Brandenburgische Technische Universität Cottbus

Fakultät Informatik

Evaluation of Design Alternatives for Flexible Elliptic Curve Hardware Accelerators

Diplomarbeit
zur Erlangung des akademischen Grades
Diplom-Informatiker

Bearbeiter:	Steffen Peter
	Matrikelnummer: 9802986
Gutachter:	Prof. Dr.-Ing. Rolf Kraemer
	Prof. Dr.-Ing. Heinrich Theodor Vierhaus
Betreuer:	Dr. Peter Langendörfer
Bearbeitungszeit:	28.08.2005 - 22.02.2006

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Frankfurt(Oder), den 22. Februar 2006

Contents

1. Introduction

Amid growing demand for mobile communication and business over networks such as the Internet, data protection and authentication are an inevitable need. With the rapid migration to wireless solutions, eavesdropping becomes ever easier and the security issues become more urgent. Cryptographic approaches have properties that provide the fundamentals for secure communication.

For a long time most cryptographic applications had been performed by symmetric key methods. Symmetric key cryptography entails all cryptographic methods that perform the decryption step with the same key as the encryption. The cryptographic algorithms are secure and fast, but they do not provide answers for important issues concerning communication over insecure channels:

- How to send an encrypted message when there is no secure way of key exchange?
- How to authenticate the origin of a digital message to a third party, an equivalent to the handwritten signature?

In 1976 Whitfield Diffie and Martin Hellman introduced an asymmetric key approach [3], which is known as public key cryptography. Hereby, the encryption and decryption steps are performed with different keys. A message that was encrypted with one key can only decrypted by a different associated key. When one key is kept secret, which is the private key, and the other one, the public key, is published openly, there are a lot of applications that can satisfy the security needs. Public key cryptography allows everyone to encrypt messages with a public key that only the person with the private key can decrypt - an idea that solves the key exchange problem. The signature issue is solved when only the owner of the private key is able to encrypt a message that everyone can decrypt with a corresponding public key.

A public key method that has become the most popular public key approach is the RSA algorithm [42], which was presented in 1978 by and named after the three developers Ron Rivest, Adi Shamir and Len Adleman. Seven years later, Neal Koblitz[22] and Victor Miller[30] independently proposed elliptic curve cryptography (ECC) as a specific algorithm for public key cryptography. Indeed, ECC achieves the same level of cryptographic strength as RSA by much shorter key lengths. These shorter key lengths improve the feasibility of public key

cryptography in mobile devices substantially, since they not only imply less bandwidth usage but also a reduction of memory and processing efforts. This is important since all secure public key approaches are connected with a very high operating expense.
Despite the short key lengths and the corresponding less computing efforts, the calculation of an ECC operation is still extremely extensive and involved. Especially on low powered mobile devices such as sensor networks or on RFID chips, the application of ECC is still an issue. Software implementations that are running on such systems require several seconds and hence a huge amount of energy for a single ECC operation.
In the last years a number of hardware accelerators for ECC operations have been proposed that improve the performance and the reduce energy consumption by orders of magnitude. Hereby, the emphasis was mostly placed on fast but tailored designs for a selected elliptic curve. The limitation to a single elliptic curve may be problematic, particularly with regard to ASICs, which are supposed to be used for years. Changed protocols and standards may cause the need for adaptations of the used curve as can also happen due to uncovered vulnerabilities in the cryptoanalysis. This is why in this work not only efficient hardware designs for a single curve implementation are investigated but also designs that can realize flexible ECC accelerators.

The primary goal of this diploma thesis is therefore the investigation of approaches for flexible and modular architectures of ECC hardware accelerators. This is preceded by extensive studies of existing methods to find suitable techniques and to identify potential problem areas. The theoretical considerations should result in efficient algorithms for the required operations. Special emphasis will be placed on investigations concerning efficient polynomial multiplications, which represent the most expensive base operation of ECC. The results of this investigations will lead to the development of new hardware designs to satisfy the requirements, such as execution time, area, and energy consumption. An efficient single curve implementation should prove the correctness of the considered algorithms and provide comparable data such as speed and size. Even though the primary focus is placed on an ASIC, an FPGA implementation should supply functional verification. Furthermore, the concept should be confirmed as the ECC design will be part of a real communication system on chip. The single curve ECC design is the basis for flexible ECC accelerators.

The rest of this thesis is organized as follows:
In Chapter 2 the cryptographic ideas, which were briefly mentioned in this introduction, are further presented and the benefits of ECC are demonstrated.
Chapter 3 introduces the finite fields that are the base fields of the elliptic curves. Since operations on these field are the functional backbone of ECC, they are comprehensively

explained and alternative fast algorithms are described and compared. Special attention is given to the multiplication operation. Based on known approaches an improved multiplication is devised that decreases the complexity of the operation in implementations considered later.
In Chapter 4 the operations on the elliptic curve are described. The algorithms for these operations, in particular the scalar point multiplication, provided there, are determined to be implemented in hardware later.
Chapter 5 applies the algorithms of Chapter 3 and 4 and presents efficient hardware implementations of the main operations required for an efficient ECC design. Also a fundamental architecture of the considered hardware design in presented.
Chapter 6 assembles the separate hardware blocks of the previous chapter to an efficient single curve ECC processor. For this, the design space for ECC hardware designs is explored. Eventually, the ECC design will be embedded into a system on chip, which is really made in silicon.
In Chapter 7 the existing ECC design will be extended to a flexible ECC accelerator. Therefore, different approaches of realizing the flexibility are discussed. The main focus is hereby placed on flexible reduction methods in the base field. Finally, the extensive considerations of the preceding chapters will lead to efficient flexible designs for the acceleration of ECC.

2. Cryptography

This chapter provides cryptographic basics and algorithms. The main focus is placed on the introduction of different approaches of the public key cryptography. The advantage of ECC over other cryptographic approaches is demonstrated by comparing the efforts for breaking the cryptographic systems.

2.1. Cryptography basics

Cryptography (from Greek kryptós, "hidden", and gráphein, "to write") is the technique of converting information into and from a format that is unreadable without secret knowledge. A classical application of cryptography in computer science is the communication over an unsecured channel. Traditionally, an entity A, named Alice, and an entity B, named Bob, communicate over this insecure channel. An insecure channel is every communication channel where it is possible for an entity E, a villain named Eve, to interfere with the communication. Possible interferences can be eavesdropping, but also a changing, delaying, dropping or emitting of packets. This compromises the security of the communication.

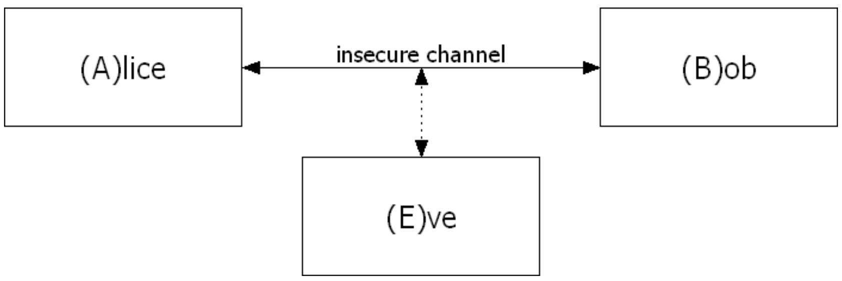


Figure 2.1.: The standard communication scenario: Alice and Bob want to communicate over an insecure channel. A third person Eve is able to eavesdrop and to modify elements of the transaction.

Desired properties of a secure communication channel are [48],[47]:

Confidentiality: When Alice sends a message to Bob, protection against unauthorized disclosure of the data to Eve should be provided.

Data integrity: It should be guaranteed that the sent data is protected against unauthorized or unintentional modifications. For example, Eve should not be able to intercept and modify data.

Authenticity: The identity claimed by or for a system entity must be verifiable. When Bob communicates with Alice, he must be sure that it is really Alice at the other end.

Non-repudiation The origin of a message must be verifiable and can be used as a proof. When Bob receives a message from Alice, she cannot deny having sent it.

Cryptography provides the tools to obtain these properties. Confidentiality is attained when plain a message is converted into an unreadable form and a reconversion is only possible with hidden secret informations. Data integrity can be assured by generating a hash value over the content. When the content changes, intentionally or unintentionally, it does not match the hash value.
Authenticity can be obtained by digital signatures or challenge-response protocols. Both approaches are based on the correct encryption or decryption of a message with a key that is exclusively known to one person.
Non-repudiation is a combination of data integrity and authenticity. Consider he case that Bob receives a message from Alice and can be sure that the content was not changed (data integrity). When it can additionally be verified that the message was original written by Alice (authenticity), it is evidence that she sent the message, which Bob has received, and she cannot deny it.

2.1.1. Hash functions

Hash functions are functions that map an input of variable length on a number of fixed bit length. This number is the hash value. These functions in particular find cryptographic application in data integrity and data authentication. It should generate a possibly unique mapping that provide the following conditions [38]:

- The input can be any length
- The length of the output of the hash function is fixed
- Computing of the function must be easy

Table 2.1.: Properties of cryptographic hash functions: modern highly secure hash functions calculate a 512 bit hash value. An attack on these hash algorithms requires 2^{256} attempts in average to be successful.

hash function	hash length [bit]	security strength
MD4	128	2^{20}
MD5	128	2^{39}[53]
SHA-1	160	2^{69}[52]
SHA-2	up to 512	up to 2^{256}
WHIRLPOOL	512	2^{256}

- The function must be 'one way' (preimage resistance). This means it should be computationally infeasible to find a message for a given hash value.
- The function should be collision resistant. This means it should be computationally infeasible to find two different messages with the same hash value.

Many different hash function have been developed in history. Well known is for example the 'Cyclic Redundancy Check' (CRC), which is mainly used for error detection in data streams or files. It is not recommended to use it to assure data integrity, since it is possible to create messages for a given hash values very efficiently. To ensure data integrity hash functions must provide a sufficient preimage and collision resistance. Hash functions that provide this properties are termed cryptographic hash functions. Popular examples of cryptographic hash functions are MD4 and MD5 [41], which calculate 128-bit hash values, and SHA-1 [6] with a 160-bit hash value. Since these hash functions have been successfully attacked ([52], [53]), stronger algorithms are recommended. Today, the SHA-2 functions [51] with hash lengths up to 512 bit and the WHIRLPOOL hash function [40] are considered as safe.

2.1.2. Digital signature

The digital signature is a method for authenticating digital information. The intention is to obtain a digital equivalent to the classic written signature on paper. It should verify that the underwriter has read and acknowledged the content. Say, Alice wants to send a message, for example an order, to Bob, and data integrity, authenticity and non-repudiation should be guaranteed. In the previous section hash functions were already described, which assure that a received message has not been changed. Indeed, it is possible for Eve to change the content and generate a new hash to pretend an original message. To eliminate this risk, for a digital signature the hash generation is followed by a second step. In this step Alice encrypts the original hash value using a key that authenticates her.

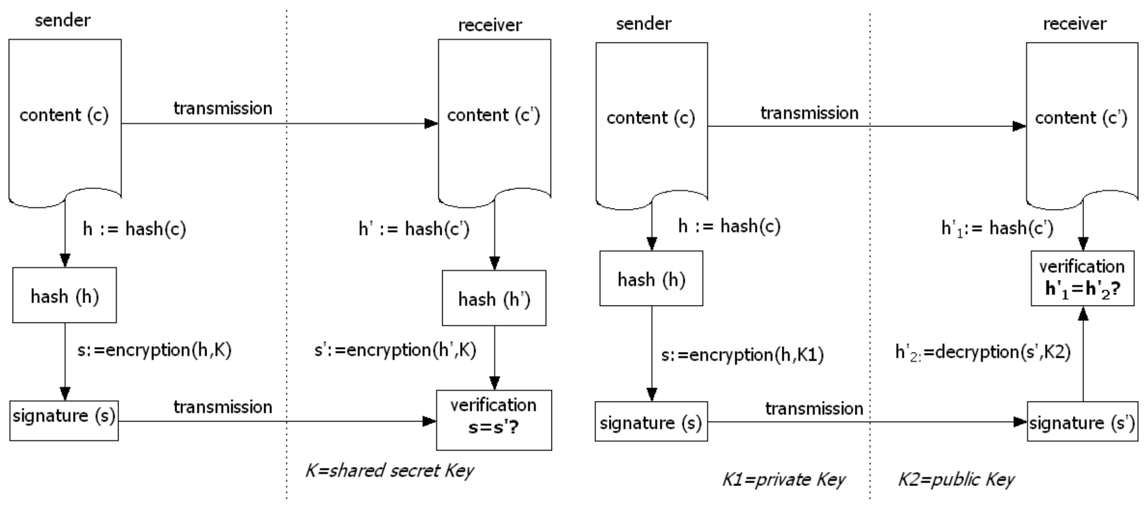


Figure 2.2.: Signature schemes realized with shared secret keys (left) and public key approaches (right). The tasks for the sender are identical in both methods: the encrypted hash value of the content is sent together with the content. The difference is on the receiver side. In the shared key approach the signature is redetermined and verified, whereby with a public key the signature is decrypted and verified against the hash of the received message.

Generally, there are two possible ways of performing this authentication:

- Both parties share one key. Alice encrypts the hash of the sent message with the same key that Bob applies to encrypt the hash of the received message. If both encryptions are identical, then it is evidence for Bob that Alice sent the text, because no one but Alice and Bob know the secret key.
- Both parties use different keys for encryption and decryption. Alice encrypts the hash value of the message with one key and sends the result together with the text. Bob can decrypt the encrypted hash with a corresponding key. To verify the received message, Bob compares the decrypted hash with the determined hash of the received message.

The shared key approach is connected with disadvantages. First, there is the problem of distributing the key. How can Alice and Bob obtain the same key without giving Eve the chance of intercepting it? In addition, this approach proves the identity of Alice to Bob but it does not prove it to a third party. This is because Bob can generate the signature himself, since he shares the same key.
The second approach where Alice has a private key and Bob uses a corresponding public key, is known as public key approach, because Bob's key can be published. Thus, everyone can verify that Alice is the origin. In addition, it is no threat when Eve intercepts the public key. One can see that the public key approach is indeed a good solution for the problem of digital

signatures, mutual authentication and key exchange. This is why public key cryptography is considered thoroughly in the following.

2.2. Public key cryptography

The public key cryptography, which is also known as asymmetric cryptography, is a cryptographic approach with different keys for encryption and decryption. One key, the public key, can be published while the other one, the private key, is kept secret. Applying this approach, it is possible for everyone to encrypt a message with the public key and only the owner of the private key can decrypt it. As mentioned before, public key cryptography is useful for digital signatures. In this application the public can decrypt a message that must have been encrypted by the owner of the private key.
To assure security, a public key cryptographic system must have the following properties:

- It must be computationally infeasible to determine the private key from the public key.
- It is not possible to reverse the public key cryptographic operation without the associated private key.

These properties can be realized with mathematical one-way functions. These 'trapdoor' operations are performed easily in one direction whereby the calculation of the reverse operation is very expensive. Until today two mathematical principles have been applied for public key cryptography:

- Factorization
- Discrete logarithm

2.2.1. Factorization problem

The multiplication of two large integers is an easy operation. The reverse operation, that is to find the factors for a given product, is far more complicated. Today no algorithm is known that solves the factorization problem in polynomial time. The first adaption of the factorization algorithm for public key cryptography is the RSA algorithm.

RSA

The RSA algorithm is probably the most well-known public key algorithm today. RSA stands for the initials of the surnames of the developers Ron Rivest, Adi Shamir and Len Adleman.
RSA uses the fact that for specific combinations of integers d and e it is true that

$m^{ed} \equiv m \mod n$. Thus, a message m can be encrypted with the public key tuple (d, n) by applying $c \equiv m^d \mod n$. The encrypted word c can be decrypted with the private key (e, n), since $c^e \mod n = m^{d^e} \mod n = m^{de} \mod n = m$.
The algorithm is based on the little Fermat theorem, which says that for any prime p and any integer a which is coprime to p it is valid that

$$a^{(p-1)} \equiv 1 \mod p$$

Thus it is also valid that $a^{i(p-1)} \equiv 1 \mod p$ for any integer i as long as a^i is coprime to p
Besides p, RSA uses another prime q so that finally:

$$n := pq$$

$$a^{i(q-1)} \equiv 1 \text{ mod } n,\ \ a^{i(p-1)} \equiv 1 \text{ mod } n \text{ and } \ \ a^{i(p-1)(q-1)} \equiv 1 \text{ mod } n$$

Thus

$$a^{i(p-1)(q-1)+1} \equiv a \mod n$$

RSA uses this fact by selecting two integer d and e so that:

$$ed \equiv 1 \mod (p-1)(q-1)$$

with other words

$$ed = i(p-1)(q-1) + 1$$

It follows

$$a^{ed} \equiv a \mod n$$

Now (e, n) is the public key and (d, n) the private one. With these keys the following operations are defined:
Encryption: $c \equiv m^d \mod n$
Decryption: $m \equiv c^e \mod n$, since $m^{d^e} \mod n = m^{de} \mod n$
This means, both encryption and decryption are performed by raising a data word m or c to power d or e respectively modulo n. For a secure system all variables should have a size of several hundreds of bits.
To break the system one must determine e for the given tuple(d, n) so that $de \equiv 1 \mod (p-1)(q-1)$. For this it is necessary to find the factors p and q for the product n. With p and q it is easy to find the private key d. Thus RSA relies on the fact that factorizing very large numbers is a problem which is extremely difficult to solve.

Concretely, the most efficient method of factorizing large numbers is Number Field Sieve [36]. The expected running time of that algorithm is proportional to [23]

$$e^{(1.9229+O(1))\cdot\ln(n)^{1/3}\cdot\ln(\ln(n))^{2/3}}$$

which is subexponential when n goes to infinity.

Despite the subexponential running time, RSA can be considered as safe when the selected keys are long enough. The largest successfully broken RSA key that has been reported is a 200-digits RSA number that could be factorized in May 2005 [18]. The process of the factorization of the 663-bit number took more than one year on a cluster of 80 2.2 GHz Opterons.

2.2.2. Discrete logarithm systems

The second important public-key cryptography scheme is the group of discrete logarithm systems. These systems rely on the idea that in a multiplicative cyclic group $\mathbb{G}$ it is very difficult to find the integer k for the given elements g and $h \in \mathbb{G}$ so that $h = g^k$, while it is easy to calculate h when g and k are known.

The DL systems operate in a finite group $(\mathbb{G}, \cdot)$. In the simplest case $\mathbb{G}$ is the set of integers modulo p, $\{0, 1, 2, ..., p-1\}$. The operation $\cdot$ denotes the multiplication modulo p on that group.

Selecting an element $g \in \mathbb{G}$, one can create a subgroup $(\langle g \rangle, \cdot)$, whereby the set $\langle g \rangle = \{g^i : 0 \leq i \leq q-1\}$ consists all powers of g, while q is the smallest nonzero integer such that $g^q = 1$. This q is called the order of the subgroup. Since the subgroup was generated by g, g is called the generator of the group.

Example: For the integers modulo $p = 11$ one obtains the group $\mathbb{G}$ with the members $\{0, 1, 2, ..., 10\}$. Now selecting one element $g = 4$, the multiplicative subgroup $\langle g \rangle$ of $\mathbb{G}$ consists of the members $\{4^1 \mod 11 = 4,\ 4^2 \mod 11 = 5,\ 4^3 \mod 11 = 9,\ 4^4 \mod 11 = 3,\ 4^5 \mod 11 = 1\} = \{1, 3, 4, 5, 9\}$. The order of this subgroup is $q = 5$.

Determining the k-th power of g, $h = g^k$, which is called discrete exponentiation, is relatively easy to calculate. The inverse operation, to determine k for given h is a much more challenging task, which is known as the discrete logarithm problem (DLP). Obviously, it is essential to use groups that provide a DLP which is very hard to solve, while the exponentiation is easy to calculate.

Example: From our group $\langle g \rangle$ let us take the exponent $k = 3$ and determine
$h = 4^3 \mod 11 = 9$. The DLP in this case is to determine k for the equation $9 = 4^k \mod 11$. In this simple example it easy to try every possible power to find the solution.

For large $k > 100$ bit this approach is obviously not feasible. Today, the best known method to calculate the discrete logarithm in subgroups is Pollard's rho algorithm [35]. The expected running time as function of the order of the subgroup q is proportional to

$$\sqrt{\frac{\pi q}{2}} \approx 1.25\sqrt{q}$$

Alternatively the Number Field Sieve (NFS) approach is applicable. This method solves the DLP for the complete based group $\mathbb{G}$. The expected running time of this algorithm is proportional to

$$e^{(1.9229+O(1))\cdot \ln(p)^{1/3}\cdot \ln(\ln(p))^{2/3}},$$

which is in contrast to Pollard's rho method subexponential in terms of the size of the parameter. But since the size p of the base group is much larger than the order q of the subgroup, the running time of both methods is roughly the same. Todays recommended key sizes are 1024 bit for p and 160 bit for q. It is interesting that the most of todays known attacks on the DLP are done applying the NFS. In practice it seems to be faster to attack the complete group $\mathbb{G}$ than the subgroup $\langle g\rangle$.
Also the largest broken DLP which is known for a 130-decimal digit prime p was computed applying the NFS. According to [20] it took three weeks on a 1.15 GHz 16-processor HP AlphaServer GS1280 to break the 430 bit key.

El Gamal encryption

An application of the discrete logarithm approach is El Gamal public-key encryption scheme. It is used when Bob wants to send a message to Alice without using a shared key. The public key that was set up by Alice is ($\mathbb{G}$, g, q, h), where $\mathbb{G}$ is the base group, g is the generator of the subgroup and q is the order of the subgroup. The fourth element h is a member of the subgroup and was determined by Alice by calculating $h = g^x \mod p$. The integer power x is the private key $(x < q)$ which is chosen arbitrarily.
To encrypt a message $m < p$, Bob does the following

- selects a random integer $(k < q)$
- computes $c_1 = g^k \mod p$
- computes $c_2 = m \cdot h^k \mod p$

The encrypted message is (c_1, c_2). To decrypt the message Alice

- computes $m = c_2 \cdot c_1^{-x} \mod p$

Alice hereby obtains the decrypted message, since

$$c_2 \cdot c_1^{-x} = \frac{c_2}{c_1^x} = \frac{m \cdot h^k}{(g^k)^x} = \frac{m \cdot (g^x)^k}{(g^k)^x} = m\frac{g^{kx}}{g^{kx}} = m.$$

When Eve eavesdrops the public key and the encrypted message, her challenge is to determine the public key x to obtain the decrypted content. This x can be calculated by solving the discrete logarithm $h = g^x \mod p$. The variables g, h and p are part of the public key. This challenge is an example for the discrete logarithm problem.

2.2.3. Elliptic curve cryptography

Generally speaking, ECC is nothing more than a special version of the discrete logarithm scheme.

The specialty lies in the underlying group. The elements of the elliptic group $\mathbb{E}$ are not natural numbers but two-dimensional points with x and y coordinates. Every point (x, y) on the curve satisfies an equation such as

$$y^2 + xy = x^3 + x^2 + b, \tag{2.1}$$

where b is a parameter defining the curve. The coordinates x and y are member of a finite field. The Equation 2.1 corresponds to the elliptic curve "B-233" as is recommended by the NIST [50]. Since not every EC implies cryptographic strengths, it is suggested to apply only such curves that have been analyzed and deemed to be secure.

To become more general, all ECs are defined by the so termed Weierstrass equation:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{2.2}$$

where $a_1, ..., a_6$ and the coordinates of the points (x, y), which satisfy the equation, are elements of a base field $\mathbb{B}$. This field $\mathbb{B}$ is not necessarily a finite field. Hence, also the field of real numbers $\mathbb{R}$ is a conceivable base field for ECs. For cryptographic applications only finite fields are applied because of the issues that are connected with the required infinite accuracy of the decimals digits, which is not a problem in finite fields. In spite of this, also ECs based on $\mathbb{R}$ are used in this thesis, in particular for presentation issues. Figure 2.3 shows an EC over $\mathbb{R}$ and one over a small finite field. A geometrical representation of the curve and the operations, which are principally identical for all curves, is obviously much more meaningful for the $\mathbb{R}$-based curves.

Independent of the applied elliptic equation and base field, all ECs satisfy the following properties, which are explained in order to understand the fundamental principles of ECC.

