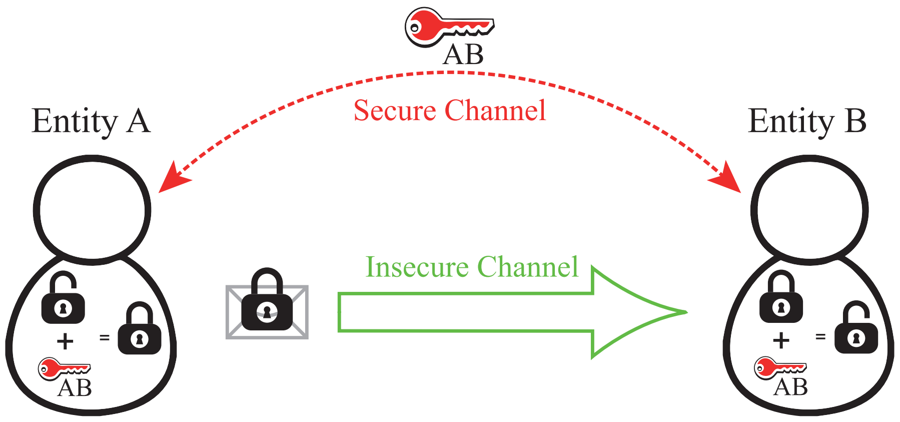


Figure 2.2.: Secure communication using symmetric-key cryptography.

using this secret key. Figure 2.2 illustrates the main idea of a symmetric-key based scheme between two entities A and B.

Note: Talking about a "secure channel" means that no other parties, except those who are supposed to participate within the conversation, are able to eavesdrop the communication. Messages sent over an "insecure channel" on the other hand can be read by any entity, including adversaries.

2.3.1. Key-Distribution Problem

Although symmetric-key cryptography works very efficiently, it suffers from a major drawback which is called the *key-distribution problem*. As already mentioned, for exchanging the secret key, a secure channel is required. Furthermore, this channel has to be authentic. As long as, for instance, only two nearby entities want to communicate with each other in a secure way, this doesn't pose a huge problem. In such a case, the participants can meet once prior communication to authenticate each other and simultaneously exchange the secret key. But when it comes to a large number of entities who want to take part in a secure communication, this prior meeting becomes much more difficult and poses a main problem, especially when the participants are distributed all over the world.

2.4. Asymmetric-Key Cryptography

In order to avoid the key-distribution problem, Whitfield Diffie and Martin Hellman [7] introduced a key-exchange protocol that makes use of *asymmetric-key cryptography* in 1976. It is based on the idea that each entity of the system receives a keypair consisting of a private and a public key. Because of using public keys, these systems are often refered to as *public-key systems*. As the name already implies, the public key of such a keypair is made public whereas the private key is only known to the entity itself. A

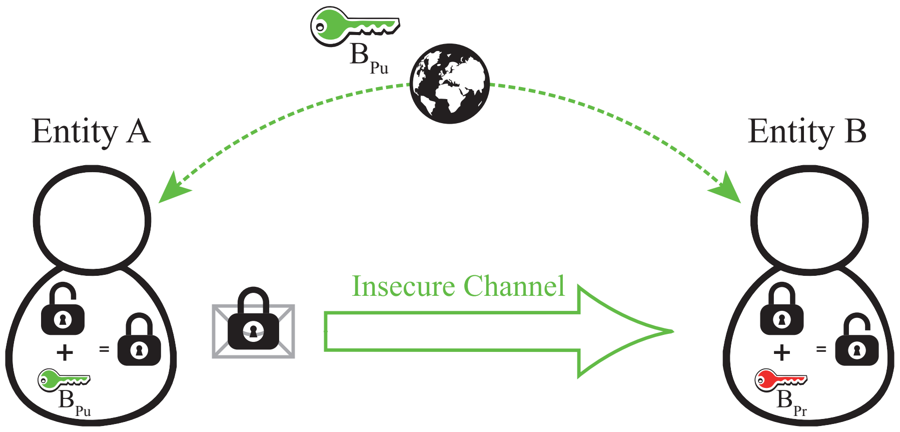


Figure 2.3.: Secure communication using public-key cryptography.

secure communication using asymmetric-key cryptography is illustrated in Figure 2.3 works as follows:

- Entity A wants to send a secure message to entity B.
- A uses the public key of B (which is available to everybody) to encrypt the message.
- Afterwards A sends the encrypted message over an (insecure) channel to B.
- B uses its private key to decrypt the message received from A.
- Communication from B to A works vice versa.

The above example describes how two entities can communicate with each other in a secure way. Because the private key of entity B is only known to itself, B is the only one who can decrypt the message.

A few small changes in this process enables an entity to sign digital data. The process is illustrated in Figure 2.4, and is divided into a *sign* and a *verify* step. It works as follows:

- Entity B wants to be sure that the received message actually comes from entity A.
- A uses its private key to *sign* the message (i.e. A applies a signature scheme).
- Afterwards A sends the signed message to entity B over an (insecure) channel.
- B uses the public key of A to *verify* the signature. If the signature is valid, B can be sure that the cryptography goals *integrity, message authentication and non-repudiation* are satisfied.
- Authentication from B to A works vice versa.

Definition 2.4.1 (Mutual Authentication)**.** *The process, when two entities want to authenticate each other, is referred to as "mutual authentication".*

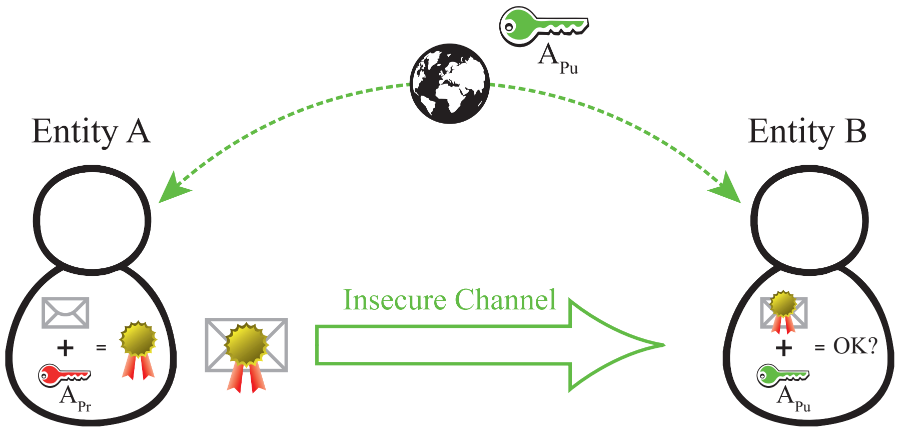


Figure 2.4.: Asymmetric-cryptography scheme for authentication.

2.4.1. Private/Public-Key Relationship

The private and public key of a keypair being used within a cryptographic scheme, are related. This relationship is in general based on a *mathematically hard problem*, for example the Integer-Factorization Problem (IFP), the Discrete Logarithm Problem (DLP) or the ECDLP.

2.4.2. Integer-Factorization Problem (IFP)

The main idea of this mathematically hard problem, shown in Equation (2.1), is that given the product n of two large prime numbers p and q, it is computationally intractable to get p and q only from n.

$$n = p \cdot q, \quad p, q \ldots prime. \tag{2.1}$$

One of the first public-key primitives is RSA system, which has been published by its inventors Ronald Rivest, Adi Shamir, and Leonard Adleman [31] in 1977. To be more precise, it has been proven that determining the private key from the public key within an RSA system is as hard as the IFP. Algorithm 2.1 describes how the RSA-keypair generation works.

2.4.3. Discrete Logarithm Problem (DLP)

Another mathematically hard problem is the DLP. Here for a given set of parameters (p, q, g) (the so called *domain parameters*) and the public key y, it is known to be "quite hard" to determine the private key x from Equation (2.2):

$$y = g^x \mod p. \tag{2.2}$$

The first Discrete Logarithm (DL) systems were already published by Diffie and Hellman in 1976. In 1984, Taher Elgamal invented digital signature and encryption systems

Algorithm 2.1 RSA-keypair generation.

```
Input: Desired bit length $l$ of the prime numbers.
Output: RSA private key $d$ and public key $(n, e)$.
1: Randomly choose two distinct prime numbers $p$ and $q$ of the same bit length $l$.
2: Compute $n = p \cdot q$.
3: Compute $\varphi(n) = (p-1) \cdot (q-1)$.
4: Choose an arbitrary integer $e$ such that $1 < e < \varphi(n)$ and $gcd(e, \varphi(n)) = 1$.
5: Compute $d \equiv e^{-1} \mod \varphi(n)$.
6: return $d, n, e$.
```

using public-key cryptography based on this problem [8]. Seven years later, the NIST published the Digital Signature Algorithm (DSA) which was finally standardized as the Digital Signature Standard (DSS). All these algorithms have their keypair generation in common, which is described in Algorithm 2.2.

Algorithm 2.2 Keypair generation for DL based cryptosystems.

```
Input: Domain parameters $(p, q, g)$.
Output: Private key $x$ and public key $y$.
1: Choose an integer $x \in [1, q-1]$.
2: Compute $y = g^x \mod p$.
3: return $x, y$.
```

2.4.4. Elliptic Curve Discrete Logarithm Problem (ECDLP)

The problem which has been treated within this work is the ECDLP. Here, for a given elliptic curve E (defined by its domain parameters) and the public key Q, which represents a point on E, it is known to be "hard" to determine the scalar d from Equation (2.3):

$$Q = d * P. \tag{2.3}$$

Equation (2.3) represents the core of all elliptic-curve based cryptosystems, in which a keypair is generated according to Algorithm 2.3. This operation is called *point multiplication* (*cf.* Section 5.2) and is the most time consuming operation in ECC. Beside the definition of the elliptic curve E itself, the domain parameters include a base point P as well as its order n.

Algorithm 2.3 Keypair generation for elliptic curve based cryptosystems.

```
Input: Elliptic curve domain parameters.
Output: Private key $d$ and public key $Q$.
1: Randomly choose an integer $d \in [1, n-1]$.
2: Compute $Q = d * P$.
3: return $d, Q$.
```

Table 2.1.: Required keylength for different crypography schemes [11].

	Security level (bits)				
Symmetric-key cryptography	80	112	128	192	256
RSA	1024	2048	3072	8192	15360
ECC	160	224	256	384	512

Although the first experiments with elliptic curves started more than 150 years ago, it took until 1984 when factoring integers using elliptic-curve properties aroused more interest from researchers. Two of them, namely Neal Koblitz [22] and Victor Miller [27], independently proposed asymmetric-cryptography schemes using elliptic curves in 1985.

Preferring Elliptic-Curve based Cryptosystems

During the last 25 years elliptic-curve cryptography became more and more popular due to a major advantage in contrast to traditional cryptosystems like RSA. And that is, because for solving the ECDLP the currently most sophisticated algorithm requires fully exponential runtime, whereas solving the IFP takes only subexponential runtime. Hence within elliptic-curve systems a much shorter key length is needed to provide the same level of security. Table 2.1 compares different asymmetric schemes and their symmetric counterpart with respect to their keylength. An extensive research on required keylengths for different cryptography schemes can be found for example in [24].

2.4.5. Timeline

Figure 2.5 displays a timeline of the most important events happened in the late 20*th* century with regard to public-key cryptography. One can see that public-key cryptography is “quite a new” achievement, especially in combination with elliptic curves.

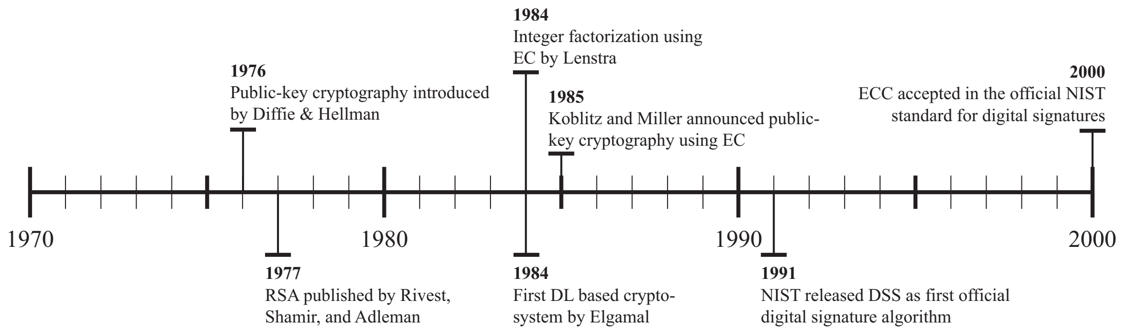

1984
Integer factorization using EC by Lenstra
1976
Public-key cryptography introduced by Diffie & Hellman
1985
Koblitz and Miller announced public-key cryptography using EC
2000
ECC accepted in the official NIST standard for digital signatures
1970
1980
1990
2000
1977
RSA published by Rivest, Shamir, and Adleman
1984
First DL based crypto-system by Elgamal
1991
NIST released DSS as first official digital signature algorithm

Figure 2.5.: Timeline of public-key cryptography.

Chapter 3

Elliptic-Curve Basics

Throughout this chapter, a brief introduction to the topic of elliptic curves will be presented. This starts with some numbertheoretical background, which is required to understand the mathematical hard problem on which ECC is based. Then the actual properties of an elliptic curve, including its definition as well as the operations which can be performed on it, will be described. Because the underlying field (discussed in Section 3.6) of an elliptic curve plays a major role in ECC, more attention will be paid to it afterwards. The chapter will close with an example, illustrating the complexity of the ECDLP.

3.1. Levels of Abstraction

When working with an elliptic curve, the highest level of abstraction is represented by the definition of the elliptic curve itself. From a geometrically point of view, this definition determines how the curve will be shaped. The next lower abstraction level contains the operations, which are performed on top of this elliptic curve. In order to carry out these elliptic curve operations, an underlying finite-field arithmetic is required, representing the lowest level of abstraction. Figure 3.1 illustrates the different abstraction levels of elliptic-curve implementations.

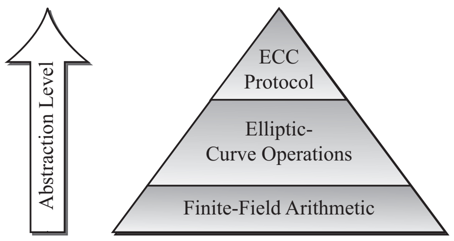


Figure 3.1.: Hierarchy of elliptic-curve implementations.

3.2. Number-Theoretic Background

In general, an elliptic curve E has to be defined over an underlying finite field K, denoted by E/K. Therefore a few mathematical basics with regard to number theory[1] have to be covered before starting with the actual topic about elliptic curves.

3.2.1. Group

From a mathematical point of view, a group denoted by $< \mathbb{S}, \diamond >$, is a set $\mathbb{S}$ of elements[2] together with an operation $\diamond$ fulfilling the following *group axioms*:

Closure Axiom. When applying operation $\diamond$ to any two elements x, y of the set $\mathbb{S}$, its result z must still be in $\mathbb{S}$, i.e.

$$\forall x, y \in \mathbb{S} : z = x \diamond y, z \in \mathbb{S}.$$

Associativity Axiom. The order of applying the operation $\diamond$ to any three elements x, y, z of the set $\mathbb{S}$ must not be of importance, i.e.

$$\forall x, y, z \in \mathbb{S} : (x \diamond y) \diamond z = x \diamond (y \diamond z).$$

Neutral/Identity-Element Axiom. For any element x of the set $\mathbb{S}$, there must exist an element e, also within $\mathbb{S}$, such that $x \diamond e = e \diamond x = e$, which is called the *neutral* or *identity element*, i.e.

$$\forall x \in \mathbb{S}, \exists e \in \mathbb{S} : x \diamond e = e \diamond x = e.$$

Inverse-Element Axiom. For any element x of the set $\mathbb{S}$, there must exist an element y in $\mathbb{S}$ such that $x \diamond y = y \diamond x = e$, which is called the *inverse element*, i.e.

$$\forall x \in \mathbb{S}, \exists y \in \mathbb{S} : x \diamond y = y \diamond x = e.$$

Abelian Group

If, in addition to the axioms defined above, a group holds the following axiom it is called an *Abelian group.*

Commutativity Axiom. The order of the elements $x, y \in \mathbb{S}$ performing the operation $\diamond$ must not be of relevance, i.e.

$$\forall x, y \in \mathbb{S} : x \diamond y = y \diamond x.$$

The two most frequently used groups are those, using the addition and multiplication as their group operation. They are called the *additive* and *multiplicative group* and are denoted by $< \mathbb{S}, + >$ and $< \mathbb{S}, \cdot >$ respectively. The identity element of $< \mathbb{S}, + >$ is denoted by 0 whereas 1 denotes the identity element of $< \mathbb{S}, \cdot >$.

[1] More details about number theory are given for example in [26].
[2] A set is a collection of distinct objects (e.g. $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$).

3.2.2. Field

A set $\mathbb{S}$ of elements together with two operations $\diamond$ and $\star$ is called a *field* denoted by $< \mathbb{S}, \diamond, \star >$ if:

- $< \mathbb{S}, \diamond >$ represents an Abelian group with the identity element denoted by 0.
- $< \mathbb{S}^*, \star >$ represents an Abelian group with the identity element denoted by 1. ($\mathbb{S}^* = \mathbb{S} \backslash \{0\}$)
- The operation $\diamond$ is distributive with respect to the operation $\star$, i.e. $\forall x, y, z \in \mathbb{S} : x \star (y \diamond z) = (x \star y) \diamond (x \star z)$.

Definition 3.2.1 (Finite Field)**.** *When the number of elements within $\mathbb{S}$ is finite, the field is called a "finite field", denoted by $\mathbb{F}$.*

The number of elements in a finite field is known as its *order*. For such a finite field $\mathbb{F}$ it holds that its order is equal to p^m, where p represents a prime number, called the *characteristic* of $\mathbb{F}$, and m an integer ≥ 1, called the dimension. Such a field is often denoted by $\mathbb{F}_{p^m}$ or $GF(p^m)$[3]. Finite fields can be distinguished by means of their order as follows.

- Finite fields with $m = 1$ are called *Prime Fields*, denoted by $\mathbb{F}_p$.
- When $m \geq 2$, finite fields are called *Extension Fields.*
- A special representative of the extension fields are the *Binary Fields* (characteristic is equal to 2), denoted by $\mathbb{F}_{2^m}$, where $m \geq 2$.

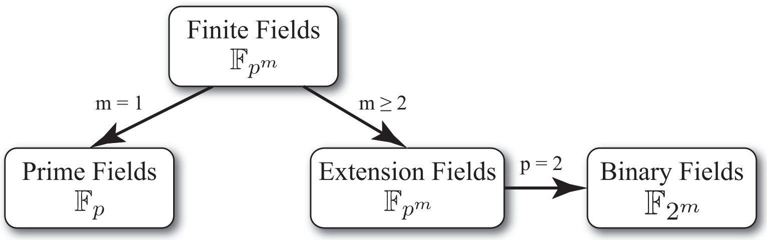

Finite Fields $\mathbb{F}_{p^m}$
m = 1
m ≥ 2
Prime Fields $\mathbb{F}_p$
Extension Fields $\mathbb{F}_{p^m}$
p = 2
Binary Fields $\mathbb{F}_{2^m}$

Figure 3.2.: Finite-Field Hierarchy.

It is a very common approach to use the additive and the multiplicative group when working with a field. Hence addition and multiplication are the two available base operations. Further required field operations, namely subtraction and division, are defined using these base operations. Subtraction on the one hand is defined using addition as depicted in Equation (3.1), where $-y$ represents the inverse element of y with regard

[3]GF is an abbreviation for "Galois Field", named after *Évariste Galois*, one of the pioneers with regard to finite fields.

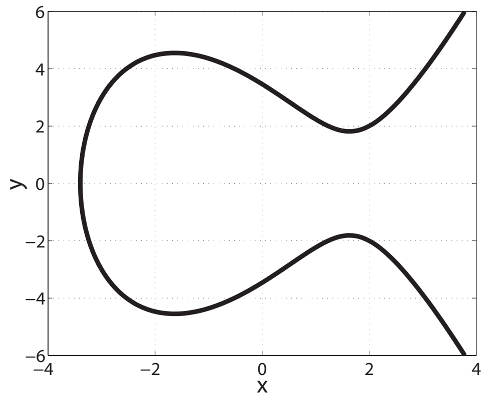


Figure 3.3.: $y^2 = x^3 - 8x + 12$.

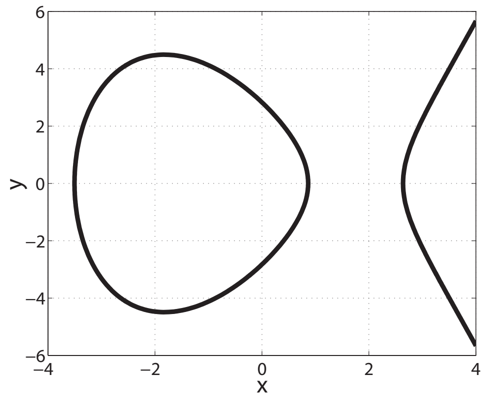


Figure 3.4.: $y^2 = x^3 - 5x + 3$.

to $< \mathbb{S}, + >$. Division on the other hand is defined using multiplication as shown in Equation (3.2), where y^{-1} represents the inverse element of y with regard to $< \mathbb{S}^*, \cdot >$:

$$x - y = x + (-y) \quad : x, y \in \mathbb{S}, \tag{3.1}$$

$$x/y = x * y^{-1} \quad : x, y \in \mathbb{S}^*. \tag{3.2}$$

3.3. Definition of an Elliptic Curve

Basically, an elliptic curve E is a cubic curve with arithmetic properties well suited for asymmetric cryptography. A curve defined over a field K can be described using the *Long Weierstrass Equation*, given in Equation (3.3):

$$E: \; y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \quad a_i \in K. \tag{3.3}$$

The graphs in Figure 3.3 and Figure 3.4 show two different elliptic curves defined over the field $\mathbb{R}$ of real numbers. The discriminant Δ of E must hold $\Delta \neq 0$ and is defined according to [11] as follows:

$$\begin{aligned} & \Delta = -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6, \\ \textit{where} \quad & d_2 = a_1^2 + 4a_2, \\ & d_4 = 2a_4 + a_1a_3, \\ & d_6 = a_3^2 + 4a_6, \qquad \textit{and} \\ & d_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2. \end{aligned}$$

The reason why elliptic curves are suitable for public key cryptography is, because an Abelian additive group can be defined on the set of points on E (together with the the point at infinity - *cf.* Section 3.4.2). Solving the DLP within this group (i.e. solving the

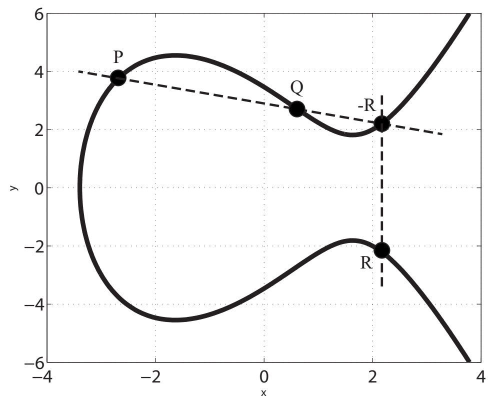


Figure 3.5.: Point addition.

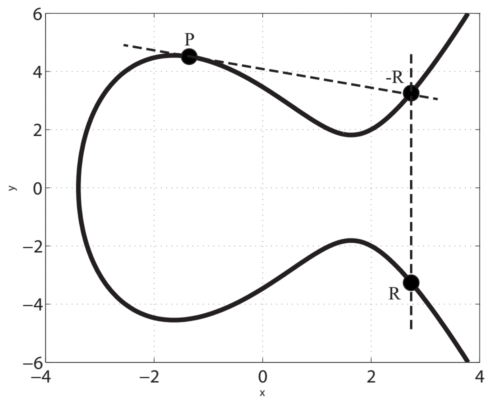


Figure 3.6.: Point doubling.

ECDLP) is known to be much harder than solving it within the group $\mathbb{Z}_p$, which is used in traditional DL based cryptography systems[4].

3.4. Elliptic-Curve Group

As already mentioned in Section 3.2.1 some axioms must be fulfilled so that a group can be defined. With regard to an elliptic curve E, the set $\mathbb{S}$ is formed by all points located on E which are represented using elements of the underlying field K. The set of points is denoted by $E(K)$. Together with the *chord-and-tangent* rule, this set forms an additive Abelian group, using the *point at infinity*, denoted by $\mathcal{O}$, as its neutral element.