An EC group is an additive finite Abelian group. This implies an additive operation is defined

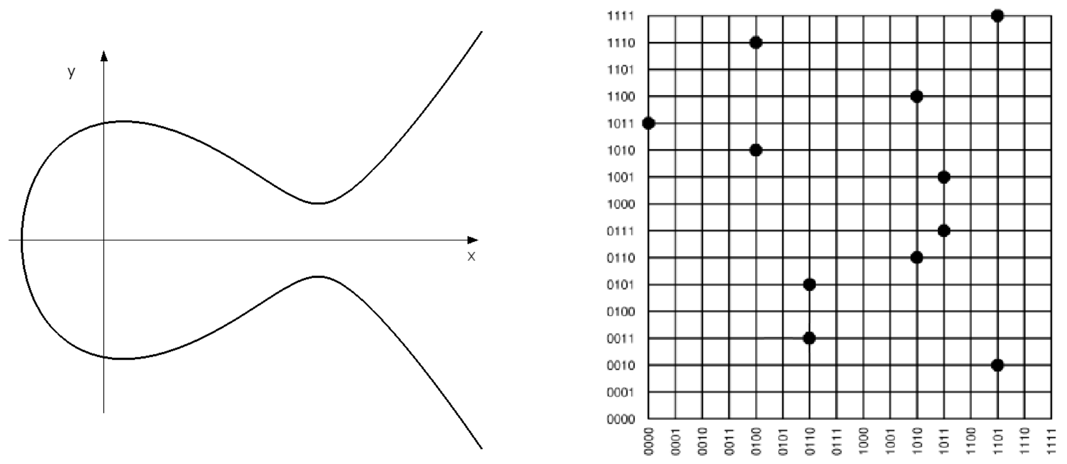


Figure 2.3.: Exemplary elliptic curves based on real numbers on the left and based on the finite field $GF(2^4)$ on the right. ECC is performed on finite fields. The typical 'EC-fish' is used for representation purposes.

in a way that every addition of two elements of $\mathbb{E}$ results in another element of the group ($C = A + B$). The term 'Abelian' means the operation is commutativ. Furthermore, an identity 0 is defined as point at infinity. Since also for every point on the curve other than infinity an additive inverse ($-A$) is defined, subtraction is possible.

Having specified the addition of two points, one can define the result of repeated addition ($Q = P + P + ... + P$) as scalar multiplication of a point by an integer k ($Q = k \cdot P$). One can create a subgroup of $\mathbb{E}$, $(\langle G \rangle, +)$, with $G \in \mathbb{E}$ and $\langle G \rangle = \{i \cdot G : 0 \leq i \leq q - 1\}$, whereby the order q is the smallest nonzero integer such that $q \cdot G = 0$. This q is finite in ECs that are based on finite fields. The subgroup $\langle G \rangle$ has similar attributes as the power operation based subgroup $\langle g \rangle$, which were described for the DLP. Analogous to the classic power operation, it is also very difficult to determine k for given $Q \in \langle G \rangle$, where $Q = k \cdot G$. This intractability of inverting scalar multiplication in $\mathbb{E}$ is named the elliptic curve discrete logarithm (ECDL) problem. The properties of the ECDL allow to use it in the same way as the classic DL. The main difference consists in having an additive group ($\mathbb{E}$,+) instead of the multiplicative group ($\mathbb{G}$,$\cdot$). Furthermore, the application of the ECDL in cryptographic algorithms is very similar.

The greatest benefit of EC become apparent when trying to solve the ECDL problem. Traditional DL can be broken with the subexponential number field sieve approach. This means, due to increasing computational power, the key sizes must grow very rapidely year by year and the keys become very long. For the ECDL problem, no equivalent of the NFS has been published. The best known way for attacking the ECDL is Pollard's rho method, which

Table 2.2.: Comparison of recommended future key sizes according to [2]. An implementation that should be used until today must already provide the corresponding degree of security today. DLP requires both field and key size to be secure.

Year	DLP		RSA	ECC	Hash size
	field	key			
−2010	1024	160	1024	160	SHA-1 (160 bit))
−2030	2048	224	2048	224	SHA-224 (224 bit)
>2030	3072	256	3072	256	SHA-256 (256 bit)

requires exponential operational effort. For EC the expected runtime as function of the order q of the subgroup is

$$\frac{\sqrt{\pi q}}{2} \approx 0.88\sqrt{q}.$$

Up to today no subexponential approach for EC has been reported. That is why a much shorter key length is adequate to provide a security level equivalent to RSA or traditional DL systems.

2.2.4. Conclusions

The preceding sections demonstrate the need for public key approaches and introduce the most important algorithms. As indicator of the strength of these methods, a factor for the expected runtime for a successful attack was specified. Even though the equations show that, due to exponential effort, the keys of the ECC are the shortest at equal security strength, they do not provide information of the actual key lengths.

Based on the equations and the expected computational power reference [23] has determined lower bounds for computationally equivalent key sizes until the year 2050. An overview that corresponds to those results is provided in the recommendations by the NIST [2]. These recommendations for the minimum key lengths that are assumed to provide sufficient security over the time are shown in Table 2.2.

Until the year 2010, ECC key lengths of 160 bits are considered as secure, whereby the equivalent secure key length for the RSA method is 1024 bit. For security systems that should be applied until 2030, an ECC key length of 224 bit is recommended and for RSA, 2024 bit. This implies that a RSA system requires the tenfold key length to provide an equal security within the next 30 years.

Indeed, these results are estimates that do not consider breakthrough in cryptoanalytics or revolutionary cryptographic hardware, such as quantum computers. An example how fast the

cryptoanlytic progress can weaken a cryptographic algorithm is the SHA-1 hash algorithm. The NIST recommendations from the year 2005 suggest SHA-1 as secure until the year 2010 but cryptgraphic progress has degraded the security strength of the algorithm so that already larger hash functions are recommended.
Despite the uncertainty concerning the concrete future key lengths and security strengths, Table 2.2 shows that the advantages of ECC compared to alternate public key approaches are significant.

3. Finite fields

Finite Fields are the basis of elliptic curve cryptography. The understanding of these fields is fundamental for efficient implementations of elliptic curves. This is why the fields and their operations are described in the following chapter. Hereby, based on previous research, different algorithms are specified and discussed. Additionally, feasible improvements are investigated to provide a basis for efficient hardware implementations. Special effort is spent on the polynomial multiplication operation. Improvements in this expensive operation, which have a significant impact on hardware designs, are described in this chapter.

3.1. Finite fields basics

Before starting with the actual operations, the basic kinds and properties of finite fields are introduced. Indeed, the fields and the algebraic background are not covered too exhaustively. For further readings [24] is recommended.
A finite field $\mathbb{F}$ is a field that contains finitely many elements. In honor of Evariste Galois, these fields are also named Galois fields (GF). The number of elements in $\mathbb{F}$ is called field order. The following operations are defined over elements of $\mathbb{F}$.

- Addition ($\mathbb{F}$,+) as additive abelian group with identity 0
- Multiplication ($\mathbb{F}\backslash\{0\}$,$\cdot$) as multiplicative abelian group with identity 1

Since distributive law also holds, every finite field is a commutative ring $(\mathbb{F}, +, \cdot)$.
There are two main kinds of finte fields, prime fields $GF(p)$ and extension fields $GF(p^m)$.

3.1.1. Prime field $GF(p)$

Prime fields $GF(p)$ are finite fields with a prime number of elements p. The elements are usually denoted as a set of integers $\{0, 1, 2, ..., p-2, p-1\}$. The arithmetics in $GF(p)$ are quite intuitive since they are performed with these natural integers.

Addition and subtraction is performed $a + b \equiv (a + b) \mod p$, which means the summation in the field is the remainder of the integer summation divided by p. Subtraction is expressed in terms of addition, while for $a - b$, a is added by the negative of b so that $a - b = a + (-b)$

Multiplication is computed the same way as addition. The product in $GF(p)$ is the remainder of the integer multiplication divided by p, $a \cdot b \equiv (a \cdot b) \mod p$.

Division is more complicated, since it is not so obvious to find $c = \frac{a}{b}$ so that $c \cdot b = a$. The most common way is to estimate the multiplicative inverse of the divisor b^{-1} ($1 = b \cdot b^{-1} \mod p$) and multiply the dividend by it, so that $c = a \cdot b^{-1}$.

Prime fields are feasible base fields for ECC. There are also special elliptic curves based on prime fields that are recommended by the NIST [50]. Hardware implementations for these fields are presented in [44] and [33]. Despite that, prime fields are mainly used in other cryptographic methods than ECC. For example RSA relies on these fields.
Even though prime fields are not supposed to be implemented in hardware in this work, they will be used from time to time in this thesis, in particular for examples, due to their arithmetic, which is easier to comprehend than polynomial field arithmetic.

3.1.2. Extension fields $GF(p^m)$

Let p be a prime and $m > 1$. $GF(p^m)$ is the field of equivalence classes of polynomials of degree at most $m - 1$ whose coefficients belong to $GF(p)$. Then the elements of the field are:

$$GF(p^m) = \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_2x^2 + a_1x + a_0 \ : \ a_i \in 0, 1, ..., p - 1\}.$$

One can imagine an element of this field as a number with m digits where every digit is number in the area $0..p - 1$.
To provide the field operations, an extension field is constructed by finding an irreducible polynomial $r(x)$ of degree m with coefficients in GF(p). Then $GF(p^m) = GF(p)[x]/(r(x))$ is a finite field with the order p^m. This means the field $GF(p^m)$ is the set of all equivalence classes of polynomials of x with coefficients out of $GF(p)$ modulo $r(x)$. This set contains exactly p^m equivalence classes and provides the additive and multiplicative operation.

Addition and subtraction is performed by standard polynomial addition and subtraction while coefficient arithmetic is done in $GF(p)$.

$$\begin{aligned} a(x) + b(x) &= (a_{m-1}x^{m-1} + ... + a_1x + a_0) + (b_{m-1}x^{m-1} + ... + b_1x + b_0) \\ &= [(a_{m-1} + b_{m-1})\text{mod } p]x^{m-1} + ... + [(a_1 + b_1)\text{mod } p]x + [(a_0 + b_0)\text{mod } p.] \end{aligned}$$

The subtraction process is analogous to the addition.

Multiplication in $GF(p^m)$ is a multiple step operation. First, a standard polynomial multiplication is performed. The coefficient arithmetic in executed in $GF(p)$. Since the result of the polynomial reduction is longer than m digits and hence is not fitting in the field, a second step is required. In this second step a reduction is performed. The result of this reduction is the remainder of the long product divided by the irreducible polynomial.

$$\begin{aligned} a(x) \cdot b(x) &= (a(x) \times b(x)) \text{mod } r(x) \\ \times &: \text{ polynomial multiplication} \\ r(x) &: \text{ irreducible polynomial} \end{aligned}$$

Division in $GF(p^m)$ is a multiplication of the dividend with the inverse of the divisor. Determining the multiplicative inverse is a very complex operation that is usually solved by applying the 'extended Euclidean algorithm'.

Special subgroups of extension fields $GF(p^m)$ are prime fields and binary extension fields. In these fields constantly $m = 1$ or $p = 2$, respectively. Even though extension fields that are not part of the two subgroups, are adequate for ECC, they are not relevant in practice. A practical application are 'optimal extension fields' [1]. These fields are fields such as $GF((2^{61} - 5)^3)$ or $GF((2^{32} - 5)^5)$, which efficiently use the sizes of registers and operations of general purpose processors.

3.1.3. Binary extension fields $GF(2^m)$

Binary finite fields are a special kind of extension fields with the prime characteristic $p = 2$. The elements of the groups are:

$$GF(2^m) = \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_2x^2 + a_1x + a_0 \;:\; a_i \in \{0,1\}\}.$$

The elements can also be written as bit vector $(a_{m-1}, a_{m-2}, ..., a_2, a_1, a_0)_2$. It corresponds to the hardware and software representations, where the m binary values are stored in a bit array of the size m. As for the general extension fields, an irreducible polynomial $r(x)$ is required to provide the field operations. This irreducible polynomial has the degree m and the coefficients are in GF(2). Then $GF(2^m) = GF(2)[x]/(r(x))$ is a finite field with the order 2^m.

Since the coefficients are delimited to zero or one, operations can be performed much easier. It can be seen in Table 3.1 that both addition and subtraction in $GF(2)$ are equivalent to the XOR operation.

Table 3.1.: Addition and subtraction in $GF(2)$ is equivalent to XOR

a	b	addition (a+b)		subtraction (a-b)		XOR (a⊕b)
		simple	mod 2	simple	mod 2	
0	0	0	0	0	0	0
0	1	1	1	-1	1	1
1	0	1	1	1	1	1
1	1	2	0	0	0	0

Addition and subtraction are performed by standard polynomial addition and subtraction. The coefficient arithmetic is done in $GF(2)$ that can be substituted by simple XOR operations.

$$\begin{aligned} a(x)+b(x) &= (a_{m-1}x^{m-1}+...+a_1x+a_0)+(b_{m-1}x^{m-1}+...+b_1x+b_0) \\ &= [(a_{m-1}+b_{m-1})\text{mod } 2]x^{m-1}+...+[(a_1+b_1)\text{mod } 2]x+[(a_0+b_0)\text{mod } 2] \\ &= [(a_{m-1}\oplus b_{m-1})]x^{m-1}\oplus ...\oplus [(a_1\oplus b_1)]x+[(a_0+b_0)] \end{aligned}$$

The subtraction is completely identical. When $a(x)$ and $b(x)$ are represented as m-bit bit arrays, addition and subtraction can be performed by a simple array-XOR:

$$a(x)+b(x)\equiv a(x)-b(x)\equiv b(x)-a(x)=a(x)\oplus b(x)$$

Multiplication in $GF(2^m)$ is the same two-step operation as in the extension fields. A polynomial multiplication is followed by a reduction step. The advantage of binary fields is again that both steps are taking profit from the faster XOR operation. Polynomial multiplication and reduction in $GF(2^m)$ are investigated in detail later.

Division in $GF(2^m)$ traditionally is performed by multiplication with the inverse of the divisor. An approach that performs a division in one step has been introduced in [46].

All further elliptic curve research in this work is based on $GF(2^m)$, because properties of these fields are especially suited for hardware implementations. Major properties are:

- Addition and subtraction are XOR operations.
- No carries are required, which usually slow down large integer arithmetics.
- The binary character of the coefficients allows an efficient representation in hardware. For example, an element in $GF(2^{163})$ can be stored in 163 flip flops.

Since elliptic curves on these fields have been standardized and analyzed for years, high level of experience and security has been attended. This provides a fundamental basis for special ECC hardware accelerators.

In the following, multiplication, squaring as a special multiplication, reduction and the operation of finding the inverse in $GF(2^m)$ are described and investigated. The addition operation is obvious, so that no further investigations are required.

3.2. Polynomial multiplication

Polynomial multiplication in $GF(2^m)$ is a very expensive operation in the base field. Applying classic algorithms, the bit complexity of this operation is $O(m^2)$. This means, double bit length leads to quadruple complexity. For bit lengths of many hundred bits this is very crucial. Advanced algorithms obtain less complexity. In this section the classic method and variations of the faster Karatsuba multiplication are discussed. Finally, improved applications of the iterative Karatsuba approach are described and compared.

3.2.1. Classic polynomial multiplication

The classic polynomial multiplication (CPM) is a straightforward translation of the classic school multiplication algorithm. Considering a multiplication of two polynomials $A(x)$ and $B(x)$, each of degree m, one obtains the product $C(x)$ of the degree $2m-1$. The coefficients of $C(x)$, $c_0 ... c_{2m-2}$, are determined in the following way: [11]

$$\begin{array}{rcccccccc}
c_0 = & a_0b_0 & & & & & & \\
c_1 = & a_0b_1 & + & a_1b_0 & & & & \\
 & & & \vdots & & & & \\
c_{m-1} = & a_0b_{m-1} & + & a_1b_{m-2} & + \ldots + & a_{m-2}b_1 & + & a_{m-1}b_0 \\
 & & & \vdots & & & & \\
c_{2m-3} = & & & & & a_{m-2}b_{m-1} & + & a_{m-1}b_{m-2} \\
c_{2m-2} = & & & & & & & a_{m-1}b_{m-2}
\end{array}$$

The estimation of the $2m-1$ bit long product out of two $m-1$ bit inputs takes m^2 partial one bit multiplications and $(m-1)^2$ additions. Since in $GF(2^m)$ one bit multiplications are AND operations and one bit additions are XOR operations, $(m-1)^2$ XOR and m^2 AND gates are required for the CPM.

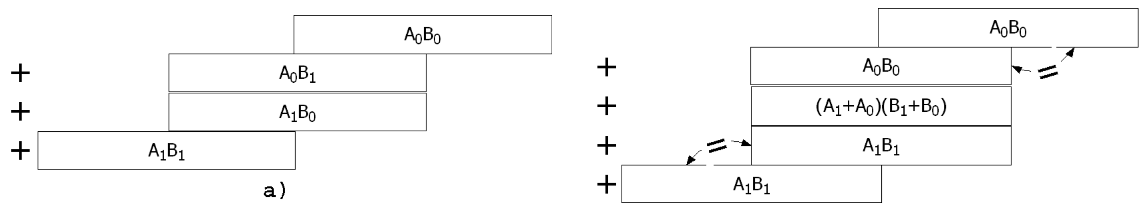


Figure 3.1.: Schematic comparison of classic polynomial method (a)) and Karatsuba multiplication (b)). CPM requires four partial multiplications and three accumulations while Karatsuba needs three different partial multiplications and four accumulation operations.

3.2.2. Karatsuba multiplication

Karatsuba introduced in [21] a multiplication approach that decreases the complexity of the operation. Originally, the method was presented to reduce the number of partial multiplications for a decimal multiplication:

$$(a + b \cdot 10^n)(c + d \cdot 10^n) = ac + [(a + b)(c + d) - ac - bd]10^n + bd \cdot 10^{2n}$$

This means that splitting the decimal factors in two equally long parts and multiplying the partial factors with the special method reduces the number of partial multiplications to three. The approach can be easily adapted to polynomial multiplications in $GF(2^m)$:

$$(a + b \cdot x^{\frac{m}{2}})(c + d \cdot x^{\frac{m}{2}}) = ac + [(a + b)(c + d) + ac + bd]x^{\frac{n}{2}} + bd \cdot x^m$$

For a polynomial multiplication of $A(x) \cdot B(x)$ where

$$A(x) = (a_{m-1}, a_{m-2}, ..., a_1, a_0)_2 = A_0 + A_1 \cdot x^{m/2}$$
$$A_0 = (a_{m-1}, a_{m-2}, ..., a_{m/2})_2 \text{ and } A_1 = (a_{m/2-1}, ..., a_1, a_0)_2$$

and corresponding B(x), one obtains

$$(A_0 + A_1 \cdot x^{\frac{m}{2}})(B_0 + B_1 \cdot x^{\frac{m}{2}}) = A_0B_0 + [(A_0 + A_1)(B_0 + B_1) + A_0B_0 + A_1B_1]x^{\frac{m}{2}} + A_1B_1 \cdot x^m$$

This operation compared to the classic multiplication is illustrated in Figure 3.1. It may be seen that an m-bit multiplication performed by the CPM corresponds to the accumulation of four partial $m/2$-bit multiplications. The alternative Karatsuba method requires the addition of five terms. But since two terms are duplicates, only three partial multiplications are required. Hence, Karatsuba takes more additions but less partial multiplications. We know that additions, especially in $GF(2^m)$, are cheap operations with linear complexity. In contrast multiplications are quite expensive. That is why the Karatsuba approach is an improvement concerning complexity.

The significant benefit of the method becomes apparent by applying it recursively. This means the partial multiplications are split into smaller partial multiplications which are split again. This recursive approach reduces the bit complexity of a multiplication to $O(m^{lg_2 3}) \approx O(m^{1.58})$. For the determination of this complexity, one can recall the original formula:

$$a_1x^m + a_0 \cdot b_1x^m + b_0 = a_1b_1x^{2m} + [(a_0 + a_1)(b_0 + b_1) + a_1b_1 + a_0b_0]x^m + a_0b_0$$

This formula consists of three m-bit-multiplications, two m-bit-additions and four 2m-bit additions. Since addition operations entail linear complexity, one can simplify the number of accumulations to ten m-bit additions. Hence, one step requires three partial multiplications and ten addition. The recursively application of that approach leads to the following conclusions, depending on the recursion level (l):

Data word size:	$s_l = 2 \cdot s_{l-1};\ s_0 = 1$
Needed single bit multiplications (ANDs):	$n_l = 3 \cdot n_{l-1};\ n_0 = 1$
Needed single bit additions (XORs):	$x_l = 3 \cdot x_{l-1} + 10 \cdot s_{l-1};\ x_0 = 0$

This means, when $l = 0$, the data word size is 1 bit, no addition is required, but one 1 bit multiplication, that is a single AND operation. At $l = 1$, 2-bit words are multiplied that apply three 1-bit multiplications and ten 1-bit additions. For $l = 5$ this sequence results in 32-bit multiplications that require 2110 XOR and 243 AND operations.
As next step the recursive sequence can be transfered into explicit formulas as function of l:

$$\begin{aligned} s_l &= 2^l \\ n_l &= 3^l \\ x_l &= 10(3^l - 2^l) \end{aligned} \tag{3.1}$$

The number of AND and XOR operations as function of the bit length m (corresponds to s_l) of a data word can finally be expressed as:

$$\begin{aligned} &\text{Number of ANDs: } m^{log_2 3} \approx m^{1.58} \\ &\text{Number of XORs: } 10 \cdot m^{log_2 3} - 10m \end{aligned} \tag{3.2}$$

Since both complexities are $O(m^{log_2 3})$, the total complexity of the classic Karatsuba method (CKM) is $O(m^{log_2 3})$.

3.2.3. Iterative Karatsuba multiplication

In [5] the Iterative-Karatsuba method (IKM) was introduced, which is an adaption of the original Karatsuba method. The IKM splits the operands into segments as in the CKM, but partial multiplications are processed iteratively. Instead of one monolith recursive

Table 3.2.: Accumulation table of the IKM [5]

Partial multiplication	Accumulations							
$[a_0 \cdot b_0][0]$					$\oplus$	$\oplus$	$\oplus$	$\oplus$
$[a_0 \cdot b_0][1]$				$\oplus$	$\oplus$	$\oplus$	$\oplus$	
$[a_1 \cdot b_1][0]$				$\oplus$	$\oplus$	$\oplus$	$\oplus$	
$[a_1 \cdot b_1][1]$			$\oplus$	$\oplus$	$\oplus$	$\oplus$		
$[a_2 \cdot b_2][0]$			$\oplus$	$\oplus$	$\oplus$	$\oplus$		
$[a_2 \cdot b_2][1]$		$\oplus$	$\oplus$	$\oplus$	$\oplus$			
$[a_3 \cdot b_3][0]$		$\oplus$	$\oplus$	$\oplus$	$\oplus$			
$[a_3 \cdot b_3][1]$	$\oplus$	$\oplus$	$\oplus$	$\oplus$				
$[(a_0 \oplus a_1) \cdot (b_0 \oplus b_1)][0]$					$\oplus$		$\oplus$	
$[(a_0 \oplus a_1) \cdot (b_0 \oplus b_1)][1]$				$\oplus$		$\oplus$		
$[(a_0 \oplus a_2) \cdot (b_0 \oplus b_2)][0]$					$\oplus$	$\oplus$		
$[(a_0 \oplus a_2) \cdot (b_0 \oplus b_2)][1]$				$\oplus$	$\oplus$			
$[(a_1 \oplus a_3) \cdot (b_1 \oplus b_3)][0]$				$\oplus$	$\oplus$			
$[(a_1 \oplus a_3) \cdot (b_1 \oplus b_3)][1]$			$\oplus$	$\oplus$				
$[(a_2 \oplus a_3) \cdot (b_2 \oplus b_3)][0]$			$\oplus$		$\oplus$			
$[(a_2 \oplus a_3) \cdot (b_2 \oplus b_3)][1]$		$\oplus$		$\oplus$				
$[(a_0 \oplus a_1 \oplus a_2 \oplus a_3) \cdot (b_0 \oplus b_1 \oplus b_2 \oplus b_3)][0]$					$\oplus$			
$[(a_0 \oplus a_1 \oplus a_2 \oplus a_3) \cdot (b_0 \oplus b_1 \oplus b_2 \oplus b_3)][1]$				$\oplus$				
	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

multiplication function, in that approach smaller multiplications are performed and the products are successively accumulated to the final result. It was also revealed that calculating the segments of the result separately can reduce the number of XOR operations.

Consider, the input words of the length m are split into four segments each. Then the product obtains eight segments of similar size $m/4$.

$$a_3a_2a_1a_0 \cdot b_3b_2b_1b_0 = c_7c_6c_5c_4c_3c_2c_1c_0$$

For the beginning, this segmentation in four pieces is nothing else than a classic Karatsuba in the second recursion level. This is why Equations 3.2 can be applied to determine the number of partial multiplications and additions. Hence, one can expect nine partial multiplication and 50 partial additions.

All operations after the unrolling are depicted in Table 3.2. The first column is the partial multiplication. Since the products have the double size, each product is written twice: one for the lower ([0]) and one for the higher part ([1]). The right columns show for which segments of c the partial results are accumulated. For example

$$c_1 = a_0 \cdot b_0[1] \oplus a_1 \cdot b_1[0] \oplus a_1 \cdot b_1[1] \oplus (a_0 \oplus a_1) \cdot (b_0 \oplus b_1)[0].$$

Surprisingly, the number of XOR operations does not correspond to the estimations. Table 3.2 shows 10 XOR operations for the determination of the factors and 42 XOR operation for the accumulation of the partial products. These are 12 operations more than expected. The reason for the discrepancy is that due to the unrolling, some additions are executed twice. But it opens the possibility of reducing the number of XOR operations by reusing summations of already accumulated partial products. For example, the summation of $a_0 \cdot b_0[1] \oplus a_1 \cdot b_1[0] \oplus a_1 \cdot b_1[1]$ can not only be used for c_1 but for c_2 and c_3 as well. In [5] a chain of additions was presented that reduces the number of XOR operations in the accumulation process from 42 XORs as shown in Table 3.2 to 29 XOR operations when reusing previous partial results. Together with the 10 XORs in factor generation, this is already less than the 40 XORs of the CKM. Even though addition is a cheap operation, the reduction of these operations accelerates hardware and software implementations.

3.2.4. Improvements of the iterative Karatsuba multiplication

Based on the IKM approach further improvements are discussed. First, an improved chain for the accumulation of the partial results $c_0..c_8$ is presented. It could be observed that the accumulation of the partial results $c_0..c_8$ as it was presented in [5] has left room for improvements. Trying other sequences, a chain could be found that only requires 24 XOR operations for the accumulation process. The corresponding sequence is depicted in Table 3.3. Together with the ten unchanged additions, which are necessary for the determination of the factors, 34 additions are required. These are six accumulations less than required for the CKM. The number of partial multiplications is unaffected.

Considering the partial multiplications are performed by the classic recursive Karatsuba method as it was described for the original IKM, the following complexities for the improved approach can be determined:

$$\begin{aligned}
\text{Number of ANDs:}\quad & 9 \cdot \tfrac{m}{4}^{log_2 3} = m^{log_2 3} \approx m^{1.58} \\
\text{Number of XORs:}\quad & 9 \cdot (8(\tfrac{m}{4})^{log_2 3} - 8\tfrac{m}{4}) + 34\tfrac{m}{4} = \\
& (10 \cdot m^{log_2 3} - 10m)\ - 4m
\end{aligned}$$

With only $4m$ XOR operations less compared to the original Karatsuba method, the approach effects in a relatively small gain in performance. However, applied recursively, this small gain can make a significant difference. Consider that the partial multiplications are also performed by the improved IKM approach. Then the recursive function for the number of XOR operations as function of the recursion level l is:

$$x_l = 9x_{l-2} + \frac{34}{4}4^{l-2},$$

Table 3.3.: This improved operation sequence reduces the number of XOR operation for the accumulation in IKM to 24.

Step	Partial product	Sequence
1	$pr = a_0 \cdot b_0$	$c_0 = pr[0]$ $c_1 = pr[0] \oplus pr[1]$
2	$pr = a_3 \cdot b_3$	$c_7 = pr[1]$ $c_6 = pr[0] \oplus pr[1]$
3	$pr = a_1 \cdot b_1$	$c_1 = c_1 \oplus pr[0]$ $c_5 = pr[1]$
4	$pr = a_2 \cdot b_2$	$c_5 = c_5 \oplus pr[0]$ $c_6 = c_6 \oplus pr[1]$
5	$pr = (a_0 \oplus a_1) \cdot (b_0 \oplus b_1)$	$c_2 = c_1 \oplus c_5 \oplus pr[1]$ $c_1 = c_1 \oplus pr[0]$ $c_5 = c_5 \oplus c_6$
6	$pr = (a_2 \oplus a_3) \cdot (b_2 \oplus b_3)$	$c_5 = c_5 \oplus pr[0]$ $c_6 = c_6 \oplus pr[1]$
7	$pr = (a_1 \oplus a_3) \cdot (b_1 \oplus b_3)$	$c_3 = c_1 \oplus c_5$ $c_4 = pr[1] \oplus c_2 \oplus c_0$ $c_5 = c_5 \oplus pr[1]$ $temp = pr[0]$
8	$pr = (a_0 \oplus a_2) \cdot (b_0 \oplus b_2)$	$temp = temp \oplus pr[1]$ $c_2 = c_2 \oplus pr[0]$ $c_3 = c_3 \oplus temp \oplus c_7 \oplus pr[0]$ $c_4 = c_4 \oplus temp \oplus c_6$
9	$pr = ((a_0 \oplus a_1) \oplus (a_2 \oplus a_3)) \cdot$ $((b_0 \oplus b_1) \oplus (b_2 \oplus b_3))$	$c_3 = c_3 \oplus pr[0]$ $c_4 = c_4 \oplus pr[1]$

which is rewritten as explicit function of the factor length:

$$x_n = \frac{34}{5} m^{log_2 3} - \frac{34}{5} m. \tag{3.3}$$

This implies that the 'Recursively Applied Iterative Karatsuba' (RAIK) results in a reduction of the XOR operations of 18% compared to classic Karatsuba. The number of AND operations is not affected.

Table 3.4.: Number of one bit XOR operations for different factor bit lengths and with different multiplication methods

Bit length	CPM	CKM	IKM	RAIK
4	9	50	34	34
8	49	190	158	129
16	225	650	586	442
32	961	2110	1982	1435
64	3969	6650	6394	4522
128	16129	20590	20078	14001
256	65025	63050	62026	42874
512	261121	191710	189662	130363

Table 3.5.: Number of one bit AND operations for different factor bit lengths and with different multiplication methods

Bit length	CPM	CKM	IKM	RAIK
4	16	9	9	9
8	64	27	27	27
16	256	81	81	81
32	1024	243	243	243
64	4096	729	729	729
128	16384	2187	2187	2187
256	65536	6561	6561	6561
512	262144	19683	19683	19683

3.2.5. Comparison

Having described different methods for multiplying polynomials in $GF(2^m)$, these approaches should be compared by the total number of operations for possible factor lengths. The considered multiplication methods are the classic polynomial multiplication (CPM), the classic Karatsuba multiplication (CKM), the iterative Karatuba multiplication (IKM), and the recursively applied iterative Karatsuba (RAIK) multiplication. The formulas for the determination of the complexity of AND and XOR operations are presented in the previous sections.

Table 3.4 and Table 3.5 provide a comparison of the methods that have been discussed so far, concerning the number of single bit XOR and single bit AND operations respectively. The graphical representation of the tables is also shown in Figures 3.2 and 3.3.

All Karatsuba based multiplications obtain the same number of AND operations. This number is always lower compared to CPM and the slope is also much slower. For the XOR operations the results are much more interesting. It is noticeable that for small factor lengths CPM requires the least number of operations. But this method is also the one that increases fastest. The slowest increment of operations is obtained by the RAIK method. Hence, it is the optimal

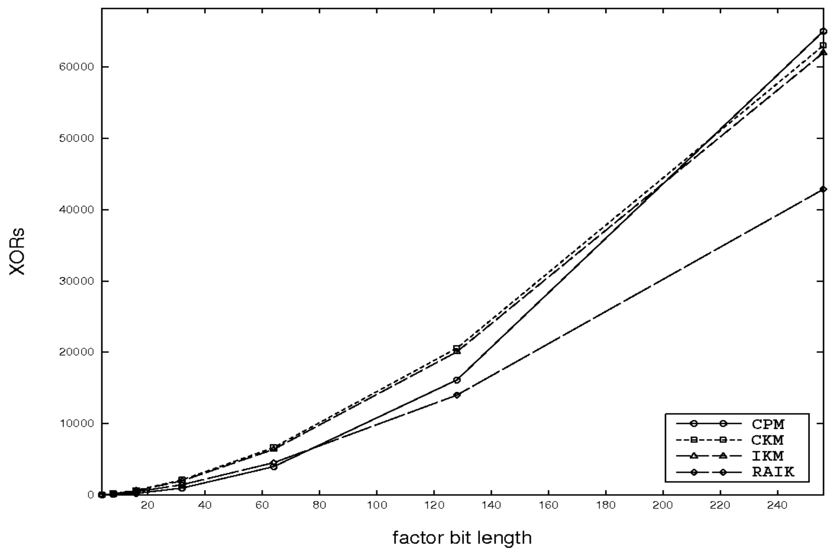


Figure 3.2.: Graphical representation of Table 3.4. The classic Karatsuba methods and CPM need a similar number of XOR operations for a multiplication. RAIK, the improved Karatsuba approach, needs 32% less XORs.

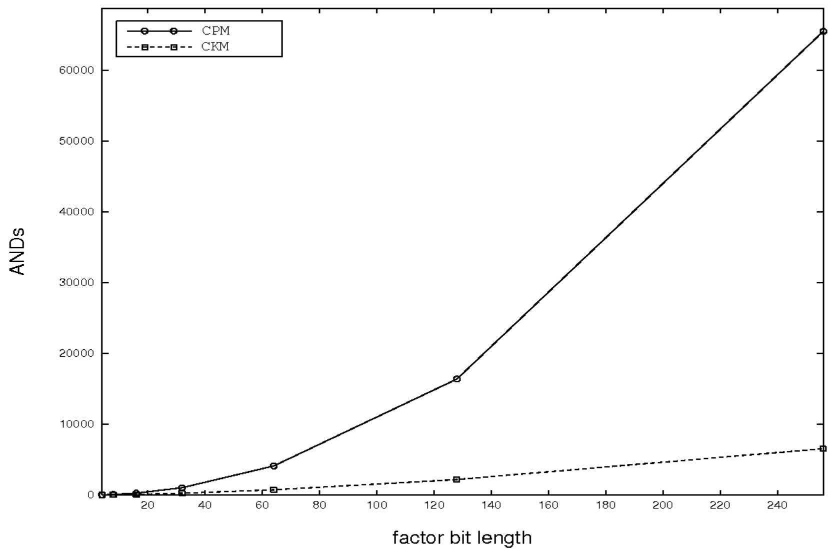


Figure 3.3.: Graphical representation of Table 3.5. The number of AND operations is significantly less for the Karatsuba approach than for CPM.

Table 3.6.: Number of one bit XOR operations for different factor bit lengths and with different multiplication methods

Bit length	CPM	CKM	RAIK	combined CKM/CPM	combined RAIK/CPM
4	9	50	34	9	9
8	49	190	129	49	49
16	225	650	442	227	217
32	961	2110	1435	841	713
64	3969	6650	4522	2843	2497
128	16129	20590	14001	9169	7505
256	65025	63050	42874	28787	24649

Table 3.7.: Number of one bit AND operations for different factor bit lengths and with different multiplication methods

Bit length	CPM	CKM	RAIK	combined CKM/CPM	combined RAIK/CPM
4	16	9	9	16	16
8	64	27	27	64	64
16	81	81	144	144	192
32	1024	243	243	576	576
64	4096	729	729	1296	1728
128	16384	2187	2187	5184	5184
256	65536	6561	6561	11664	15552

method for larger factor lengths.

The fact that for small factor lengths the fewest XOR operations are obtained by the CPM, raises the idea of substituting deep recursion levels of the Karatsuba multiplications with CPM. For example with RAIK, a 64-bit multiplier uses 16-bit partial multipliers which use 4-bit partial multipliers. Following the Karatsuba method, the next step would apply 34 single bit XORs and 9 single bit ANDs for the calculation of the 4-bit partial multiplication. The CPM only requires 9 XORs and 16 ANDs. This means, using the CPM for the 4-bit partial multiplication instead of RAIK reduces the total number of operations from 43 to 25. Since the small partial multipliers are the base elements of every Karatsuba multiplier, the impact of this substitution is significant. The concrete numbers of XOR and AND operations using this combined approach are represented in Table 3.6 and 3.7. The numbers are estimated by applying the recursive formulas that were presented in the previous sections. The combined method for both two-segment-Karatsuba and four-segment-Karatsuba are depicted.

As expected, the number of XOR operation is reduced by the combined method. For a 256-bit RAIK multiplier the decrease is 43% from 42878 to 24649 XOR operations. In contrast, the number of AND operations is rising. For the 256-bit-example by 77% from 6561 to 11664

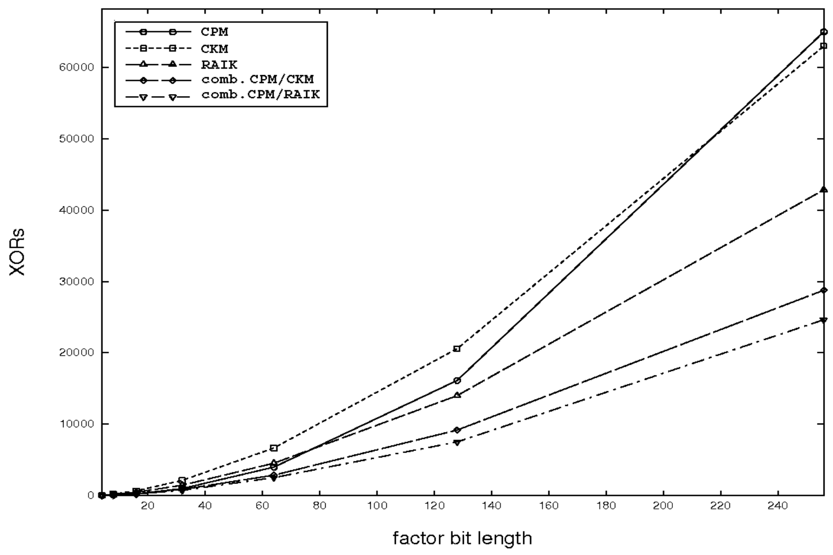


Figure 3.4.: Graphical representation of Table 3.6. The combined approaches require the least number of XOR operations, the classic approaches the most.

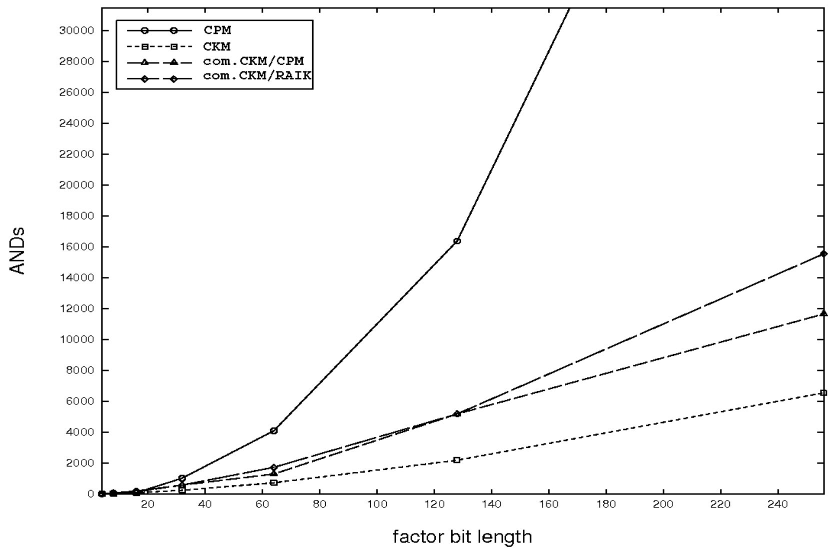


Figure 3.5.: Graphical representation of Table 3.7. The classic Karatsuba multiplication requires the least AND operations. The combined methods need more ANDs than the CKM, but much less than the CPM.

AND operations. Eventually, there are about 5000 AND operations more to compute but about 18000 XOR operations less. Effectively, that results in a reduction of 13000 total operations.

3.3. Polynomial squaring

Polynomial squaring in $GF(2^m)$ can be done with standard polynomial multiplication as described in Section 3.2, as it is obvious that $c(x) = a(x)^2 = a(x) \cdot a(x)$. But applying special properties of the $GF(2^m)$ can lead to implementations with much less complexity compared to standard multiplication.
In the most known GF(2) implementations (e.g. [43]) the following expression is used:

$$c(x) = a(x)a(x) = \sum_{i=0}^{m-1} a_i x^i \cdot \sum_{i=0}^{m-1} a_i x^i = \sum_{i=0}^{m-1} a_i x^{2i} \tag{3.4}$$

This expression is valid, since the finite fields $GF(2^m)$ are commutative rings with prime characteristic $p = 2$. [24] proved that for every commutative ring with prime characteristic p it is true that $(a+b)^p = a^p + b^p$. They applied the properties of binomial coefficients.
Since

$$\begin{pmatrix} p \\ i \end{pmatrix} = \frac{p!}{i!(p-i)!} = p\,\frac{(p-1)!}{i!(p-i)!} \equiv 0 \mod p$$

it follows that

$$(a+b)^p = a^p + \begin{pmatrix} p \\ 1 \end{pmatrix} a^{p-1}b + ... + \begin{pmatrix} p \\ p-1 \end{pmatrix} ab^{p-1} + b^p = a^p + b^p$$

And since $p = 2$ for $GF(2^m)$ Formula 3.4 is valid.

One can rewrite the coherence from Formula 3.4 to

$$(a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x^1 + a_0x^0)^2 = a_{m-2}x^{2(m-2)} + a_{m-1}x^{2(m-1)} + ... + a_1x^2 + a_0x^0$$

and see what actually happens. Every bit a_i from the input word at position i is moved to position $2i$ in the square. The other bits are filled with zeros. With other words, a 0-bit is inserted between every input bit. The complexity is $O(m)$ because it is simply a reassignment of the bits.

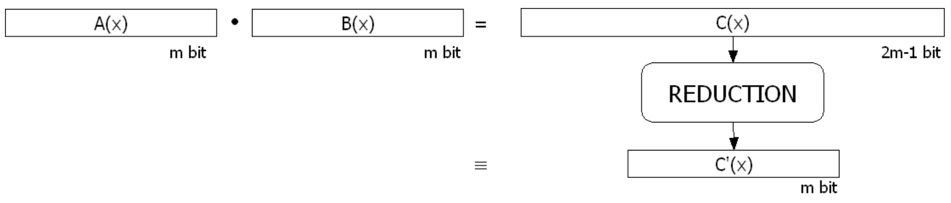


Figure 3.6.: Principle of the reduction operation: after a multiplication of two m bit polynomials, the result C(x) is too long. It must be reduced to an equivalent inside the field.

3.4. Reduction

Both polynomial multiplication and polynomial squaring result in data words that have a higher degree than the maximum field degree. To make them fit in the field, they have to be reduced. The fundamental scheme can be seen in Figure 3.6. The multiplication of two m bit data words results in a $2m-1$ bit product $C(x)$. The process of determining the equivalent element in the field is termed reduction. The operation corresponds to the modulo operation in prime fields. That is why the classic reduction method in $GF(2^m)$ is to divide the $C(x)$ by the irreducible. The remainder of this division is the reduced product $C'(x)$. There are also other methods for the reduction process that do not require a complete division. The multiplicative reduction and a fast reduction that has linear complexity are described in this section.

3.4.1. Polynomial division

The first obvious solution for the reduction problem, is to perform a standard polynomial division in order to estimate the remainder. The polynomial division based reduction is shown as Algorithm 3.1, and an example is shown in Figure 3.7.

Algorithm 3.1: Polynomial division as reduction implementation

```
   input  : $c(x)$, $m = degree(field)$, $r(x)$
   output: reduced polynomial $= c(x) \mod r(x)$
1  for $i = degree(c)$ downto $m$ do
2      if $c_i = 1$ then
3          $c(x) \leftarrow c(x)$ XOR $r(x) \cdot x^{i-m}$;
4      end
5  end
6  return $(c(x))$
```

The algorithm does not compute the quotient but the remainder that eventually is the reduced value. In this algorithm every overlapping bit is reduced successively from left to right. Thus, the maximum number of iterations of the inner loop is $deg(a)-deg(r)+1$. After a multiplication of two m bit words, the product has a bit length of $2m-1$. Then $2m-1-m+1 = m$ iterations

```
1 1 0 1 1 1 : 1 0 1 1
1 0 1 1
---------------------
  1 1 0 1 1
  1 0 1 1
---------------------
    1 1 0 1
    1 0 1 1
---------------------
      1 1 0
---------------------
```

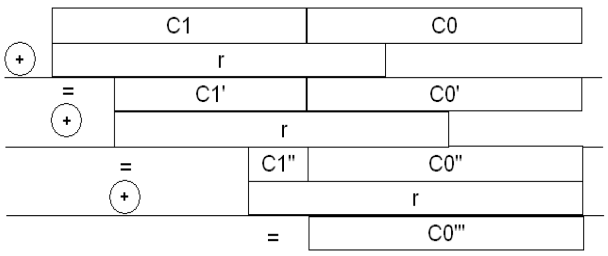


Figure 3.7.: Example of a simple polynomial reduction using polynomial division to compute the remainder.

of the inner loop must be performed in maximum. Since each of the possible m iterations is connected with a m bit XOR operation, the total effort of this method is $O(m^2)$.