3.4.1. Chord-and-Tangent Rule

Basically, the chord-and-tangent rule describes how two points on an elliptic curve have to be added, resulting in a third point located on the curve. In order to explain this addition, let's assume an elliptic curve E, defined over the field $\mathbb{R}$ of real numbers containing all points with coordinates x and y fulfilling:

$$E: \; y^2 = x^3 - 8x + 12 \quad x, y \in \mathbb{R}$$

Adding two distinct points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ on the elliptic curve E, results in a third point $R = (x_R, y_R)$, i.e.

$$R = P + Q, \quad P, Q \in E(K), P \neq Q \tag{3.4}$$

Figure 3.5 illustrates this *point addition* and is described in the following:

- Draw a straight line through the two points P and Q.

[4]Within the group $\mathbb{Z}_p$ all operations are performed modulo a large prime number p. The elements within $\mathbb{Z}_p$ are therefore $(0, 1, 2, \ldots, p-1)$.

- Take the third intersection of this line with the elliptic curve. This point is denoted by $-R$ and represents the negative element of R.
- Mirroring that intersection on the x-axis gives you the point R.

When the two points P and Q represent the same point, the addition operation is called *point doubling*, i.e.

$$R = P + Q = 2P, \quad P, Q \in E(K), P = Q \tag{3.5}$$

Figure 3.6 shows the point doubling and is described in the following:

- Draw the tangent in the point P.
- Take the second intersection of the tangent with the elliptic curve. This point once again is denoted by $-R$ and represents the negative of the sum of the point P with itself.
- To get the doubled point R, this intersection has to be mirrored on the x-axis.

3.4.2. The Point at Infinity $\mathcal{O}$

Point addition and point doubling does not always work as well as described in Section 3.4.1. Think of the situation when trying to add two points P and Q where $Q = -P$. The straight line drawn from P to Q results in a parallel line with regard to the y-axis which does not intersect the elliptic curve at a third point. The same problem arises when one tries to double a point where the tangent is also a parallel line to the y-axis (i.e. $y_P = 0$) and hence does not intersect the elliptic curve at a second point. These cases are illustrated in Figure 3.7 and Figure 3.8, respectively.

In order to get around this problem, a point, called the *point at infinity*, denoted by $\mathcal{O}$, has to be introduced. This point serves as the identity element within the additive elliptic-curve group, fulfilling the group axioms as follows.

Addition with identity element: $P + \mathcal{O} = \mathcal{O} + P = P, \quad P \in E(K)$.

Addition with inverse element: $P + (-P) = \mathcal{O}, \quad P \in E(K)$.

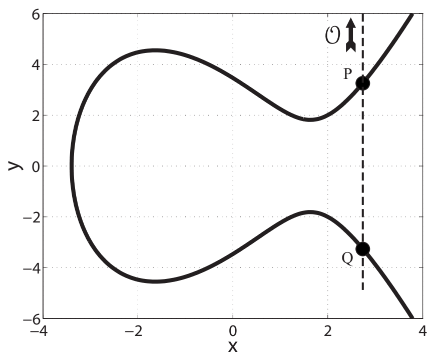


Figure 3.7.: Point addition with $\mathcal{O}$.

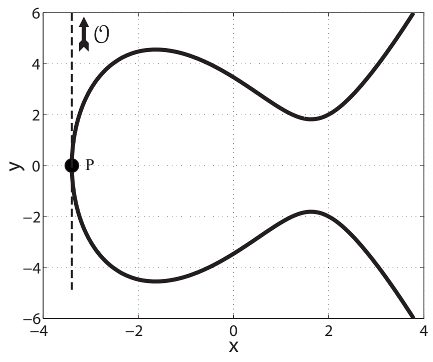


Figure 3.8.: Point doubling with $\mathcal{O}$.

Table 3.1.: Projective-coordinate types.

Name	**c**	**d**	**Correspondence**
Standard Projective	1	1	$(X, Y, Z)_{\mathbb{P}} \mapsto (x, y)_{\mathbb{A}} = (X/Z, Y/Z)$
Jacobian Projective	2	3	$(X, Y, Z)_{\mathbb{P}} \mapsto (x, y)_{\mathbb{A}} = (X/Z^2, Y/Z^3)$
López-Dahab Projective	1	2	$(X, Y, Z)_{\mathbb{P}} \mapsto (x, y)_{\mathbb{A}} = (X/Z, Y/Z^2)$

3.5. Choosing Point Coordinates

Because the point multiplication (*cf.* Section 5.2) consists of point additions and point doublings, it is of great relevance to implement these two operations applying the most efficient finite-field arithmetic. Basically, implementations can be distinguished with regard to the type of coordinates being used to represent the elliptic-curve points. The most intuitive way would be to use *affine coordinates* like in the previous sections for demonstration purposes. The set of points using affine coordinates over a finite field is denoted by $\mathbb{A}(K)$, representing all points according to:

$$\mathbb{A}(K) = \{(x, y) : x, y \in K\}.$$

When using affine coordinates, a division within the underlying finite field is required in every point addition and point doubling operation. Because in general division is much more time consuming in contrast to the remaining arithmetic, it should be avoided if possible. Therefore many implementations use *projective coordinates*, which circumvent the division at the expense of an additional coordinate Z. The set of points using projective coordinates over a finite field is denoted by $\mathbb{P}(K)$, representing all points according to:

$$\mathbb{P}(K) = \{(X, Y, Z) : X, Y, Z \in K\}. \tag{3.6}$$

Throughout the remainder of this work, affine coordinates will be written using lower-case letters, whereas capital letters will be used to represent projective ones. Transformations between affine and different projective-coordinate types can be carried out using the relationship

$$(X, Y, Z)_{\mathbb{P}} \mapsto (x, y)_{\mathbb{A}} = (X/Z^c, Y/Z^d),$$

where c and d represent positive integers. Table 3.1 summarizes the most popular types of projective coordinates together with their correspondences to affine coordinates. When choosing point coordinates different to affine ones, the definitions of the elliptic curves, presented in the following section, have to be altered accordingly.

3.6. Underlying Field

Elliptic-curve based implementations have to define several implementations. First, the actual curve itself has to be defined. Second, the underlying field, on which the curve operations are performed, has to be fixed. Basically, any field K can be chosen to perform the point addition and point doubling. Using the field $\mathbb{R}$ of real numbers like in the previous sections is quite handy for demonstration purposes, but poses problems with regard to speed and accuracy due to round-off errors. When it comes to practical implementations of cryptosystems based on elliptic curves, efficient arithmetic operations are required. The two most popular representitives, which have also been taken into consideration within this work, are operations over binary fields $\mathbb{F}_{2^m}$ and over prime fields $\mathbb{F}_p$.

3.6.1. Binary-Field Operations

For an underlying binary field $\mathbb{F}_{2^m}$, Equation (3.3) can be simplified using a so-called *admissible change of variables* (the substitution is shown in Equation (3.7)).

$$(x,y) \mapsto \left(a_1^2 x + \frac{a_3}{a_1}, a_1^3 y + \frac{a_1^2 a_4 + a_3^2}{a_1^3}\right). \tag{3.7}$$

The simplified description of the resulting elliptic curve is given in Equation (3.8) and holds as long as $a_1 \neq 0$. For further information, we refer to [11].

$$\begin{aligned} y^2 + xy = x^3 + ax^2 + b, &\qquad \textit{where } a, b \in \mathbb{F}_{2^m} \textit{ and} \\ \Delta = b. & \end{aligned} \tag{3.8}$$

Point Addition over $\mathbb{F}_{2^m}$

The geometrically illustration of the point operations from Section 3.4.1 was mentioned only for demonstration purposes. Using the linear equation, which can be set up running through the two given points P and Q, together with the elliptic-curve definition E, enables one to determine the algebraic relations between P, Q, and R. Equation (3.13) shows the resulting coordinates when adding the points over a binary field $\mathbb{F}_{2^m}$:

$$\begin{aligned} x_R &= \lambda^2 + \lambda + x_P + x_Q + a, \\ y_R &= \lambda(x_P + x_R) + x_R + y_P, \\ \text{with} \qquad \lambda &= \left(\frac{y_P + y_Q}{x_P + x_Q}\right). \end{aligned} \tag{3.9}$$

Point Doubling over $\mathbb{F}_{2^m}$

Similar to the point addition, the relations for the doubled point $R = 2P$ can be determined by intersecting the tangent running through P and the elliptic-curve definition E. Equation (3.10) shows the relations for the doubled point:

$$\begin{aligned} x_R &= \lambda^2 + \lambda + a, \\ y_R &= x_P^2 + \lambda x_R + x_R, \\ \text{with} \qquad \lambda &= x_P + \frac{y_P}{x_P}. \end{aligned} \tag{3.10}$$

Table 3.2.: Finite-field operations count for point addition and point doubling for an underlying binary field $\mathbb{F}_{2^m}$. Coordinates: $\mathcal{A}$ = affine, $\mathcal{P}$ = standard projective, $\mathcal{J}$ = Jacobian projective, $\mathcal{L}$ = López-Dahab projective; Operations: D = division, M = multiplication.

Point Addition		**Point Doubling**	
$\mathcal{A}+\mathcal{A}\to\mathcal{A}$	$1D, 1M$	$2\mathcal{A}\to\mathcal{A}$	$1D, 1M$
$\mathcal{P}+\mathcal{P}\to\mathcal{P}$	$13M$	$2\mathcal{P}\to\mathcal{P}$	$7M$
$\mathcal{J}+\mathcal{J}\to\mathcal{J}$	$14M$	$2\mathcal{J}\to\mathcal{J}$	$5M$
$\mathcal{L}+\mathcal{L}\to\mathcal{L}$	$14M$	$2\mathcal{L}\to\mathcal{L}$	$4M$

As already mentioned in Section 3.5, using projective coordinates instead of affine coordinaates eliminates the required field division at the expense of other field operations. Table 3.2 compares the point addition and the point doubling over $\mathbb{F}_{2^m}$ using different coordinate types with respect to the required field operations [11]. Because in $\mathbb{F}_{2^m}$ additions and squarings are cheap and can be performed in a single clock cycle (finite-field operations will be discussed throughout Chapter 4), only multiplications and divisions are considered.

3.6.2. Prime-Field Operations

For a finite field with a characteristic not equal to 2 or 3, which is true for the prime fields used within elliptic curve cryptography systems, it is possible to simplify the Long Weierstrass Equation using the admissible change of variables from Equation (3.11).

$$(x, y) \mapsto \left(\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1x}{216} - \frac{a_1^3 + 4a_1a_2 - 12a_3}{24}\right). \tag{3.11}$$

This substitution leads to the simplified version of the elliptic-curve definition over $\mathbb{F}_p$ and is depicted in Equation (3.12). Throughout the remainder of this work, the simplified elliptic-curve definitions for both an underlying binary and prime field will be used.

$$\begin{aligned} &y^2 = x^3 + ax + b, \qquad \textit{where } a, b \in \mathbb{F}_p \textit{ and} \\ &\Delta = -16(4a^3 + 27b^2). \end{aligned} \tag{3.12}$$

Point Operations over $\mathbb{F}_p$

Analogous to the point operations in $\mathbb{F}_{2^m}$, the algebraic equations for the point addition and point doubling in $\mathbb{F}_p$ can be derived. The appropriate relations are given in Equation (3.13) and Equation (3.14), respectively.

$$\begin{aligned} x_R &= \left(\frac{y_Q - y_P}{x_Q - x_P}\right)^2 - x_P - x_Q, \\ y_R &= \left(\frac{y_Q - y_P}{x_Q - x_P}\right)(x_P - x_R) - y_P. \end{aligned} \tag{3.13}$$

Table 3.3.: Finite-field operations count for point addition and point doubling for an underlying prime field $\mathbb{F}_p$. Coordinates: $\mathcal{A}$ = affine, $\mathcal{P}$ = standard projective, $\mathcal{J}$ = Jacobian projective; Operations: D = division, M = multiplication, S = squaring.

Point Addition		**Point Doubling**	
$\mathcal{A}+\mathcal{A}\to\mathcal{A}$	$1D, 1M, 1S$	$2\mathcal{A}\to\mathcal{A}$	$1D, 1M, 2S$
$\mathcal{P}+\mathcal{P}\to\mathcal{P}$	$12M, 2S$	$2\mathcal{P}\to\mathcal{P}$	$7M, 3S$
$\mathcal{J}+\mathcal{J}\to\mathcal{J}$	$12M, 4S$	$2\mathcal{J}\to\mathcal{J}$	$4M, 4S$

$$\begin{aligned} x_R &= \left(\frac{3x_P^2+a}{2y_P}\right)^2 - 2x_P, \\ y_R &= \left(\frac{3x_P^2+a}{2y_P}\right)(x_P - x_R) - y_P. \end{aligned} \tag{3.14}$$

The required field operations in $\mathbb{F}_p$ to perform the elliptic-curve arithmetic are compared in Table 3.3 with regard to their point representation [11].

3.7. Point Multiplication

Because the security of elliptic curve cryptosystems relies on the assumed intractability of the ECDLP, defined in Equation (2.3), the most important operation with regard to elliptic curves is the so called *point multiplication*. For a given point P on an elliptic curve E and a scalar k this means adding P to itself k times, i.e.

$$Q = k * P = \underbrace{P + P + P + \ldots + P}_{\text{k-times}}. \tag{3.15}$$

Because of the scalar k the point multiplication is also known as *scalar multiplication*. In order to explain why solving the ECDLP is that hard, let's take a look at a short example taken from [11]. Let E be an elliptic curve given in Equation (3.7) defined over the prime field $\mathbb{F}_{29}$. All points located on E are those shown in Table 3.4 together with the point at infinity $\mathcal{O}$.

$$E:\ y^2 = x^3 + 4x + 20, \quad x, y \in \mathbb{F}_{29}.$$

Table 3.4.: Points in $E(\mathbb{F}_{29})$.

$(0,7)$	$(2,6)$	$(4,10)$	$(6,12)$	$(10,4)$	$(14,6)$	$(16,2)$	$(19,13)$	$(24,7)$
$(0,22)$	$(2,23)$	$(4,19)$	$(6,17)$	$(10,25)$	$(14,23)$	$(16,27)$	$(19,16)$	$(24,22)$
$(1,5)$	$(3,1)$	$(5,7)$	$(8,10)$	$(13,6)$	$(15,2)$	$(17,10)$	$(20,3)$	$(27,2)$
$(1,24)$	$(3,28)$	$(5,22)$	$(8,19)$	$(13,23)$	$(15,27)$	$(17,19)$	$(20,26)$	$(27,27)$

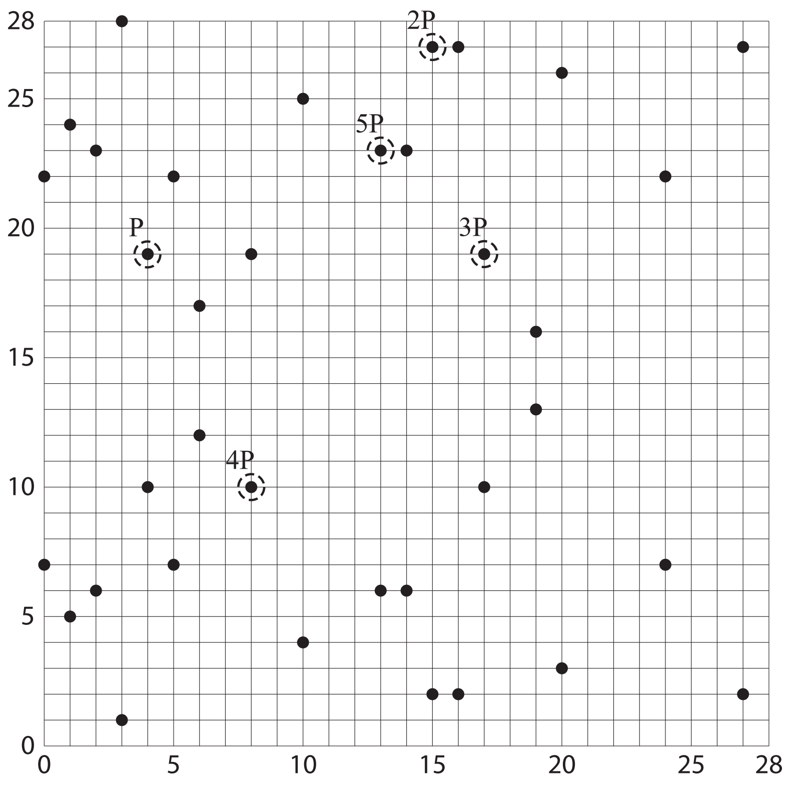


Figure 3.9.: Point cloud for $y^2 = x^3 + 4x + 20$ over $\mathbb{F}_{29}$.

In order to describe how the point multiplication using the sample point $P(4, 19)$ and the scalar $k = 5$ works, the resulting points are given below and are illustrated in Figure 3.9 within a grid. As one can see from the figure, adding a point to itself multiple times does not follow any regularity. This “chaotic”, almost random looking “jumping around” of points illustrates the difficulty to invert the point multiplication (i.e. solving the ECDLP).

Performing a brute-force attack to solve the ECDLP is always possible, of course. But when choosing an underlying finite field containing enough elements such that a brute-force attack is not feasible within an adequate period of time, solving the ECDLP becomes a *really hard problem.*

$$P = (4, 19) \quad 4P = (8, 10) \tag{3.16}$$

$$2P = (15, 27) \quad 5P = (13, 23) \tag{3.17}$$

$$3P = (17, 19) \tag{3.18}$$

Chapter 4

Finite-Field Arithmetic

Due to the fact that elliptic-curve operations are performed using underlying finite-field operations, it goes without saying that an efficient implementation of the field arithmetic is mandatory for a sophisticated elliptic-curve based cryptosystem. Hence, throughout this chapter, algorithms calculating the field operations given in Table 4.1 will be considered. At first, the operations in $\mathbb{F}_{2^m}$ will be treated, followed by those in $\mathbb{F}_p$. Beside the actual operations, some field-specific information will be presented, including for instance how the elements in each field are represented.

Table 4.1.: Finite-field operations in $\mathbb{F}_{2^m}$ and $\mathbb{F}_p$ required for ECC.

	$a, b \in \mathbb{F}_{\mathbf{2^m}}$	$a, b \in \mathbb{F}_{\mathbf{p}}$
Addition/Subtraction	$a \oplus b$	$a \pm b \mod p$
Multiplication	$a \cdot b \mod f(x)$	$a \cdot b \mod p$
Squaring	$a^2 \mod f(x)$	$a^2 \mod p$
Inversion/Division	$a^{-1} \mod f(x)$	$a/b \mod p$

4.1. Arithmetic in $\mathbb{F}_{2^m}$

Elements in $\mathbb{F}_{2^m}$ are represented using a bit string of the form shown in Equation (4.1), i.e. they can be treated as a m-dimensional vector:

$$a = (a_{m-1}\, a_{m-2} \dots a_2\, a_1\, a_0), \quad \text{where } a_i \in \{0, 1\}, \text{ and } i = 0 \dots m-1. \tag{4.1}$$

In order to know how this bit string has to be interpreted, a basis has to be chosen first. Basically, the following two common basis representations exist.

Normal-Basis Representation

If a normal basis of the form

$$x, x^2, x^4, \ldots, x^{2^{m-2}}, x^{2^{m-1}} \tag{4.2}$$

is used, the bit string within Equation (4.1) represents the $\mathbb{F}_{2^m}$ element shown in Equation (4.3):

$$a_0 x + a_1 x^2 + a_2 x^4 + \ldots + a_{m-2} x^{2^{m-2}} + a_{m-1} x^{2^{m-1}}. \tag{4.3}$$

Here x is selected such that the elements within Equation (4.2) are linearly independent.

Although this representation looks a little bit inconvenient, its major advantage in contrast to the polynomial-basis representation is that the squaring of field elements can be performed by a simple rotation of its bit string, i.e.

$$\begin{aligned} a &= (a_0\, a_1\, a_2 \ldots a_{m-2}\, a_{m-1}), \\ a^2 &= (a_{m-1}\, a_0\, a_1\, a_2 \ldots a_{m-2}). \end{aligned} \tag{4.4}$$

This might be of great interest, especially for hardware designs where squaring (or exponentiation) is the most important operation like it is in the case of RSA. Because field squaring is not that important for the ECC implementation within this work and field multiplication is considered to be a more complex operation when using this basis, normal basis-representation has not further been taken into account. Detailed information about the normal-basis representation as well as its arithmetic can be found for example in [30].

Polynomial-Basis Representation

When using a polynomial-basis representation, the elements of $\mathbb{F}_{2^m}$ are the polynomials of a degree at most $m-1$, as illustrated in Equation (4.5):

$$a(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0 \ : a_i \in \{0, 1\}. \tag{4.5}$$

Hence the bit string, given in Equation (4.1), simply contains the binary coefficients a_i of the corresponding polynomial. In order to perform arithmetic in $\mathbb{F}_{2^m}$, an *irreducible polynomial* or *field polynomial* $f(x)$ has to be chosen first.

Definition 4.1.1 (Irreducible Polynomial)**.** *Choose $f(x) \in \mathbb{F}_{2^m}$ of degree m. $f(x)$ is said to be irreducible over $\mathbb{F}_{2^m}$ if it cannot be factored as a product of two polynomials with degree less than m.*

For further information on finding such a polynomial, which exists for any m, we refer to, e.g., [11]. Operations in $\mathbb{F}_{2^m}$ have to be performed modulo $f(x)$. The irreducible polynomial $f(x)$ can also be written as within Equation (4.6). This representation is of interest, because it will be used in later sections.

$$f(x) = x^m + r(x) \tag{4.6}$$

Throughout the remainder of this work, a polynomial $a(x) \in \mathbb{F}_{2^m}$ is often written using a sole a, as long as it is understood from the context.

4.1.1. Addition and Subtraction

Addition in $\mathbb{F}_{2^m}$ is performed like any other addition of polynomials. Hence adding two polynomials $a(x)$ and $b(x)$ using binary coefficients is a simple bitwise addition of their coefficients. Because the operation is performed on binary coefficients only, addition and subtraction are identical over $\mathbb{F}_{2^m}$ and can be computed using an *exclusive or*, denoted by $\oplus$. Algorithm 4.1 shows the addition in $\mathbb{F}_{2^m}$ for two m-bit wide operands.

Algorithm 4.1 Addition and subtraction in $\mathbb{F}_{2^m}$.

```
Input: $a = (a_{m-1}, \dots, a_1, a_0)$, $b = (b_{m-1}, \dots, b_1, b_0)$, $a_i, b_i \in \{0, 1\}$.
Output: $c = (c_{m-1}, \dots, c_1, c_0) = a + b$, $c_i \in \{0, 1\}$.
1: for $i = 0$ to $m - 1$ do
2:     $c_i = a_i \oplus b_i$.
3: end for
4: return c.
```

4.1.2. Multiplication

Multiplication in $\mathbb{F}_{2^m}$ has to be performed modulo an irreducible polynomial as shown in Equation (4.7):

$$c(x) = a(x) \cdot b(x) \mod f(x). \tag{4.7}$$

This can basically be done in two different ways:

Multiplication without interleaved reduction: Given the two input polynomials $a(x)$, $b(x) \in \mathbb{F}_{2^m}$, the product $c'(x) = a(x) \cdot b(x)$ is calculated within the first step. Afterwards, $c'(x)$ gets reduced using the irreducible polynomial $f(x)$ resulting in $c(x) = c'(x) \mod f(x)$.

Multiplication with interleaved reduction: Within this method the multiplication is performed using an interleaved reduction. This means that the multiplication and the subsequent reduction are combined and hence for two given polynomials $a(x)$ and $b(x)$ of degree $m-1$, the (large) intermediate result $c'(x)$ of degree $2m-2$ must not be stored.

A very promising representative for the second type of multiplication is the *shift-and-add* method [11], which is described in the following.

Shift-and-Add Algorithm

Basically, this multiplication method is very similar to the multiplication performed by paper and pencil (i.e. the *school method*). Here the multiplicand a gets added to itself b times using straightforward shift and add operations. An example using the values $a = 10_d$ and $b = 13_d$ is given in Figure 4.1.