3.4.2. Multiplicative reduction

The multiplicative reduction is the second possible reduction method. Hereby, repeatedly the product of the overlapping part $C1(x)$ of $C(x)$ and the reduction polynomial $r(x)$ is subtracted from $C(x)$. 'Overlapping' are the bits of $C(x)$ on higher positions than the field degree. In Figure 3.7 and 3.8 overlapping part is $C1$, and C0 are the bits that are within the field boundaries. Since $deg(C0(x)) = m$, the coherence between C, C0, C1 can be written as

$$C(x) = C1(x)x^m + C0(x).$$

The multiplicative reduction is shown as Algorithm 3.2 and represented in Figure 3.8. By this, with every iteration at least the highest bit is reduced. This is working, because in finite fields the following property holds true:

$$C(x) \equiv C(x) - i \cdot r(x) \bmod r(x)$$

for every i, and hence, also

$$C(x) \equiv C(x) - C1(x) \cdot r(x). \tag{3.5}$$

This shows that the result of the subtraction of the product of the irreducible polynomial and

Algorithm 3.2: Multiplicative reduction

```
  input  : $a(x)$, $m = degree(a)$, $r(x)$
  output: reduced polynomial $a(x) \mod c(x)$
1 $C1(x) \leftarrow C(x)/x^m$;
2 while $C1(x) <> 0$ do
3     $C(x) \leftarrow C(x)$ XOR $[C1(x) \cdot r(x)]$;
4     $C1(x) \leftarrow C(x)/x^m$;
5 end
6 return $(C(x))$
```

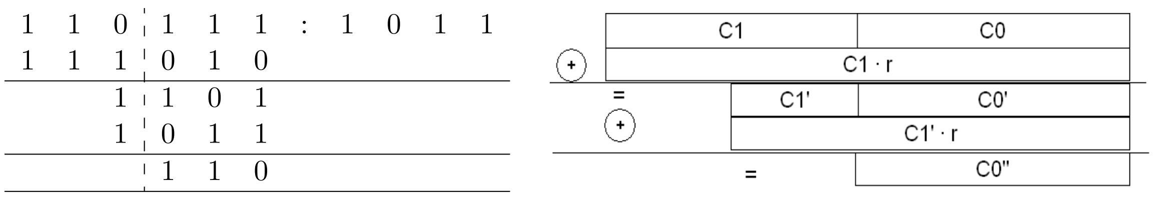


Figure 3.8.: Simple example for multiplicative reduction. Exploiting the special properties of NIST irreducible polynomials, the number of steps is always limited to two.

the overlapping part from $C(x)$ is still equivalent to $C(x)$ in the finite field. And since

$$\begin{aligned} C(x) &= C1(x)x^m + C0(x), \text{ and} \\ r(x) &= x^m + r_{m-1}x^{m-1} + ... + r_0 \end{aligned} \tag{3.6}$$

it follows that

$$\begin{aligned} C(x) &\equiv C(x) - C1(x) \cdot r(x) \\ &\equiv [C1(x)x^m + C0(x)] - [C1(x) \cdot (x^m + r_{m-1}x^{m-1} + ... + r_0)] \\ &\equiv [C1(x)x^m + C0(x)] - [C1(x)x^m + C1(x) \cdot (r_{m-1}x^{m-1} + ... + r_0)] \\ &\equiv C0 - C1(x) \cdot (r_{m-1}x^{m-1} + ... + r_0) \\ &\equiv C1(x) \cdot r_{m-1}x^{m-1} + ... + C1(x) \cdot r_0 + C0(x) \end{aligned} \tag{3.7}$$

This implies that $C1(x)$ is not multiplied by x^m anymore but by x^{m-1} in maximum. This way a new polynomial C'(x) can be defined:

$$C'(x) = C1(x) \cdot r_{m-1}x^{m-1} + ... + C1(x) \cdot r_0 + C0(x)$$

which has a smaller degree than $C(x)$. This $C'(x)$ can be expressed as

$$C'(x) = C1'(x)x^m + C0'(x)$$

where $deg(C0'(x)) = m - 1$ and $deg(C1'(x)) < deg(C1(x))$. This simplified representation shows that in every multiplicative step at least one bit is reduced. When only one bit is cut in each iteration, still $m - 1$ steps are required. The $m - 1$ multiplications of factor size m result in a complexity of $O(m \cdot m^{log_2 3})$, which is more expensive than reduction by division.
But taking a look to commonly used reduction polynomials reveals an interesting feature. As the five irreducible polynomials for the ECs recommended by the NIST (see Table 3.8) all reduction polynomials are either trinomials (three coefficients set) or pentanomials (five coefficients). This simplifies the multiplication remarkably. For example the multiplication

Table 3.8.: Reduction polynomials from NIST elliptic curves [50]

Recommended field	reduction polynomial $r(x)$
B-163	$x^{163} + x^7 + x^6 + x^3 + 1$
B-233	$x^{233} + x^{74} + 1$
B-283	$x^{283} + x^{12} + x^7 + x^5 + 1$
B-409	$x^{409} + x^{87} + 1$
B-571	$x^{571} + x^{10} + x^5 + x^2 + 1$

of the overlapping part ($C1(x)$) with the trinomial $x^{233} + x^{74} + 1$ is done by performing $C1(x) + C1(x)x^{74} + C1(x)x^{233}$. Efficient implementations can realize this operation with only two shifts and two XOR-operations. At least a full multiplier is not required.
The commonly used reduction polynomials have another property which is very beneficial. The second highest coefficient set in the polynomial in a field $GF(2^m)$ is always smaller than $m/2$. When substituting the coefficients of the reduction polynomial in Equation 3.7 according to this fact, it follows that a single iteration reduces more than $m/2$ bit of $C1(x)$. Thus, after two steps C(x) is completely reduced.
This brings complexity of the multiplicative reduction down to two multiplications. The multiplications can be substituted by few shifts and XOR-operations. This is why the total complexity for the reduction in commonly used fields is $O(m)$.

3.4.3. Fast reduction

The fast reduction is a special application of the multiplicative reduction for known reduction polynomials. The idea is, when the reduction polynomial is known, at design time a chain of XOR operations can be build that performs the reduction within one step as a direct mapping of the long input word.
Figure 3.9 shows an unrolled multiplicative reduction for the 233-bit field with the reduction polynomial $r(x) = x^{233} + x^{74} + 1$. As depicted, the result of the reduction process $C0''$ is

$$C0'' = C0' + C1'(72 - 0) + C1'(72 - 0) << 74 \tag{3.8}$$

The operation $<< 74$ shifts the word 74 bit to the left and fills the right bits with zeros. It is equal to a multiplication by x^{74}. The brackets show the number of the start and end bit of the considered chunk in the word.
Substituting C0' and C1' in Equation 3.8 one obtains

$$C0'' = C0 + C1(158 - 0) << 74 + C1 + C1(231 - 159) + C1(231 - 159) << 74$$

and further

$$C0'' = C(232 - 0) + C(390 - 233) << 74 + C(464 - 233) + C(464 - 392) + C(464 - 392) << 74.$$

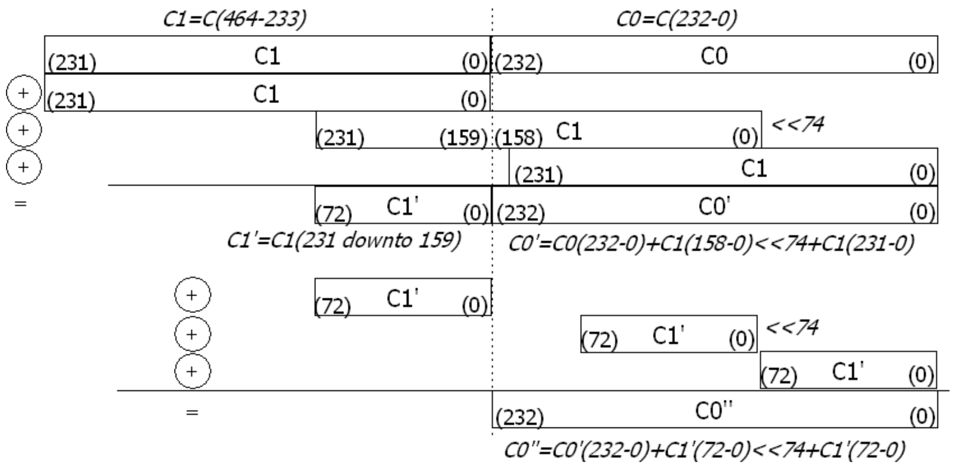


Figure 3.9.: Reduction of a 465 bit word in $GF(2^m)$. Finally the reduced $C0''$ can be determined by a direct mapping.

The final result can be split into five slices, where each slice is a direct accumulation of slices of C. Finally, the XOR operations that are required for mapping C onto the reduced C'' in $GF(2^{232})$ are:

$$\begin{aligned}
C''(232) &= C(232) + C(391) \\
C''(231-147) &= C(231-147) + C(390-306) + C(464-380) \\
C''(146-74) &= C(146-74) + C(233-305) + C(379-307) + C(464-392) \\
C''(73) &= C(73) + C(306) \\
C''(72-0) &= C(72-0) + C(305-233) + C(464-392).
\end{aligned}$$

The graphical representation of this mapping is shown as Figure 3.10.

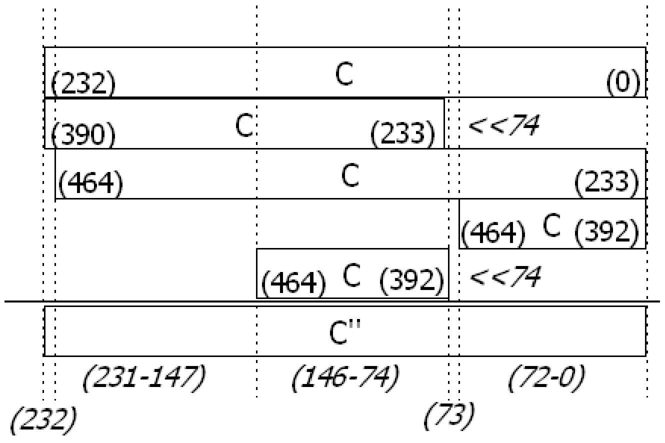


Figure 3.10.: Graphical representation of the direct mapping required for the fast reduction in $GF(2^{232})$.

This way, a combination of XOR operations can be derived for every reduction polynomial. For Pentanomials more slices have to be accumulated, but the approach is the same.
Since the fast reduction is a cheap method for the reduction, it is very often applied in both software and hardware implementations (e.g. [54], [43]). Of course, the gain of speed must be paid with a loss in flexibility, because for every reduction polynomial an specialized XOR combination has to be determined and realized.

3.5. Modular multiplicative inversion

The modular multiplicative inversion faces the problem of determining the multiplicative inverse in a finite field. The multiplicative inverse is the element $i(x) \in GF(2^m)$ that fulfills the equation $a(x) \cdot i(x) \equiv 1$ mod r(x) for a given element $a(x)$. The element $i(x)$ is termed the multiplicative inverse of $a(x)$ in the field.
The inversion is required for the division operation, when the division is performed by multiplying the dividend with the inverse of the divisor.
There are two main approaches for calculating the multiplicative inverse in large finite fields. The first one is based on the Euclidean algorithm and the other one on the so called Fermat's little theorem. Also a direct division is known, which can substitute the operation of finding the multiplicative inverse. All three approaches are described and compared in the following.

3.5.1. Extended Euclidean algorithm

The original Euclidean algorithm determines the greatest common divisor of two integers. It is the base for the extended Euclidean algorithm, which can be used to determine the multiplicative inverse. The algorithms will be described for prime fields since the operations are easier to comprehend. Generally, the algorithms in $GF(2^m)$ are the same. A version for binary fields is also presented at the end of this section.

The Euclidean algorithm determines the greatest common divisor (GCD) of two integers without factoring. It was presented by Euclid around 300 BC. The algorithm uses the fact that $gcd(a, b) = gcd(a - qb, b)$, while $q = \lfloor \frac{a}{b} \rfloor$.
The algorithm determines a sequence of remainders r_n , whereby the r_0 and r_1 are initialized

with a and b, respectively. The last nonzero remainder is the GCD. The concrete formulas are:

$$q_n = \lfloor r_{n-1}/r_n \rfloor$$
$$r_n = r_{n-2} \bmod r_{n-1};\ r_0 = a;\ r_1 = b$$
$$r_n = r_{n-2} - r_{n-1} \cdot q_{n-1};\ r_0 = a;\ r_1 = b$$

The extended Euclidean algorithm (EEA) extends the original Euclidean algorithm by tracking additional values to obtain the $ax + by = gcd(a,b)$. When a and b are relatively prime, which means $gcd(a,b) = 1$, this algorithm can be used for estimating the multiplicative inverse of $a \mod b$.
If it is assured that

$$ax + by = 1,$$

and since

$$by \equiv 0 \mod b,$$

then it follows

$$ax + by \equiv 1 \mod b$$
$$ax + 0 \equiv 1 \mod b$$
$$ax \equiv 1 \mod b.$$

Hence, x is the multiplicative inverse of a modulo b.
Additional to r_n and q_n the EEA determines the values s_n and t_n in every step.

$$s_n = s_{n-2} - q_{n-1} \cdot s_{n-1};\ s_0 = 1;\ s_1 = 0$$
$$t_n = t_{n-2} - q_{n-1} \cdot t_{n-1};\ t_0 = 0;\ t_1 = 1$$

Then it is provided that

$$a \cdot s_n + b \cdot t_n = r_n.$$

An example of computing the inverse of 17 in GF(37) is presented as Table 3.9:
Hence, $-13 \equiv +24$ mod 37 is the multiplicative inverse of 17 in $GF(37)$. It can be verified by calculating $24 \cdot 17 = 408 \equiv 1 \mod 37$.
For the estimation of the inverse it is not required to track all values. For example, there is no need for calculating s, because we only want to know $ax \equiv 1 \mod b$. Also, only the values of the last two iterations must be stored instead of the complete table. This simplifies the algorithm.

Table 3.9.: Example for the calculation of the inverse of 17 in GF(37). Since, finally $1{=}6 \cdot 37 - 13 \cdot 17$, $-13 \equiv 24$ in GF(37) is the inverse of 17 in this field.

i	0	1	2	3	4
r_i	37	17	3	2	1
q_i		2	5	1	2
s_i	1	0	1	-5	6
t_i	0	1	-2	11	-13
			$3 = 37 - 2 \cdot 17$	$2 = (-5)37 + 11 \cdot 17$	$1 = 6 \cdot 37 - 13 \cdot 17$

A version of the EEA for $GF(2^m)$ is in principle the same. Due to the fact that the divisions and multiplications can be substituted by shift operations, the performance is improved significantly. The concrete algorithm for the determination of multiplicative inverse in $GF(2^m)$ is shown in Algorithm 3.3. Hereby, the multiplication by x^q in Line 9 is equivalent to a shift left operation by q in a binary representation.

Algorithm 3.3: Extended Euclidean algorithm in $GF(2^m)$

```
   input  : $a(x), r(x) \in GF(2^m)$
   output: $a(x)^{-1} \bmod r(x)$
1  $r_0(x) = r(x)$; $r_1(x) = a(x)$;
2  $t_0(x) = 0$;$t_1(x) = 1$;
3  while $b(x) \neq 1$ do
4      $q = deg(r_0(x)) - deg(r_1(x))$;
5      if $q < 0$ then
6          $r_0(x) \leftrightarrow r_1(x)$; $t_0(x) \leftrightarrow t_1(x)$;
7          $q \leftarrow (-q)$;
8      end
9      $r_0(x) = r_0(x) + r_1(x) \cdot x^q$;
10     $t_0(x) = t_0(x) + t_1(x) \cdot x^q$;
11 end
12 return $(t_1(x))$
```

With every iteration of the loop at least one bit of r_0 or r_1 is cleared, from the highest to the lowest bits. This is why the algorithm requires up to $2m$ iterations.

A number of variations of the EEA have been presented in the literature whereby the improvements mostly concern special target computer architectures. Algorithm 3.4 shows a binary version of the EEA as it has been described in [9]. Hereby, only a single right shift operation is required, which substitutes the divisions of the original algorithm. In contrast to the variable shift operation by q in the Algorithm 3.3, the constant shift by one simplifies potential hardware designs. Multiplications are not required at all. Even though the binary EEA is very suitable for compact hardware designs, referring to [9] software implementations of this algorithm on general purpose processors provide a worse performance than the classic EEA. The theoretical complexity of the runtime is not distinct.

Algorithm 3.4: Binary extended Euclidean algorithm [9]

```
input  : $a(x), r(x) \in GF(2^m)$
output: $a(x)^{-1} \bmod r(x)$
1  $U \leftarrow a(x)$;$V \leftarrow r(x)$;
2  $T_0 \leftarrow 0$;$T_1 \leftarrow 1$;
3  while $U$ is even do
4      $U \leftarrow U/x$;
5      if $g1$ is even then $T_1 \leftarrow T_1/x$;
6      else $T_1 \leftarrow (T_1 + r(x))/x$;
7      if $u = 1$ then return($T_1$);
8      if $deg(U) < deg(V)$ then $T_0 \leftrightarrow T_1$; $U \leftrightarrow V$;
9      $U \leftarrow U + V$; $T_1 \leftarrow T_0 + T_1$;
10     Goto step 3;
11 end
```

3.5.2. Fermat's method

The basic principle of the Fermat inversion is that in finite fields repetitive multiplication eventually comes back to the origin element $a(x)$ with $a(x)^n \equiv a(x) \mod r(x)$. The idea is illustrated in Figure 3.11. The product that occurs just before closing the circle is 1, because $1 \cdot a(x) = a(x)$, and the preceding product is the multiplicative inverse $i(x)$ of $a(x)$, since $i(x) \cdot a(x) \equiv 1 \mod r(x)$.

The straightforward application of this idea would be the repeated multiplication until the 1 is found. Then the previous element is the inverse. This approach is prohibitive in huge fields such as $GF(2^m)$ as it would take infinite time to find the 1. But when a specific exponent is known that closes the cirlce, fast exponentiation algorithms can be applied, which can compute the inverse. The specific exponent for the inversion in $GF(2^m)$ can be derived from the Fermat's theorem, which says:

$$\text{If}\, gcd(a, p) = 1, \text{ then } a^{p-1} \equiv 1\ (mod\, p).$$

The equivalent meaning in $GF(2^m)$ is:

$$a(x)^{2^m - 1} \equiv 1$$

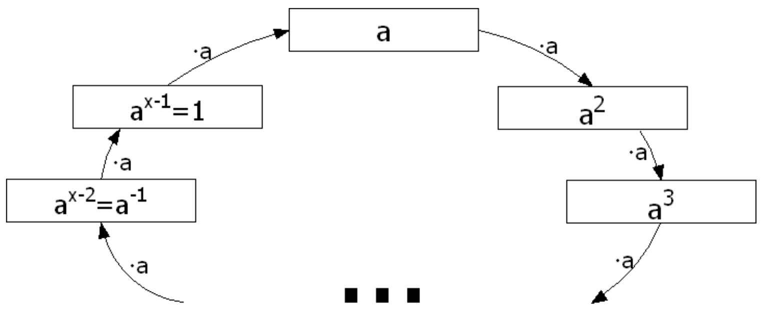


Figure 3.11.: The basic idea of Fermat based inversion: repetitive multiplication leads back to the element.

because every nonzero element $a(x) \in GF(2^m)$ is relative prime to the order 2^m. This implies that $a(x)^{2^m} = a(x)$.
Since

$$a(x)^{2^m-1} = a(x)^{2^m-2} \cdot a(x) \equiv 1,$$

it follows that

$$a(x)^{2^m-2} \equiv a(x)^{-1}. \tag{3.9}$$

Hence, the Fermat based algorithms compute the modular multiplicative inverse by modular exponentiation of $a(x)$ by $2^m - 2$.
Since simple $2^m - 2$ times multiplying $a(x)$ is not feasible, more sophisticated methods are required to calculate $a(x)^{2^m-2}$. In [10] a comprehensive survey of fast exponentiation methods is provided. Approaches that are suitable for finding the multiplicative inverse in $GF(2^m)$ are described below.

Square-and-multiply algorithm

An application of a binary exponentiation algorithm is the so called square-and-multiply algorithm [29]. It bases on the idea that the exponent can be written in a binary representation as $2^m - 2 = (11...110)_2$ with a length of m positions. It is a binary number with $m - 1$ ones and an appended zero. Considered this, one can derive

$$a(x)^{2^m-2} = \prod_{i=1}^{m-1} a(x)^{2^i}.$$

This means, that the factors $a(x)^{2^1}, a(x)^{2^2}, a(x)^{2^3}...a(x)^{2^{m-1}}$, which are determined by repeated squarings, are successively multiplied to obtain $a(x)^{2^m-2}$. Hereby, the $m-1$ factors correspond to the $m - 1$ set bits in the binary representation of the exponent. The algorithm is shown as Algorithm 3.5. This algorithm requires $m - 2$ multiplications and $m - 1$ squaring operations in $GF(2^m)$.

Algorithm 3.5: Square-and-multiply inversion

```
input  : $a(x) \in GF(2^m)$
output: $b(x) = a(x)^{-1}$
1 $a(x) \leftarrow a(x)^2$;
2 $b(x) \leftarrow a(x)$;
3 for $i \leftarrow 2$ to $m-1$ do
4     $a(x) \leftarrow a(x)^2$;
5     $b(x) \leftarrow b(x) \cdot a(x)$;
6 end
```

Improved Fermat based inversion

A classic approach to accelerate exponentiations is to use addition chains. The goal is to create the possibly shortest addition chain that generates the needed exponent. An addition chain is a sequence of positive integers e_0, e_1, e_2,..., e_k, where $e_0 = 1$, e_k is the desired exponent, and for every j, $1 < j \leq k$ there exist elements $i_1 < j$ and $i_2 < j$ where $e_{i_1} + e_{i_2} = e_j$. This means that every element in the chain is the summation of two earlier elements. In case of exponentiation, every element represents the multiplication of two previous elements, $a^{a_j} = a^{e_{i_1}} + a^{e_{i_2}}$. The first element is $a(x)^1$ and the desired element is $a(x)^{2m-2}$. Estimating the shortest addition chain for a given exponent is a NP hard problem [4]. Hence, finding the shortest chain for very long exponents such as $2^{232} - 2$ is not possible in practice due to the calculation time. Another problem is that standard processes of estimating minimal addition chains do not consider constraints as much faster execution of squarings compared to normal multiplications.

Despite this, it is possible to find addition chains that provide much less calculation effort than the standard square-and-multiply approach. Itoh and Tsujii [17] proposed an algorithm for the computation of multiplicative inverses where the number of multiplications is decreased to $N_M \leq 2\,[log_2(m-1)]$ whereby the number of squarings N_S remains at $m-1$.

Before starting the description of the method, two operations will be introduced for convenience.

$$\Delta : \qquad \Delta n = 2^n - 1 = 2^{\overbrace{(111...11)}^{n}{}_2}$$

$$\triangleleft : \qquad a(x) \triangleleft n = a(x)^{2^n}$$

is a n times repeated square operation.

Applying the new operations, the desired product for the inversion can be written as:

$$\begin{aligned} a(x)^{2^m-2} &= 2^{(\overbrace{111...11}^{m-1}0)_2} \\ &= \left(a(x)^{2^{(\overbrace{111...11}^{m-1})_2}} \right)^2 \\ &= a(x)^{\Delta(m-1)} \lhd 1 \end{aligned}$$

The computation of $a(x)^{\Delta m-1}$ can be simplified by reusing previously calculated terms, since $a(x)^{\Delta j} = a(x)^{\Delta i_1} \lhd i_2 \cdot a(x)^{\Delta i_2}$ in case $j = i_1 + i_2$. This is the basis for a shorter addition chain, which must be determined for every m separately.

In case of $m = 233$, an exemplary optimized chain is shown in Algorithm 3.6. It computes the inverse in the field $GF(2^{233})$ requiring 10 field multiplications and 232 squaring operations. In the given example the algorithm computes successively the powers of two of the Δ-operator, $a(x)^{\Delta 1}$, $a(x)^{\Delta 2}$, $a(x)^{\Delta 4}$, ..., $a(x)^{\Delta 128}$ and accumulates the corresponding partial powers to obtain $a(x)^{\Delta 232}$.
This approach is connected with significantly fewer multiplication operations, which substantially accelerates the computation of the multiplicative inverse. The major disadvantage is that the calculation chain must be determined separately for every field size.

Algorithm 3.6: Optimized Fermat based multiplicative inversion in $GF(2^{233})$

```
   input  : $a(x) \in GF(2^m)$
   output: $a(x)^{-1} = a(x)^{2^{233}-2}$
1  $a(x)^{\Delta 1} \leftarrow a(x) \lhd 1$;
2  $a(x)^{\Delta 2} \leftarrow a(x)^{\Delta 1} \lhd 1 \cdot a(x)^{\Delta 1}$;
3  $a(x)^{\Delta 4} \leftarrow a(x)^{\Delta 2} \lhd 2 \cdot a(x)^{\Delta 2}$;
4  $a(x)^{\Delta 8} \leftarrow a(x)^{\Delta 4} \lhd 4 \cdot a(x)^{\Delta 4}$;
5  $a(x)^{\Delta 16} \leftarrow a(x)^{\Delta 8} \lhd 8 \cdot a(x)^{\Delta 8}$;
6  $a(x)^{\Delta 32} \leftarrow a(x)^{\Delta 16} \lhd 16 \cdot a(x)^{\Delta 16}$;
7  $a(x)^{\Delta 64} \leftarrow a(x)^{\Delta 32} \lhd 32 \cdot a(x)^{\Delta 32}$;
8  $a(x)^{\Delta 128} \leftarrow a(x)^{\Delta 64} \lhd 64 \cdot a(x)^{\Delta 64}$;
9  $a(x)^{\Delta 192} \leftarrow a(x)^{\Delta 128} \lhd 64 \cdot a(x)^{\Delta 64}$;
10 $a(x)^{\Delta 224} \leftarrow a(x)^{\Delta 192} \lhd 32 \cdot a(x)^{\Delta 32}$;
11 $a(x)^{\Delta 232} \leftarrow a(x)^{\Delta 224} \lhd 8 \cdot a(x)^{\Delta 8}$;
12 return $a(x)^{\Delta 232} \lhd 1$
```

3.5.3. Shantz division

Inversions in Elliptic Curve cryptography are usually used as one step for division operations in the field. The inversion step is followed by a field multiplication to obtain $\frac{a(x)}{b(x)} = a(x) \cdot b(x)^{-1}$. Shantz [46] proposed an algorithm that determines the quotient directly without the combined inversion and multiplication.

The algorithm actually is a derivation of the EEA. During the iterations, the invariant $A \cdot a(x) \equiv U \cdot b(x) \mod r(x)$ is maintained. When A is reduced to 1, corresponding to the EEA, the obtained obtained result is: $a(x) \equiv U \cdot b(x) \mod r(x)$ so that $U = \frac{a(x)}{b(x)} \mod r(x)$.

Algorithm 3.7: Shantz division algorithm [46]

```
input  : $a(x),\ b(x) \in GF(2^m)$, irreducible $r(x) \in GF(2^m)$
output: $a(x)/b(y)$
1  $M \leftarrow r(x),\ A \leftarrow a(x),\ B \leftarrow r(x),\ U \leftarrow b(x),\ V \leftarrow 0$;
2  while $A \neq B$ do
3      if $A$ even then
4          $A \leftarrow A/2$;
5          if $U$ even then $U \leftarrow U/2$; else $U \leftarrow (U+M)/2$;
6      else if $B$ even then
7          $B \leftarrow B/2$;
8          if $V$ even then $V \leftarrow V/2$; else $V \leftarrow (V+M)/2$;
9      else if $A > B$ then
10         $A \leftarrow (A-B)/2$;
11         $U \leftarrow U - V$;
12         if $U < 0$ then $U \leftarrow U + M$;
13         if $U$ even then $U \leftarrow U/2$; else $U \leftarrow (U+M)/2$;
14     else
15         $B \leftarrow (B-A)/2$;
16         $V \leftarrow V - U$;
17         if $V < 0$ then $U \leftarrow V + M$;
18         if $V$ even then $V \leftarrow V/2$; else $V \leftarrow (V+M)/2$;
19     end
20 end
21 return $(U)$
```

The division as it is shown in Algorithm 3.7 has the same complexity as the standard EEA. Up to $2m-1$ iterations of the loop have to be performed. The benefit of this division algorithm is that no subsequent field multiplication is required, because it computes the final quotient, instead of the inverse of the divisor as the EEA.

4. Elliptic curve cryptography

Elliptic curve cryptography (ECC) has become the probably most efficient and powerful method in the field of public key cryptography. Although studies have emphasized the advantages, rather less applications in practice have been presented. Certainly, one reason is the difficulty of understanding the mathematical ideas and the realization of a reliable and high-performance implementation. The difficulty is caused by different interdependent layers and operations. The main layers are visualized in the pyramid of ECC operations, which can be seen in Figure 4.1. The basic layers up to the finite field operations have already been discussed in the previous chapters. In this chapter the operations on the ECs are described in more detail. It starts with the introduction of the two base operations, point addition and doubling, and goes on with scalar point multiplication, which is the most important ECC operation. The presented algorithm of the Montgomery point multiplication is determined to be implemented in hardware in the following chapters.

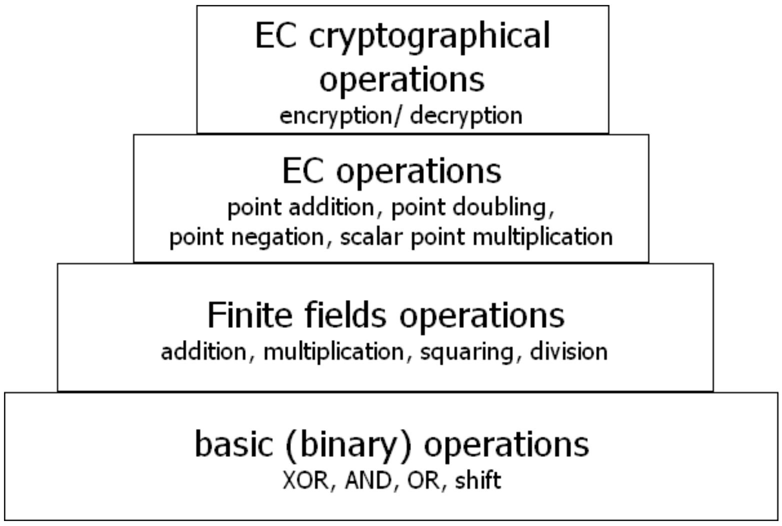

EC cryptographical operations
encryption/ decryption
EC operations
point addition, point doubling,
point negation, scalar point multiplication
Finite fields operations
addition, multiplication, squaring, division
basic (binary) operations
XOR, AND, OR, shift

Figure 4.1.: The pyramid of ECC as shown in [34]. The high cryptographic protocols apply the EC operations, in particular EC point multiplication. The EC operation use finite field operations, which apply the standard binary operations as XOR or AND.

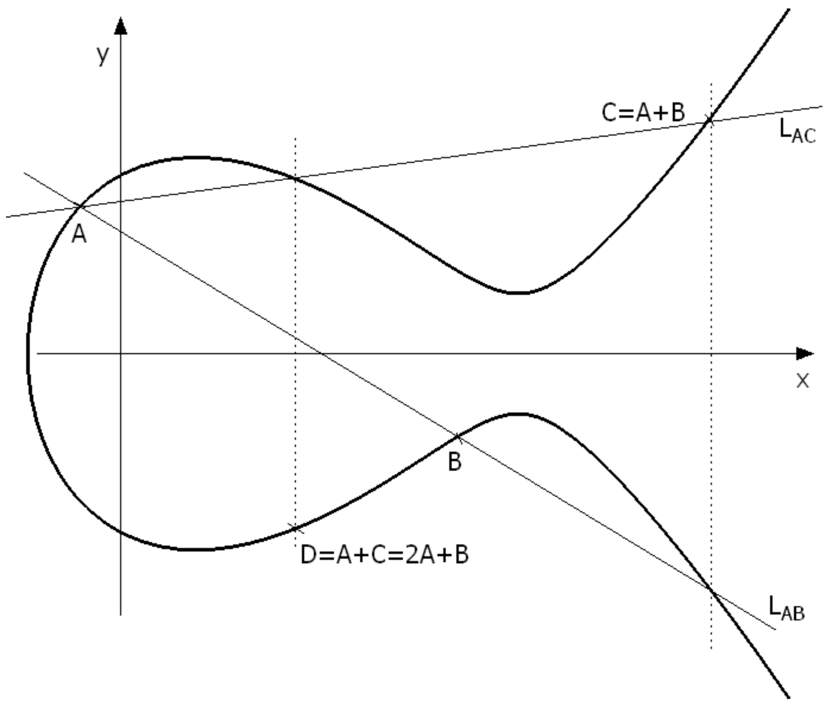


Figure 4.2.: ECC point addition: the summation of the two points A and B is geometrically constructed by determining third intersection point (L_{AB}) of the line through A and B, and mirroring this point on the x-axis to achieve C=A+B.

4.1. ECC operations

The two basic operations for ECC are:

Point addition: Determine the point $C = A + B$ on the curve for the given points A and B, whereby $A \neq B$.

Point doubling: Determine the point $C = A + A = 2A$ on the curve for the given point A.

The addition of two points on an elliptic curve on real numbers can be geometrically constructed by the following method:

1. Draw a line L through both input points.
2. Mark the third intersection point of L and the elliptic curve.
3. Invert the Y coordinate of the third intersection point.

The obtained point C is the summation point ($C = A + B$) of the two input points. The method is depicted in Figure 4.2.
The point doubling operation, which is shown in 4.3, is done analogous. The only difference is the first point of the enumeration, the construction of the intersection line L. For the point

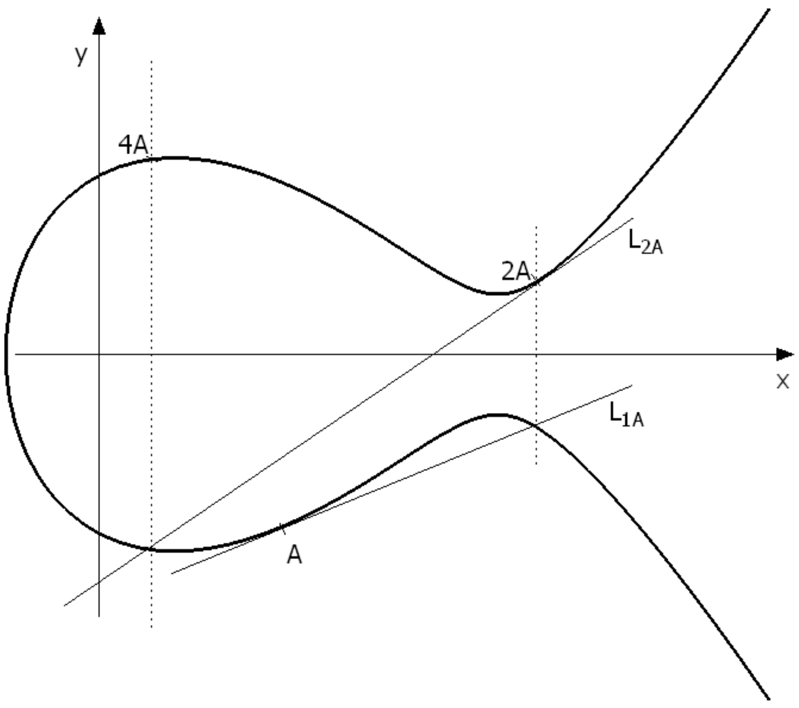


Figure 4.3.: ECC point doubling: the point A is geometrically doubled by determining the second intersection point (L_{1A}) of the tangent on the curve through A. This point L_{1A} is finally mirrored on the x-axis to achieve the point 2A.

doubling L is the tangent of the elliptic curve in A. As it is for the normal addition, the summation point for the point doubling is the inverse of the additional intersection point of L and the EC.

The principle approach is the same for all elliptic curves. The concrete formulas depend on the specific equation and properties of the curve. In prime fields $GF(p)$ where the elliptic curve is defined by the formula

$$y^2 = x^3 + ax + b,$$

the concrete calculation process for the addition of points $A = (x_A, y_A)$ and $B = (x_B, y_B)$ is:

$$\lambda = \begin{cases} \frac{y_B - y_A}{x_B - x_A} & \text{for point addition} \\ \frac{3x_A^2 + a}{2y_A} & \text{for point doubling} \end{cases}$$

$$x_C = \lambda^2 - x_A - x_B$$

$$y_C = -y_A + \lambda(x_A - x_C).$$

The coordinates x_C and y_C of the summation point C are calculated applying the value λ. This λ corresponds to the slope of the line L. The determination of λ is the only difference between point adding and doubling.

In the binary finite fields $GF(2^m)$ the operations are very similar. The elliptic curve for these fields is defined by

$$y^2 + xy = x^3 + yx^2 + b,$$

and the coefficient algebra is, due to the modulo 2 operations, simpler. Thus, the the concrete calculation is done as follows:

$$\lambda = \begin{cases} \frac{y_B + y_A}{x_B + x_A} & \text{for point addition} \\ x_A \frac{y_A}{x_A} & \text{for point doubling} \end{cases}$$

$$x_C = \begin{cases} \lambda^2 + \lambda + x_A + x_B + a & \text{for point addition} \\ \lambda^2 + \lambda + a & \text{for point doubling} \end{cases}$$

$$y_C = y_A + x_C + \lambda(x_A + x_C).$$

Hence, one division, one multiplication, one square and several XOR operations are required to perform point addition or doubling. Thus, all field operations that were described in the previous chapter are necessary for one basis ECC point operation. The by far most expensive operation is the division, which is required for the computation of λ. One approach to avoid the division and to accelerate the EC operations is to use projective coordinates.
In projective coordinates the affine coordinates (x, y) are mapped onto a three-dimensional point (X, Y, Z). The third dimension Z is the denominator of the division. Its current value is stored in each step instead of performing the division. The division is finally performed at the transformation from the projective coordinates back to the affine coordinates. Thus, many EC point operation can be subsequently performed without the need for a division.
Hereby a possible mapping is:

$$(x, y) \rightarrow (X, Y, Z) = (x, y, 1)$$
$$(X, Y, Z) \rightarrow (x, y) = (\frac{X}{Z}, \frac{Y}{Z})$$

In [27] alternative improved mappings were discussed. Different mappings lead to a different numbers of field operations for the additions and doublings. Table 4.1 shows the required number of multiplications and squarings for point addition and doubling. The fastest mapping is

$$(X, Y, Z) \rightarrow (x, y) = (\frac{X}{Z}, \frac{Y}{Z^2}).$$

Table 4.1.: Number of field operations for squaring and addition applying different projective coordinates [27].

Projective coordinates	Doubling #Mult.	Doubling #Sqr.	Adding #Mult.	Adding #Sqr.
$(x/z, y/z)$	7	5	12	1
$(x/z, y/z^2)$	4	5	9	4
$(x/z^2, y/z^3)$	5	5	10	4

The optimized algorithms for point doubling and addition on ECs on $GF(2^m)$ as they were proposed in [27] are shown in 4.1 and 4.2, respectively.

Algorithm 4.1: Projective elliptic doubling algorithm

```
    input  : $A = (X_A, Y_A, Z_A)$
    output: $2A = (X_C, Y_C, Z_C)$
1   $Z_C \leftarrow Z_A^2 \cdot X_A^2$;
2   $X_C \leftarrow X_A^4 + b \cdot Z_A^4$;
3   $Y_C \leftarrow bZ_A^4 \cdot Z_C + X_C \cdot (a \cdot Z_B + Y_A^2 + bZ_A^4)$;
4   return $((X_C, Y_C, Z_C))$
```

Algorithm 4.2: Projective elliptic adding algorithm

```
    input  : $A = (X_A, Y_A, 1)$, $B = (X_B, Y_B, Z_B)$
    output: $A + B = (X_C, Y_C, Z_C)$
1   $T_1 \leftarrow Y_A \cdot Z_B^2 + Y_B$;
2   $T_2 \leftarrow X_A \cdot Z_B + X_A$;
3   $T_3 \leftarrow T_2 \cdot Z_B$;
4   $T_4 \leftarrow T_2^2 \cdot (T_3 + a \cdot Z_B^2)$;
5   $T_5 \leftarrow T_1 \cdot T_3$;
6   $Z_C \leftarrow T_3^2$;
7   $X_C \leftarrow T_1^2 + T_4 + T_5$;
8   $T_6 \leftarrow X_C + X_A \cdot Z_C$;
9   $T_7 \leftarrow X_C + Y_A \cdot Z_C$;
10  $Y_C \leftarrow T_5 \cdot T_6 + Z_C \cdot T_7$;
11  return $((X_3, Y_3, Z_3))$
```

4.2. EC point multiplication

Having the point addition operations defined and efficient algorithms found, now the multiplication can be defined. The elliptic curve point multiplication (ECPM) is not a multiplication of two points, but a scalar multiplication of a point on the EC by an integer.

The basic idea is to perform addition operations repeatedly to obtain the product kP, whereby k is the integer and P is the point on the EC:

$$\mathrm{kP} = \overbrace{\mathrm{P} + \mathrm{P} + ... + \mathrm{P}}^{\text{k times}}.$$

In cryptographic applications k is randomly selected from the interval $[1, q-1]$ whereby q is the group order of the EC. It can be assumed that $k \approx 2^m$ and therefore k has a length of hundreds of bits. Hence, the straightforward repeated addition is not feasible.

Classic double-and-add algorithm

The factor k can be written as summation of powers of two:

$$k = \sum_{i=0}^{t} k_i 2^i,$$

whereby the vector $(k_t, ..., k_1, k_0)_2$ is the binary representation of k.
The ECPM can therefore be written as:

$$kP = \sum_{i=0}^{t} k_i (2^i) P.$$

Algorithm 4.3: Double-and-add kp multiplication [13]

```
  input  : $k = (k_{t-1}, ..., k_1, k_0)_2$ with $k_{t-1} = 1$, $P = (x, y) \in E(GF2^m)$
  output: $kP$
1 $Q \leftarrow 0$;
2 for $i = t-1$ downto 0 do
3     $Q \leftarrow 2Q$;
4     if $k_i = 1$ then
5         $Q \leftarrow Q + P$;
6     return $Q$
7 end
8 return $(Q = Mxy(X_1, Z_1, X_2, Z_2))$
```

The terms $2^i P$ are efficiently computed by repeated point doublings. Based on this, an algorithm can be derived that determines the resulting points $2^i P$ of the repeated point doublings and adds the points where the corresponding $k_i = 1$. The algorithm, which is shown in Algorithm 4.3, requires $t-1$ point doublings and up to $t-2$ point additions. According to Table 4.1, the fastest mapping requires

$(t-1) \cdot 5$	squaring operations for the point doublings,
$(t-1) \cdot 4$	multiplication operations for the point doublings,
up to $(t-2) \cdot 4$	squaring operations for the point additions,
up to $(t-2) \cdot 9$	multiplication operations for the point additions.

This implies that applying the double-and-add approach requires up to $(9t - 13)$ squaring operations and up to $(13t - 22)$ field multiplications for a ECPM. For the conversion of the projective coordinates back to affine coordinates $((X, Y, Z) \rightarrow (X/Z, Y/Z^2))$ additional three multiplications, one inversion and one squaring operation must be performed. Assuming that in average every second bit of k is set and $t \approx m$, the total average effort is approximately $8.5m$ multiplications, $7m$ squarings, and one inversion.

Montgomery point multiplication

The fastest known ECPM algorithm for ECs over $GF(2^m)$ is Montgomery point multiplication (MPM). The approach was introduced in [31]. An improved version, which is described in the following, was presented in [28]. The algorithm, which is shown as Algorithms 4.4 and 4.5 requires approximately $6m$ multiplications, $5m$ squaring operations, and one inversion.
It is based on the idea that it is not required to compute the y values during the double-and-add phase, since the final value y_1 can be determined after the loop out of the x values of the points $P_1 = kP = (x_1, y_1)$ and $P_2 = (k+1)P = (x_2, y_2)$ for the given point $P = (x, y)$:

$$y_1 = y + \frac{x_1 + x}{x} \left[(x_1 + x)(x_2 + x) + x^2 + y\right] + y \tag{4.1}$$

Algorithm 4.4: Basic idea of Montgomery kP multiplication [28]

```
    input  : $k = (k_{t-1}, ..., k_1, k_0)_2$ with $k_{t-1} = 1$, $P = (x, y)$
    output: $kP$
 1  $P_1 \leftarrow P$; $P_2 \leftarrow 2P$;
 2  It is always maintained that $P_2 = P_1 + P$
 3  for $i = t - 2$ downto 0 do
 4      $T \leftarrow P_2$;
 5      if $k_i = 1$ then
 6          $P_2 \leftarrow P_1 + P_2$; $P_1 \leftarrow 2T$;
 7      else
 8          $P_2 \leftarrow 2P_1$; $P_1 \leftarrow P_1 + T$;
 9      end
10  end
11  convert to affine coordinates and compute $y_1$ as in Equation 4.1
12  return ($kP = convert_{xy}(P, P_1, P_2)$)
```

The basic idea can be seen as Algorithm 4.4. In this algorithm, the relationship $P_2 = P_1 + P$ is valid over the whole process. This is why the the x-coordinate of the point $P_3 = (x_3, y_3) = P_1 + P_2$ can be computed without requiring any y-coordinates using

$$x_3 = x + \frac{x_1}{x_1 + x_2} + \left(\frac{x_1}{x_1 + x_2}\right)^2. \tag{4.2}$$

The division in the equation can be avoided by applying the projection $(X, Z) \rightarrow x = \frac{X}{Z}$. The conversion back to affine coordinates is performed once at the end of the algorithm. In the double-and-add phase, only the representatives for x_1 and x_2, which are the tuples (X_1, Z_1) and (X_2, Z_2) respectively, must be stored. The complete algorithm is depicted as Algorithm 4.5.

Algorithm 4.5: Complete Montgomery kP multiplication [28]

```
input  : $k = (k_{t-1}, ..., k_1, k_0)_2$ with $k_{t-1} = 1$, $P = (x, y) \in E(F_{2^m})$
output: $kP = (x_1, y_1)$
1  $X_1 \leftarrow x$; $Z_1 \leftarrow 1$; $X_2 \leftarrow x^4 + b$; $Z_2 \leftarrow x^2$;
2  for $i = t - 2$ downto 0 do
3      if $k_i = 1$ then
4          $T \leftarrow Z_1$; $Z_1 \leftarrow (X_1 Z_2 + X_2 Z_1)^2$; $X_1 \leftarrow xZ_1 + X_1 X_2 T Z_2$;
5          $T \leftarrow X_2$; $X_2 \leftarrow X_2^4 + bZ_2^4$; $Z_2 \leftarrow T^2 Z_2^2$;
6      else
7          $T \leftarrow Z_2$; $Z_2 \leftarrow (X_2 Z_1 + X_1 Z_2)^2$; $X_2 \leftarrow xZ_2 + X_1 X_2 T Z_1$;
8          $T \leftarrow X_1$; $X_1 \leftarrow X_1^4 + bZ_1^4$; $Z_1 \leftarrow T^2 Z_1^2$;
9      end
10 end
11 $x_1 \leftarrow X_1 / Z_1$;
12 $y_1 \leftarrow y + (x + x_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1 Z_2)]/(xZ_1 Z_2)$;
13 return $((x_1, y_1))$
```

The algorithm requires one addition and one doubling operation in every iteration of the loop, independent whether k_i is set or not. An point addition hereby implies four multiplications and one squaring operation, and the doubling requires two multiplications and four squarings. The loop is executed $t - 1$ times, whereby t is the binary length of k. The initialization needs three squarings, and the computation of the affine coordinates (x_1, y_1) requires ten multiplications, one squaring, and one inversion. Hence, the total number of operations is $(t - 1)(6M + 5S) + 3S + 10M + 1I$. Since $t \approx m$, one can approximate the total effort to $6m$ field multiplications, $5m$ squarings and one inversion.

5. Hardware designs for ECC

In this chapter hardware solutions for the mathematical algorithms and approaches, described in the previous chapter are elaborated. First, a general processor design is presented that provides a flexible base for further development. This is followed by the discussion of each of the individual function blocks. Thereby, the design space for the individual function blocks and the ECC processor as a whole is explored.

5.1. Hardware development work flow

Before starting with the concrete discussion of hardware designs for ECC, the work flow, the design applications and target platforms that are used for this thesis are described.
First, the considered algorithms were realized by pure software implementations. These C-implementations that are running on a standard PC practically prove the correctness of the algorithms and provide intermediate results for debugging of the hardware designs developed in the following. The correctness of the results is ensured by comparisons with results delivered by the MIRACL[45] library.
Based on the C routines, functional blocks in VHDL and corresponding test benches were build. The designs were simulated by Cadence NC-Sim ver. 5.00[14]. With the simulation results it can be verified that the VHDL blocks, which are running in a behavioral simulation, are performing the operations correctly.
Until this point the design flow is identically for both implementations for the FPGA version as well as for the ASIC version. Our primary purpose is a ASIC implementation in our in-house $0.25\mu m$ CMOS-Technology[16]. Concurrently, the designs should be synthesized and tested on a FPGA. In this work a Xilinx XC2VP70-7[56] is used. Running designs on a FPGA allow to test tangible implementations at real speed. Additionally, the synthesized designs provide comparable numbers such as speed and used area on the FPGA. This is necessary, since the majority of the hardware designs that have been reported in the literature were implemented on FPGAs. The FPGA designs described in this thesis are synthesized and implemented by Xilinx ISE 6.1[57]. The result numbers of the FPGA designs are taken from the reports generated by this tool.
But since the main focus is on the ASIC implementation, the discussion of implementations

and design alternatives is always done in terms of silicon properties. Results of FPGAs are only noted when remarkably different behavior compared to the ASICs is to expect.
The ASIC implementations are synthesized with a library of our in-house $0.25\mu m$ CMOS-Technology by Synopsys Design Analyzer ver. 2004.12. As result of this step netlists with concrete timing informations for our technology are obtained. The netlists with timing information can be tested and verified again with NC-Sim. Based on the netlists the Design Analyzer reports area requirements and the maximum speed of this pre-layout design. Unless it is explicitly written, the area and the timings noted in the text refer to these reports.
The power consumption is determined by Cadence PrimePower ver. 2005.06[49]. Primepower provides a gate level power estimation based on parameters of the technology library and real test pattern. This results in very precise power estimations, since the real transitions for the calculation are considered instead of merely statistical assumptions.
As final step, the layout is build out of the synthesized netlists applying Cadence Encounter v3.30[15]. With the resulting post-layout design the effects of routing and clock trees can be determined. Eventually, a post-layout simulation with NC-Sim verifies the correct functionality of the design with the given speed.

5.2. Processor architecture

The primary target of the architecture of the ECC processor discussed in this work is to provide a flexible system that can easily be used and adapted to various algorithms. It is not in first place to obtain the most efficient and optimal implementation for every approach, that is investigated.

The processor architecture that provides the basis for the hardware design is shown in Figure 5.1. The functional units (FU) are communicating via a transport network, which connects the components. The access on the transport network is controlled by a controller unit. Every clock cycle it determines which function unit is supposed to write on the bus and which components read the data. This approach corresponds to the transport triggered architecture (TTA) [37].

Transport triggered architecture

TTA processors are not programmed by describing the operations of the FUs but the data transport between the units.