The shift-and-add method in $\mathbb{F}_{2^m}$ is based on the idea that within each iteration, the partial product $x^i \cdot b(x) \mod f(x)$ is calculated and depending on whether the current

```
      1010b* 1101b
---------------------
      1010
     0000
    1010
   1010
---------------------
   10000010
```

Figure 4.1.: *Paper and pencil* multiplication over $\mathbb{F}_{2^8}$.

bit of a is set or not, it is added to the result c. This is true, because for two binary polynomials $a(x)$ and $b(x)$ their product is equal to:

$$a(x) \cdot b(x) = a_{m-1}x^{m-1}b(x) + \ldots + a_2x^2b(x) + a_1xb(x) + a_0b(x)$$

In order to determine the intermediate result $x^i \cdot b(x) \mod f(x)$ within each iteration, the following idea is used. Given the polynomial in Equation (4.8), its multiplication with x results in Equation (4.9). The remainder of the polynomial division by $f(x)$ is shown in Equation (4.10). Therefore the intermediate result $x^i \cdot b(x) \mod f(x)$ can be computed by a simple shift operation followed by an addition with r, depending on the Most Significant Bit (MSB) of b. Algorithm 4.2 illustrates the multiplication process.

$$\begin{aligned} b(x) &= b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \ldots + b_2x^2 + b_1x + b_0, && (4.8)\\ b(x) \cdot x &= b_{m-1}x^m + b_{m-2}x^{m-1} + \ldots + b_2x^3 + b_1x^2 + b_0x && (4.9)\\ &\equiv b_{m-1}r(x) + b_{m-2}x^{m-1} + \ldots + b_2x^3 + b_1x^2 + b_0x \mod f(x). && (4.10) \end{aligned}$$

Algorithm 4.2 Shift-and-add multiplication in $\mathbb{F}_{2^m}$.

Input: $(a_{m-1} \ldots a_1\, a_0), (b_{m-1} \ldots b_1\, b_0)$, $a_i, b_i \in \{0, 1\}$.
Output: $(c_{m-1} \ldots c_1\, c_0)$ such that $c(x) = a(x) \cdot b(x) \mod f(x)$.

```
1:  if $a_0 = 1$ then
2:     $c = b$.
3:  else
4:     $c = 0$.
5:  end if
6:  for $i = 1$ to $m - 1$ do
7:     $b = b \cdot x \mod f(x)$.
8:     if $a_i = 1$ then
9:        $c = c + b$.
10:    end if
11: end for
12: return c.
```

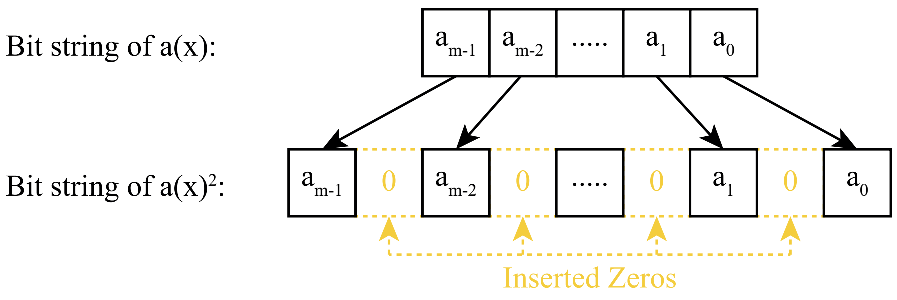


Figure 4.2.: Squaring in $\mathbb{F}_{2^m}$ using a polynomial-basis representation.

4.1.3. Squaring

The most obvious way the realize squaring in $\mathbb{F}_{2^m}$ would be to use the multiplier with twice the same input, of course. But because squaring a binary polynomial is a linear operation, it can be implemented much more efficiently. For a given binary polynomial

$$a(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0, \quad a_i \in \{0, 1\},$$

its squared value is equal to:

$$a(x)^2 = a_{m-1}x^{2m-2} + a_{m-2}x^{2m-4} + \cdots + a_1x^2 + a_0.$$

Hence squaring in $\mathbb{F}_{2^m}$ is analogous to inserting zeros between all coefficients as illustrated in Figure 4.2. When the irreducible polynomial $f(x)$ is fixed, the subsequent reduction step of the squared polynomial can be interleaved easily. The following example shows the squaring operation in $\mathbb{F}_{2^5}$ with the interleaved reduction using a reduction polynomial $f(x) = x^5 + x^2 + 1$ and a field element $a(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$. In order to obtain Equation (4.12) from Equation (4.11), a simple polynomial division is performed.

$$\begin{aligned} c(x) &= a(x)^2 \\ &= a_4x^8 + a_3x^6 + a_2x^4 + a_1x^2 + a_0 && (4.11) \\ &\equiv a_2x^4 + (a_4 + a_3)x^3 + (a_4 + a_1)x^2 + a_3x + a_4 + a_0 \mod f(x) && (4.12) \end{aligned}$$

As one can see from the example above, squaring in $\mathbb{F}_{2^m}$ can be executed using an *exclusive-or-network* as long as the irreducible polynomial is fixed.

4.1.4. Inversion and Division

One always has to keep in mind that the division of two field elements a and b can also be determined using an inversion with a subsequent multiplication, i.e. $a/b = a \cdot b^{-1}$. Hence in the literatur, algorithms for calculating the inverse of a field element can be found most of the time. Remember that the inverse a^{-1} of an element a in $\mathbb{F}_{2^m}$ is defined according to Equation (4.13).

$$a^{-1}a \equiv 1 \mod f(x), \quad a \in \mathbb{F}_{2^m} \qquad (4.13)$$

Inversion in general is the most complex and time-consuming field operation and there exist basically two different approaches to calculate it: one based on the *Extended Euclidean Algorithm* and one based on *Fermat's Little Theorem*. Inversion using the Euclidean algorithm is much more efficient, but requires additional logic in both, the datapath and the controlpath. Because the inversion based on Fermat's little theorem only requires field multiplications and squarings, just a few changes in the control logic are required to implement it. For the point representation within the binary-field design, projective coordinates have been chosen and hence only a single inversion is required during the scalar-multiplication process (for the transformation from projective to affine coordinates). Consequently, the second approach has been selected and is described in the following. For further information about the Euclidean-algorithm based approach, see Section 4.2.4.

Fermat's Little Theorem based Inversion

Fermat's little theorem, as given in Equation (4.14) for an arbitrary finite field $\mathbb{F}_q$ and the binary field $\mathbb{F}_{2^m}$, can be used to compute the inverse of a field element.

$$\begin{aligned} a^{-1} &= a^{q-2}, \quad && a \in \mathbb{F}_q, \\ a^{-1} &= a^{2^m-2}, \quad && a \in \mathbb{F}_{2^m}. \end{aligned} \tag{4.14}$$

Because $2^m - 2 = \sum_{i=1}^{m-1} 2^i$, Equation (4.14) becomes Equation (4.15). Hence the inverse element a^{-1} can be computed using $m-1$ squarings and $m-2$ multiplications:

$$a^{-1} = a^{\sum_{i=1}^{m-1} 2^i} = \prod_{i=1}^{m-1} a^{2^i}. \tag{4.15}$$

For the binary field $\mathbb{F}_{B163}$, it can be shown that the number of required squarings and multiplications can be greatly reduced resulting in Algorithm 4.3 which requires 9 multiplications and 162 squarings to perform a field inversion in $\mathbb{F}_{B163}$ [11].

4.2. Arithmetic in $\mathbb{F}_p$

For the ECDSA prime fields are of great interest, because on the one hand they are required to compute elliptic-curve operations (if one uses a prime field as the underlying field) and on the other hand the ECC protocol-level operations make use of $\mathbb{F}_p$ arithmetic (this will become clearer throughout Chapter 5). The elements in $\mathbb{F}_p$ are the integers, given in Equation (4.16):

$$\mathbb{F}_p = \{0, 1, 2, \ldots, p-2, p-1\}. \tag{4.16}$$

The value p is called the *modulus* of $\mathbb{F}_p$ and all operations in the field are performed modulo this prime number. Basically for an element $a \in \mathbb{F}_p$ infinite numbers of possible representations exist (i.e. $a, a \cdot p, a \cdot 2p, a \cdot 3p, \ldots$), however, using the value within $[0, p-1]$ requires the least memory and is therefore the best choice for a sophisticated implementation of a prime architecture.

Algorithm 4.3 Inversion in $\mathbb{F}_{2^m}$ using Fermat's little theorem.

Input: $a \in \mathbb{F}_{2^m}$, m odd.
Output: $a^{-1} \in \mathbb{F}_{2^m}$.

```
1: $A = a^2$, $B = 1$, $x = (m-1)/2$.
2: while $x \neq 0$ do
3:     $A = A \cdot A^{2^x}$.
4:     if $x$ is even then
5:         $x = x/2$.
6:     else
7:         $B = B \cdot A$, $A = A^2$, $x = (x-1)/2$.
8:     end if
9: end while
10: return $B$.
```

4.2.1. Addition and Subtraction

As well as for binary fields, the addition in $\mathbb{F}_p$ constitutes the easiest operation, i.e.

$$c = a + b \mod p, \quad a, b, c \in \mathbb{F}_p. \tag{4.17}$$

Because the resulting sum c might be equal to or greater than p, the modulus has to be subtracted accordingly to gain the most memory-saving representation of c.

The subtraction $a - b$ in the prime field can be written using the addition and the negative element $-b$ according to:

$$c = a - b \mod p = a + (-b) \mod p, \quad a, b, c \in \mathbb{F}_p.$$

Once again the difference has to be checked whether it is smaller than 0 or not and has to be corrected accordingly.

4.2.2. Multiplication

The multiplication in $\mathbb{F}_p$, as given in Equation (4.18), can be performed in two steps. For two integers a and b of width m, the intermediate result $c' = a \cdot b$ is determined during the first step whereas in the second step c' gets reduced by p to gain $c = c' \mod p$.

$$c = a \cdot b \mod p, \quad a, b, c \in \mathbb{F}_p \tag{4.18}$$

As soon as the width of the operands a and b increases (e.g. $m = 192$, as it is in $\mathbb{F}_{P192}$), attention must be paid to the carry propagation of the addition, which is required to add the partial products during the multiplication. This is due to the fact that this addition is often part of the critical path within designs and limits the maximal possible frequency. Therefore many designs found in the literature process the large numbers "word by word"[1](this is also known as *multi-precision arithmetic*).

[1]Typical values for the width of processor words are 8, 16, or 32 bits.

The two most commonly used methods to perform the multiplication word by word are mentioned in the following: The *operand scanning form*, which is also known as the *paper and pencil* method, has already been described throughout Section 4.1.2. In contrast to the operand scanning form, the second one, known as the *product scanning form*, computes all terms required for a single digit of the final product at the same time. Therefore the product scanning form needs less memory access and might be a better choice with respect to a hardware design. Detailed descriptions of the two multiplication methods can be found for example in [11].

Although multiplying large operands can be circumvented by processing them "word by word", the resulting product (which has to be reduced afterwards) is of width up to $2m$ and hence requires a lot of memory to be stored. In order to stay competitive with the field multiplication in $\mathbb{F}_{2^m}$, multiplication in $\mathbb{F}_p$ must also be interleaved with modular reduction. Luckily modular multiplication is a very central operation in cryptography and due to the popularity of RSA cryptosystems, which require a modular exponentiation (i.e. multiple prime-field multiplications), research with regard to field multiplications has been pushed a lot during the last centuries. An improved multiplication algorithm, which is considered to be one of the most efficient ones, is due to Peter Montgomery [28] and is described in the following.

Montgomery Multiplication

The main idea, developed by Montgomery, is to transform numbers from the integer domain $\mathbb{Z}$ into another number domain, called the *Montgomery domain*, denoted by $\mathbb{M}$. When choosing the transformation parameters wisely, the modular multiplication in $\mathbb{M}$ can be performed using very simple operations only (i.e. field addition and shift-operations, which are quite cheap especially in hardware). Throughout the following explanation, values within the $\mathbb{Z}$ domain will be represented using lower-case letters, whereas for values in the $\mathbb{M}$ domain capital letters will be used. The transformed n-bit value $X \in \mathbb{M}$ is sometimes called the *image* of x and can be illustrated using its binary representation as $X = (X_{n-1}\ X_{n-2}\ \dots\ X_1\ X_0)$. An image X of the integer x is defined according to:

$$X = x \cdot R \mod M, \quad x \in \mathbb{Z}, X \in \mathbb{M}, \tag{4.19}$$

where R is a constant called the *Montgomery radix* and M is the n-bit wide modulus fulfilling $2^{n-1} \leq M < 2^n$. The Montgomery multiplication basically calculates the so called *Montgomery product* (*cf.* Equation (4.20)), which results in another image Z:

$$MonPro(X, Y) = Z = X * Y = X \cdot Y \cdot R^{-1} \mod M. \tag{4.20}$$

Therefore it must hold that $gcd(R, M) = 1$, i.e. R and M have to be *co-prime.* If R is chosen such that $R = 2^n$ (which is recommended, because then the ouput is bounded by $2M$ according to [20]), $gcd(R, M) = 1$ is true as long as M is an odd value (which is true in general for cryptography purposes). Already many different variations exist in the literature to calculate the Montgomery product, a few of them are compared in [9]. Algorithm 4.4, taken from [19], describes how the Montgomery product can be determined. It contains simple additions and subtractions together with divisions by 2, which can easily be executed in hardware using shift operations. Also the modular division by 2 can be calculated using a conditional addition together with a shift operation according to Equation (4.21). Throughout the whole algorithm, the value of U is bounded by $2M$ and can therefore be corrected afterwards easily with a single subtraction. This correction step starts in line number 6.

Algorithm 4.4 Montgomery multiplication.

```
Input: $2^{n-1} < M < 2^n$, $gcd(M, 2) = 1$, $0 \leq X, Y < M$.
Output: $Z = X \cdot Y \cdot R^{-1} \mod M$, $0 \leq Z < M$.
 1: $A = Y$, $U = 0$, $V = X$.
 2: for $i = n$ downto 1 do
 3:     $A = (A - A_0)/2$.
 4:     $U = (U + A_0 V)/2 \mod M$.
 5: end for
 6: if $U \geq M$ then
 7:     $Z = U - M$.
 8: else
 9:     $Z = U - 0$.
10: end if
11: return $Z$.
```

$$\frac{X}{2} \mod M = \begin{cases} \frac{X}{2}, & X_0 = 0 \\ \frac{X+M}{2}, & X_0 = 1 \end{cases} \tag{4.21}$$

In order to transform the values from $\mathbb{Z}$ to $\mathbb{M}$ and vice versa, the same algorithm as for the Montgomery multiplication can be used. Value $x \in \mathbb{Z}$ can be transformed to $X \in \mathbb{M}$ by applying the Montgomery multiplication to x and the value $R^2 \mod M$, i.e. the squared and reduced value of the Montgomery radix (which can be precomputed as long as the modulus M does not vary). Transforming an image X back to its integer representation x can be achieved by executing the Montgomery-multiplication algorithm using the image and the value 1. The two transformations are given in Equation (4.22) and (4.23), respectively:

$$\mathbb{Z} \mapsto \mathbb{M}: \quad X = MonPro(x, R^2) = x \cdot R^2 \cdot R^{-1} \mod M = x \cdot R \mod M, \tag{4.22}$$

$$\mathbb{M} \mapsto \mathbb{Z}: \quad x = MonPro(X, 1) = X \cdot 1 \cdot R^{-1} \mod M = x \cdot R \cdot R^{-1} \mod M. \tag{4.23}$$

Another advantage of the Montgomery representation of numbers is that it dos not affect modular addition and modular subtraction algorithms. Hence, as long as no division is required, operations can be performed in the Montgomery domain with just a single transformation from $\mathbb{Z} \mapsto \mathbb{M}$ at the beginning of the calculation and a second one from $\mathbb{M} \mapsto \mathbb{Z}$ at the end.

4.2.3. Squaring

The squaring operation in $\mathbb{F}_p$ has been performed using the field multiplication with twice the same input according to Equation (4.24) and hence is not further treated herein:

$$a^2 \mod p = a \cdot a \mod p, \quad a \in \mathbb{F}_p. \tag{4.24}$$

4.2.4. Inversion and Division

As already mentioned in Section 4.1.4, one possibility to determine the inverse of a field element is based on the extended Euclidean algorithm and its variations. Because this approach requires quite computationally expensive operations, another amendment of it will be presented throughout the next two subsections.

Extended Euclidean based Inversion

The extended Euclidean algorithm is an enhancement of the classical Euclidean algorithm which is used to determine the *greatest common divisor (gcd)* of two non-negative integers a and b according to Algorithm 4.5.

Algorithm 4.5 Euclidean algorithm.

```
Input: $a, b \in \mathbb{N}^*$, $a \geq b$.
Output: $\gcd(a, b)$.
1: while $b \neq 0$ do
2:     $r = a \mod b$, $a = b$, $b = r$.
3: end while
4: return $a$.
```

The extension of this algorithm causes it, not only to yield the $\gcd(a, b)$, but also the two integers x and y satisfying

$$ax + by = \gcd(a, b).$$

Because the actual result of the extended Euclidean algorithm is not that interesting for the field inversion, the appropriate algorithm is solely given in Appendix A.1. Nevertheless, an amendment to it, listed in Algorithm 4.6, can be used to determine the inverse element $a^{-1} \mod p$ (taken from [11]). This works as long as the modulus p is prime and $a \in [1, p-1]$ (i.e. $\gcd(a, p) = 1$).

Using the extended Euclidean algorithm to determine the inverse of a field element suffers from the major drawback that it uses a computationally expensive division in line number 4. In order to get around this problem, a binary version, called the *Binary GCD Algorithm*, was used which will be explained in the following.

Algorithm 4.6 Inversion in $\mathbb{F}_p$ using the extended Euclidean algorithm.

```
Input: Prime $p$, $a \in [1, p-1]$.
Output: $a^{-1} \mod p$.
1: $u = a$, $v = p$.
2: $x_1 = 1$, $x_2 = 0$.
3: while $u \neq 1$ do
4:     $q = \lfloor v/u \rfloor$, $r = v - qu$, $x = x_2 - qx_1$.
5:     $v = u$, $u = r$, $x_2 = x_1$, $x_1 = x$.
6: end while
7: return $x_1 \mod p$.
```

Extended Binary GCD Algorithm

The main idea of this approach to determine $a^{-1} \mod p$ is that the computationally expensive division, required to determine the $\gcd(a, b)$ of two field elements a and b, is replaced by cheaper operations, namely add and shift operations. Detailed information about the computation of the gcd using these operations as well as the inversion process (i.e. $a^{-1} \mod p$) can be found in Appendix A.2.

Apart from the calculation of the inverse element, the extended binary gcd algorithm can be used to determine the quotient $a/b \mod p$ by simply changing the initialization values. A very promising version performing this division is due to Kaihara *et al.* [19] and is listed in Algorithm 4.7.

All required operations for the aforementioned algorithm are easy to implement in hardware. The *modular-halving* operation, $a/2 \mod p$, has already been explained in Section 4.2.2. The only operation which has not been treated yet, is the *modular-quartering* operation, i.e. $a/4 \mod p$. But this quotient can also be determined using cheap add and shift operations according to Equation (4.25):

$$\frac{a}{4} \mod p = \begin{cases} \frac{a}{4}, & a \equiv 0 \mod 4 \\ \frac{a-p}{4}, & a \equiv 1 \mod 4 \text{ and } p \equiv 1 \mod 4 \\ \frac{a-p}{4}, & a \equiv 3 \mod 4 \text{ and } p \equiv 3 \mod 4 \\ \frac{a+2p}{4}, & a \equiv 2 \mod 4 \\ \frac{a+p}{4}, & a \equiv 1 \mod 4 \text{ and } p \equiv 1 \mod 4 \\ \frac{a+p}{4}, & a \equiv 3 \mod 4 \text{ and } p \equiv 3 \mod 4 \end{cases}. \tag{4.25}$$

Keep in mind that the modular reduction for a modulus $m = 2^i, i \in N^*$ can be determined according to

$$a \mod m \mathrel{\widehat{=}} a \,\&\, (m-1), \tag{4.26}$$

where “&” represents a bitwise AND operation.

Algorithm 4.7 Extended binary GCD algorithm for division in $\mathbb{F}_p$.

```
Input: Prime modulus $p$ of width $n$, $2^{n-1} < p < 2^n$, $0 \leq x < p$, $0 < y < p$.
Output: $z \equiv x/y \mod p$.
 1: $a = y$, $b = p$, $u = x$, $v = 0$, $\rho = n$, $\delta = 0$.
 2: while $\rho \neq 0$ do
 3:   while $a \mod 2 = 0$ do
 4:     $a >> 1$, $u = u/2 \mod p$, $\rho = \rho - 1$, $\delta = \delta - 1$.
 5:   end while
 6:   if $\delta < 0$ then
 7:     $t = a$, $a = b$, $b = t$, $t = u$, $u = v$, $v = t$, $\delta = -\delta$.
 8:   end if
 9:   if $(a + b) \mod 4 = 0$ then
10:     $q = 1$.
11:   else
12:     $q = -1$.
13:   end if
14:   $a = (a + qb) >> 2$, $u = (u + qv)/4 \mod p$, $\rho = \rho - 1$, $\delta = \delta - 1$.
15: end while
16: if $b = 1$ then $z = v - 0$ else $z = p - v$ end if
17: return $z$.
```

4.3. Applied Algorithms

Table 4.2 summarizes the algorithms, which have been used to implement the finite-field operations in $\mathbb{F}_{2^m}$ and in $\mathbb{F}_p$, required for the ECC.

Table 4.2.: Applied algorithms for the finite-field operations in $\mathbb{F}_{2^m}$ and in $\mathbb{F}_p$.

	$\mathbb{F}_{\mathbf{2^m}}$	$\mathbb{F}_{\mathbf{p}}$
Addition / Subtraction	Exclusive-or relation 4.1.	Addition with subsequent subtraction of modulus 4.2.1.
Multiplication	Shift-and-add with interleaved reduction 4.2.	Montgomery multiplication 4.4.
Squaring	Exclusive-or-network 4.1.3.	Montgomery multiplication with twice the same input 4.4.
Inversion / Division	Fermat's little theorem based 4.3.	Extended binary GCD algorithm4.7.

Chapter 5

Elliptic-Curve Cryptography

This chapter will start by presenting the elliptic-curve based digitial signature protocol, which has been implemented within this work, namely ECDSA. Both the signature generation and the signature-verification process will be explained in detail. Due to the popularity of ECDSA already many institutions, like the NIST, standardized elliptic-curves and the appropriate underlying finite fields, that are well suited for cryptography purposes. The implemented curves, namely the B-163 and the P-192, are two of these representitives and will be handled afterwards. Due to the fact that the point multiplication dominates the execution time of ECC protocols, different approaches to implement it will be considered in Section 5.2. Throughout the verification process of ECDSA (*cf.* Section 5.1.2), the scalar multiplication has to be performed twice. Therefore at the end of this chapter an algorithm is given, which computes the required operation much more effectively than using two successive point multiplications.

5.1. Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is the elliptic-curve counterpart of the DSA and has already been standardized by many different institutions (e.g. by the NIST in the year 2000 [30]). Beside the definition of the elliptic curve itself, including the underlying finite field, ECDSA requires a prime-field arithmetic, a hash function, and a Random Number Generator (RNG) to provide its full functionality. Figure 5.1 shows a block diagram of the required components for an ECDSA implementation.

Although the RNG and the hash function are required to process the ECDSA in general, they have not been implemented within this work. Hence the required ephemeral key for the signature generation as well as the hash value of the message which should be signed/verified, are assumed to be given and have to be provided as an input.

Most ECC protocols, like ECDSA, require further arithmetic beside the one for elliptic curves, nevertheless the operations ontop of elliptic curves (i.e. the point multiplication including point addition and point doubling) are the most important operations in general. That is because they dominate the execution time of an elliptic-curve based

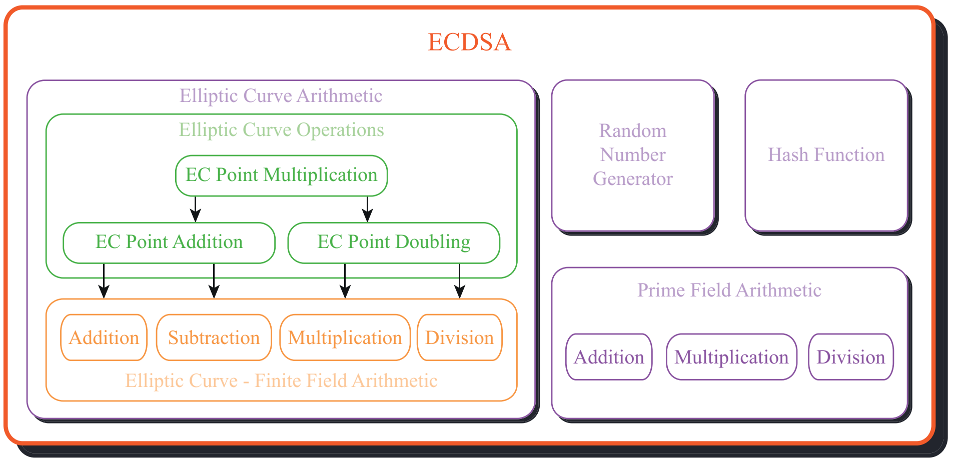


Figure 5.1.: Block diagram of the components required for an ECDSA implementation.

cryptosystem.