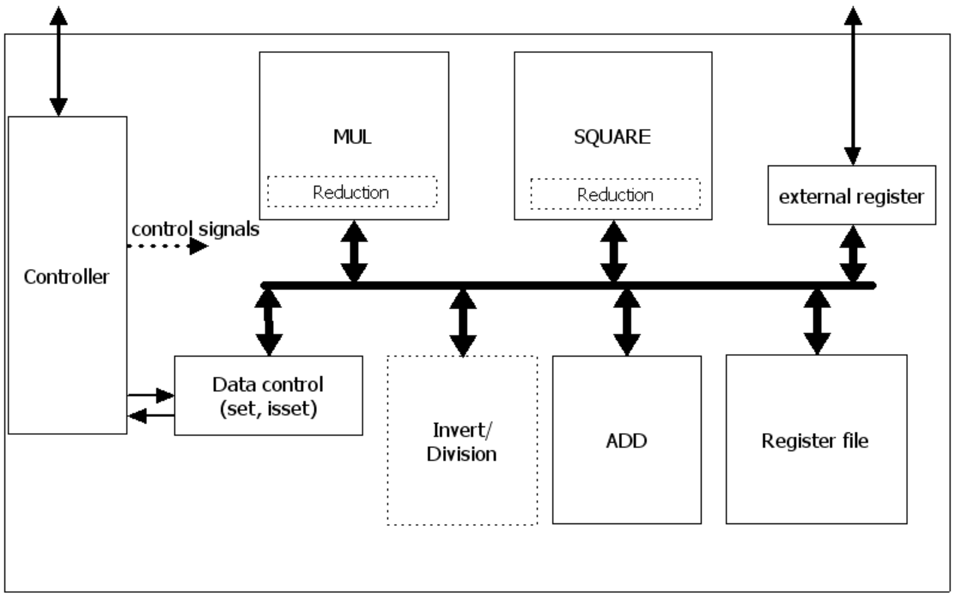


Figure 5.1.: The intended basic ECC design with Multiplier (MUL), square unit, adder (ADD), external register, internal registers, inverter and data control as functional units. These FUs are controlled by the controller unit. The data words are transmitted over a system bus structure. The Invert/Divison block is dotted because it is not sure if it really required.

For an addition ($a := a + b$) this means that instead of the operational instruction $add(register_a,\ register_b)$ the following transports will be described:

$$register_a \rightarrow ADD_{in0}$$
$$register_b \rightarrow ADD_{in1}$$
$$ADD_{result} \rightarrow register_a$$

TTA has properties that are very beneficial for application specific hardware and cryptographic processors in particular. The major benefits of the TTA are :

Explicit parallelism: Every FU works independent from the others. This allows to set up an FU while another one is calculating and to perform concurrent operations. Which FUs are calculating is explicit determined at compile time.

Forwarding network: The transport over the transport network from an FU to the register file generally is the same as the transport from one FU to another one. This allows faster execution times because on critical paths the data is transfered directly between the FUs without need for buffering it in the register file. The forwarding is explicitly controlled by the program.

Flexibility: Adding or removing FUs is very easy. Once an FU is connected to or disconnected from the transport network, only the software must be adapted. There is no need to change the network or the register file.

Especially the flexibility in combination with the straightforward transport network are properties that satisfy the needs of the evaluations to be done in this work.
But the TTA approach has some disadvantages. In the small example above, one can see that the controller unit or the compiler, which generates the program that controls the processor, must be very sophisticated. It is not only important to control the mutually exclusive access to the bus structure but to know the execution times of the FUs and to ensure that no results are overwritten before they are used. In the addition example it would lead to incorrect results if the result is fetched before the addition is done. Various problems of TTA-compilers have been addressed in [19]. It should not be part of this thesis to implement a TTA compiler. Since the considered ECC algorithms are not very extensive, manual work is sufficient to obtain efficient translations into the processor language.

Basic ECC processor design

Figure 5.1 already shows the whole processor design and the main FUs, intended to be implemented into the chip. The blocks with solid surface are integral part of the design while the integration of the dotted-surface blocks depends on used algorithms and results of further investigations.
The considered units of the chip are:

Polynomial multiplier: Performs a polynomial multiplication in $GF(2^m)$. The reduction is considered to be part of the multiplication unit.

Square unit: Since the squaring operation is much cheaper than the multiplication, a special square unit is feasible.

Addition unit: Performs a polynomial addition.

Control unit: Controls the assignments to the bus. It is also the place where the program runs, which controls the whole processor. The EC algorithms as point additions, doublings and multiplications are performed by this block.

Data control unit: It is the connection port between data path and control path. This block can set bits in a data word and can check whether specified bits are set. This affects the execution path in the control block.

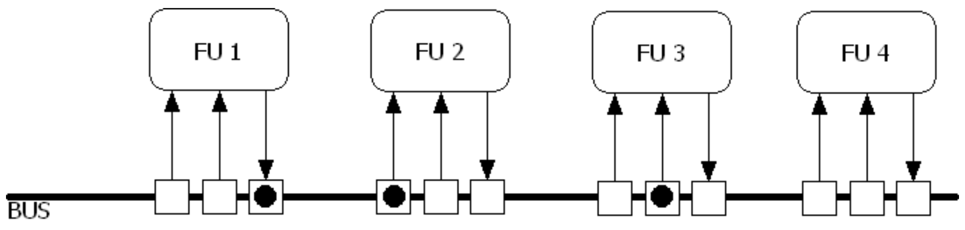


Figure 5.2.: Model of the TTA. Functional units are connected to the bus by sockets that can be enabled and disabled. Enabled input sockets not only transport the data but trigger the operations. In this example FU1 transfers the data to FU2 and FU3.

Registers: Internal buffer for data words. Since data words are long, as few as possible of these registers should be spent.

External register: It represents the port to the external world.

Inversion/division unit: Performs one field division or inversion. It requires further investigations to decide whether this block is necessary. Alternatively this operations could be performed by the control unit applying the other components.

This listing of functional units and the described design is an introducing overview of the units and components discussed below. The kind and number of components in the final design depends on the results of this evaluations.

5.2.1. Transport network

A crucial design decision is the choice of the kind of the internal transport bus. In case of the ECC design it obtains special importance as data words of several hundred of bits must be transported. Since TTA has been chosen, the actual bus only needs to transport data. Neither address logic nor logic for requesting and granting the bus access is required. The bus access controll is predetermined and performed by the controller unit.

Figure 5.2 depicts the principle idea of the bus. In the shown example four FUs with two inputs and one output each are connected to the bus. The controll unit determines which port is plugged. In the example, the output of FU1 is forwarded to the first input of FU2 and the second input of FU3. The question is now, how to realize this idea practically.

Multiplexed buses

The first implementation idea is a multiplexed bus. The approach is depicted in Figure 5.3. Here, all write operations on the bus are sent to one large multiplexer which selects the bus signal. This signal represents the bus and is connected to all possible read ports. The input

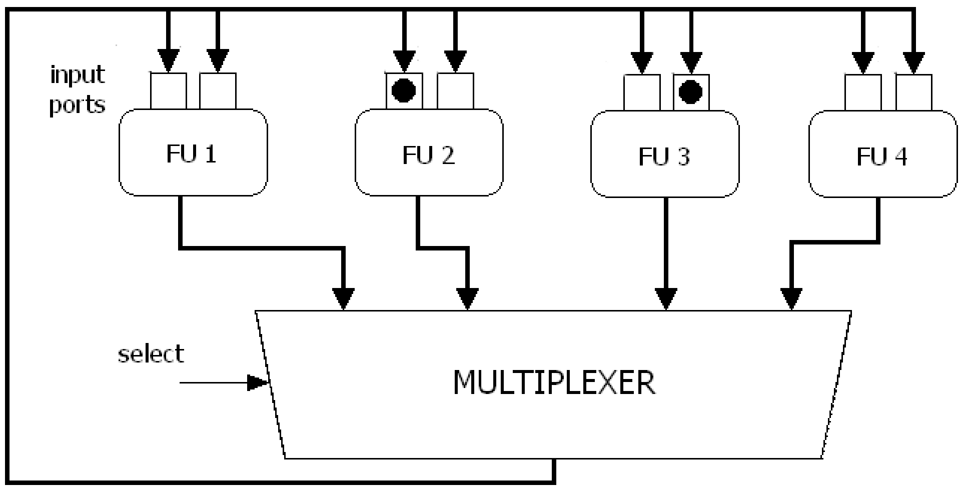


Figure 5.3.: Implementation of the TTA bus with a large multiplexer. The multiplexer decides which word is set onto the bus. The word on the bus can be fetched by the input ports that are controlled over select signals.

ports can separately fetch the word from the bus.

The advantages of the multiplexed bus are:

- It is straightforward implementation of the basic idea. First, the bus signal is multiplexed, then it is led to all input ports. Mutually exclusive access on the bus is guaranteed.
- It is easy to understand and to debug. Also the testing is not problematic.
- It is independent from the technology. This approach can be realized with standard gates.

But there are also some disadvantages:

- The multiplexer block can become very large. For the ECC, ten writing devices with 500 bit each are possible.
- Routing is a problem. First all output signals must be connected to the multiplexer, then the output of the multiplexer is connected to the input signals.
- Lack of flexibility. When an FU should be added or removed, the multiplexer must be changed.

Tristate buses

Another approach is a tristate bus. On these buses, the writing devices are directly connected to the bus. The outputs that are not supposed to write propagate a high impedance signal. This way, one writing signal can be read on the whole bus. In the example that is shown in Figure 5.4, the writing port of FU1 propagates its signal to the bus, whereas the other outputs present a high impedance ('Z').

This approach obtains the following advantages:

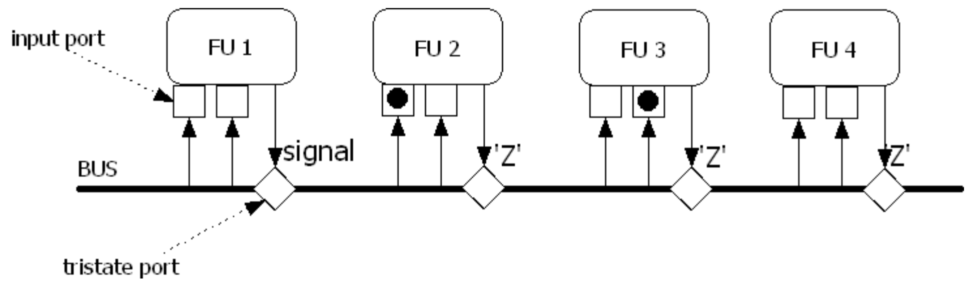


Figure 5.4.: Implementation of the TTA bus with tristate bus. The output determined to write applies the signal to the bus while all other potential output ports must present a high impedance. The input ports are independently controlled by control signals.

- No large multiplexer required.
- Routing should be much easier. Figuratively, it is just one wire that is connected to all inputs and outputs of the FUs.
- Very flexible because adding an FU is as easy as connecting it to the wire. No other hardware block must be changed.

But there are also some negative properties:

- Tristate drivers, which write either the signal or 'Z', are large and slow. For the 10x500 bit bus, 5000 of these tristate drivers are required.
- Lack of observability and testability. Especially the integration of scan chains is problematic on tristate buses.
- Dependence from technology. Not every technology provides tristate gates.

Comparison

For the comparison of multiplexer and tristate bus, we implemented an experimental 571-bit ECC system using a multiplexer bus as well as one with a tristate bus. The resulting size and latency on an ASIC and FPGA were measured and are shown in Table 5.1. It is interesting to see that the multiplexer bus requires less gates and is therefore smaller after synthesizing. Because of the reduced routing problem, the tristate bus design could be routed with higher core utilization, so that finally both approaches are resulting in similar area after routing (5.8mm^2 to 5.7mm^2). For the ASIC implementations the resulting speed does not differ mentionable.
On the FPGA the timing differs in evidence. While the longest path in the multiplexer design takes $15ns$, it is $18ns$ for the tristate bus. In contrast many LUTs (look-up tables) could be substituted with TBUFs (tristate buffer), which are rarely used in most designs. That indicates the tristate design is more space efficient on FGPAs.

Table 5.1.: Comparison of tristate and multiplexer bus for a 571 bit system

on ASIC	Size[mm^2]	After routing[mm^2]	Size [kgates]	Speed [ns]
mux	3.5	5.8	116	10
tristate	3.8	5.7	124	10
on FPGA	**4LUTs**	**FFs**	**TBUFs**	**Speed [ns]**
mux	20055	7459	0	15
tristate	15167	7318	5710	18

Partitioned bus structure

Since routing can be a serious problem in particular for large ECs on ASICs, it is worth mentioning the bus partitioning technique. Instead of transmitting a complete word in one clock cycle, the data word will be partitioned into n parts. Thus, transmitting one word will take n clock cycles.

Speed is not the only disadvantage of this approach. Additional space is required for the switching to and from the bus. Also additional flip flops or modified logic is needed in case combinatoric logic needs the complete data word for the calculation.

For the estimation of the impact of the partitioned bus structure, an adapted design with a 144 bit physical bus as an alternative to the 571 bit multiplexer has been implemented. This design needs four clock cycles for the transmission of one data word. The synthesis results of this design are shown in Table 5.2. Compared to the original multiplexer design on the ASIC it is noticeable that the small bus design takes more gates. Despite that, the area after routing is twelve percent smaller than the one with the large bus. On FPGAs no benefits could be discovered since numbers for logic and flip flops are larger than those of the corresponding large bus design. Also the timing is affected negatively, due to the additional switching logic.

Table 5.2.: Synthesis results for the partitioned 4x144 bit bus.

on ASIC	Size[mm^2]	After routing[mm^2]	Size [kgates]	Speed [ns]
part	3.6	5.1	121	10
on FPGA	**4LUTs**	**FFs**	**TBUFs**	**Speed [ns]**
mux	22402	8010	0	16

5.3. Reduction

In Section 3.4 three methods of reducing overlapping bits were described. In the following, practical implementation of those three approaches are discussed, in order to find the most efficient reduction algorithm for the ECC design.

Polynomial division

The first described method for the reduction process is the computation of the remainder of a polynomial division as demonstrated in Algorithm 3.1. A hardware implementation of this approach is possible. All one needs are two large registers which store $a(x)$ and $r(x)$, whereby the r-register should be able to perform a shift right operation, a XOR-operation block that executes the operations of the inner loop and a logic that controls the condition. Even though a possible hardware design is clear, we have not implemented it, because the solution would require $m-1$ iterations of the inner loop. This would imply that $m-1$ clock cycles are needed to perform a reduction. Since the reduction has to be performed after every multiplication and squaring operation, the polynomial reduction is not a feasible method for the reduction.

Multiplicative reduction

The second described method is the multiplicative reduction. Two multiplications of the overlapping part with the reduction polynomial and two attached XOR operation are required for for the whole reduction process. Since the reduction must be performed after every multiplication, with this method the actual multiplication is followed by two other multiplications to reduce the result. Even though it does not sound very efficient, it is sometimes applied for flexible designs [7] and very area efficient implementations [44], because the method is not bound to an irreducible polynomial nor special hardware blocks are required. For this approach an adaption of the data path around the multiplier is sufficient. This approach is contemplated in Section 7.2, where flexible reduction methods are evaluated.

Fast reduction

The third method is the fast reduction which performs a reduction within one step. For a specific polynomial a direct combination of XOR operations hereby reduces the long word as it was described in Section 3.4.3. This allows to implement a combinatorial logic that computes the reduction within one clock cycle. Hence, this method is also known as combinatorial or hardwired reduction.
Beside the speed and efficiency this approach is also connected with a serious problem. For every field size and every reduction polynomial a separate logic must be realized. Despite this

Size [bit]	Area [mm^2]
163	0.045
233	0.034
283	0.076
409	0.059
571	0.159

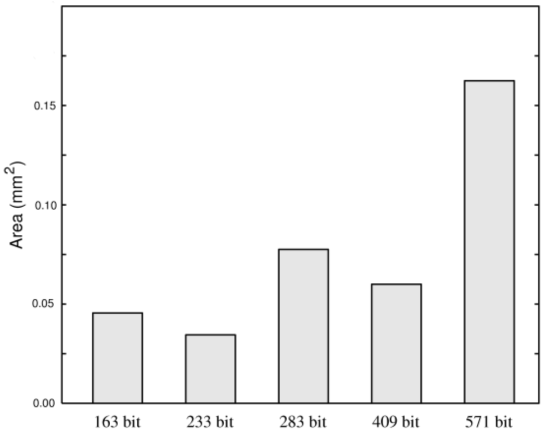


Figure 5.5.: Area consumption of combinatorial reduction blocks for the five binary NIST ECs B-163 to B-571. The area is not increasing steadily, due to reduction polynomials of different complexity.

drawback, it is very often used for ECC hardware implementations [43]. This is due to the surpassing speed whereby the required area and additional logic is very low. Applying this method, the reduction has been rendered to a quite cheap operation.
For the evaluation of this method, we implemented combinatorial reduction blocks for each of the five NIST polynomials and measured the resulting area. From the results which are depicted in Figure 5.5 one can see that even the reduction block for the 571 bit field is with $0.16mm^2$ relatively small. It is interesting that the 233 bit reduction is smaller than the 163 bit version and the 409 bit block is smaller than the 283 bit reduction block. This is due to the complexity of the reduction polynomial. Table 3.8 shows that the 163, 283 and 571 bit field irreducible polynomials are pentanomials while the 233 and 409 bit polynomials are trinomials. This is why especially area efficient ECC hardware implementations take use of trinomials to reduce the complexity of the system.

5.4. Polynomial multiplication

In this section the methods for polynomial multiplication, which were introduced in Section 3.2 are implemented and compared. This way the impact of the different approaches for the computation of the multiplication are investigated. The main focus hereby is placed on improved implementations of the iterative Karatsuba approach. Finally, the results are compared with special emphasis on required silicon area, since area consumption is a good practical indicator for the number of operations integrated onto the design.

Table 5.3.: Silicon areas for classic polynomial and Karatsuba multipliers

Factor size	Area [mm^2]	
[bit]	CPM	CKM
4	0.002	0.003
8	0.007	0.010
16	0.030	0.033
32	0.12	0.12
64	0.47	0.34
128	2.0	1.0
256	7.7	3.1

5.4.1. Combinatorial multipliers

Combinatorial multipliers are multipliers that deliver results instantaneous, independent from clock cycles. This implies that for sequential circuits factors can be set to the inputs of the multiplier and the product can be used within the same clock cycle. The silicon areas for combinatorial multiplication blocks for the classic methods CPM and CKM in different sizes are shown as Table 5.3.

It is interesting that the size of smaller multipliers is less for CPM than for CKM. This affirms the theoretical considerations from Section 3.2.5, where the operational expenses of the multiplication methods were compared. The practical results in particular affirm the considerations of complexity for the operations. A theoretical complexity of $O(n^2)$ for CPM and $O(n^{1.58})$ for CKM was derived. This means for a doubled input size the resulting area should about quadruple (2^2) or triple ($2^{1.58}$), respectively. These quotients fit very well to the results represented in Table 5.3.

The fact that CPM results in smaller units for shorter bit lengths can be applied in a way that the small CPM units are used as base for the recursive Karatsuba. This leads to significant smaller multipliers. The theoretical results of this approach were shown in Figure 3.4. These assumptions should be practically verified by implementing 128 and 256 bit recursive Karatsuba multipliers that are based on CPM of different bit lengths. The results of this investigation are depicted in Table 5.4.

Table 5.4.: Area consumption of recursive Karatsuba multipliers based on different CSM units.

Base [bit]	128 bit [mm^2]	256 bit [mm^2]
1	1.03	3.09
2	0.87	2.70
4	0.79	2.46
8	0.90	2.76
16	1.00	3.08

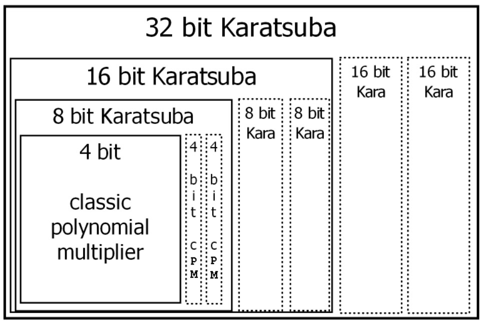


Figure 5.6.: Structure of a 32 bit recursive classic Karatsuba multiplier. It applies three 16 bit CKM which apply three 8 bit CKMs, each, that finally apply three 4 bit CPMs. The thin dotted blocks correspond to the thick ones.

As expected, the area is reduced by applying CPMs as basic operation of the recursive Karatsuba process. It is also not surprising that for larger base blocks this advantage is negated. The optimum for the investigated multipliers is a CPM unit that processes 4 bit factors. This means 8 bit multiplier use 4 bit CPM, and are applied themselves in 16 bit Karatsuba multipliers which are used in 32-bit versions and so on. The structure of a 32-Bit version of such a multiplier is shown in Figure 5.6.

Another approach presented in the theoretical considerations is the RAIK method, which splits the data word in four parts (in contrast to two parts as in CKM) and applies the results of iterative Karatsuba to reduce the number of XOR operations. Based on the 4 bit CKM, which is known as optimal, a 16 bit RAIK as well as a 64 bit and 256 bit version were implemented. The exemplary structure of the 64 bit RAIK multiplier unit is shown as Figure 5.7. In contrast to the 64 bit or 256 bit version, for the 128 bit RAIK the configuration is not that obvious, because the repeated splitting into four parts does not lead to the optimal 4 bit CSM. This is why different configurations of the underlying 8 bit partial multiplier have been evaluated. The results are shown as Table 5.5. The best solution is to split the 8 bit multiplier in two 4 bit parts by the CKM method and finally use the 4 bit CPM blocks. It is very interesting that the best 128 bit combinatorial multiplier is a recursive combination of all described multiplication methods - RAIK, CKM and CPM.

Table 5.5.: Area consumption of recursive Karatsuba multipliers based on different classic multipliers.

Multiplier	Area [mm^2]
raik64_raik16_cpm4	0.23
raik256_raik64_raik16_cpm4	2.16
raik128_raik32_cpm8	0.86
raik128_raik32_ckm8_cpm4	0.75
raik128_raik32_raik8_cpm2	0.79

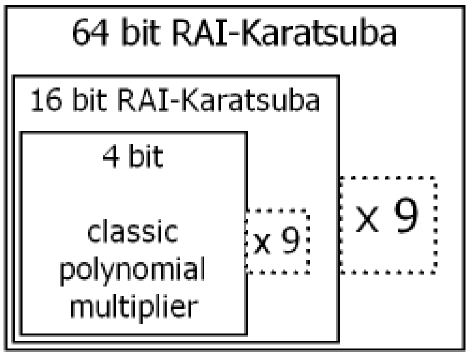


Figure 5.7.: Structure of a 64 bit RAIK multiplier that bases on nine 16 bit RAIKs which apply 4 bit CPMs

5.4.2. Iterative Karatsuba designs

Combinatorial multipliers become very large and slow for longer factors. To counter this problem, in [5] the iterative Karatsuba design was presented. It uses smaller combinatorial multiplication blocks, and applies them repeatedly following the Karatsuba method in order to perform a larger polynomial multiplication. The IKM design for a 233 bit multiplication unit presented in [5] is the starting point for the investigations concerning improved IKM designs that will be the most important functional unit of the whole ECC design.
It consists of three main parts (Figure 5.8):

Selection logic: selects and combines the factors of the partial multiplication.

Partial multiplier: performs the partial multiplication within one clock cycle.

Accumulation block: computes the final product by accumulating the partial products.

The partial multiplier is a combinatorial multiplier as described in the last section. The higher the segmentation the smaller is the required partial multiplier.
Probably the greatest benefit of the iterative Karatsuba is the increased flexibility concerning required total time, potential clock frequencies and needed silicon area. The results of the initial 233 bit multiplier designs as reported in [5] are shown in Table 5.6. One can see the one-clock multiplier requires the least total time but is by far the largest one. Interestingly

Table 5.6.: Parameters of different 233 bit iterative Karatsuba multipliers [5]

Multiplier setup	Area [mm^2]	Required clock cycles	Period [ns]
recursive	6.28	1	19
2 segments	2.18	3	15
4 segments	1.52	9	10
8 segments	1.67	27	9

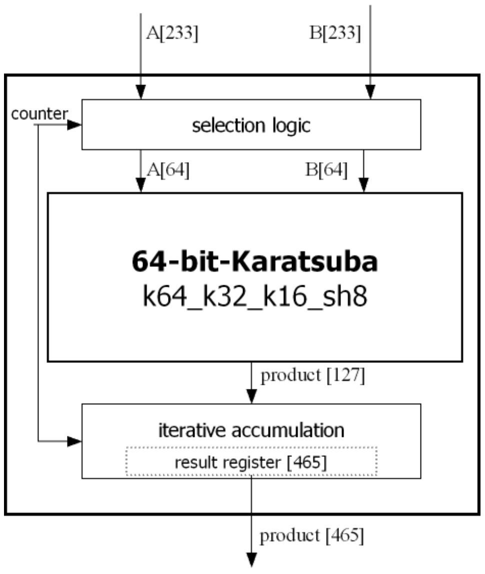


Figure 5.8.: Initial 233 bit iterative Karatsuba design. A counter controls the selection logic, which selects the factors of the partial multiplication, and the accumulation process. Nine clock cycles are required for a 233 bit multiplication with this setup.

the four-segment Karatsuba implementation is smaller than the eight-segment multiplier. The logic for the selection of partial factors and accumulation of partial products has a remarkable impact on area consumption in versions with higher segmentation. For the eight segment version these logic parts take more than 75% of the chip area, since the other components, partial multiplier and result register, require merley 0.08mm^2 and 0.11mm^2, respectively. In absolute numbers the selection and accumulation part takes 0.15mm^2 for the two segment IKM, 0.39mm^2 for the four segment IKM and 0.59mm^2 for the eight segment IKM.

The area scales so much with segmentation because of the complicate data path. For the selection and accumulation processing a sequence has to be executed like the one shown in Table 3.3. These sequences reduce the number of total executed XOR operations, but lead to an irregular data path structure.

Hence, a very regular structure, for the accumulation, which is the more complicated task, is described in the following. The issue is to accumulate (XOR) the product of a partial multiplication in specific positions of the result field of the complete multiplication as it is depicted in Figure 3.2. The positions depend on the current cycle of the multiplication.

Consider for a binary polynomial multiplication with the factor size m, which is divided into four segments, each so that the corresponding partial multiplication has a size of n x n, where the segment size $n = m/4$. In case of a 256 bit multiplication the partial multiplier has a factor size of 64 bit. Applying IKM for this fourfold segmentation requires nine clock cycles to compute the final result of the multiplication. In each clock cycle a partial product of the size

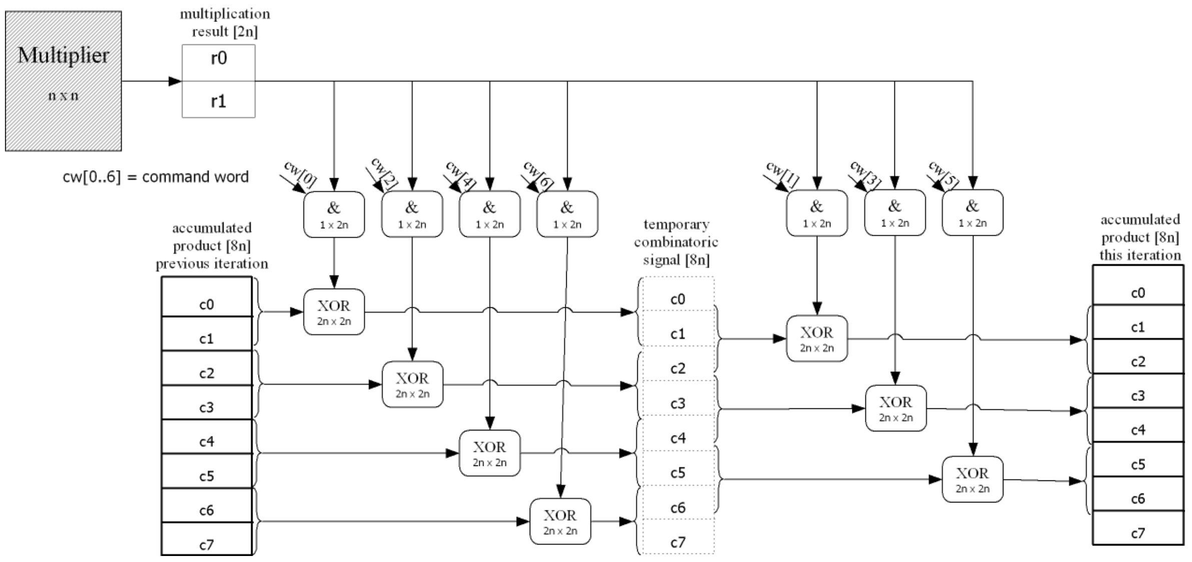

Multiplier
n x n
multiplication result [2n]
r0
r1
cw[0..6] = command word
cw[0]
cw[2]
cw[4]
cw[6]
cw[1]
cw[3]
cw[5]
& 1 x 2n
accumulated product [8n] previous iteration
c0
c1
c2
c3
c4
c5
c6
c7
XOR 2n x 2n
temporary combinatoric signal [8n]
accumulated product [8n] this iteration

Figure 5.9.: Configurable structure for the accumulation of partial products in the IKM process with embedded reduction operation. This design allows only to store four registers (c0 to c3) in flip-flops instead of eight.

of $2n$ is computed, which has to be accumulated to determined positions in the full product. For four segment IKM, seven different positions are possible. The positions can be represented by a seven bit command word which is generated by a small controller block. These command words depend on the current clock cycle of the multiplication. The data path is organized as shown in Figure 5.9. If a command bit is set, the partial product is forwarded to the corresponding XOR operation, else the XOR operation is performed with zeros which results in no change at the relative position. Because of the overlapping XOR operations it is necessary to perform this process in two stages. The intermediate result after the first stage is not stored but is forwarded directly to the second stage. The result of the second stage is stored in the result register, which is used again in the next iteration.
The selection process is done in the same way. Small control words determine the XOR operations that have do be executed. The results of both the new selection and accumulation approach compared to the original method are listed in Table 5.7. One can see that area increasing effect of higher segmentation is very small in the new implementation.

After discussing the partial elements of a full iterative Karatsuba multiplier, these parts will be assembled to one unit. Firstly, the interface of the multiplier block had to be modified in order to use the multiplication block as functional unit of the ECC design. The initial multiplier, as seen in Figure 5.8, has two data inputs and it is necessary to keep it set for the whole multiplication process. The current chip design does not allow these kinds of steady input signals over the bus. This is why the multiplier needs two additional input registers, which

Component	Area[mm^2]
IKM core	0.53
Wrapper	0.18
Summation	0.71

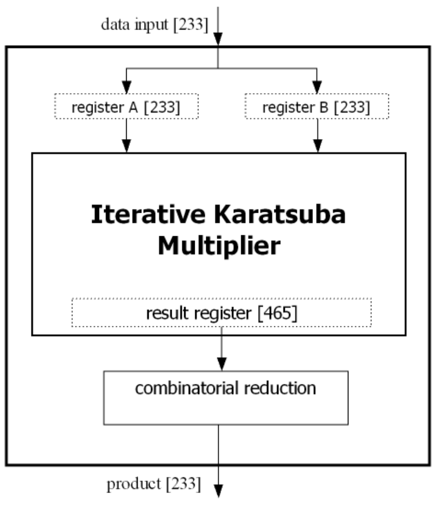


Figure 5.10.: Wrapped 233 bit four segment IKM design. The standardized 233 bit input and output simplify the bus access.

keep the factors during the execution of the multiplication. Another necessary change concerns the output signals. In the initial design the output signal has the doubled size $(2m-1)$, since the product has not been reduced. To match with the bus width, which corresponds to to the normal word width m, it is necessary to reduce the product before writing it onto the bus.
A way to apply these changes is to build a wrapper around the multiplier, which keeps the input signals and reduces the output signals. The data path of this multiplier and wrapper combination and the corresponding area consumption is shown in Figure 5.10.

It is questionable whether it is necessary to store the double sized unreduced intermediate product, as it is done in this design. The register is used as buffer for accumulation as seen in Figure 5.9. In case of the four segment IKM holds the intermediate accumulation result of the nine partial multiplication steps $(acc = p_0 + p_1 + ... + p_7 + p_8)$. The accumulation result finally is reduced to the result of the complete multiplication. This means for the multiplication $c = a \cdot b$, c is $((p_0 + p_1 + ... + p_7 + p_8) \mod r)$, where p_i are the partial accumulations and r is the reduction polynomial. Since it is also valid that $c = (p_0 \mod r + p_1 \mod r + ... + p_7 \mod r + p_8 \mod r)$, one can perform a reduction after every iteration step. The idea is depicted

Table 5.7.: Area consumption in mm^2 of selection and accumulation task for 233 bit IKM compared to the original method.

	Selection	Accumulation	Summation sel. + acc.	Original method
2 segment	0.05	0.08	0.13	0.15
4 segment	0.05	0.09	0.14	0.39
8 segment	0.06	0.10	0.16	0.59

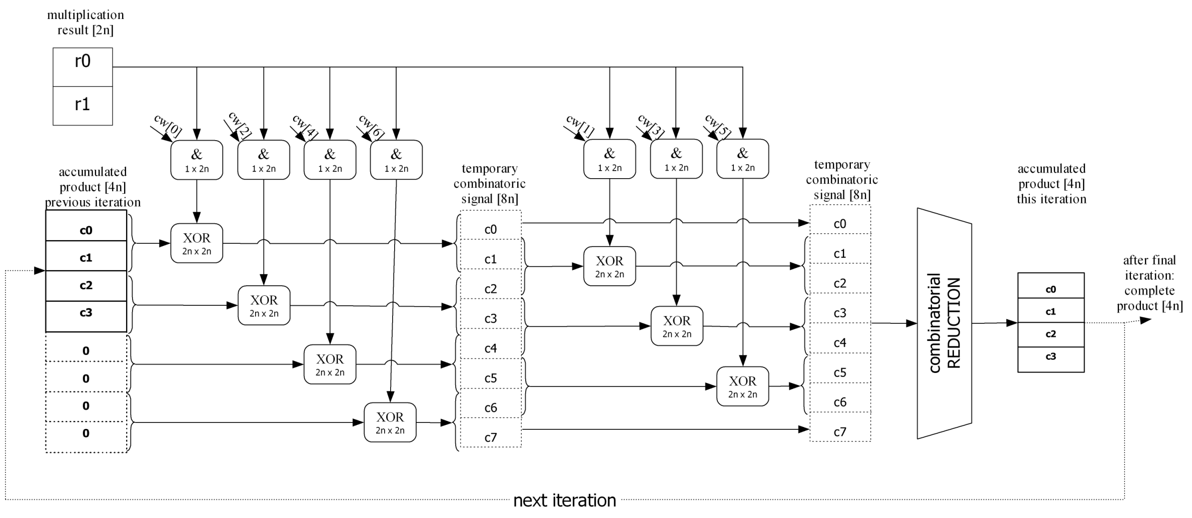

multiplication result [2n]
r0
r1
cw[0]
cw[2]
cw[4]
cw[6]
cw[1]
cw[3]
cw[5]
&
1 x 2n
XOR
2n x 2n
accumulated product [4n] previous iteration
c0
c1
c2
c3
0
temporary combinatoric signal [8n]
c4
c5
c6
c7
combinatorial REDUCTION
accumulated product [4n] this iteration
after final iteration: complete product [4n]
next iteration

Figure 5.11.: Regularly accumulation process with integrated reduction block. By this method registers for the long accumulated product can be saved.

Multiplier	Area[mm^2]
233 bit version	0.62

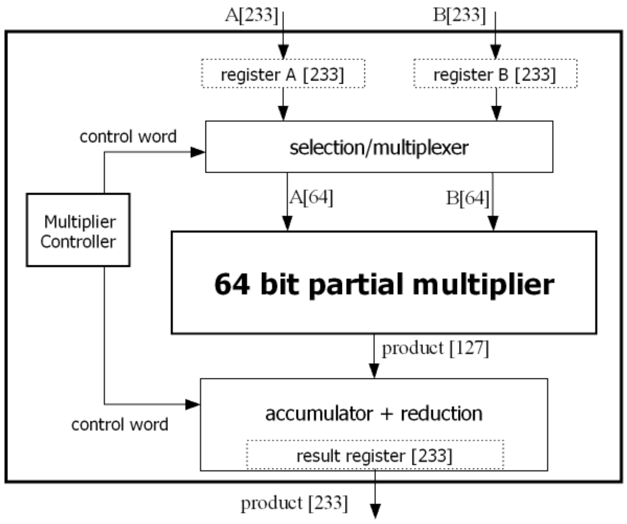


Figure 5.12.: Block diagram of the final multiplication unit. The reduction is part of the accumulation. The 'Multiplier Controller' controls the selection and accumulation. The 233 bit version of this multiplier has a size of 0.62mm^2.

in Figure 5.11. Indeed, in this approach it is no longer possible to maintain the separation of multiplier and wrapper. Thus, even the reduction is part of the multiplier. For a single elliptic curve this design appears really good. For the 233 bit EC the silicon area is 0.62mm^2, which is 14% less than the design of Figure 5.10.

Results

In addition to the 233 bit four segment multiplication unit, we implemented a set of multipliers of different sizes and configurations. Since the reduction step is integral part of the multiplier, reduction polynomials must be selected. In this section multiplier units are presented that are tailored for the recommended NIST curves. For every size, three setups of the IKM unit were implemented: the two segment configuration which requires three clock cycles for a muliptlication, the four segment multiplier with nine clock cycles and the eight segment setup with 27 clock cycles. The results are reported in Table 5.8 and illustrated in Figure 5.13.

The area was determined after synthesizing for our technology. The power consumption was identified applying the multiplications with random factors followed by an power analysis of the internal transition changes. The effective energy is the average power multiplied with the duration of one multiplication.

Figure 5.13 shows that both silicon area and energy consumption are increasing with larger word sizes. It is not surprising that more silicon is required for the less segmented multiplier versions, because the internal combinatorial multiplier is larger for these fast versions. It is remarkable that the needed energy for a complete multiplication is less for faster multipliers. Even though larger multipliers consume more power, due to of the shorter runtime, ultimately they require less energy.
Based on the data, it is possible to answer the question to what extend area and energy consumption is affected by a change of the word size. Figure 5.13 depicts that area and energy grow more than linear with the word size. A logarithmic regression analysis of the data in Table 5.8 has resulted that the area is proportional to the factor $x^{1.27}$, whereby x stands for the word size. This means that a threefold word length results in a fourfold increase of the area. For the energy consumption a factor could be estimated that is proportional to $x^{1.29}$. For both estimations the correlation coefficient is about 0.9. The variation is caused by different multiplication configurations and reduction polynomials, which differ in complexity. Based on this statistical results it is possible to predict area and power consumption for polynomial multipliers with other factor sizes.

Table 5.8.: Comparison of different multipliers. The power consumption was estimated for a clock frequency of 33MHz

Size [bit]	Segments	Configuration of combinatorial multiplier	Cycles	Area [mm^2]	Power [mW]	Energy [nWs]
163	2	raik96_raik24_cpm6	3	0.79	47.9	4.31
163	4	raik48_raik12_cpm3	9	0.45	31.6	8.53
163	8	raik24_cpm6	27	0.35	18.5	14.99
233	2	raik128_raik32_ckm8_cpm4	3	1.17	64.5	5.80
233	4	raik64_raik16_cpm4	9	0.62	42.9	11.58
233	8	raik32_ckm8_cpm4	27	0.44	22.8	18.47
283	2	raik160_raik40_ckm10_cpm5	3	1.60	94.1	8.47
283	4	raik80_raik20_cpm5	9	0.85	45.6	12.31
283	8	raik40_ckm10_cpm5	27	0.62	31.3	25.35
409	2	raik224_raik56_ckm14_cpm7	3	2.87	159.9	14.39
409	4	raik112_raik28_cpm7	9	1.38	81.7	22.06
409	8	raik56_ckm14_cpm7	27	0.85	44.6	36.13
571	2	raik320_raik80_raik20_cpm5	3	4.35	277.6	25.0
571	4	raik160_raik40_ckm10_cpm5	9	2.10	141.8	38.3
571	8	raik80_raik20_cpm5	27	1.31	82.9	67.14

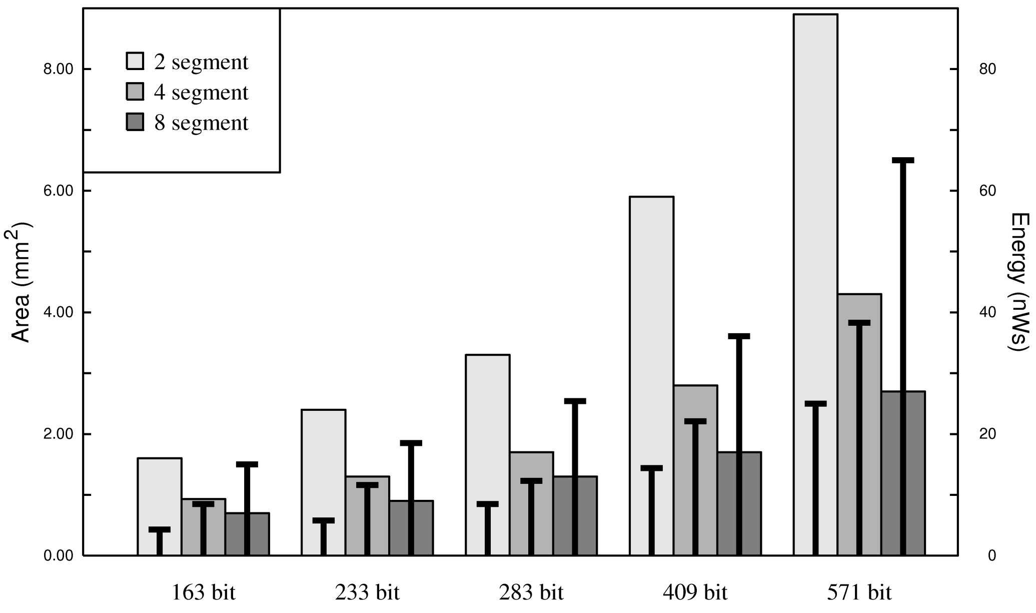


Figure 5.13.: Area and energy consumptions of different multiplier setups. The bars represent the silicon area and the lines show the corresponding energy for one complete multiplication.

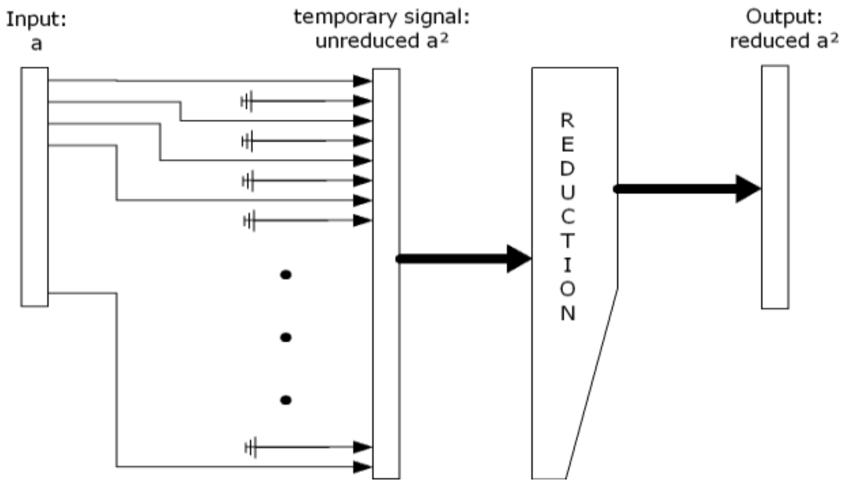


Figure 5.14.: Squaring operation as subsequent performed squaring and reduction. The actual squaring inserts a zero between the input bits. The long intermediate result is reduced by a reduction block.

5.5. Polynomial squaring

As written in Section 3.3 the straightforward way of squaring in $GF(2^m)$ is to insert a 0 between every bit of the input word. Since after squaring, as after multiplication, the word has the doubled size, a reduction step after squaring is necessary.
In the simplest case the functionality can be realized within a two step logic. In the first step the bits of the input word are assigned to the even positions of an internal signal of the square unit. The odd positions are set to zero, precisely they are connected to ground on the chip. In the second step a reduction step is performed to achieve the final reduced result of the squaring operation. For known ECs, the combinatorial reduction is the best choice for the reduction step. Thus, squaring and reduction together can be done in one clock cycle. A design of such squaring and reduction is shown in Figure 5.14.

The described design can be improved by developing a combinatorial reduction block instead of using a general one. The idea is to exploit the fact that every second bit is zero, to minimize combinatorial logic of the reduction. For example for the considered field $GF(2^3)$ with the reduction polynomial $x^3 + x + 1$, a multiplication result has a degree of five $(x_4, x_3, x_2, x_1, x_0)$. The standard combinatorial reduction process performs the reduction process of the result in the following way, whereas the presentation as vector should improve the readability:

$$\begin{pmatrix} x_2^{'} \\ x_1^{'} \\ x_0^{'} \end{pmatrix} = \begin{pmatrix} x_2 + x_4 \\ x_1 + x_3 + x_4 \\ x_0 + x_3 \end{pmatrix}$$

Size [bit]	Area [mm^2]	Relative to full reduction
163	0.018	-59%
233	0.012	-64%
283	0.032	-58%
409	0.016	-72%
571	0.063	-61%

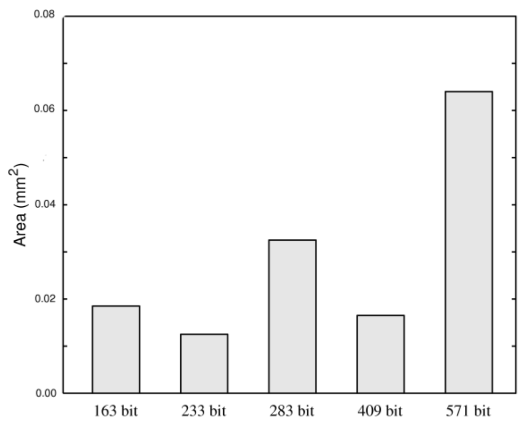


Figure 5.15.: Area consumption of optimized squaring with reduction. 233 and 409 bit implementations are relatively small because of the simple structure of the reduction polynomial. The area for this squaring blocks is less than 50% of the area for a full reduction block.

But since after a squaring operation the odd positions are zero, one can simplify the reduction:

$$\begin{pmatrix} x_2^{'} \\ x_1^{'} \\ x_0^{'} \end{pmatrix} = \begin{pmatrix} x_2 + x_4 \\ x_1 + x_3 + x_4 \\ x_0 + x_3 \end{pmatrix} \Rightarrow \begin{pmatrix} x_2 + x_4 \\ 0 + 0 + x_4 \\ x_0 + 0 \end{pmatrix} = \begin{pmatrix} x_2 + x_4 \\ x_4 \\ x_0 \end{pmatrix}$$

Hence, in the considered field $GF(2^3)$ the complete squaring and reduction can be expressed as

$$\begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix}^2 = \begin{pmatrix} x_1 + x_2 \\ x_2 \\ x_0 \end{pmatrix}$$

Similar changes can be made for every field. Figure 5.15 shows the resulting silicon area for the five NIST polynomials.

The needed area for a combinatorial squaring with reduction is not steadily growing with increasing bit length of the factor. The reason is the same as in the normal combinatorial reduction. The 233 bit and 409 bit reduction polynomial are trinomials and the others are pentanomials, and thus they are more complicated and consequently consume more area. Comparing with the results of the full reduction, it is also remarkable that a combined square and reduction block is significantly smaller than the corresponding detached reduction. For all polynomials the area is at least 50% smaller.

Beside the advantages of small area and fast execution time, this combinatorial approach is connected with the disadvantage of inflexibility. Since the reduction polynomial is embedded in the squaring block, it is not possible to use one of these squaring blocks for more than one field. But for single field implementations it is the best choice, because the discussed very small units perform the squaring operation in $GF(2^m)$ within one clock cycles.

5.6. Modular multiplicative inversion

In Section 3.5 two general methods for the computation of the modular multiplicative inversion were described: the extended Euclidean algorithm and the Fermat method. The pure EEA has lost relevance, since the Shantz division has been presented. This division algorithm computes the whole division instead of merely a single inversion. Hereby, the same computational effort as for the EEA is required. This is why the EEA will not be considered as potential hardware implementation. In this section hardware designs for the Shantz division and the Fermat method are investigated.

5.6.1. Fermat method

The Fermat method for the computation of the multiplicative inverse is a successively performed combination of field multiplications and squaring operations. Thus, additional hardware blocks are not necessarily required because a multiplier and a square unit are already parts of the ECC design. This is why a program that is executed in the controller block is sufficient for the implementation of the inversion operation.