Like any other digital signature system, ECDSA consists of a signature-generation algorithm and its verification counterpart which are described in the following.

5.1.1. Signature Generation

The generation of a digital signature according to ECDSA is given in Algorithm 5.1. In contrast to the verification algorithm, it uses a RNG, which is required so that the same input message does not always result in the same digital signature and hence does not leak any information about the private key being used. The only elliptic-curve operation within this process can be found in line number 3, namely the point multiplication.

Algorithm 5.1 ECDSA signature generation.

```
Input: Domain parameters $(a, b, P, n)$, private key $d$, message $m$.
Output: Signature $(r, s)$.
1: Compute $e = HASH(m)$.
2: Randomly choose an integer $k \in [1, n-1]$.
3: Compute $k * P = (x_1, y_1)$.
4: Convert $x_1$ to an integer $\overline{x}_1$.
5: Compute $r = \overline{x}_1 \mod n$.
6: if $r = 0$ then Go to step 2.
7: Compute $s = (e + d \cdot r)/k \mod n$.
8: if $s = 0$ then Go to step 2.
9: return (r, s).
```

Despite the fact that not all provided domain parameters are used within the aforementioned algorithm, they are actually required, because they define the shape of the elliptic curve and are needed throughout the elliptic-curve operations.

5.1.2. Signature Verification

The verification process of the ECDSA is listed in Algorithm 5.2. Here the only elliptic-curve operations can be found in line number 5, namely two point multiplications and a subsequent point addition. One can combine these three elliptic-curve operations into the so-called *multiple point-multiplication* (*cf.* Section 5.3).

Algorithm 5.2 ECDSA signature verification.

```
Input: Domain parameters $(a, b, P, n)$, public key $Q$, message $m$, signature $(r, s)$.
Output: Acceptance or rejection of signature.
 1: if $r \notin [1, n-1]$ or $s \notin [1, n-1]$ then return Reject signature.
 2: Compute $e = HASH(m)$.
 3: Compute $u_1 = e/s \mod n$.
 4: Compute $u_2 = r/s \mod n$.
 5: Compute $X = u_1 * P + u_2 * Q = (x_1, y_1)$.
 6: if $X = \mathcal{O}$ then return Reject signature.
 7: Convert $x_1$ to an integer $\overline{x}_1$.
 8: Compute $v = \overline{x}_1 \mod n$.
 9: if $v = r$ then return Accept signature.
10: else return Reject signature.
```

Observing Algorithm 5.1 and Algorithm 5.2 reveals that the signature generation and the signature-verification process both require more or less the same operations (except of generating random numbers). Therefore the components needed during signature generation and signature verification can be shared among these two modes of operation.

The following proof explains why for a valid keypair d and Q, the verification process works correctly [11]. Keep in mind that the public key Q is determined according to $Q = d * P$ and the modular division, i.e., $x/y \mod n$, can be written using the multiplication and the inverse element y^{-1}.

ECDSA Verification Process Proof:

$$\begin{aligned}
s &\equiv (e + d \cdot r)/k \mod n \\
&\equiv (e + d \cdot r) \cdot k^{-1} \mod n \\
k &\equiv s^{-1} \cdot (e + d \cdot r) \mod n \\
&\equiv \underbrace{s^{-1} \cdot e}_{u_1} + \underbrace{s^{-1} \cdot r}_{u_2} \cdot d \mod n \\
&\equiv u_1 + u_2 \cdot d \mod n \\
X &= u_1 * P + u_2 * Q = u_1 * P + u_2 \cdot d * P = (x_1, y_1) \\
&= \underbrace{(u_1 + u_2 \cdot d)}_{k} * P
\end{aligned}$$

And hence $v = x_1 \mod n = r$ iff the correct keypair has been used.

$\square$

5.1.3. Applied NIST Standards

Due to the widely spread usage of elliptic curves within cryptosystems, institutions like the NIST standardized curves with certain properties that are well suited for cryptography purposes. The two curves being used within this work, namely the *NIST B-163* and the *NIST P-192*, are recommend in [30] and have been implemented according to [13]. The standard defines their domain parameters, specifying how the elliptic curve looks like as well as the underlying finite field.

Basically, NIST provides 15 elliptic curves which are recommended for the use within cryptosystems. They offer different levels of security based on the length of the parameters being used. The mentioned curves have been chosen, because they process the smallest values among those, recommended by NIST and hence require the fewest resources with regard to area and runtime. Furthermore, these curves are absolutely sufficient for a state-of-the-art cryptosystem.

NIST B-163 Curve

The underlying finite field of this curve is a binary field $\mathbb{F}_{2^m}$ with $m = 163$. The domain parameters of $\mathbb{F}_{B163}$ are described and given in Table 5.1. Some of them are written using hexadecimal values showing the 163-bit wide vectors containing the coefficients of the polynomial basis-representation as described in Section 4.1.

Table 5.1.: NIST B-163 domain parameters.

		Domain Parameter Descriptions
a, b	...	Coefficients describing the elliptic curve.
x	...	The x coordinate of the base point P.
y	...	The y coordinate of the base point P.
n	...	The order of the base point P.
$f(x)$	...	Reduction polynomial.

		Domain Parameter Values
a	=	1
b	=	0x 00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD
x	=	0x 00000003 F0EbA162 86A2D57E A0991168 D4994637 E8343E36
y	=	0x 00000000 D51FBC6C 71A0094F A2CDD545 B11C5C0C 797324F1
n	=	0x 00000004 00000000 00000000 000292FE 77E70C12 A4234C33
f(x)	=	0x 00000008 00000000 00000000 00000000 00000000 000000C9

NIST P-192 Curve

The P-192 curve works on an underlying finite field $\mathbb{F}_{P192}$ using 192-bit wide values. The domain parameters describing the curve are given in Table 5.2.

Table 5.2.: NIST P-192 domain parameters.

Domain Parameter Descriptions		
a, b	...	Coefficients describing the elliptic curve.
x	...	The x coordinate of the base point P.
y	...	The y coordinate of the base point P.
n	...	The order of the base point P.
p	...	Prime field order.
Domain Parameter Values		
a	=	-3
b	=	0x 64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1
x	=	0x 188DA80E B03090F6 7CBF20EB 43A18800 F4FF0AFD 82FF1012
y	=	0x 07192B95 FFC8DA78 631011ED 6B24CDD5 73F977A1 1E794811
n	=	0x FFFFFFFF FFFFFFFF FFFFFFFF 99DEF836 146BC9B1 B4D22831
p	=	0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFF

5.2. Point Multiplication

Throughout this section, the point multiplication will be considered, i.e. the operation computing $k * P$. Basically many different approaches exist to determine the result of the scalar multiplication, some of them require precomputations, others do not. Because any kind of precomputation requires additional storage, it is not of interest within this work. For further information about point-multiplication algorithms not handled in the following subsections, we refer for example to [11] or [15].

5.2.1. Naive Point-Multiplication Approach

For a given scalar k and an elliptic curve point P the most obvious way to perform a point multiplication within the additive elliptic-curve group would be, to add P to itself k-times. For this approach, listed in Algorithm 5.3, $k - 1$ point additions would be required. Due to the huge number of point additions, this would lead to an unacceptable runtime proportional to k.

Algorithm 5.3 Naive point-multiplication approach.

```
Input: Elliptic-curve point $P \in E(K)$, scalar $k$.
Output: $Q = k * P$.
1: $Q = P$.
2: for $i = k - 1$ downto 1 do
3:     $Q = Q + P$.
4: end for
5: return $Q$.
```

5.2.2. Double-and-Add Algorithm

Improvements concerning the required runtime can be made by using the so-called *double-and-add* algorithm (with regard to the multiplicative group, it is known as the *square-and-multiply* algorithm). It uses a scalar k given in the binary expansion, i.e.

$$k = \sum_{i=0}^{t-1} k_i \cdot 2^i, \quad k_i \in \{0, 1\},$$

and performs the scalar multiplication according to Algorithm 5.4. The main idea of this algorithm is, to add P to itself depending on whether the current bit of k is set or not and considering the current position of the bit by doubling the value of Q.

Algorithm 5.4 Left-to-right double-and-add algorithm.

```
Input: Elliptic curve point $P \in E(K)$, scalar $k = (k_{t-1}, \dots, k_0)_2$.
Output: $Q = k * P$.
1: $Q = \mathcal{O}$.
2: for $i = t - 1$ downto 0 do
3:    $Q = 2Q$.
4:    if $k_i = 1$ then
5:       $Q = Q + P$.
6:    end if
7: end for
8: return $Q$.
```

A major problem with regard to security arises, because the runtime depends on the Hamming weight of k. Hence a potential adversary is able to gain information about k, using for instance a timing analysis or a Simple Power Analysis (SPA). This irregularity poses a considerable threat, especially when k represents any kind of secret like it does within the ECDSA.

Nevertheless this algorithm would require $\lceil \log_2 k \rceil$ point doublings and up to $\lceil \log_2 k \rceil$ point additions. Assuming that the *zeros* and *ones* in k are uniformly distributed, the number of point additions becomes $\left\lceil \frac{\log_2 k}{2} \right\rceil$.

5.2.3. Montgomery Ladder Algorithm

An algorithm, which performs the scalar multiplication within a fixed amount of time, was proposed by Peter L. Montgomery [29]. According to its inventor it is called the *Montgomery Ladder* and is given in Algorithm 5.5.

Although this point-multiplication algorithm does not depend on the Hamming weight of the scalar k, one has to keep in mind that also the underlying abstraction levels (i.e. point addition and point doubling as well as the finite-field arithmetic) must not leak any information about the runtime, such that the scalar multiplication can be performed regularly. The constantly required number of point additions and point doublings for the Montgomery Ladder algorithm is equal to $\lceil \log_2 k \rceil$.

Algorithm 5.5 Montgomery Ladder algorithm.

```
Input: Elliptic curve point $P \in E(K)$, scalar $k = (k_{t-1}, \dots, k_0)_2$.
Output: $Q = k * P$.
 1: $Q = \mathcal{O}$, $R = P$.
 2: for $i = t - 1$ downto 0 do
 3:     if $k_i = 1$ then
 4:         $Q = Q + R$.
 5:         $R = 2R$.
 6:     else
 7:         $R = R + Q$.
 8:         $Q = 2Q$.
 9:     end if
10: end for
11: return $Q$.
```

5.2.4. X-Coordinate Only Montgomery Ladder Algorithm over $\mathbb{F}_{2^m}$

A very promising improvement to the algorithm described in Section 5.2.3 using an underlying binary field $\mathbb{F}_{2^m}$ is due to Julio López and Ricardo Dahab [25], based on an idea by Peter Montgomery [29]. Their scalar-multiplication algorithm is applicable for arbitrary scalars k and does not need any kind of precomputations. Basically it is based on the idea, that for two distinct points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ whose difference $P_4 = (x_4, y_4) = P_2 - P_1$ is known, the x-coordinate of the sum $P_3 = (x_3, y_3) = P_1 + P_2$ can be obtained only from the x-coordinates of P_1, P_2 and P_4 according to Equation (5.1). This is true, as long as the relationship $P_2 - P_1$ is maintained invariant, because then the x-coordinate of P_4 satisfies $x_4 = x_P$.

$$x_3 = \begin{cases} x_P + \left(\frac{x_1}{x_1+x_2}\right)^2 + \frac{x_1}{x_1+x_2}, & P_1 \neq P_2 \\ x_1^2 + \frac{b}{x_1^2}, & P_1 = P_2 \end{cases} \tag{5.1}$$

López and Dahab provide two different versions of their algorithm, one uses affine coordinates whereas the other one is based on projective ones and is listed in Algorithm 5.6. Furthermore, they present a formula, given in Equation (5.2), which can be used to restore the y-coordinate of the resulting point Q subsequent to the whole scalar-multiplication procedure. Because within the ECDSA the y-coordinate of the resulting point is not required, no further attention has been paid to its recovery.

$$y_Q = (x_1 + x_P) \cdot \{(x_1 + x_P) \cdot (x_2 + x_P) + x_P^2 + y_P\}/x_P + y_P \tag{5.2}$$

For the algorithm a scalar k is assumed, where the MSB must be set. This is quite handy, because with this assumption the first iteration of the *for-loop* can be skipped by initializing X_1, Z_1, X_2 and Z_2 to their appropriate values and hence point additions and point doublings with the point at infinity $\mathcal{O}$ do not have to be handled. The operations *Madd* and *Mdouble*, performing the point addition and the point doubling using projective coordinates, are given in Algorithm 5.7 and Algorithm 5.8, respectively.

Algorithm 5.6 X-Coordinate only Montgomery Ladder algorithm over $\mathbb{F}_{2^m}$.

```
Input: Elliptic curve point $P(x_P, y_P) \in E(\mathbb{F}_{2^m})$, scalar $k = (1, k_{t-2}, \ldots, k_0)_2$.
Output: $x_Q$ of $Q = (x_Q, y_Q) = k * P$.
1: $X_1 = x_P,\ Z_1 = 1,\ X_2 = x_P^4 + b,\ Z_2 = x_P^2$.
2: for $i = t-2$ downto $0$ do
3:    if $k_i = 1$ then
4:       $(X_1, Z_1) = Madd(X_1, Z_1, X_2, Z_2)$, $(X_2, Z_2) = Mdouble(X_2, Z_2)$.
5:    else
6:       $(X_2, Z_2) = Madd(X_2, Z_2, X_1, Z_1)$, $(X_1, Z_1) = Mdouble(X_1, Z_1)$.
7:    end if
8: end for
9: return $x_Q = X_1/Z_1$.
```

Algorithm 5.7 *Madd* operation.

```
Input: $X_1$, $Z_1$, $X_2$, $Z_2$, $x_P$.
Output: $(X_3, Z_3) = Madd(X_1, Z_1, X_2, Z_2)$.
1: $X_3 = X_1 \cdot Z_2$.
2: $T_1 = X_2 \cdot Z_1$.
3: $Z_3 = X_3 + T_1$.
4: $Z_3 = Z_3^2$.
5: $T_1 = X_3 \cdot T_1$.
6: $X_3 = x_P \cdot Z_3$.
7: $X_3 = X_3 + T_1$.
8: return $(X_3, Z_3)$.
```

Algorithm 5.8 *Mdouble* operation.

```
Input: $X_1$, $Z_1$, $b$.
Output: $(X_3, Z_3) = Mdouble(X_1, Z_1)$.
1: $X_3 = X_1^2$.
2: $T_1 = Z_1^2$.
3: $Z_3 = X_3 \cdot T_1$.
4: $X_3 = X_3^2$.
5: $T_1 = T_1^2$.
6: $T_1 = T_1 \cdot b$.
7: $X_3 = X_3 + T_1$.
8: return $(X_3, Z_3)$.
```

Table 5.3.: Comparison of point-multiplication algorithms with regard to the number of required point additions and point doublings.

	Elliptic-Curve Operations	
Algorithm	**Additions**	**Doublings**
Naive point-multiplication approach	k	-
Double-and-add[1]	$\left\lceil \frac{\log_2 k}{2} \right\rceil$	$\lceil \log_2 k \rceil$
Montgomery Ladder	$\lceil \log_2 k \rceil$	$\lceil \log_2 k \rceil$
X-coordinate only Montgomery Ladder[2]	$\lceil \log_2 k \rceil$	$\lceil \log_2 k \rceil$

Although the number of required point additions and point doublings is equal to those from Algorithm 5.5, the x-coordinate only version requires significantly less field operations to perform the two elliptic-curve operations (*cf.* Section 3.6). One iteration of the algorithm requires 6 multiplications for both the point addition and the point doubling (field additions and squarings are neglected because they require significantly less operations than the multiplication). Furthermore, if the elliptic curve and the point with which the scalar k has to be multiplied (i.e. the base point in ECDSA) are maintained constant, two of these multiplications represent multiplications with a constant value and hence can be possibly simplified.

5.2.5. Comparison of Point-Multiplication Algorithms

In order to get an overview of the presented scalar-multiplication algorithms determining the value of $k * P$, Table 5.3 compares them with regard to the required number of point additions and point doublings.

For the design based on the underlying binary field $\mathbb{F}_{B163}$, the x-coordinate only version of the Montgomery Ladder has been chosen, whereas for the design based on the prime field $\mathbb{F}_{P192}$ the Montgomery Ladder has been selected. The provided algorithms have only been used during the digital signature generation of the ECDSA design. For the verification process it is required to perform the point multiplication twice. In order not to double the runtime due to the two time-consuming scalar multiplications, the approach presented in the next section has been used.

5.3. Multiple Point-Multiplication

The most evident approach to perform the elliptic-curve operations required in the ECDSA signature verification would be, to use two point multiplications together with a subsequent point additioin (*cf.* Section 5.1.2). A more convenient and faster way is

[1]For the point additions it has been assumed, that the number of *zeros* and *ones* in the binary expansion of k is uniformly distributed.

[2]Although the number of required elliptic-curve operations is equal to the standard Montgomery Ladder, significantly less finite-field operations are desired.

the so-called *multiple point-multiplication* which is also known as *Shamir's trick* and was originally proposed by E. Straus [32]. Algorithm 5.9, taken from [10], describes how the multiple scalar-multiplication works.

Algorithm 5.9 Multiple Point-Multiplication.

```
Input: $P, Q \in E(K)$, $k = (k_{t-1}, \ldots, k_0)_2$ and $l = (l_{t-1}, \ldots, l_0)_2$.
Output: $R = k * P + l * Q$.
1: $R = \mathcal{O}$.
2: $Z = P + Q$.
3: for $i = t - 1$ downto 0 do
4:    $R = 2R$.
5:    if $k_i = 1$ and $l_i = 0$ then
6:       $R = R + P$.
7:    else if $k_i = 0$ and $l_i = 1$ then
8:       $R = R + Q$.
9:    else if $k_i = 1$ and $l_i = 1$ then
10:      $R = R + Z$.
11:   end if
12: end for
13: return $R$.
```

Although the runtime of this algorithm depends on the *joint Hamming weight*[3] of k and l, this does not pose a threat within the ECDSA verification, because no secret key is part of the verification process and hence might be eavesdropped by an adversary using any kind of side-channel attack. The multiple point-multiplication requires λ point doublings and up to λ point additions, where $\lambda = \max(\lceil \log_2 k \rceil, \lceil \log_2 l \rceil)$. One little drawback, which is accepted due to the reduced execution time, is, that the sum of the two points P and Q has to be precomputed and stored during the whole algorithm.

[3]The joint Hamming weight of two scalars $k = (k_{t-1}, \ldots, k_0)_2$ and $l = (l_{t-1}, \ldots, l_0)_2$ is the number of nonzero bit vectors (k_i, l_i) such that $(k_i, l_i) \neq (0, 0)$.

Chapter 6

Design Implementation

Throughout this chapter, the implementation of two ASIC designs, which provide a signature generation and a signature verification-process according to ECDSA will be presented. One of them is based on an elliptic-curve arithmetic using $\mathbb{F}_{B163}$ as the underlying finite field and the other one uses $\mathbb{F}_{P192}$. Both designs have been worked out processing the operands at full precision.

The chapter starts with the basic design idea, followed by a short insight into the golden model. Next, the architecture and the HDL model implementations of the finite-field arithmetics will be described. Throughout Section 6.8, the applied chip interface will be addressed, followed by the verification of the HDL model. Finally a brief overview of the steps, which have been performed during the back-end design will be given.

6.1. Introduction

Basically, the implementation of a chip design can be roughly subdivided into the steps illustrated in Figure 6.1. During this chapter the individual steps of this block diagram will be covered in terms of the implemented designs, focussing on the implementation of the finite-field arithmetic for both the binary field $\mathbb{F}_{B163}$ and the prime field $\mathbb{F}_{P192}$.

6.2. Design Idea / Basic Architecture

Many ECDSA designs, currently available in the literature, are in general based on a core logic, working on a 8, 16, or 32-bit wide datapath where the Arithmetic Logic Unit (ALU) includes components like an adder, a Multiply Accumulate (MAC) unit and some further components. This is the case, because they process the large operands "word by word" (i.e. *multi precision*). Furthermore, they usually contain any kind of memory as well as a controlling unit, which is responsible for creating the control signals, required to calculate the correct operation during each clock cycle. A sketch of the required components for this design approach is given in Figure 6.2. Due to the narrow and small datapath, the most complexity herein is located in the controlpath.

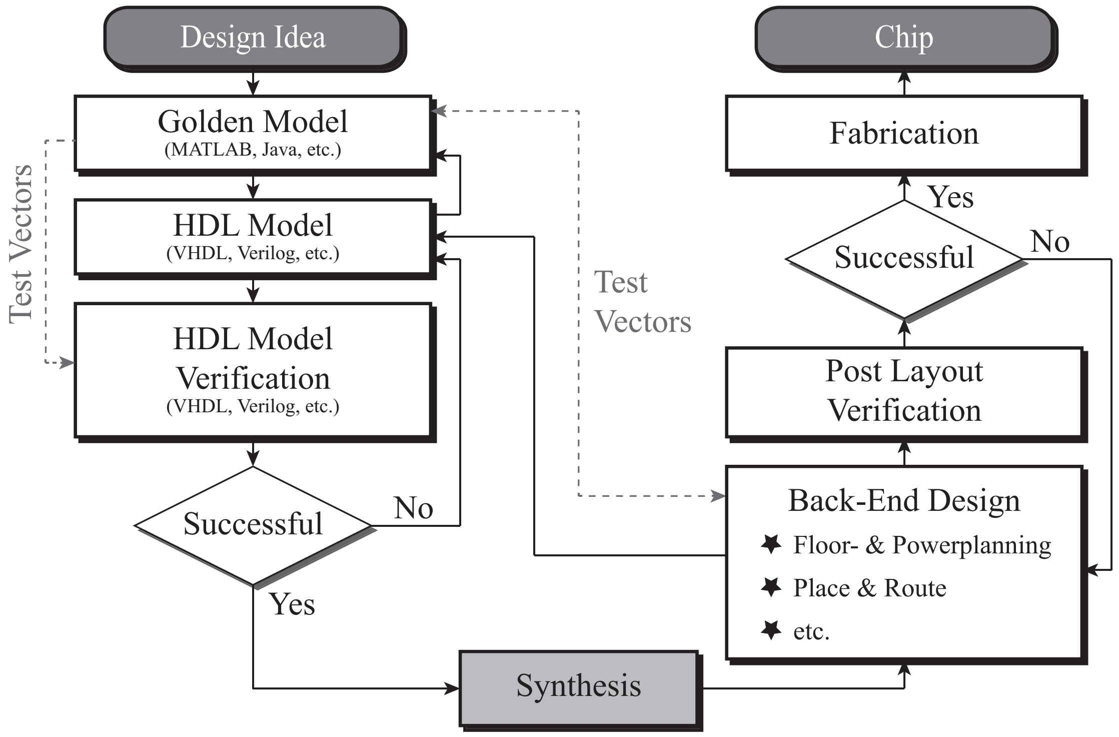


Figure 6.1.: Basic design flow for a chip development.

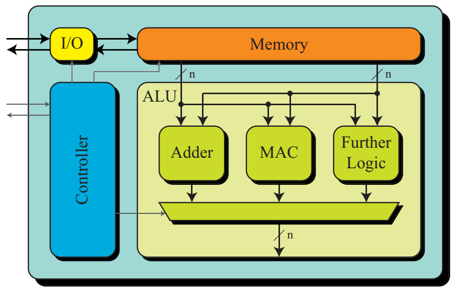


Figure 6.2.: Processor architecture with n usually equal to 8, 16, or 32.

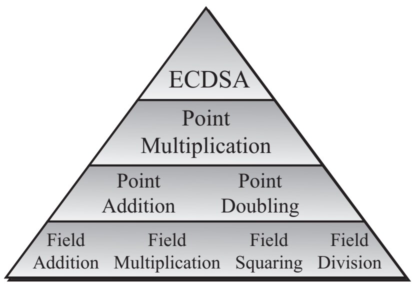
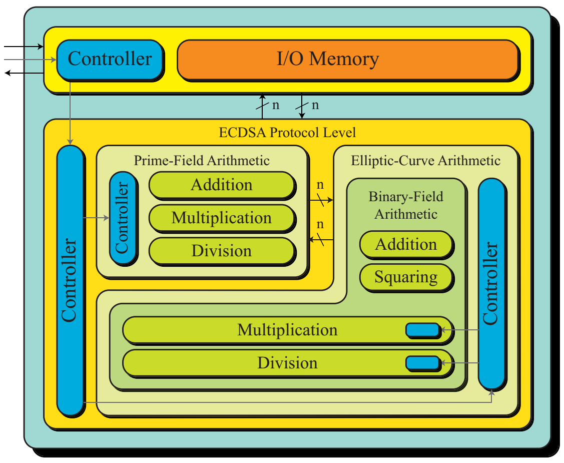
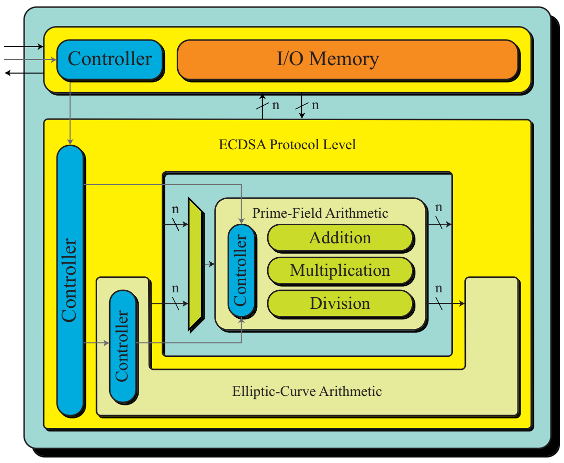


Figure 6.3.: Hierarchy of the ECDSA protocol.



(a) $\mathbb{F}_{B163}$ design.



(b) $\mathbb{F}_{P192}$ design.

Figure 6.4.: Design overviews of the full precision ECDSA designs using $\mathbb{F}_{B163}$ and $\mathbb{F}_{P192}$ as the underlying finite field for the elliptic-curve operations.

The designs implemented within this work are based on a different architecture. In general, the focus of the thesis lies on a competitive implementation of the elliptic-curve arithmetic using $\mathbb{F}_{B163}$ as the underlying finite-field. The design therefor processes the 163-bit wide operands at their full bit width (i.e. *full precision*). In order to implement two compareable designs, one working on $\mathbb{F}_{B163}$ and the other one on $\mathbb{F}_{P192}$ as the underlying finite field, also the prime-field version was designed at full precision. Because the cell library of the manufacturer[1], where the chips have been fabricated, didn't support the use of Random Access Memories (RAMs), standard-cell registers have been used as storage elements.

Due to the fact that the ECDSA is built up in a very hierarchical structure (*cf.* Figure 6.3), also the chip architecture has been developed according to this. This means that the lowest hierarchy level (i.e. the finite-field arithmetic) was implemented at first. Most of the field operations are implemented in separate modules, including their own control logic as well as the required memories. The higher hierarchy levels then only consist of a controlling unit, which is responsible for putting together the low level operations correctly, and some registers required for intermediate results of the appropriate algorithms.

Because the arithmetic in a prime field on the one hand is required for the underlying finite field of the elliptic curve within the prime design, and on the other hand for the modular operations on the ECDSA protocol level in both designs, a universally applicable prime-field arithmetic has been developed (*cf.* Section 6.6). For the design using $\mathbb{F}_{P192}$ as the underlying finite field, this prime-field arithmetic has been shared among the

[1]The final designs have been manufactured at the *LFoundry GmbH* in Landshut.

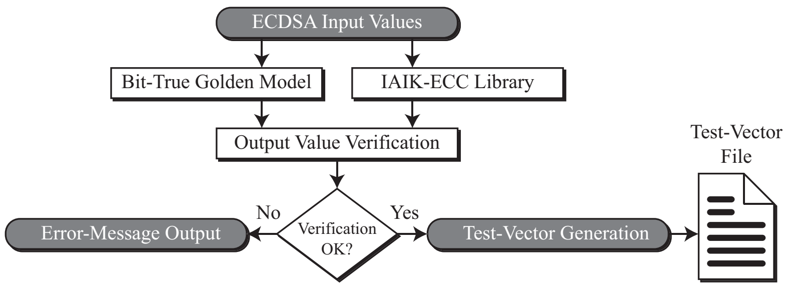


Figure 6.5.: Block diagram of the verification process of the bit-true golden model.

elliptic-curve operations and the ECDSA protocol-level operations, whereas the design based on $\mathbb{F}_{B163}$ only uses it for the ECC protocol-level operations. A design overview working on the full bit width based on $\mathbb{F}_{B163}$ and $\mathbb{F}_{P192}$, which is subdivided into the different hierarchy levels are illustrated in Figure 6.4a and Figure 6.4b, respectively.

6.3. Golden Model

The development of the golden model (also known as the *high-level model*) was the first step which was carried out after finishing the study on elliptic curves. Basically it should fulfill the following two requirements:

Bit-True Model: Because debugging a HDL model is not as convenient as debugging an implementation written in, for example MATLAB, Java or C++, a model which performs exactly the same bitwise operations as the later developed HDL model was targeted. With the use of this golden model, verification of the HDL model and debugging becomes much easier.

Test-Vector Generation: Beside the before mentioned requirement, the golden model was used to generate test vectors for both the verification of the HDL-model and the post-layout simulation.

The Institute for Applied Information Processing and Communications (IAIK) developed a *Java*-library[2] which provides well-known cryptography algorithms based on elliptic curves, including ECDSA. Using this library, the correctness of the bit-true high-level model could easily be verified as illustrated in Figure 6.5. For that reason and for the fact that *Java* provides the *BigInteger*[3] data type, *Java* as a programming language was chosen for the golden model.

For the implementation of the high-level model, the hierarchical structure of the elliptic-curve arithmetic, required for ECDSA, was maintained. A small example, illustrating the implementation in the high-level model is given in Listing 6.1. It shows

[2]`http://jce.iaik.tugraz.at/sic/Products/Core-Crypto-Toolkits/ECCelerate`

[3]The BigInteger data type provides arithmetic for integers of abitrary length, based on a two's complement representation. This is quite handy, especially when designing cryptosystems, where large integers play a major role.

the golden-model implementation of the point addition from Algorithm 5.6. Line number 7 to 13 show the finite-field operation calls of the underlying Galois field.

Listing 6.1: Golden-model implementation of the point addition used by the x-coordinate only scalar multiplication algorithm which is due to López and Dahab.

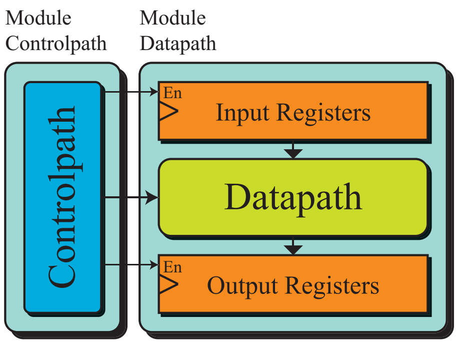
```
1  public MECPoint mAdd(BigInteger _x, BigInteger _x1, BigInteger _z1,
       BigInteger _x2, BigInteger _z2)
   {
3     BigInteger xAdd = null;
      BigInteger zAdd = null;
5     BigInteger t1   = null;

7     xAdd = gf.multiply(_x1, _z2);
      t1   = gf.multiply(_x2, _z1);
9     zAdd = gf.add(xAdd, t1);
      zAdd = gf.square(zAdd);
11    t1   = gf.multiply(xAdd, t1);
      xAdd = gf.multiply(_x, zAdd);
13    xAdd = gf.add(xAdd, t1);

15    return new MECPoint(xAdd, zAdd);
   }
```

6.4. HDL Model

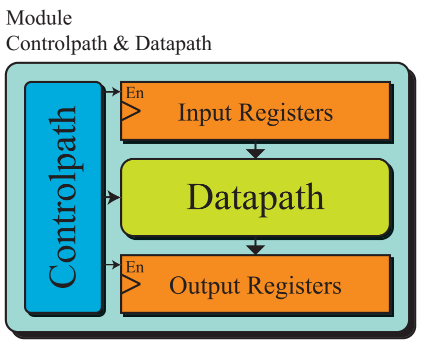
Basically, for the description of an integrated circuit a so-called Hardware Description Language (HDL) is required. The two most popular representatives for a HDL are Very High Speed Integrated Circuit Hardware Description Language (VHDL) and *Verilog*. Because the designs within this work have been carried out using the workflow at the *Integrated Systems Laboratory* of the ETH Zurich and their design flow, in general, is based on the use of VHDL, it has been chosen as the applied HDL.

As already mentioned, most of the operations within the ECDSA hierarchy (i.e. finite-field operations, elliptic-curve operations, etc.) have been implemented in separate modules. Therefore the whole design consists of a *"main module"* containing many submodules. In general, such a module can be subdivided into a datapath and a controlpath. The implementation of the controlpath can be done within a separate module as shown in Figure 6.6a. The approach which was chosen within this thesis, combines the datapath and the controlpath in a single module as illustrated in Figure 6.6b. The main advantage of the second approach compared to the first one is that it does not require passing all the control signals from the control logic to the datapath and therefore the portlists (i.e. the interface of the module) become much simpler.

The implementation of the control logic can be done either in a *hardwired way*, or based on a set of available instructions, controlled by a microprogram which can, for example, be stored in a Read Only Memory (ROM). The first control paradigm uses so-called Finite State Machines (FSMs) to control the datapath flow, mostly using a



(a) Separated modules.



(b) Single module.

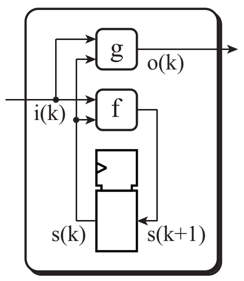
Figure 6.6.: Controlpath and datapath in separated modules and a single module, including input and output registers.

hierarchical structure to get around too large and confusing FSMs. According to [17] three different types of FSMs exist:

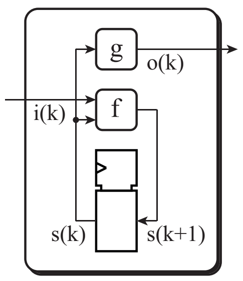
- Mealy machine
- Moore machine
- Medvedev machine

The three FSM types differ in the way they calculate the output from the input and from the internal state. Figure 6.7a to Figure 6.7c illustrate the data-dependency graphs of the three FSM types. A description of the parameters used in these figures is given in the following:

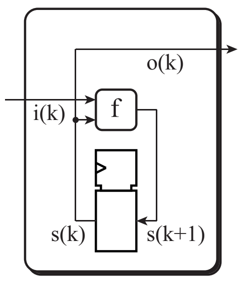
The comparison between Figure 6.7b and Figure 6.7c shows that Medvedev machines are a special subtype of Moore machines where the output is equal to the current state (i.e. there exists no output function at all). Mealy and Moore machines can be described according to Equation (6.1) and Equation (6.2), respectively:



(a) Mealy machine.



(b) Moore machine.



(c) Medvedev machine.

Figure 6.7.: Different types of FSMs.

$s(k)$	$\dots$ k-th state (i.e. the *current state*)
$s(k+1)$	$\dots$ $(k+1)$-th state (i.e. the *next state*)
$i(k)$	$\dots$ k-th input
$o(k)$	$\dots$ k-th output
f	$\dots$ transition or next-state function
g	$\dots$ output function

$$\begin{aligned} o(k) &= g(i(k), s(k)), \\ s(k+1) &= f(i(k), s(k)). \end{aligned} \tag{6.1}$$

$$\begin{aligned} o(k) &= g(s(k)), \\ s(k+1) &= f(i(k), s(k)). \end{aligned} \tag{6.2}$$

Because the output of the Moore machine only depends on the current state, no unregistered output values exist within this type of FSM, i.e. multiple Moore machines can be connected in series without increasing the critical path. This is a major advantage of Moore machines and hence it is, in general, recommended in the literature to use this type of FSM. Furthermore it should be targeted to keep the output logic (i.e. $g(s(k))$) as simple as possible, such that not too much combinational logic becomes part of a subsequent transition function with regard to its longest path. All FSMs within this thesis are implemented according to the Moore paradigm, trying to keep the output logic as simple as possible.

6.5. $\mathbb{F}_{B163}$ Arithmetic Implementation

Two of the required finite-field operations in $\mathbb{F}_{B163}$, namely the addition (i.e. also the subtraction) and the squaring, are quite easy to implement and therefore there is no need to spend a lot of time developing sophisticated algorithms to determine their results.

As described in Section 4.1.1, addition and subtraction in $\mathbb{F}_{2^m}$ can be performed using a simple exclusive-or operation as illustrated in Figure 6.8.

Because the irreducible polynomial for $\mathbb{F}_{B163}$ is fixed according to the NIST B-163 standard, the squaring operation was implemented using the idea from Section 4.1.3. In order to get the correct coefficients for the squared polynomial, a universally applicable polynomial division was implemented in the golden model using the B-163 reduction polynomial, resulting in a quite cheap exclusive-or network. Most of the output bits of the squaring operation (i.e. the bits of the squared and reduced polynomial) only depend on two or three of the input bits. Just a few of the output bits depend on more than three input bits, but none requires more than five. A sketch of the implementation of the squaring operation in $\mathbb{F}_{B163}$ is shown in Figure 6.9.

Because the implementation of these two operations is fully combinational, both can be performed within a single clock cycle. Furthermore they only require cheap exclusive-or standard cells, what keeps their area within a reasonable amount. Table 6.1 summarizes their area (obtained after synthesis) and timing properties.

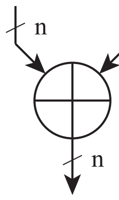


Figure 6.8.: Addition in $\mathbb{F}_{B163}$.

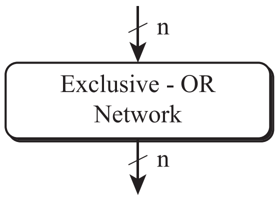


Figure 6.9.: Squaring in $\mathbb{F}_{B163}$.

6.5.1. Multiplication

Throughout Section 4.1.2 a multiplication algorithm in $\mathbb{F}_{2^m}$ with interleaved reduction was presented. An adaption of the described algorithm, which is suitable for a hardware implementation is due to Beth *et al.* [5] and is given in Algorithm 6.1. The bit string

Algorithm 6.1 MSB first multiplier for $\mathbb{F}_{2^m}$.

```
Input: $a = (a_{m-1}, \dots, a_1, a_0), b = (b_{m-1}, \dots, b_1, b_0) \in \mathbb{F}_{2^m}$, irreducible polynomial
    $f(x) = z^m + r(x)$.
Output: $c(x) = a(x) \cdot b(x) \mod f(x)$.
1: $c \leftarrow 0$.
2: for $i = m - 1$ downto $0$ do
3:     $c \leftarrow (c << 1) + c_{m-1} r$.
4:     if $b_i = 1$ then
5:         $c \leftarrow c + a$.
6:     end if
7: end for
8: return $c$.
```

containing the binary coefficients of $r(x)$ in $\mathbb{F}_{B163}$ is equal to $(0 \dots 0\,1\,1\,0\,0\,1\,0\,0\,1)$, and therefore the addition with $r(x)$ becomes a simple bitwise inversion at those four positions, where $r_i = 1$. Figure 6.10 shows a block diagram of the bit-serial multiplier in $\mathbb{F}_{B163}$.

As already mentioned before, the design for the point multiplication using an underlying binary field is based on projective coordinates. Because addition and squaring can be executed within a single clock cycle, the bit-serial multiplication, which requires 163 clock cycles, constitutes the major bottleneck throughout the elliptic-curve opera-

Table 6.1.: Area and timing properties of the addition and squaring operation in $\mathbb{F}_{B163}$.

Operation	**Timing**	**Area**			
		Comb.	Non-comb.	Entire	
Addition	Fully combinational	435 GE	-	435 GE	4 075 μm^2
Squaring	Fully combinational	521 GE	-	521 GE	4 881 μm^2

tions. Therefore the previously mentioned algorithm was adapted to process multiple bits within a single clock cycle by partly unrolling the loop within Algorithm 6.1. Figure 6.11 shows the area/time - tradeoff for the different implementations where both area and time values have been normalized to the bit-serial implementation. In spite of this improvement, the multiplication still requires definitely more clock cycles than the other field operations and therefore dominates the execution time of the scalar multiplication. For the final implementation of the multiplier in $\mathbb{F}_{B163}$ the digit-serial version processing 4 bits within a single clock cycle was used. Hence 41 cycles are required to determine the product of two 163 bit wide operands. The desired area of the 4-bit multiplier, obtained after synthesis, is 3802 GE.

6.5.2. Inversion

The implementation of the inversion in $\mathbb{F}_{B163}$ was done using Fermat's little theorem as described in Section 4.1.4. Therefore only the already described field operations are required, together with some temporary registers and a simple control unit. Including 10 field multiplications and 162 squarings, the inversion operation results in altogether 572 cycles and requires 5 420 GE. Because the inversion is only required once during the whole point-multiplication process (for the conversion from López - Dahab projective coordinates to affine coordinates), the fact that the inversion is quite time-consuming does not pose a major problem with regard to the timing.

6.6. $\mathbb{F}_{P192}$ Arithmetic Implementation

The observation of Algorithm 4.4 and Algorithm 4.7 shows that both require more or less the same operations to determine their results. Marcelo Kaihara and Naofumi Takagi [19] used this fact, to develop a design which uses a combined datapath for both the Montgomery multiplication and the division based on the extended GCD algorithm. A modified version of their design was implemented, which includes a modular addition and hence provides all the field operations in $\mathbb{F}_p$, required for the elliptic-curve operations as well as for the ECDSA protocol level. Because the field multiplication and

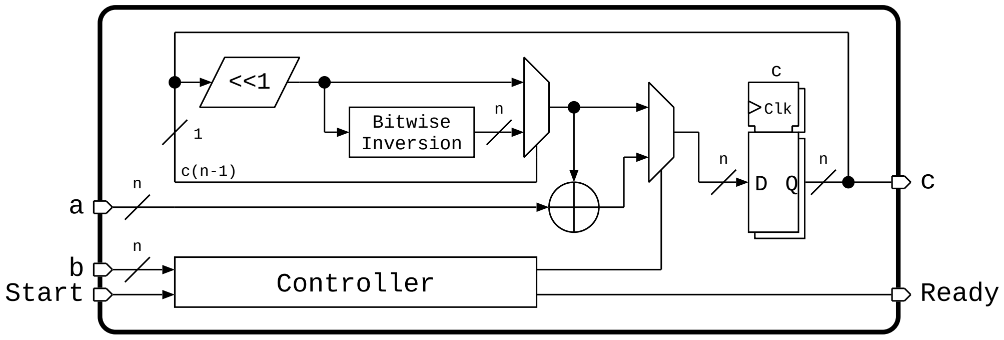


Figure 6.10.: Implementation of the multiplier in $\mathbb{F}_{B163}$.

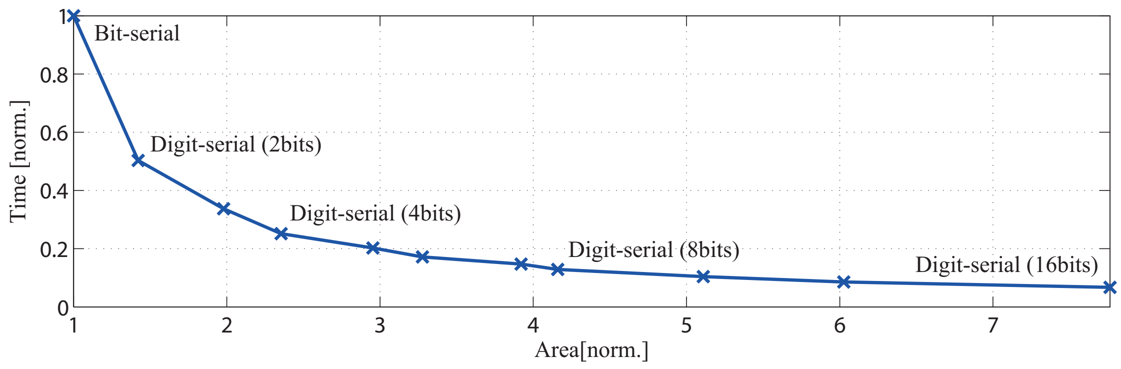


Figure 6.11.: Area/time - tradeoff for the multiplier in $\mathbb{F}_{B163}$.

the division, executed according to the aforementioned algorithms, can be determined using addition and shift operations only, the addition becomes the most important operation. In contrast to the arithmetic in $\mathbb{F}_{2^m}$, a full precision implementation of the arithmetic in $\mathbb{F}_p$ suffers from a major problem with regard to the addition, namely the carry-propagation problem. In order to process the 192-bit wide operands on their full bit width, special attention must be paid to it. To keep the maximal possible frequency of their design within a reasonable range, Kaihara and Takagi made use of a redundant binary representation which will be described in the following.

6.6.1. Redundant Binary-Number Representation

The two standard representations within binary-number systems are basically the one's and the two's complement. As soon as the width of the numbers being processed increases significantly, attention must be paid to the carry-propagation problem when adding or subtracting two numbers. Because there is the possibility that the carry has to be propagated from the Least Significant Bit (LSB) all the way up to the MSB, it is very likely that the maximal possible frequency for such a number arithmetic is determined by the addition/subtraction. In order to get around this problem, a redundant binary-number representation has been chosen.

Note: During the remainder of this work, the coefficients next to the radix of a binary number will be called *digits*, whereas a single binary *bit* can also just describe parts of a digit used within a redundant binary representation. When using a non-redundant representation, the expressions *digit* and *bit* can be used interchangeable.

In general, standard binary-number representation systems use a single bit to represent one digit of a binary number. The main idea of a redundant representation is, to use more than just a single bit to describe one digit, which makes it possible to describe a single binary number in many different ways. This redundancy allows one to perform the addition/subtraction without propagating the carry through all the digits.

The representation chosen within this work uses a Signed-Digit (SD) representation proposed by Avizienis [4]. It uses the digit set $\{-1, 0, 1\}$ and due to the radix being used, this representation is called a Signed-Digit Radix-2 (SD2) representation. Throughout the remainder of this work, -1 will be denoted by $\bar{1}$. An n-digit SD2 integer X can be represented using a bit string of the form

$$x = (x_{n-1}\, x_{n-2}\, \ldots\, x_2\, x_1\, x_0), \quad x_i \in \{\bar{1}, 0, 1\},$$

which represents the value

$$X = \sum_{i=0}^{n-1} x_i \cdot 2^i.$$

Hence it is almost equal to a standard unsigned integer except that the coefficients x_i can be $\bar{1}$ too. In contrast to the standard one's and two's complement representations, where the sign is depicted using the MSB of a number, the SD2 representation does not require a special sign bit.

This redundant representation enables one to represent a single number using many different ways. The following example shows this by displaying the decimal value 3 using four different 4-digit SD2 representations.

$$\begin{aligned}
(0011)_{SD2} &= 2 + 1 = 3\\
(010\bar{1})_{SD2} &= 4 - 1 = 3\\
(1\bar{1}0\bar{1})_{SD2} &= 8 - 4 - 1 = 3\\
(1\bar{1}\bar{1}1)_{SD2} &= 8 - 4 - 2 + 1 = 3
\end{aligned}$$

The only integer which is uniquely represented using SD2 digits is 0. The major benefit of the SD2 representation in contrast to the Standard-Binary (SB) representation is that this redundancy allows one to perform an addition where the carry is only propagated one digit to the left. Hence, using the SD2 representation eliminates the carry-propagation chains at the expense of the redundant representation of the operands. This does not say that addition using SD2 is more efficient than using SB in general, but as soon as the width of the integers increases, it is quite obvious that the carry-propagation problem, which is only present when using SB, represents the major bottleneck with regard to the critical path.

Redundant Binary Encoding

Basically, there are several ways to represent the digit set $\{\bar{1}, 0, 1\}$ of the SD2 representation using two bits. The encoding shown in Table 6.2 has been chosen due to the following advantages:

- The bits x^H and x^L, representing one SD2 digit, can be interchanged without altering the actual value of the SD2 number.
- In order to negate a SD2 number only the signs of all non zero digits have to be changed. This can be done by inverting the bits x^H and x^L individually.
- The SB representation is one of the valid SD2 representations and can easily be achieved by setting all x_i^L bits to 1 and the x_i^H bits to the actual SB value.

Table 6.2.: Encoding for the SD2 digits using 2 bits.

$\mathbf{x^H}$	$\mathbf{x^L}$	**SD2 Digit**
0	0	$\bar{1}$
0	1	0
1	0	0
1	1	1

Conversion between Redundant and Non-Redundant Representation

As already mentioned in the previous section, the conversion from SB to SD2 representation is not necessary, because the SB representation is also a valid SD2 representation where all x_i^L bits are set to 1. Nevertheless for the conversion vice versa a subtraction with carry propagation is required. This can of course be done at the very end of all required $\mathbb{F}_p$ operations and works as follows:

- Let x_{SD2} be an n-digit wide SD2 number.
- Let x_{pos} be an unsigned binary n-bit wide number, where all bits are set to 1 according to the positive bits in x_{SD2}.
- Let x_{neg} be an unsigned binary n-bit wide number, where all bits are set to 1 according to the negative bits in x_{SD2}.
- Compute the difference $x' = x_{pos} - x_{neg}$ resulting in the $n+1$-bit wide two's complement binary number, representing the same value as x_{SD2}.
- Because x' can also be negative, the modulus has to be added accordingly when working in a prime field $\mathbb{F}_p$ to stay within $[0, p-1]$.

Figure 6.12 shows the schematic of a circuit which performs this conversion, getting the x_i^H and x_i^L bits of an appropriate SD2 number as inputs. In order not to lower the maximal possible frequency because of this carry-propagating addition, its calculation has been split up such that it is performed during multiple cycles.

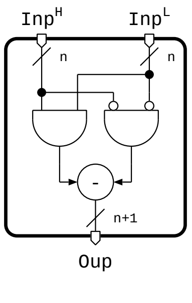


Figure 6.12.: Schematic for the conversion from SD2 to SB.

Table 6.3.: Addition rules for the carry-propagation-free addition, based on the SD2 representation.

Case	x_i	y_i	x_{i-1}, y_{i-1}	c_i	s_i
I	0	0	-	0	0
	1	$\bar{1}$		0	0
	$\bar{1}$	1		0	0
II	1	1	-	1	0
III	$\bar{1}$	$\bar{1}$	-	$\bar{1}$	0
IV	0	1	neither is $\bar{1}$	1	$\bar{1}$
	1	0	otherwise	0	1
V	0	$\bar{1}$	neither is $\bar{1}$	0	$\bar{1}$
	$\bar{1}$	0	otherwise	$\bar{1}$	1

Carry-Propagation-Free SD2 Addition

The carry-propagation-free addition (*cf.* for example [4] or [34]), for two operands x and y, consisting of the digits x_i and y_i, basically works in two successive steps. During the first step an intermediate sum digit s_i and an intermediate carry digit c_i are determined according to:

$$x_i + y_i = 2 \cdot c_i + s_i. \tag{6.3}$$

In the second step the intermediate sum digit s_i and the intermediate carry digit from the preceding position c_{i-1} are summed up to get the final sum digit z_i, i.e.

$$z_i = s_i + c_{i-1}. \tag{6.4}$$

In order to avoid generating a carry in the second step, the values of the intermediate sum digit s_i and the intermediate carry digit c_i have to be determined during the first step such that s_i and c_{i-1} both will never neither be 1 nor $\bar{1}$. This is possible because the SD2 numbers (01) and ($1\bar{1}$) both represent the value 1 and ($0\bar{1}$) and ($\bar{1}1$) both represent the value -1. Table 6.3 summarizes the addition rules for the SD2 arithmetic and the different cases are described in the following:

Case I : This is the simplest case, because when the operand digits $(x_i y_i) \in [(00), (1\bar{1}), (\bar{1}1)]$, the intermediate sum s_i and the intermediate carry c_i are both 0, independently of the preceding operand digits $(x_{i-1} y_{i-1})$.

Case II : When x_i and y_i are both 1, a positive carry is generated.

Case III : This is the counterpart to the previous case where both operand digits are negative and hence a negative carry is generated.

Case IV : The fourth case is the first *more interesting* one. Because when $(x_i y_i) \in [(01), (10)]$ the preceding operand digits $(x_{i-1} y_{i-1})$ have to be examined whether there is the possibility of a positive or negative carry. If there is the possibility of a positive carry, $(c_i s_i)$ have to be set to $(1\bar{1})$ to stop propagating the carry. If the possibility of a negative carry exists, $(c_i s_i)$ have to be set to (01). When neither a positive nor a negative carry is possible, then $(c_i s_i)$ can either be set to (01) or $(1\bar{1})$.

Case V : In the second interesting case, i.e. $(x_i y_i) \in [(0\bar{1}), (\bar{1}0)]$, carry propagation has to be avoided too by checking the previous operand digits. If in this case the possibility of a positive carry exists, $(c_i s_i)$ have to be set to $(0\bar{1})$. In order to avoid propagating the carry when the possibility of a negative carry exists, $(c_i s_i)$ have to be set to $(\bar{1}1)$. Once again setting $(c_i s_i)$ to $(0\bar{1})$ or $(\bar{1}1)$ does not matter when neither a positive nor a negative carry is possible.

Determining whether there is the possibility of a positive carry or not, as mentioned in the cases above, can be achieved by checking the preceding operand digits. If at least one of the previous digits is 1 and the other one is either 1 or 0, then the possibility of a positive carry exists. In contrast to this, to check if a negative carry might exist, at least one of the preceding digits has to be $\bar{1}$ and the other one must either be 0 or $\bar{1}$.

SD2 Full Adder

The first approach to implement the addition rules for two SD2 digits according to Table 6.3 was a simple look-up table. With that it was up to the synthesizer to generate a suitable logic for adding the SD2 digits. The resulting logic for a component adding two single SD2 digits, required 19 GE. Processing the $\mathbb{F}_{P192}$ operands at their full bit width, requires 193 of these addition cells. In order to reduce the required area of the SD2 full adder, another approach for the design of the *digit-adding-cell*, which is due to Kim *et al.* [21], was implemented. It turned out that this design only requires 15.7 GE after synthesis, i.e. 18% less than the previous one. The schematic of the cell is shown in Figure 6.13.

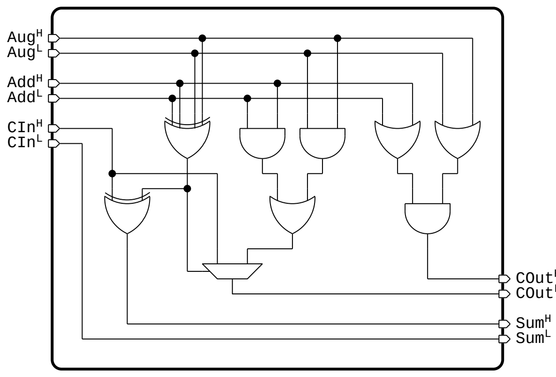


Figure 6.13.: A single cell adding two SD2 digits (`SD2FACell`).

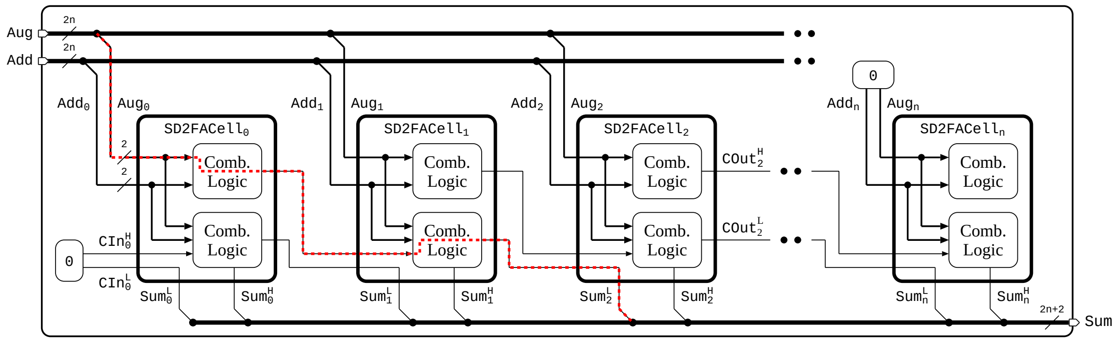


Figure 6.14.: The critical path of the SD2 full adder (SD2FA).

Beside the two operand digits $\mathtt{Aug_i}$ and $\mathtt{Add_i}$, which are both expressed using the bits $(\mathtt{Aug^H_i}, \mathtt{Aug^L_i})$ and $(\mathtt{Add^H_i}, \mathtt{Add^L_i})$ respectively, the adder cell gets a "carry digit" from the preceding cell to inform the current cell whether there is the possibility of a positive or negative carry. As one can see from Figure 6.14, the critical path of the SD2 full adder does not exceed the logic of two adder cells, independently of the width of the operands. Without any timing constraints, the design of the 192-digit SD2 full adder results in a critical path of 0.8 ns (*cf.* Listing 6.2), i.e. a maximal possible frequency of $f_{max} = 1.2$ GHz.

Listing 6.2: Maximal timing results after synthesis for the 192-digit full adder.

```
Point                                       Incr       Path
-------------------------------------------------------------
input external delay                        0.00       0.00 f
AddSD2xDI[383] (in)                         0.00       0.00 f
U1874/O (MOAI1S)                            0.17       0.17 f
U1873/O (MOAI1S)                            0.37       0.55 r
U2918/O (INV1S)                             0.10       0.65 f
U2919/O (MOAI1S)                            0.18       0.83 f
SumSD2xDO[384] (out)                        0.00       0.83 f
data arrival time                                      0.83
-------------------------------------------------------------
(Path is unconstrained)
```

6.6.2. SD2 Modular Addition

The modular addition using SD2 numbers is actually based on an idea by Takagi *et al.* [33], and was adapted for the present prime-field arithmetic. Let M be the n-bit wide modulus of the prime field satisfying $2^{n-1} < M < 2^n$, and let furthermore A and B be two $(n+1)$-digit wide SD2 numbers within $[-M, M]$. Then the modular addition,

$$C = A + B \mod M,$$

Table 6.4.: Operands of the modular addition using SD2 numbers including their appropriate intervals and digit values.

Value	**Interval**	**Width**	**Digits**							
			$n+2$	$n+1$	n	$n-1$	$n-2$	$\dots$	1	0
M	–	n	–	–	–	1	M_{n-2}	$\dots$	M_1	M_0
A	$[-M, M]$	$n+1$	–	–	A_n	A_{n-1}	A_{n-2}	$\dots$	A_1	A_0
B	$[-M, M]$	$n+1$	–	–	B_n	B_{n-1}	B_{n-2}	$\dots$	B_1	B_0
C'	$[-2M, 2M]$	$n+2$	–	C'_{n+1}	C'_n	C'_{n-1}	C'_{n-2}	$\dots$	C'_1	C'_0
C''	$[-M, M]$	$n+3$	C''_{n+2}	C''_{n+1}	C''_n	C''_{n-1}	C''_{n-2}	$\dots$	C''_1	C''_0
C	$[-M, M]$	$n+1$	–	–	C_n	C_{n-1}	C_{n-2}	$\dots$	C_1	C_0

can be performed using three steps. During the first step, a carry-propagation-free addition (*cf.* Section 6.6.1) is performed, resulting in the $(n+2)$-digit wide sum

$$C' = A + B, \quad C' \in [-2M, 2M].$$

In order to keep the result within the interval $[-M, M]$, the most significant two digits of C', i.e. $(C'_{n+2}\,C'_{n+1})$, are examined during the second step and 0, M or $-M$ is added to C' accordingly as their value is zero, negative or positive, i.e.:

$$C'' = \begin{cases} C' + 0, & (C'_{n+2}\,C'_{n+1}) = 0, \\ C' + M, & (C'_{n+2}\,C'_{n+1}) < 0, \\ C' - M, & (C'_{n+2}\,C'_{n+1}) > 0. \end{cases}$$

Using the standard SD2 full adder, this gives an $(n+3)$-digit wide intermediate sum, which is within $[-M, M]$. Due to the redundant representation, the most significant four digits have to be examined and rewritten (this process is denoted by *most significant digits correction*, i.e. `MSDCorr`) according to Equation (6.5) before C'' can be truncated to $(n+1)$ digits, i.e.

$$(C''_{n+3}\,C''_{n+2}\,C''_{n+1}\,C''_n) = \gamma = \begin{cases} (0\,0\,0\,1), & \gamma = 1, \\ (0\,0\,1\,0), & \gamma = 2, \\ (0\,0\,1\,1), & \gamma = 3, \\ (0\,0\,0\,\bar{1}), & \gamma = -1, \\ (0\,0\,\bar{1}\,0), & \gamma = -2, \\ (0\,0\,\bar{1}\,\bar{1}), & \gamma = -3. \end{cases} \tag{6.5}$$

Table 6.4 summarizes the operands and (intermediate) results of the modular addition, together with their intervals, widths and digits.

6.6.3. Combined Arithmetic

Now that all finite-field operations required for the elliptic-curve arithmetic (squaring within $\mathbb{F}_{P192}$ is performed using the multiplier with twice the same input) can be performed using SD2 numbers, the combined datapath based on the idea of Kaihara *et al.* [19] has been implemented. A major advantage of such a combined datapath is that for a potential adversary it makes it much more difficult to eavesdrop information by the use of side-channel attacks, because all operations use partly the same components of the datapath. The datapath of the architecture, combining the logic for the Montgomery multiplier, the field division based on the extended GCD algorithm and the modular addition described in the previous section, is shown in Figure 6.15. The appropriate algorithm performing the $\mathbb{F}_{P192}$ operations is listed in Algorithm 6.6.1. As already shown in Equation (4.21) and Equation (4.25), the required operations performing the modular halving and the modular quartering, denoted by `ModHalv` and `ModQuar`, can be determined using add and shift operations too.

Basically, the following algorithm is split up into four different steps. The *Init Step* and the *Output Step* are of importance for all modes of the algorithm, whereas the two steps in between, namely the *Run Step* and the *Correction Step*, are only of interest during the prime-field multiplication and the division. The time-consuming *Run Step* is designed, such that one iteration of it can be performed within a single clock cycle. The whole modular arithmetic requires 33 kGE. This is mainly due to the higher amount of memory, which is required because of the redundant representation and due to the combinational logic, working on the large digit width. Originally the algorithm by Kaihara *et al.*uses a *mixed radix-4/2 approach*, i.e. performing 2 digits during one clock cycle when possible and 1 digit otherwise. Because this would enable an adversary to eavesdrop information about the operands using a timing analysis, the algorithm was adopted to perform all operations within a constant time by inserting some dummy operations where necessary. `MultDummy` and `DivDummy` represent the dummy operations for the multiplication and the division, both requiring a single clock cycle. For the *Run Step* this results in a cycle count (for an n-digit wide number) equal to n for the multiplication and equal to $2n+5$ for the division.

Algorithm 6.6.1 Combined arithmetic for SD2 numbers in $\mathbb{F}_p$.

```
Input: $mode \in \{0,1,2,3,4\}$,
    prime modulus $M : 2^{n-1} < M < 2^n$,
    operands $X, Y : -M < X, Y < M$ ($Y \neq 0$ when $mode = 1$).
Output: $mode = 0 : Z = XY2^{-(n+2)} \mod M$, with $Z \in [-M, M]$,
    $mode = 1 : Z = X/Y \mod M$, with $Z \in [-M, M]$,
    $mode = 2 : Z = X + Y \mod M$, with $Z \in [-M, M]$,
    $mode = 3 : Z = X - Y \mod M$, with $Z \in [-M, M]$,
    $mode = 4 : Z = X + Y$, with $Z \in [-2M, 2M]$.
 1: ——— Init Step ———
 2: $A = Y, P = 2^{n+1}, D = 1, s = 1$.
 3: if $mode = 0$ then $B = \bar{1}, U = 0, V = X$.
 4: if $mode = 1$ then $B = M, U = X, V = 0$.
 5: if $mode = 2$ then $U = X + Y \mod M$.
```

```
6: if $mode = 3$ then $U = X - Y \mod M$.
7: if $mode = 0$ or $mode = 1$ then goto Run Step else goto Output step.
8: ——— Run Step ———
9: while $p_0 \neq 1$ do
10:    if $A \equiv 2 \mod 4$ then
11:       $A >> 1$, $U = \texttt{ModHalv(U, M)}$.
12:       if $s = 0$ then
13:          if $d_1 = 1$ then $s = 1$.
14:          $D >> 1$.
15:       else
16:          $D << 1$, $P >> 1$.
17:       end if
18:    else
19:       if $([a_1a_0] + [b_1b_0]) \mod 4 = 0$ then $q = 1$ else $q = -1$.
20:       if $mode = 0$ or $s = 0$ or $d_0 = 1$ then
21:          $A = (A + qB) >> 2$, $U = \texttt{ModQuar(U + qV, M)}$.
22:          if $s = 1$ then
23:             $D << 1$.
24:             if $mode = 0$ and $p_1 = 0$ then $P >> 2$, Compute MultDummy.
25:             else $P >> 1$, if $p_0 = 1$ then $s = 0$.
26:          else
27:             $D >> 1$, if $d_0 = 1$ then $s = 1$.
28:          end if
29:       else
30:          $\{A = (A + qB) >> 2, B = A\}$, $\{U = \texttt{ModQuar(U + qV, M)}, V = U\}$.
31:          if $d_1 = 0$ then $s = 0$.
32:          $D >> 1$.
33:       end if
34:    end if
35: end while
36: ——— Correction Step ———
37: if $mode = 0$ and $s = 1$ then
38:    $U = \texttt{ModHalv(U, M)}$.
39: else if $mode = 1$ then
40:    while $D \neq 0$ do
41:       $D >> 1$.
42:       Compute DivDummy.
43:    end while
44:    if $([b_1b_0] = 3$ or $[b_1b_0] = -1)$ then
45:       $V = -V$.
46:    end if
47: end if
48: ——— Output Step ———
49: if $mode = 1$ then $Z = V$.
50: else if $mode = 4$ then $Z = X + Y$.
51: else $Z = U$.
52: return $Z$.
```

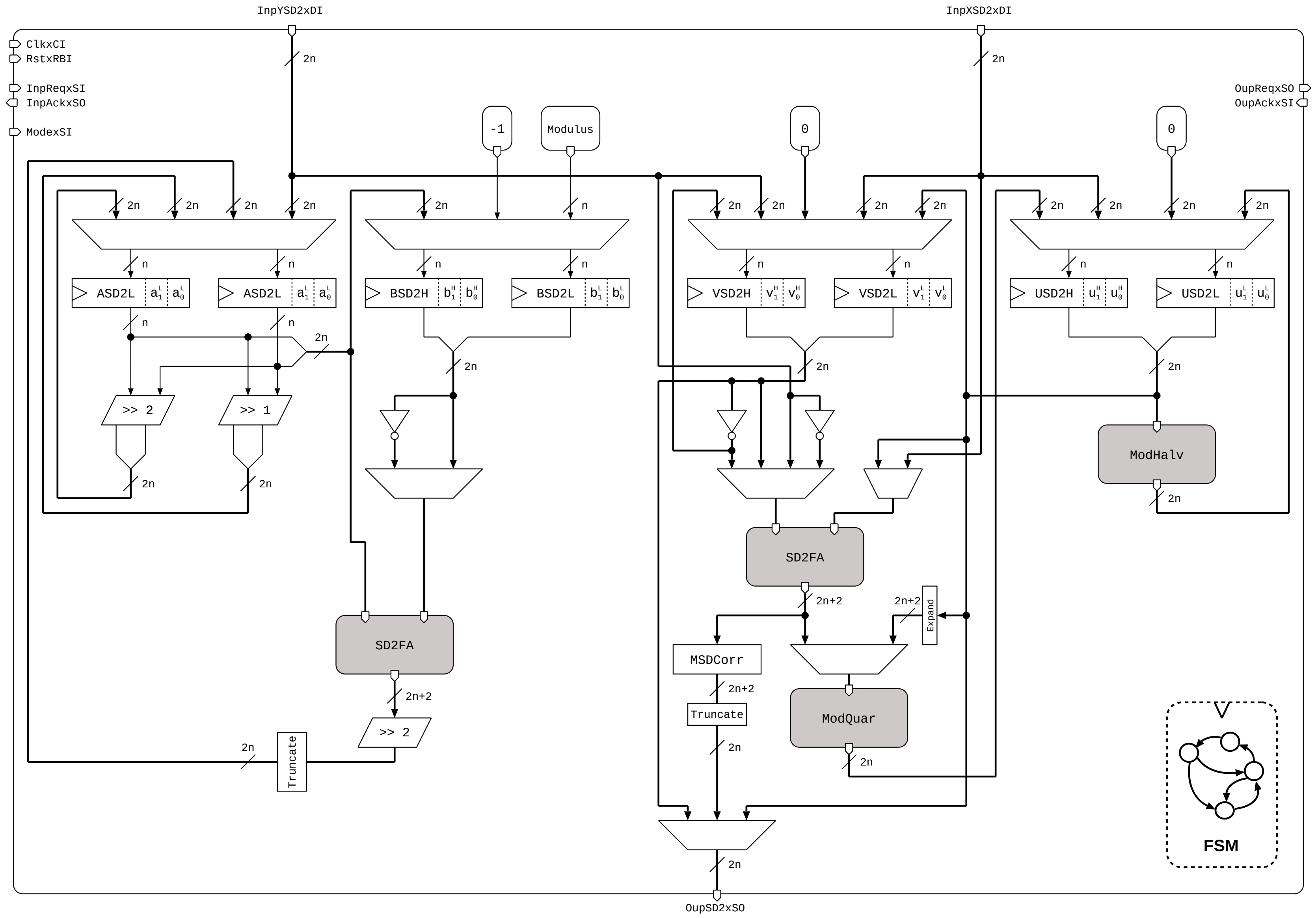


Figure 6.15.: Combined datapath for the prime-field operations based on SD2 numbers.

6.7. Higher-Level Operations

As already mentioned in Section 5.2.5, the point multiplication for the binary design was implemented using the projective version of the x-coordinate only algorithm which is due to López and Dahab (*cf.* Algorithm 5.6). Therefor five 163-bit wide registers are required to perform the elliptic-curve operations. Using the x-coordinate only paradigm for the multiple-point multiplication does not work in the same way like within the point-multiplication process[4], because the difference of the points, performing the point addition is not known (*cf.* Section 5.2.4). Hence, the other point coordinates have to be processed too. Making use of projective coordinates would circumvent the time-consuming field inversion in $\mathbb{F}_{2^m}$, but requires further memory for the additional point coordinates[5]. Due to the fact that using affine coordinates during the multiple-point multiplication does not need any further registers for its main algorithm in comparison to performing the scalar multiplication based on the x-coordinate only method using projective coordinates (the registers storing the z-coordinates during the point multiplication are used to store the y-coordinates throughout the multiple-point multiplication), affine coordinates have been chosen for the calculation of the multiple-point multiplication within the binary-field based design. Nevertheless, two additional registers are required for the multiple-point multiplication to store a temporary point at the beginning of the algorithm (*cf.* Algorithm 5.9). In the prime-field based design, the implementation of the field division only requires about twice the runtime of the multiplication and hence, the use of projective coordinates can not really be justified. Therefore for the P-192 design affine coordinates have been chosen for both the point multiplication and the multiple-point multiplication.

6.8. AMBA APB - Interface

The ECDSA designs implemented in this work operate on relatively wide numbers (i.e. 163-bit and 192-bit values, respectively). Hence, it goes without saying that for writing inputs to and reading outputs from the chip an adequate interface is required. One state of the art protocol family, especially for SoC designs, is the family of the Advanced Microcontroller Bus Architecture (AMBA) protocols [3]. The AMBA Advanced Peripheral Bus (APB) interface is one representative of this family and has been chosen for the communication with the chip within this work. The APB protocol is a two phase transaction protocol, taking at least two clock cycles. It is a protocol with a quite low complexity where signal transitions occur during a rising clock edge. Furthermore, it is compatible with the following protocols within the AMBA family:

- AMBA Advanced High-performance Bus (AHB)
- AMBA Advanced High-performance Bus Lite (AHB-Lite)

[4]Finding a way to apply the x-coordinate only formulae to the multiple-point multiplication would be a nice breakthrough.

[5]Using projective coordinates with a common-z coordinate would lower the additional memory to just a single 163-bit wide register.

Table 6.5.: AMBA APB Signal Definitions.

Signal	Direction	Width	Description
PClk	IN	1	Clock signal
PResetn	IN	1	Reset signal; Active LOW
PSel	IN	1	Select signal; Indicates that for the specified APB unit a data transfer is required.
PEnable	IN	1	Enable signal; Starts the second phase of the communication and indicates that the data is available.
PWrite	IN	1	Write/Read signal. Indicates whether a write or read operation takes place. (HIGH... Write, LOW... Read)
PAddr	IN	7	Address signal. Specifies the address to which should be written or from which should be read respectively.
PWData	IN	8	Data write signal. Holds the data which should be written to the APB unit.
PRData	OUT	8	Data read signal. Holds the data which should be read from the APB unit.

- AMBA Advanced Extensible Interface (AXI)
- AMBA Advanced Extensible Interface Lite (AXI4-Lite)

The required signals for the AMBA APB interface are given in Table 6.5. The two main phases of the protocol are listed in the following:

Setup Phase: During this phase, a valid address is assigned. Furthermore, the APB device has to be selected and the operation type (i.e. writing or reading) has to be defined.

Enable Phase: Within the second stage, the APB device gets enabled and the data will be written to or read from it.

All in all, a maximum of 32 pins were available for the communication between the chip and its environment. Using a data bus width of 8 bits for both, the writing and reading bus, together with 7 address bits, 3 control bits, the clock and the reset signal, this results in 28 required bits overall.

Timings

The timing for the write access is shown in Figure 6.16. During the setup cycle (between T2 and T3) the address of the data and the data itself get assigned. Furthermore, the control signals PSel and PWrite indicate that a write access will follow. The following cycle represents the enable cycle, during which the PEnable signal is set to HIGH indicating that certain data should be written to the assigned address.

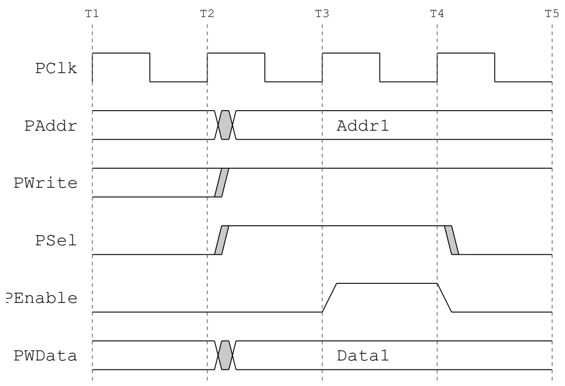


Figure 6.16.: AMBA APB - Write [3].

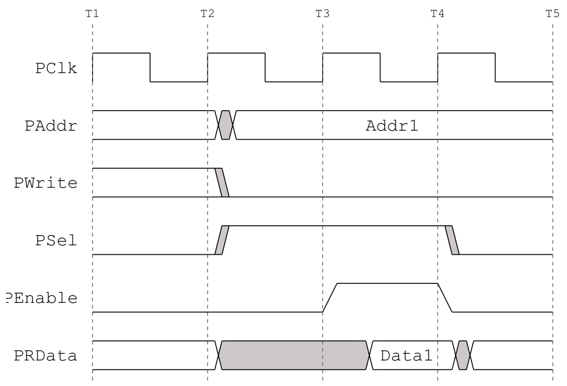


Figure 6.17.: AMBA APB - Read [3].

In contrast to the write access, the APB unit has to ensure that during the read access the data, which has been requested during the setup cycle (between T2 and T3), is available while the enable cycle (between T3 and T4). The timing for the read access is displayed in Figure 6.17.

I/O Addresses

The addresses required for writing data via the AMBA interface into the chip are listed in Table 6.6. Those for reading data from the chip can be seen in Table 6.7. All data is given using *little-endian* format (i.e. numeric significance increases with increasing addresses).

Control Byte

The inputs for the calculations as well as their outputs differ with regard to the specified mode of operation (i.e. signature generation and signature verification). In order to define the mode of operation and to keep control over the calculation a *Control Byte* has been implemented which can be accessed at address 120. Furthermore, this byte contains the state-bit defining whether verification of a signature was successfully or

Table 6.6.: Inputs and Writing Addresses.

Value		Prime		Binary	
Sign	Verify	Width	Addresses	Width	Addresses
Private Key	Public Key (x)	192	0 . . . 23	163	0 . . . 20
Message Digest	Message Digest	192	24 . . . 47	163	21 . . . 41
Ephemeral Key	Public Key (y)	192	48 . . . 71	163	42 . . . 62
-	Signature (r)	192	72 . . . 95	163	63 . . . 83
-	Signature (s)	192	96 . . . 119	163	84 . . . 104

Table 6.7.: Outputs and Reading Addresses.

Value		Prime		Binary	
Sign	Verify	Width	Addresses	Width	Addresses
Signature (r)	-	192	0...23	163	0...20
Signature (s)	-	192	24...47	163	21...41

not. Currently, only the lowest four bits of this *Control Byte* are used. Table 6.8 describes the bits of the *Control Byte*.

6.9. Verification of the HDL Model

In order to verify the correctness of the HDL model, for most of the design units file-based test benches have been implemented. The test benches, purely developed in VHDL, read the test inputs together with the expected outputs from a single file, which has been created by the golden model before. Afterwards they provide a set of required input parameters for the current Device Under Test (DUT) and compare the actual outputs with the expected ones. A summary of the results of this comparison is then written in a report file. In comparison to other test bench approaches (e.g. using Tcl[6]-scripts to provide the test bench with sample inputs), the one applied in here is known to be much faster and hence very suitable when executing thousands of test runs. Figure 6.18 shows a sketch of the HDL model verification process.

The simulation of the VHDL modules as well as the post-layout simulation have been performed using *ModelSim*[7] by *Mentor Graphics*.

Table 6.8.: Control Byte - Bit Explanation.

Bit	Name	Description
x_0	Mode	Defines which mode of operation will be used. (LOW... Signature Generation, HIGH... Signature Verification)
x_1	Start	When set to HIGH the calculation starts.
x_2	Ready	As soon as the calculation finishes this bit will become HIGH.
x_3	State	During *verification mode*, this bit will define whether the signature has been accepted or not (LOW... Signature rejected, HIGH... Signature accepted)

[6]Tcl is a scripting language and many digital logic simulator manufacturer use it to provide an interface to communicate with the HDL implementations.

[7]`http://model.com/`

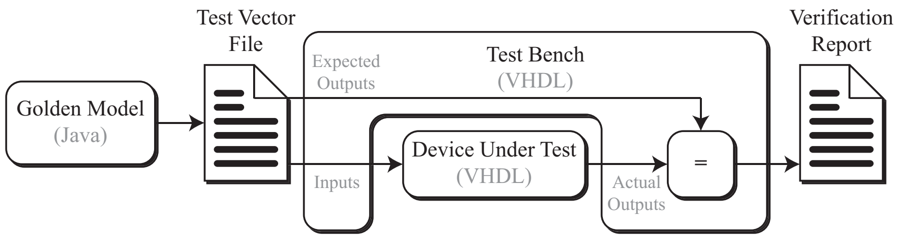


Figure 6.18.: Block diagram of the verification process of the HDL-Model.

6.10. Backend Design

For both ECDSA implementations, the backend design has been performed. Because the setup for the target technology (150 nm @ LFoundry) was not yet finished during the development of the designs, the workflow was done using the 180 nm technology by United Microelectronics Corporation (UMC) at first. As soon as the design flow for the actual target technology finished, the gained build scripts have been altered from the UMC to the LFoundry technology.

Basically, throughout the backend design the gate-level netlist, gained from synthesis[8] was used in order to create the layouts of the two chips. Therefor, roughly the following design steps have been accomplished:

Floorplanning: Physical properties like the width and the height of the chips are defined during this step. Moreover, the placement of potentially available macro-cells (which is not the case within the present designs) is done herein.

Power Planning: In order to have a uniformly distributed power consumption on the chips, a well-conceived placement of the power wires is required. This includes the definition of a power ring as well as a suitable number of power stripes.

Standard-Cell Placement: As soon as the power planning has finished, the standard cells can be placed. This is in fact an important step, because afterwards the utilization of the core area can be determined quite precisely.

Routing: One of the latter steps during the backend design is represented by the routing process. After that, the final netlist is more or less finished and can be exported for the post-layout simulation.

The backend design, in general, is a very iterative process, i.e. many steps have to be performed multiple times to obtain the desired results. Once the final netlist has been generated, it can be used to perform the post-layout simulation by applying the test benches, generated for the HDL model simulation.

[8]To receive the gate-level netlist from the HDL model, the DesignCompiler tool by Synopsys has been used.

Chapter 7

Results

Throughout this chapter, the results of the design implementations will be presented. We carried out two ASIC designs, which are able to perform a digital signature generation as well as a signature verification according to ECDSA. The required elliptic-curve operations for the first design are accomplished ontop of the binary field $\mathbb{F}_{B163}$, whereas the second design uses the prime field $\mathbb{F}_{P192}$ as the underlying finite field. The results contain information about the required resources with regard to area, timing, power and the maximal possible clock frequency. Furthermore, the results of the binary-field based design and the prime-field based design will be compared.

At the beginning of the chapter, the area and timing requirements will be considered for the different ECDSA hierarchy levels, starting with the finite-field arithmetics and then transfering over to the elliptic-curve operations and the ECDSA protocol-level operations. Next, some information about the power and energy consumption as well as the maximal possible clock frequency will be given. The chapter will close with a screenshot of the final layout of both chips.

7.1. Area and Timing Results

Although the synthesizer being used within this thesis[1] gives the results of the area analysis in μm^2, they have been transformed to Gate Equivalents (GEs) in order to be comparable to designs, implemented using different technologies. With regard to the runtime, the results of the implemented algorithms are given in required clock cycles.

Note: Although the final chips have been implementend using the 150 nm technology by the *LFoundry GmbH*[2], the area and timing results listed throughout this section are obtained by the use of the 180 nm technology by *UMC*[3]. That is because the workflow

[1]For the synthesis of the VHDL model the *Design Compiler* tool by Synopsys has been used.
[2]`http://www.lfoundry.com/`
[3]`http://www.umc.com/`

Table 7.1.: Area/timing requirements for the operations in $\mathbb{F}_{B163}$.

Operation	**Cycles**	**Area**			
		Comb.	Non-comb.	Entire	
Addition	1	435 GE	-	435 GE	4 075 μm^2
Squaring	1	521 GE	-	521 GE	4 881 μm^2
Multiplication	41	2 827 GE	975 GE	3 802 GE	35 642 μm^2
Inversion	572	2 552 GE	2 868 GE	5 420 GE	50 806 μm^2

for the LFoundry technology at the Integrated Systems Laboratory[4] was not yet set up when running these tests.

7.1.1. Finite-Field Arithmetic

Because the finite-field operations form the lowest hierarchy level of the ECDSA structure, they are considered at first. In order to get an overview of the different modules, implementing the required field operations in $\mathbb{F}_{B163}$, Table 7.1 summarizes their area and timing requirements. Because addition and squaring are implemented only by the use of exclusive-or cells, both operations can be performed within a single clock cycle and do not require any sequential logic at all. Figure 7.1 shows the area distribution of the field operations in $\mathbb{F}_{B163}$. It illustrates that the multiplication and the inversion take most of the area. This is due to the fact that in contrast to the addition and the squaring, the multiplication and the inversion require both combinational and sequential logic as well as a more complex control logic.

As described in Section 6.6.3, all required operations in $\mathbb{F}_{P192}$ use mostly the same datapath. Therefore the area requirements for this combined arithmetic are given in Table 7.2 and illustrated in Figure 7.2. Due to the redundant binary representation, which was used in order to circumvent the carry-propagation problem during the addition of the 192-bit wide operands, the SD2 arithmetic requires quite a lot memory. Also, because of the large values being processed, the combinational logic takes much more area in the $\mathbb{F}_{P192}$ arithmetic than within the $\mathbb{F}_{B163}$ arithmetic.

The modular addition as well as the standard addition using SD2 values can be performed within a single clock cycle. Multiplication and division in $\mathbb{F}_{P192}$ require 194 and 391 clock cycles, respectively.

7.1.2. ECC Arithmetic

Because the elliptic-curve operations have been implemented using two different underlying finite fields, also the ECC results are separated with regard to them.

[4]`http://www.iis.ee.ethz.ch/`

Table 7.2.: Area distribution of the combined SD2 arithmetic working in $\mathbb{F}_{P192}$.

Component	**Volume**	**Area**			
		Comb.	Non-comb.	Entire	
ModHalv	1	2 131 GE	-	2 131 GE	19 980 μm^2
ModQuar	1	4 979 GE	-	4 979 GE	46 676 μm^2
SD2FA	2	5 910 GE	-	5 910 GE	55 405 μm^2
Top level	-	15 412 GE	11 368 GE	26 780 GE	251 046 μm^2
Entire				39 800 GE	373 107 μm^2

Table 7.3.: Area requirements for the ECC arithmetic over $\mathbb{F}_{B163}$.

	Area			
Component	Comb.	Non-comb.	Entire	
Finite-field operations	6 335 GE	3 843 GE	10 178 GE	95 412 μm^2
Top level	17 686 GE	6 718 GE	24 404 GE	228 778 μm^2
Entire	24 021 GE	10 561 GE	34 582 GE	324 185 μm^2

Based on $\mathbb{F}_{B163}$

The ECC working on $\mathbb{F}_{B163}$ as the underlying finite field uses projective coordinates for the representation of the elliptic-curve points when performing the point multiplication. It is based on the x-coordinate only algorithm described in Section 5.2.4 and hence does not need to store the y-coordinate at all. Altogether this algorithm requires five 163-bit wide registers to calculate the x-coordinate of the resulting point of the scalar multiplication. The required area for the whole ECC working on $\mathbb{F}_{B163}$ as the underlying finite field is given in Table 7.3 and the area distribution is illustrated in Figure 7.3.

Due to the fact, that the x-coordinate only algorithm can only be applied under certain conditions (*cf.* Section 5.2.4), which are not the case when performing the multiple-point multiplication during the ECDSA signature-verification process, the control logic can also perform the elliptic-curve operations (i.e. the point doubling and the point

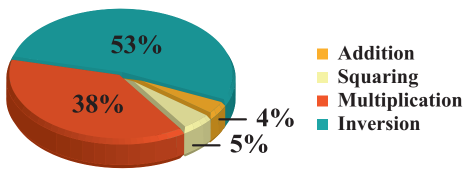


Figure 7.1.: Area distribution of the finite-field operations in $\mathbb{F}_{B163}$.

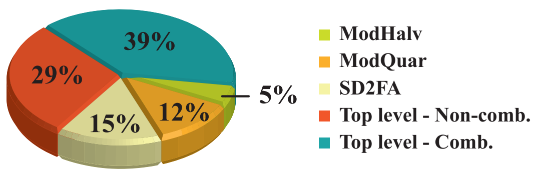


Figure 7.2.: Area distribution of the SD2 arithmetic.

Table 7.4.: Timing requirements for the point addition and the point doubling over $\mathbb{F}_{B163}$ using LD projective coordinates and affine coordinates.

	Projective Coordinates				Affine Coordinates			
	ECDouble$_{\text{proj}}$		**ECAdd$_{\text{proj}}$**		**ECDouble$_{\text{aff}}$**		**ECAdd$_{\text{aff}}$**	
Operation	Volume(#)	Cycles	#	Cycles	#	Cycles	#	Cycles
Addition	1	1	2	2	5	5	9	9
Squaring	4	4	1	1	2	2	1	1
Multiplication	2	82	4	164	2	82	2	82
Inversion	-	-	-	-	1	572	1	572
Entire		87		167		661		664

addition) using affine coordinates. Therefor the registers which hold the z-coordinates during the point multiplication are used to hold the y-coordinates when performing the multiple-point multiplication. Hence exactly the same number of registers are required for the point addition and the point doubling using projective coordinates and affine coordinates. Table 7.4 summarizes the required finite-field operations based on the two different point-representation types. The overall runtimes for the point multiplication and the multiple-point multiplication are listed in Table 7.5. Because the multiple-point multiplication is not performed within a constant runtime, only the maximal required clock cycles are given.

Based on $\mathbb{F}_{P192}$

Working with the SD2 arithmetic allows to perform the elliptic-curve operations at full precision at the expense of a higher memory consumption. Because the conversion from SD2 to SB requires an addition with carry propagation (*cf.* Section 6.6.1), converting the operands into the standard-binary representation after each prime-field operation would eliminate the advantage gained from the redundant representation. Therefore the SD2 representation was maintained during the whole ECDSA calculation.

Using projective coordinates instead of affine coordinates is recommended, because in general the inversion in a prime field is much more time-consuming than the other field operations. Because for the implemented prime-field arithmetic the field division only requires approximately twice the runtime of the multiplication, there would be

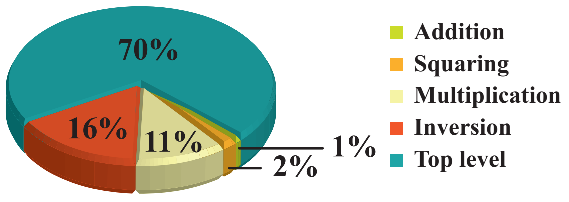


Figure 7.3.: Area distribution of the ECC arithmetic over $\mathbb{F}_{B163}$.

Table 7.5.: Timings of the point multiplication and the multiple-point multiplication and the required operations based on $\mathbb{F}_{B163}$.

Point Multiplication			**Multiple-Point Multiplication**		
Operation	Volume	Cycles	Operation	Volume	Cycles
Addition[a]	1	1		-	-
Squaring[a]	2	2		-	-
Multiplication[b]	1	41		-	-
Inversion[b]	1	572		-	-
ECDouble$_{\text{proj}}$	162	14 094	ECDouble$_{\text{aff}}$	163	107 743
ECAdd$_{\text{proj}}$	162	27 054	ECAdd$_{\text{aff}}$	$\leq$ 164	$\leq$ 108 896
Entire		41 764	Entire		$\leq$ 216 639

[a]Required during the first point doubling when the MSB is set.
[b]Required for the conversion from projective to affine coordinates.

Table 7.6.: Required field operations and the appropriate timings for the elliptic-curve arithmetic over $\mathbb{F}_{P192}$. Field operations: D = division, M = multiplication, S = squaring, A = addition.

	Required Field Operations	Cycles
ECDouble	1D, 4M, 2S, 8A	1563
ECAdd	1D, 4M, 1S, 6A	1367

no advantage by the use of projective coordinates (*cf.* Section 3.6.2). Hence, affine coordinates have been used for the point representation of the elliptic-curve points.

The required field operations for the point addition and the point doubling and their appropriate runtimes are listed in Table 7.6. The multiplications with the constants required for the point doubling (*cf.* Equation (3.14)) are performed in terms of two consecutive field additions. The multiplication and the addition are done within the Montgomery domain. Because the division works on standard integers, three additional field multiplications are required to transform the operands of the division from the Montgomery domain into the standard domain and the quotient vice versa.

The timing results for the point multiplication according to Algorithm 5.5 and the multiple-point multiplication based on $\mathbb{F}_{P192}$ are given in Table 7.7. The runtime of the multiple-point multiplication depends on its input values but requires at most a single point addition more than the scalar multiplication[5]. The area consumption of the ECC arithmetic including the distribution with regard to the finite-field arithmetic and the top level is illustrated in Figure 7.4. Although there is no additional logic in the top

[5]Further improvements can be made to the multiple-point multiplication using the Joint Sparse Forms (JSFs) to represent the scalars. We refer to [10] for detailed information.

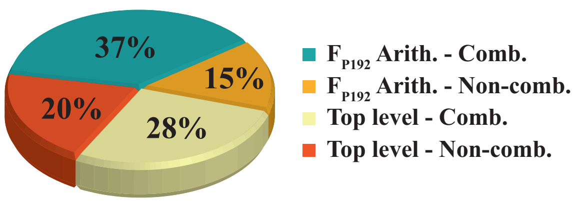
Table 7.7.: Runtime requirements for the point multiplication and the multiple-point multiplication over $\mathbb{F}_{P192}$.

	Required Operations	Cycles
Point Multiplication	192 ECDouble, 192 ECAdd	562 560
Multiple-Point Multiplication	192 ECDouble, $\leq$ 193 ECAdd	$\leq$ 563 927

level than the selection logic as well as some temporarily registers, it requires a lot of area due to the large signal widths.

Figure 7.4.: Area consumption of the ECC arithmetic over $\mathbb{F}_{P192}$.

Component	Area
$\mathbb{F}_{P192}$ arithmetic comb.	28 432 GE
$\mathbb{F}_{P192}$ arithmetic non-comb.	11 368 GE
Top level comb.	22 190 GE
Top level non-comb.	15 828 GE
Entire	77 818 GE



7.1.3. ECC Runtimes as a Function of Finite-Field Operations

The dependence of the runtimes of the elliptic-curve operations with regard to the finite-field operations is of great interest. Therefore Figure 7.5 shows the runtimes of the elliptic-curve operations, required for the signature generation and the verification according to ECDSA with regard to the needed finite-field operations. Their overall runtime has been normalized to one.

Because of the use of projective coordinates in the point multiplication, the runtime of the signature generation using $\mathbb{F}_{B163}$ as the underlying finite field is determined heavily by the field multiplication. When using affine coordinates, as within the signature-verification process of the binary-field based design, the field inversion dominates the runtime of the required elliptic-curve operation (i.e. the multiple-point multiplication). Because the division in the prime-field based design only needs about twice the runtime of the multiplication, both field multiplication and division determine the runtime of the elliptic-curve operations, although affine coordinates have been used.

7.1.4. ECDSA Protocol Level

Beside the ECC arithmetic based on the finite field $\mathbb{F}_{B163}$, the ECDSA signature generation and signature-verification process requires some prime-field operations at the protocol level. Therefore the SD2 prime-field arithmetic working on 163-bit wide operands was used to enable performing the entire digital signature algorithm. Due to the fact that the prime-field arithmetic based on SD2 numbers does not imply any special kind of modulus, it has been shared among the ECC operations and the ECDSA protocol-level

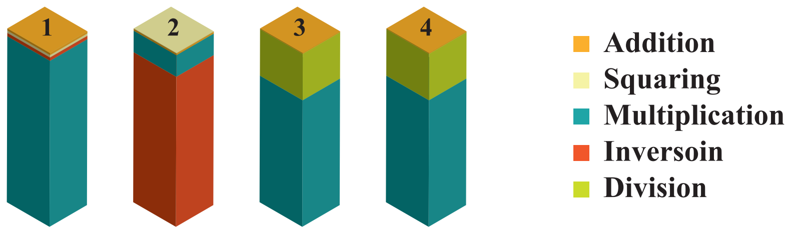


Figure 7.5.: Required runtime for the different finite-field operations during the elliptic-curve arithmetic (The sum of all operations are normalized to one). 1 - Point multiplication (*PM*) using López Dahab projective coordinates based on $\mathbb{F}_{B163}$, 2 - Multiple point multiplication (*MPM*) using affine coordinates ($\mathcal{A}$) based on $\mathbb{F}_{B163}$, 3 - *PM* using $\mathcal{A}$ based on $\mathbb{F}_{P192}$, 4 - *MPM* using $\mathcal{A}$ based on $\mathbb{F}_{P192}$.

operations within the prime design. For the conversion between SD2 and SB numbers a module, implementing the idea mentioned in Section 6.6.1, has been designed. Furthermore an AMBA APB interface was designed, which enables the chips to communicate with their environment using an eight bit wide data signal (further information about the interface can be found in Section 6.8) and holds the 163/192bit input and output values during the entire calculation. The overall timing and area properties of both ECDSA designs are given in Table 7.8 and Table 7.9, respectively.

7.2. Power and Energy Consumption

In order to get an estimation of the power and energy consumption of the two designs, the EDA tools of the design flow at the *Integrated Systems Laboratory* have been used. The first results of the power estimation were gained by setting a single global toggle-activity to all internal nodes of the circuit. Due to the fact that this assumption is not correct and should only be used for very rough estimations, another approach using a so-called Value Change Dump (VCD) file was carried out in order to get a more precise power estimation.

A VCD file contains the toggle information for all signals during a simulation run and therefore predicts the power consumption of a design much more correctly than setting a global toggle-activity. Table 7.10 summarizes the power consumption for both designs based on the two aforementioned power-estimation variants working on a frequency of

Table 7.8.: Timing requirements for the ECDSA designs based on $\mathbb{F}_{B163}$ and $\mathbb{F}_{P192}$.

	Based on $\mathbb{F}_{B163}$	Based on $\mathbb{F}_{P192}$
Signature Generation	42 264 cycles	563 147 cycles
Signature Verification	$\leq$ 217 306 cycles	$\leq$ 564 710 cycles

Table 7.9.: Area requirements for the ECDSA designs based on $\mathbb{F}_{B163}$ and $\mathbb{F}_{P192}$.

Component	Based on $\mathbb{F}_{B163}$		Based on $\mathbb{F}_{P192}$	
ECC arithmetic	34 582 GE	18 %	37 093 GE	36 %
Modular arithmetic	33 227 GE	16 %	37 309 GE	36 %
SD2-to-SB converter	2 051 GE	2 %	2 568 GE	2 %
ECDSA top level	12 818 GE	14 %	15 653 GE	15 %
AMBA interface	9 250 GE	10 %	11 350 GE	11 %
Entire	91 928 GE	100 %	103 973 GE	100 %

Table 7.10.: Power consumption of the ECDSA designs ($f = 100$ MHz, $U_{DD} = 1.8$ V).

	Binary Design	Prime Design
Global Toggle-Activity	92 mW	113 mW
VCD File Based	43 mW	61 mW

100 MHz and a supply voltage of 1.8 V. Due to the fact that both designs use a *full precision* arithmetic, they consume more power than *multi-precision* designs found in the literature. Especially the binary design has a very short runtime for the signature-generation process (approx. 42 kcycles), hence the required energy, i.e. 18 μJ, is quite low. The signature generation using $\mathbb{F}_{P192}$ as the underlying finite field requires about 340 μJ.

7.3. Critical Path

Because the critical path runs through the prime-field arithmetic and the two designs make use of it, both have the same critical path in common, i.e. approx. 10 ns. This results in a maximal possible frequency of 100 MHz. In contrast to the prime-field arithmetic the critical path of the binary-field arithmetic is equal to 4 ns, i.e. it is possible to clock it with up to 250 MHz.

7.4. Layout

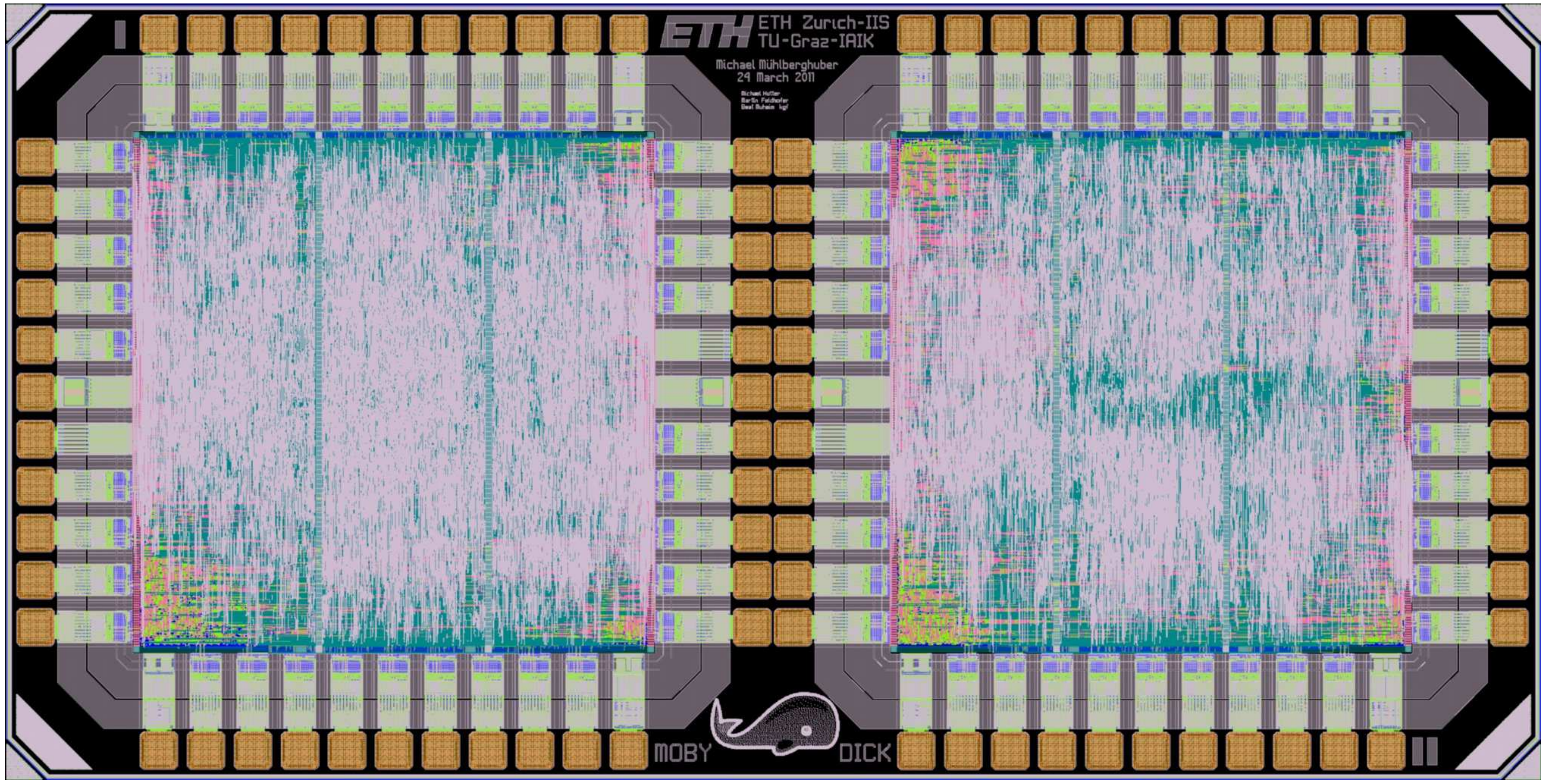


Figure 7.6.: Layout - Left: Prime-field based design, Right: Binary-field based design.

Chapter 8

Conclusion

Comparing different ECC implementations is, in general, very difficult, because there are so many hierarchy levels (i.e. finite-field arithmetic, elliptic-curve operations, ECC protocol level) where developers can vary their designs. Although the basic trend of ECC implementations during the last years is towards multi-precision designs, we decided to implement two full precision ECDSA designs, one based on a binary field and one based on a prime field. The reason therefore laid in the fact that on the one hand binary-field operations are well suited for hardware designs and can therefore also be implemented at full precision without exceeding a reasonable range with regard to the required resources. On the other hand a generally applicable prime-field arithmetic was targeted which was then used for both, the elliptic-curve operations based on the prime field and the ECDSA protocol-level operations in both designs. Furthermore, the encapsulation of the particular finite-field operations in separated modules allows altering them easily without touching any other parts of the designs.

The domain parameters, specifying the two elliptic curves as well as the underlying finite field for the binary and the prime design, have been taken from a standard, published by the NIST, namely the B-163 and the P-192, respectively. All field operations in both designs have been implemented with an interleaved reduction, hence neither a subsequent reduction step has to be performed, nor the (large) intermediate results have to be stored.

In order to create two fairly comparable designs, we decided to implement the prime-field arithmetic at full precision too. Using a standard binary representation for the 192-bit wide operands would have increased the critical path due to the field addition significantly. Hence we decided to implement the prime-field operations using a redundant binary representation. This redundancy allows a carry-free addition at the expense of doubling the required memory for storing the operands. Beside the redundant representation of the operands in the prime design, the combinational logic for selecting the appropriate operands at the full bit width needs a decisive amount of area.

Summarizing we can say that we provide the first ASIC designs of ECDSA, one working on top of a binary field and the other one working on top of a prime field, which have been implemented by the same designers using the equivalent design workflow and EDA tools. Therefore comparisons between the different designs are more suitable than comparing designs from different designers using various work flows and technologies. Furthermore the two designs offer both a signature generation and a signature-verification mode. Basically, binary-field operations are much easier to implement in hardware than prime-field operations, hence as long as there is a choice we suggest using $\mathbb{F}_{2^m}$ as the underlying finite field of the elliptic-curve operations. When it comes to an ECC protocol where modular operations are necessary (e.g. ECDSA), using a prime field as the underlying finite field based on a universally applicable field arithmetic might become justified. That is because the respective hardware is needed either way and might be shared. Nevertheless, the prime fields used for the elliptic-curve arithmetic are often of a special type (i.e. a special modulus), such that reduction becomes easier and therefore the modular operations at the lowest hierarchy level and at the ECC protocol level might differ.

In comparison to multi-precision implementations, the full precision designs presented herein require more resources with regard to area and power. This was expected due to the wide datapaths. But when it comes to a runtime analysis, especially the binary design performs the signature generation quite efficiently, i.e. in about 42 kcycles and can even be lowered by optimizing the field multiplication in $\mathbb{F}_{B163}$.

8.1. Future Work

Some possible tasks to improve the binary and the prime design presented in this work are given in the following.

8.1.1. Simultaneous $\mathbb{F}_{2^m}$ Operations

Addition and squaring in the binary field are implemented completely combinational. Because their critical path is quite narrow and these operations are often required in a subsequent way during the elliptic-curve arithmetic, they could be performed within a single clock cycle easily. Moreover, both operations might be appended in the last, or prefixed in the first cycle of the multiplication such that further cycles can be saved.

8.1.2. X-Coordinate Only Multiple-Point Multiplication

A very promising improvement with regard to the runtime would be to apply the x-coordinate only approach presented in Section 5.2.4 to the multiple-point multiplication. Therefore the difference of the two points, applying the point addition and the point doubling, has to be known during the elliptic-curve operations. We are not sure yet if this is even possible, but this would lower the number of required clock cycles for the signature-verification process in the ECDSA significantly.

8.1.3. Multiple Clock Domains

Both designs are implemented using a single clock domain only. Because the critical path within the prime-field arithmetic is more than twice as long as the critical path within the ECC arithmetic based on $\mathbb{F}_{B163}$, inserting another clock domain would speed up the binary design significantly.

8.1.4. Simplified SD2 Addition

The addition of two SD2 numbers is one of the major operation in the prime-field arithmetic. Beside the standard addition of two arbitrary SD2 numbers, adding the modulus to and subtracting it from any SD2 number is required quite often. Because the negative of an integer, which is given in the standard representation, can be expressed with SD2 digits using only a single $\bar{1}$, addition/subtraction of the modulus can be achieved in a simpler way than using the standard SD2 adder. For detailed information on the simpler addition which would reduce the required area, see for example [33].

8.1.5. Montgomery Inverse

Due to the fact that the field division in $\mathbb{F}_{P192}$ according to the extended binary GCD algorithm (*cf.* Section 4.2.4) requires an input value given in the standard domain, but the field multiplication operates on values given in the Montgomery domain, three additional multiplications are needed for each division to convert the operands between the different domains. Using the Montgomery inversion (for further information see for example [11]) circumvents these multiplications by the use of operands within the Montgomery domain. Nevertheless it has to be verified first, whether the Montgomery inversion can be implemented as efficient as the division using the extended binary GCD algorithm.

8.2. Outlook

Full precision ECC designs are more or less only competitive with multi-precision implementations when using a binary field as the underlying finite field. What might be of interest is a mixed design, including a full precision ECC arithmetic working on a binary field and a multi-precision modular arithmetic which performs the protocol-level operations. This would combine the runtime benefits with regard to the binary field and the elliptic-curve operations and the low area requirements for the rarely used modular arithmetic. Nevertheless, an ECC design working on the full bit width, will never require as less power as a multi-precision implementation. But when runtime or energy consumption play a major role in the requirements of a design, full precision might be worth consideration.

Appendix A

Algorithms

A.1. Extended Euclidean Algorithm

Algorithm A.1, taken from [11], illustrates the extended Euclidean algorithm which determines x, y and $\gcd(a, b)$ satisfying $ax + by = \gcd(a, b)$. A modification of this

Algorithm A.1 Extended Euclidean algorithm.

```
Input: $a, b \in \mathbb{N}^*$, $a \leq b$.
Output: $d = \gcd(a, b)$, $x$ and $y$ satisfying $ax + by = \gcd(a, b)$.
1: $u = a$, $v = b$.
2: $x_1 = 1$, $y_1 = 0$, $x_2 = 0$, $y_2 = 1$.
3: while $u \neq 0$ do
4:     $q = \lfloor v/u \rfloor$, $r = v - qu$, $x = x_2 - qx_1$, $y = y_2 - qy_1$.
5:     $v = u$, $u = r$, $x_2 = x_1$, $x_1 = x$, $y_2 = y_1$, $y_1 = y$.
6: end while
7: $d = v$, $x = x_2$, $y = y_2$.
8: return $(d, x, y)$.
```

algorithm, which is presented in Section 4.2.4, can be used to determine the inverse of a field element $a^{-1} \in \mathbb{F}_p$ and $b = p$, because during the last iteration of the loop where u is non-zero, the integers u, x_1 and y_1 satisfy $ax_1 + py_1 = 1$ (i.e. $ax_1 \equiv 1 \mod p$ and hence $a^{-1} \equiv x_1 \mod p$).

A.2. Binary GCD Algorithm

The main idea of the binary gcd algorithm is to replace the required division in the extended Euclidean algorithm with cheaper operations, namely subtractions and shift-operations. Algorithm A.2 shows this process [11].

The algorithm, given in Listing A.3 is an extension to the previous one and calculates the inverse of an element in $\mathbb{F}_p$ without the requirement of the computationally expensive field divisions [11].

Algorithm A.2 Binary GCD algorithm.

Input: $a, b \in \mathbb{N}^*$.

Output: $\gcd(a, b)$.

```
1: $u = a$, $v = b$, $e = 1$.
2: while $u$ and $v$ are even do
3:    $u >> 1$, $v >> 1$, $e << 1$.
4: end while
5: while $u \neq 0$ do
6:    while $u$ is even do
7:       $u >> 1$.
8:    end while
9:    while $v$ is even do
10:      $v >> 1$.
11:   end while
12:   if $u \geq v$ then
13:      $u = u - v$.
14:   else
15:      $v = v - u$.
16:   end if
17: end while
18: return $e \cdot v$.
```

Algorithm A.3 Inversion in $\mathbb{F}_p$ using the binary GCD algorithm.

```
Input: Prime $p$, $a \in [1, p-1]$.
Output: $a^{-1} \mod p$.
u = a, v = p.
$x_1 = 1$, $x_2 = 0$.
while $u \neq 1$ and $v \neq 1$ do
    while $u$ is even do
        $u = u/2$.
        if $x_1$ is even then
            $x_1 = x_1/2$.
        else
            $x_1 = (x_1 + p)/2$.
        end if
    end while
    while $v$ is even do
        $v = v/2$.
        if $x_2$ is even then
            $x_2 = x_2/2$.
        else
            $x_2 = (x_2 + p)/2$.
        end if
    end while
    if $u \geq v$ then
        $u = u - v$, $x_1 = x_1 - x_2$.
    else
        $v = v - u$, $x_2 = x_2 - x_1$.
    end if
end while
if $u = 1$ then
    return $x_1 \mod p$.
else
    return $x_2 \mod p$.
end if
```

Appendix B

Pinout and Pin Description

Both designs have been wrapped into a QFN56 package, where only 44 pins have been used. Because both chips use the identical interface protocol, only a single pinout and the appropriate pin descriptions are given in Figure B.1 and Table B.1, respectively. For further information about the applied interface protocol, see Section 6.8.

Table B.1.: Pin Descriptions.

Pin Name	Pin Numbers	Direction	Description
PClkxCI	44	IN	Clock signal.
PResetnxRBI	45	IN	Active-low asynchronous reset.
PWDataxDI	2-6, 9-11	IN	Data write bus.
PRDataxDO	13, 16-20, 23, 24	OUT	Data read bus.
PAddrxDI	25, 26, 30-34	IN	Address bus of the.
PSelxSI	37	IN	Select signal of the AMBA interface.
PEnablexSI	38	IN	Enable signal of the AMBA interface.
PWritexSI	39	IN	Mode signal of the AMBA interface.
ECDSAStartxSO	40	OUT	Start signal of the ECDSA algorithm. Becomes 1, when the actual ECDSA calculation begins.
ScanEnxTI	46	IN	Scan enable pin. Enables the scan inputs of the scan registers.
ScanInxTI	57, 48, 51	IN	Scan inputs.
ScanOutxTO	52, 53, 54	OUT	Scan outputs.

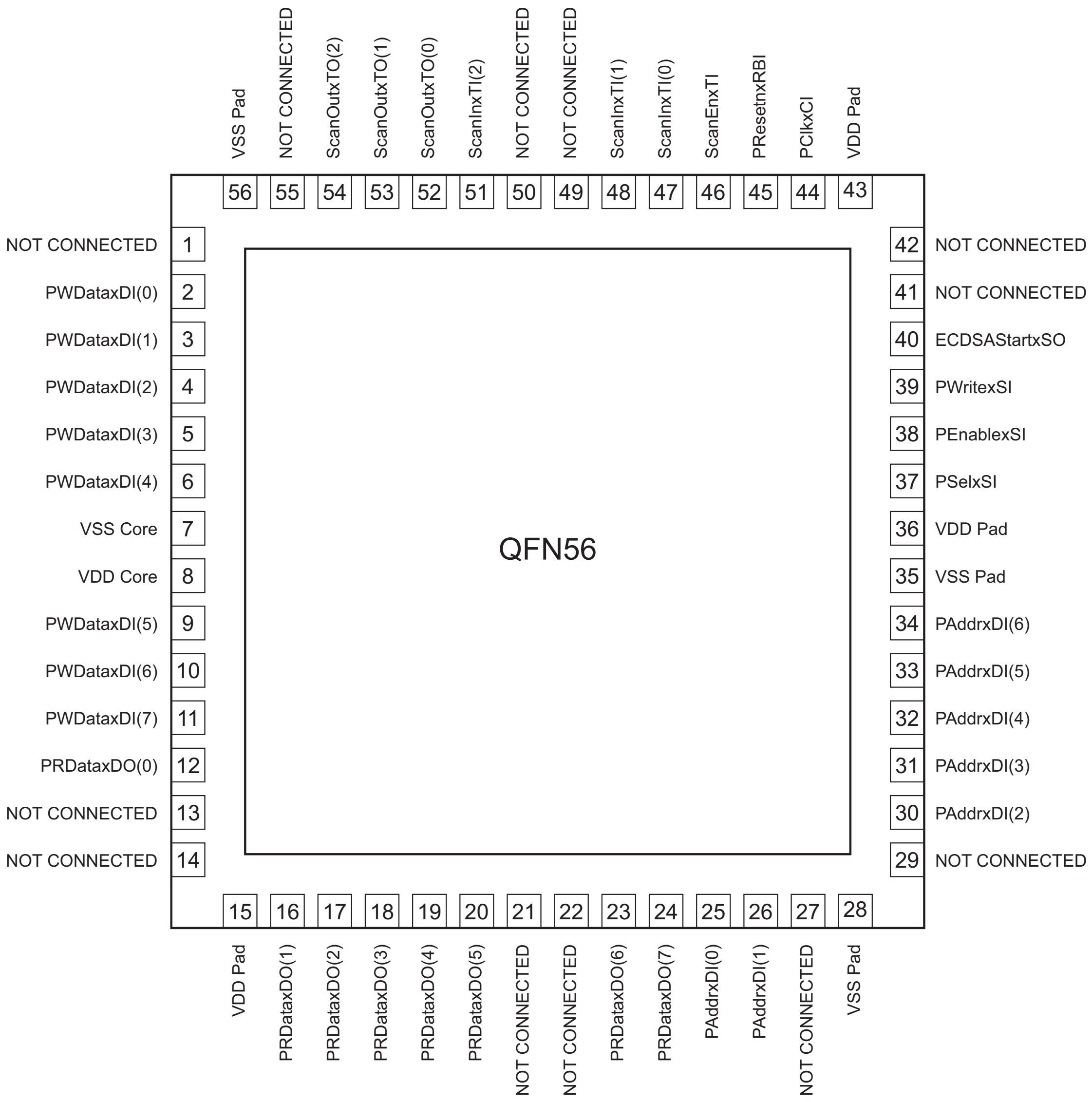


Figure B.1.: Pinout.

Acronyms

AHB	Advanced High-performance Bus
AHB-Lite	Advanced High-performance Bus Lite
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASIC	Application Specific Integrated Circuit
AXI	Advanced Extensible Interface
AXI4-Lite	Advanced Extensible Interface Lite
DL	Discrete Logarithm
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
DUT	Device Under Test
ECC	Elliptic-Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
EDA	Electronic Design Automation
FSM	Finite State Machine
GE	Gate Equivalent
HDL	Hardware Description Language
IAIK	Institute for Applied Information Processing and Communications
IFP	Integer-Factorization Problem

JSF	Joint Sparse Form
LSB	Least Significant Bit
MAC	Multiply Accumulate
MSB	Most Significant Bit
NIST	National Institute for Standards and Technology
RAM	Random Access Memory
RNG	Random Number Generator
ROM	Read Only Memory
SB	Standard-Binary
SD	Signed-Digit
SD2	Signed-Digit Radix-2
SoC	System on Chip
SPA	Simple Power Analysis
UMC	United Microelectronics Corporation
VCD	Value Change Dump
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Bibliography

[1] Gordon B. Agnew, Ronald C. Mullin, and Scott A. Vanstone. An Implementation of Elliptic Curve Cryptosystems Over F2155. *IEEE Journal on Selected Areas in Communications*, 11:804–813, 1993.

[2] ANSI. *ANSI X9.62-2005 - Public Key Cryptography for the Financial Service Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).*

[3] ARM. AMBA Open Specifications. Website, 2011. `http://www.arm.com/products/system-ip/amba/amba-open-specifications.php`.

[4] Algirdas Avizienis. Signed-Digit Number Representation for Fast Parallel Arithmetic. *IRE Transactions on Electronic Computers*, 1961.

[5] Thomas Beth and Dieter Gollmann. Algorithm Engineering for Public Key Algorithms. *IEEE Journal on Selected Areas in Communications*, 7:458–465, May 1989.

[6] Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. Technical report, Ecole Normale Supérieure, 1999.

[7] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[8] Taher Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[9] Çetin Kaya Koc and Tolga Acar. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16:26–33, 1996.

[10] Johann Großschädl, Alexander Szekely, and Stefan Tillich. The energy cost of cryptographic key establishment in wireless sensor networks. In *Computer and Communications Security*, pages 380–382, 2007.

[11] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag New York, Inc., 2004.

[12] Michael Hutter, Marcel Medwed, Daniel Hein, and Johannes Wolkerstorfer. Attacking ECDSA-Enabled RFID Devices. Technical report, Institute for Applied Information Processing and Communications (IAIK), 2009.

[13] IEEE Std 1363-2000. *Standard Specifications for Public-Key Cryptography.* New York, 2000.

[14] Tetsuya Izu, Bodo Möller, and Tsuyoshi Takagi. Improved Elliptic Curve Multiplication Methods Resistant against Side Channel Attacks. Technical report, Fujitsu Laboratories Ltd., 2002.

[15] Marc Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. CHES, pages 135–147, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] Aleksandar Jurisic and Alfred Menezes. Elliptic Curves and Cryptography. Technical report, Certicom Corp., 2005.

[17] Hubert Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication.* Cambridge University Press, New York, NY, USA, 1st edition, 2008.

[18] Marcelo E. Kaihara and Naofumi Takagi. A VLSI Algorithm for Modular Multiplication/Division. *IEEE Symposium on Computer Arithmetic*, 16, 2003.

[19] Marcelo E. Kaihara and Naofumi Takagi. A Hardware Algorithm for Modular Multiplication/Division. *IEEE Transactions on Computers*, 54(1):12–21, 2005.

[20] Marcelo E. Kaihara and Naofumi Takagi. Bipartite Modular Multiplication Method. *IEEE Transactions on Computers*, 57:157–164, 2008.

[21] Yun Kim, Bang-Sup Song, J. Grosspietsch, and S. F. Gillig. A carry-free 54bx54b multiplier using equivalent bit conversion algorithm. *IEEE Journal of Solid-state Circuits*, 36:1538–1545, 2001.

[22] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[23] Yong Ki Lee and Ingrid Verbauwhede. A Compact Architecture for Montgomery Elliptic Curve Scalar Multiplication Processor.

[24] Arjen K. Lenstra and Eric R. Verheul. Selecting Cryptographic Key Sizes. *Journal of Cryptology*, 14:255–293, 1999.

[25] Julio López and Ricardo Dahab. Fast Multiplication on Elliptic Curves over GF(2m) without Precomputation. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, pages 316–327, London, UK, 1999. Springer-Verlag.

[26] Alfred Menezes, Scott Vanstone, and Paul Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[27] Victor S. Miller. Use of elliptic curves in cryptography. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[28] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44:519–519, 1985.

[29] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. Technical report, American Mathematical Society, 1987.

[30] NIST. *Digital Signature Standard (DSS) (FIPS PUB 186-3).* National Institute of Standards and Technology, 2009.

[31] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[32] Ernst Straus. Addition Chains of Vectors. *American Mathematical Monthly*, 71:806–808, 1964.

[33] Naofumi Takagi and Shuzo Yajima. Modular multiplication hardware algorithms with a redundant representation and their application to rsa cryptosystem. *IEEE Trans. Comput.*, 41:887–891, July 1992.

[34] Naofumi Takagi, Hiroto Yasuura, and Shuzo Yajima. High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree. *IEEE Transactions on Computers*, 34, September 1985.

[35] Annette Werner. *Elliptische Kurven in der Kryptographie.* Springer-Verlag Berling Heidelberg, 2002.