Square-and-multiply

The first feasible algorithm for the inversion by the Fermat method, is the repeated square-and-multiply algorithm, which is shown as Algorithm 3.5. Even though the method requires $m-2$ multiplications and $m-1$ square operations, it is a reasonable approach because it does not require additional hardware and has a very simple and flexible program. Only the number of the iterations of the loop must be adapted when the field size changes.
A possible program for a hardware environment with one multiplier, a square unit and the registers A and B, which are connected with one m bit wide bus, is shown as Algorithm 5.1.

Algorithm 5.1: Square-and-multiply inversion

```
input  : $A = a(x) = (a_{m-1}, a_{m-2}, ..., a_1, a_0)_2$
output: $B = a(x)^{-1}$
1 $sq_a$ denotes the input of the square unit and $sq_r$ their output, $mul_a$ and $mul_b$ are the
  two inputs of the multiplier and $mul_r$ the result of the last multiplication
2 $A \Rightarrow sq_a$ ;
3 $sq_r \Rightarrow sq_a, mul_a$ ;
4 for $i \leftarrow 2$ to $m-1$ do
5     $sq_r \Rightarrow sq_a, mul_b$ ;
6     $mul_r \Rightarrow mul_a$ ;
7 end
8 $sq_r \Rightarrow mul_b$ ;
9 $mul_r \Rightarrow B$;
```

In an assumed hardware design with a field multiplier that requires C_M clock cycles for a multiplication and is working on a single bus, this algorithm does not only require C_M clock cycles for one multiplication but also additional 2 cycles for setting up each multiplication. All but the first one, squaring operations can be done concurrently to a multiplication, so that actually the squaring does not require extra time. Hence, this algorithm requires

$$(C_M + 2) \cdot (m - 2) + 2 \tag{5.1}$$

clock cycles.

The two clock cycles for the setup of the next multiplication can be saved when the factors of the following multiplication are already known during the previous operation. In Algorithm 5.1 this is not possible due to the dependence of the next multiplication on the result of the previous one. But the multiplication chain can be split up into two independent multiplication chains:

$$\prod_{i=1}^{m-1} a(x)^{2^i} = \prod_{i=1}^{m/2} a(x)^{2^{2i}} a(x)^{2^{2i-1}} = \prod_{i=1}^{m/2} a(x)^{2^{2i}} \prod_{i=1}^{m/2} a(x)^{2^{2i-1}}$$

These two product chains can be calculated parallel. Thus, the next multiplication of one chain can be set up while the previous operation of the other chain is still being performed. Only the first and the last multiplication, which merges both chains, require additional setup cycles. Hence, the total required time is reduced to

$$C_M \cdot (m - 2) + 6 \tag{5.2}$$

clock cycles.

Itoh-Tsujii-inversion

In Section 3.5.2 Itoh-Tsuhii's improved approach of the Fermat based inversion has been described. It reduces the number of multiplications to not more than $2\,[log_2(m-1)]$. The concrete number of multiplication depends on the specific field size, since the operations flow must be determined separately for every field. Referring to Algorithm 3.6, which shows the case of $m = 233$, ten multiplications and 232 squarings are required. Considering that a multiplication requires N_M clock cycles for the execution and additional two cycles for the setup of every multiplication and every square operation takes one cycle, a complete inversion requires less than

$$(N_M + 2) \cdot 2\,[log_2(m-1)] + m \tag{5.3}$$

clock cycles. In practice, this theoretic upper bound is not reached, because usually less multiplications are required, the two setup cycles are not always necessary, and some squaring

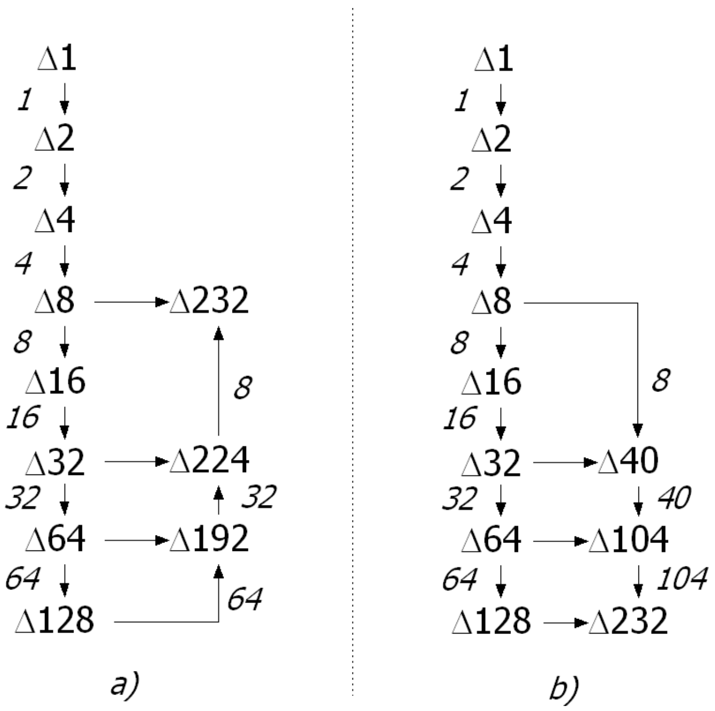


Figure 5.16.: Graphical representation of algorithm 3.6 on the left that require less square operation (numbers on the edges). On the right the modified flow that does not need saving the intermediate results (nodes), since the flow is straight downwards.

operations can be performed parallel to a multiplication.

An issue, which has not been mentioned yet, is the required number of registers for the intermediate results. The square-and-multiply Algorithm 5.1 needs two registers: the input A and the register B for the computed result. The Itoh-Tsujii-Inversion as is shown exemplary for the 233 bit field requires more registers. Figure 5.16 a) shows the flow diagram of the algorithm, whereby the nodes are the intermediate results and the edges are noticed by the required square operations. The notation is used that was introduced in Section 3.5.2. For example $a(x)^{\Delta 232}$ is computed by $a(x)^{\Delta 224} \lhd 8 \cdot a(x)^{\Delta 8}$, which implies eight shift operations in addition to the multiplication. The algorithm determines the powers of two of the Δ-operator, which corresponds to the downward direction. Then, the required intermediate results are accumulated to obtain $a(x)^{\Delta 232}$. If this is processed upwards, as it is depicted in the diagram, exactly $m - 1$ shift operations must be performed, as it can be determined by accumulating the labeled edges. The disadvantage of this approach is that the intermediate results, which are in the example $a(x)^{\Delta 8}$, $a(x)^{\Delta 32}$, and $a(x)^{\Delta 64}$, must be stored.
Three additional 233 bit wide registers are very expensive. Figure 5.16 b) shows an alternative flow that works with two registers: one register contains the computed powers of two of the Δ-operator, whereby the other register obtains the accumulated intermediate results and

Table 5.9.: Estimations, according to Equation 5.4 and actual clock cycles of the Itoh-Tsujii based Inversion for the NIST field sizes.

Field size [bit]	Estimation [clock cycles]	Required [clock cycles]
163	406	248
233	522	316
283	603	382
409	804	494
571	1057	684

eventually the final solution. Thus, the intermediate results are accumulated in time and no additional storing is required. This method is connected with the disadvantage of additional shift operations as it can be observed by adding the labeled edges in the diagram. In the 232 bit example, the number of squaring operations increases to 280. In worst case, when the accumulation register recalculates the powers of two, $(m-1)/2$ additional square operations lead to total maximum square number of $N_S \leq 1.5m$. With the same assumed hardware as before, the maximum total computation time is

$$(C_M + 2) \cdot 2\left[log_2(m-1)\right] + 1.5m \tag{5.4}$$

clock cycles.

The actual number of clock cycles for the register saving inversion in the five NIST fields is shown in Table 5.9. Hereby a multiplier has been applied that requires nine clock cycles for one field multiplication. Squaring operations are performed concurrently to multiplications and the setup cycles for the multiplier are avoided as far as possible.

Shantz division

The Shantz division, computes a field division without need for the computation of the inverse. The algorithm as it is shown as Algorithm 3.7. An implementation as program that runs in the controller block is not feasible as it requires various additional functional units, e.g. shifter, and new comparators, such as compare for equality. In addition the complicated program flow and the loop that is repeated up to $2m$ times renders the idea of an execution in the controller block very slow.

In [12] a separate block for the execution of the algorithm has been proposed. The described design concurrently executes the operations of the loop and terminates therefore in a maximum of $2m$ clock cycles. The actual runtime depends on the actually computed values and varies from division to division. Hence, at compile time it has to be presumed that the operation finishes after $2m$ clock cycles.

Table 5.10.: Silicon area for division blocks for different field sizes

Size [bit]	Area [mm^2]
163	0.38
233	0.56
283	0.69
409	1.01
571	1.45

A similar hardware design for the known 233 bit field, written in VHDL and synthesized for the IHP 0.25μm technology, requires silicon area of about 0.56mm^2. The corresponding areas for other field sizes are given in Table 5.10. Even though additional silicon and less performance, the division approach is not useless at all. A separate FU allows to perform other operations concurrently. In addition, the execution time does not depend on the speed of the multiplication and the square unit. In example on hardware designs with very slow multiplier or squaring operations that require more than one clock cycle for one computation, the Shantz division is the faster implementation.

Another benefit of the division block is less power consumption. Even with the preferred four segment multiplier, the division block requires merely a third of the total energy, in particular because no expensive multiplication must be performed.

Table 5.11.: Comparison of Itoh-Tsujii based inversion approach to the Shantz division for the 233 bit EC

Method	Area[mm^2]	Time[μs]	Power[mW]	Energy[nWs]
Itoh-Tsujii inversion and division	< 0.1	9.8	38	372
Shantz division	0.56	14.0	9	126

6. Efficient hardware designs for ECPM

In this chapter the results of the previous chapter are combined to build a cryptographic coprocessor that accelerates ECC. This coprocessor executes elliptic curve point multiplication for the 233 bit curve B-233 recommended by the NIST. This 233 bit implementation is exemplary for every arbitrary elliptic curve. At the end of this chapter the ECC module is integrated in a network communication and cryptographic system on chip. The performance of the ECC unit of this chip that is actually produced in silicon is eventually compared to other known ECC implementations in software and hardware.

6.1. EC point multiplication

As described earlier, the ECC processor contains hardware blocks for the field operations. These field operations as polynomial multiplication, squaring and addition are controlled by the controller block. The algorithms for the point operations on the actual elliptic curve, as the point multiplication, are part of the program that is executed by this controller.

In this section the Montgomery point multiplication (MPM) algorithm is applied as it is presented in Algorithm 4.5. As described, it is the fastest known algorithm for ECPM in $GF(2^m)$. The algorithm is executed as a program running in the controller unit, and applies the other blocks that perform the base field operations. Based on the result of the investigations in the last chapter, different combination of these field operation blocks are feasible and are discussed in the following. The aim is not only to find an area efficient and fast implementation for the point multiplication but also to show a comprehensive design space for ECC hardware implementations.

Initially, the MPM, as it is shown in Algorithm 4.5, can be subdivided in three phases:

Initialization: Initial setting of the projective coordinates.

Loop: Determination of the result of the point multiplication in projective coordinates. The operations in the loop are repeated up to $m - 1$ times.

Finalization: Determination of the final affine coordinates (x, y) out of the projective coordinates.

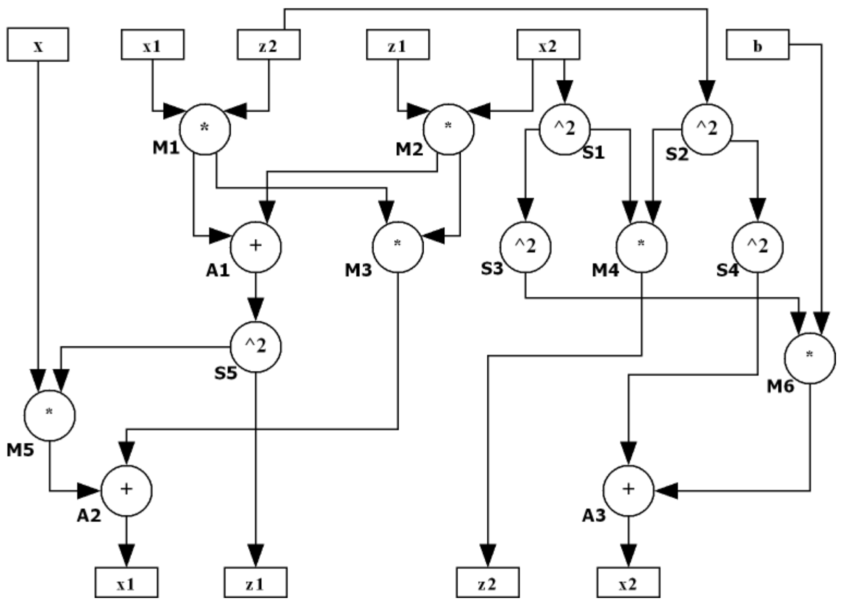


Figure 6.1.: Flow chart of the inner loop of the Montgomery point multiplication for the case $k_i = 1$. It requires six multiplications, three additions and five squaring operations.

For an ECPM, initialization and finalization phase are performed once, whereby the operations of the inner loop are repeated $m - 1$ times. Hence, the largest amount of time is spent in the inner loop. This also implies that it is very important to optimize the operations flow in the loop. A clock cycle wasted in the inner loop implies $m - 1$ wasted clock cycles for a complete point multiplication.

A flow chart of the operations flow for one iteration of the inner loop in case $k_i = 1$ is shown in Figure 6.1. The flow for $k_i = 0$ is the same considered x1 and x2 as well as z1 and z2 are exchanged on input and output. Thus, for the following investigations it is sufficient to treat the case $k_i = 1$. The results can perfectly be applied for the other case.

The flow diagram depicted in Figure 6.1 already demonstrates an optimized flow of the inner loop. It reuses intermediate terms if possible in order to reduce the number of operations. Hence, one iteration of the inner loop of the MPM algorithm requires 14 operations in $GF(2^m)$:

- 6 multiplication operations $M_1 ... M_6$
- 5 squaring operations $S_1 ... S_5$
- 3 addition operations $A_1 ... A_3$

The longest paths are M_1, A_1, S_5, M_5, A_3 as well as M_2, A_1, S_5, M_5, A_3 which contain five operations each. This is why the lowest bound for computing one iteration is the time required for subsequently executing two multiplications, two additions, and one squaring.

Assuming each operation takes one clock cycle, the number of hardware units can be determined that are needed to perform the loop within the lower bound of five cycles. Since six multiplications must be computed, it is impossible to finish within five cycles with only one multiplier. At least two multipliers are required. The diagram also shows that every squaring operation is followed by at least one other operation. To achieve the five clock cycles, the last squaring must be finished after four cycles. Hence, at least two squaring units are required in order to perform the aimed five squarings in time.
Using two multipliers, two square units, and one adder the following schedule can be generated:

cycle 1	M_1, M_2, S_1, S_2
cycle 2	M_3, M_4, A_1, S_3, S_4
cycle 3	S_5, M_6
cycle 4	M_5, A_3
cycle 5	A_2

Even though this schedule demonstrates the possibility of performing one iteration in five clock cycles, there are some reasons why it is probably not done. Beside the fact that a one clock multiplier is very large and additionally a second one is required, there is still the question how to transport the data to the operating units in time. Multiplications need two assignments as well as additions while a squaring requires only one input. In the first cycle of the schedule this results in six required assignments for the two multiplications and the two squaring operations. Using a one-write-multiple-read bus there is still the need for four m bit wide buses ($X_1 \Rightarrow M_1$, $X_2 \Rightarrow M_2\&S_1$, $Z_1 \Rightarrow M_2$, $Z_2 \Rightarrow M_1\&S_2$). Obviously, that would result in a very large design, and it is very questionable whether it is routable at all, but it shows the lower bound concerning the timing and an upper bound of hardware usage.

Time analysis with slower multipliers

As shown in Section 3.2 the multipliers that compute the polynomial product in one clock cycle are very large. Hence, it is reasonable to discuss implementations of the inner loop using smaller multipliers which require more than one clock cycle to complete.
Together with the variables that have already been mentioned, there are the following parameters that affect the performance of the system:

- Speed of one polynomial multiplication
- Number of multiplication units
- Number of buses

Table 6.1.: Numbers of clock cycles needed to complete the inner loop of an MPM using one adder, one square unit, and multiplier with four different speeds: a) one clock multiplier, b) three clock multiplier, c) nine clock multiplier, d) 27 clock multiplier. For each multiplier is

Multipliers:	1	2	3
one bus	20	20	20
two buses	10	10	10
three buses	8	8	8
four buses	8	7	7

a)

Multipliers:	1	2	3
one bus	20	18	18
two buses	19	12	11
three buses	19	12	11
four buses	19	11	10

b)

Multipliers:	1	2	3
one bus	56	32	27
two buses	55	30	23
three buses	55	30	23
four buses	55	29	22

c)

Multipliers:	1	2	3
one bus	164	86	63
two buses	163	84	59
three buses	163	84	59
four buses	163	83	58

d)

For the estimation of the concrete impact of the variables, we wrote a C program that determines optimal schedules for one iteration of the inner loop under different combinations of the system parameters. Even though the program works with some simplifying constraints - for example time and space for storing the values are not considered - it returns comparable values for analyzing the different setups. The program uses the flow chart of Figure 6.1 and estimates the fastest schedule, which determines the required time. It is considered that the number of squaring and adding units is one each.

Setups with one to three multipliers, one to four buses and multipliers that require one, three, nine and 27 clock cycles for one multiplication were investigated. The results of these estimations are depicted in Tables 6.1a) to d). Table 6.1a) shows the results for very fast multipliers that take only one clock cycle to complete, while the other tables show the times for slower multipliers.

The fastest version requires seven clock cycles instead of the five clock cycles of the previously described optimal setup. This is due to the number of squaring units that is limited to one in the current investigation. Another reason is that one clock cycle is considered for the final assignment of the variables. When the results of one iteration are not stored into registers but forwarded directly into the functional units of the next iteration, this additional clock cycle can be saved.

Evaluating the numbers in Tables 6.1, the following conclusions can be drawn:

- The number of buses is important while using fast (Table 6.1a)) or many multipliers (Table 6.1b), column three). Using slow multipliers the benefit of multiple buses is limited. This was expected, because the relative amount of time needed for transferring the data increases with the calculation power.

- The benefit of multiple units is high for slow multipliers (Table 6.1c) and 6.1d)) but not mentionable for the fast ones. These results were also expected, since the transportation is the bottleneck for the setups with fast multipliers.
- The speed of the multiplies is only relevant when the data can be transported fast enough. With only one bus, the performance with the one clock multipliers is the same as for the three clock versions. In contrast, the required cycles for one multiplier unit and multiple buses scales very much with the speed of the multiplies.

The results are not surprising, but the concrete numbers can help to reduce the design space when additional constraints are emerging. For example, assume that due to the area consumption only one bus and one multiplier are allowed. In this case, the one clock multiplier has no benefit compared to the three clock version. Hence, there is no need to consider the large one clock multiplier in further investigations.

EC point multiplication for ECC 233

In the following the MPM is implemented for a 233 bit EC. Due to routing issues, the number of buses is constraint to one. The set of 233 bit polynomial multiplier configurations was described and measured in Table 5.8. An application of these results, in particular the silicon area, together with the estimation of required clock cycles of the different setups are combined in Table 6.2. Only plausible setups are listed.

Table 6.2.: Expected parameters for ECC designs with polynomial multiplier of different configurations.

Segments	Cycles per mul	Number of muls	Area [mm^2] of muls	Expected total clock cycles
2	3	1	1.2	20
4	9	1	0.6	56
4	9	2	1.3	32
8	27	1	0.4	164
8	27	2	0.9	86

Configurations using two multipliers are worse in time and area consumption compared to the single multiplier solutions applying the corresponding next larger multiplier. Hence, the multiple multiplier setups are not worth further consideration. The remaining three configurations are single multiplier setups that show straightforward the trade-off between time and area: the faster a setup is, the larger is the design.

Having chosen the number of functional units, the size of the register file has to be determined. Since one single 233 bit register has a silicon area of 0.1mm^2 and consumes power in every clock cycle, it is recommended to keep the number of registers as small as possible. Figure 6.1 shows the six variables required for the inner loop. These are the four temporary variables

Table 6.3.: Program and schedule of the inner loop of the MPM algorithm in case $k_i = 1$. It requires 56 clock cycles.

clk	Schedule	Program	Comments
0			first multiplication started before
1	M1 S2	$Z2 \Rightarrow SQ$	start square operation $S2(= Z2^2)$
2		$SQR \Rightarrow Z2$	store the result in register $Z2$
3			
4			
5			
6		$Z1 \Rightarrow MULA$	set the first factor of $M2$
7		$X2 \Rightarrow MULB,\ SQ$	set the second factor of $M2$
8	S1	$MULR \Rightarrow X1$	store the result of $M1$ into register $X1$
9		$SQR \Rightarrow X2$	
10			
11			
12	M2		
13			
14			
15		$Z2 \Rightarrow MULA$	
16		$X2 \Rightarrow MULB,\ SQ$	
17	S4	$MULR \Rightarrow Z1$	
18		$SQR \Rightarrow X2$	
19			
20			
21	M4		
22			
23	S3	$Z2 \Rightarrow SQ$	
24		$SQR \Rightarrow MULA$	
25		$B \Rightarrow MULB$	
26		$MULR \Rightarrow Z2$	
27			
28			
29			
30	M6		
31			
32			
33		$X1 \Rightarrow MULA$	
34		$Z1 \Rightarrow MULB$	
35		$MULR \Rightarrow AX$	
36	A3	$X2 \Rightarrow xAX$	
37		$AX \Rightarrow X2$	
38		$X1 \Rightarrow AX$	
39	M3 A1	$Z1 \Rightarrow xAX$	
40		$AX \Rightarrow SQ$	
41	S5		
42		$SQR \Rightarrow MULA,\ Z1$	
43		$X \Rightarrow MULB$	
44		$MULR \Rightarrow AX$	
45			
46			
47			
48	M5		
49			
50	A2		
51		$K \Rightarrow TESTBIT$	test next bit of factor
52		$Z2 \Rightarrow MULA$	start first mul. of next iteration
53		$MULR \Rightarrow xAX$	
54		$AX \Rightarrow MULB,\ X1$	
55			one clock cycle for the loop

(x1, x2, z1, z2), the parameter of the curve (b) and the x-coordinate of the base point (x). For the finalization the y-coordinate of the base point is required. Additionally, the factor of the multiplication (k) must be kept in memory. Altogether, at least eight 233 bit registers are required for an MPM. Consequently, we are considering to exploit the properties of the functional units and the forwarding bus architecture. Such beneficial properties are:

- Setting up the next multiplication while the previous is still running.
- Keeping intermediate results in FUs instead of writing them into register.
- Forwarding from one FU to the next one without storing in register.

These benefits allow to realize an MPM without additional registers. The concrete program with the schedule for the inner loop in case $k_i = 1$, applying the nine cycle multiplier, is shown in Table 6.3. A complete iteration as depicted requires 56 clock cycles. The theoretical fastest execution time for the six multiplications is 54 clock cycles. Hence, the multiplier is idling for only two clock cycles. In case that the factor bit is not set even these two idle cycles can be avoided due to an improved instruction flow. The utilization rate for square unit and adder are much less. Actually, the only intersection of duty time of the two blocks occurs while moving a word from the adder to the square unit. This is why both blocks can be combined to one FU. Thus, the complexity of the bus structure can additionally be reduced, since one block less is accessing it.

Corresponding schedules have been developed for the initialization and finalization task as well as for the setups with a two segment multiplier or an eight segment multiplier. The finalization phase requires one inversion. The results are shown in Table 6.4. It is remarkable that the two-segment multiplier version is slower than expected. The time for storing and managing the intermediate results has more impact than considered in the estimations presented in Table 6.1b).

Table 6.4.: Required number of clock cycles for a 233 bit ECPM with with polynomial multiplication blocks of different speeds. The clock cycles for the loop are considered as m times duration for one average iteration, for the final phase the time for an inversion is enlisted separately.

Segments	Clock cycles for				
of mult.	one mul	Init	Loop	Final	Total time
2	3	9	232·32=7424	49+276=325	7758
4	9	9	232·55=12760	110+314=424	13193
8	27	9	232·163=37816	308+493=801	38626

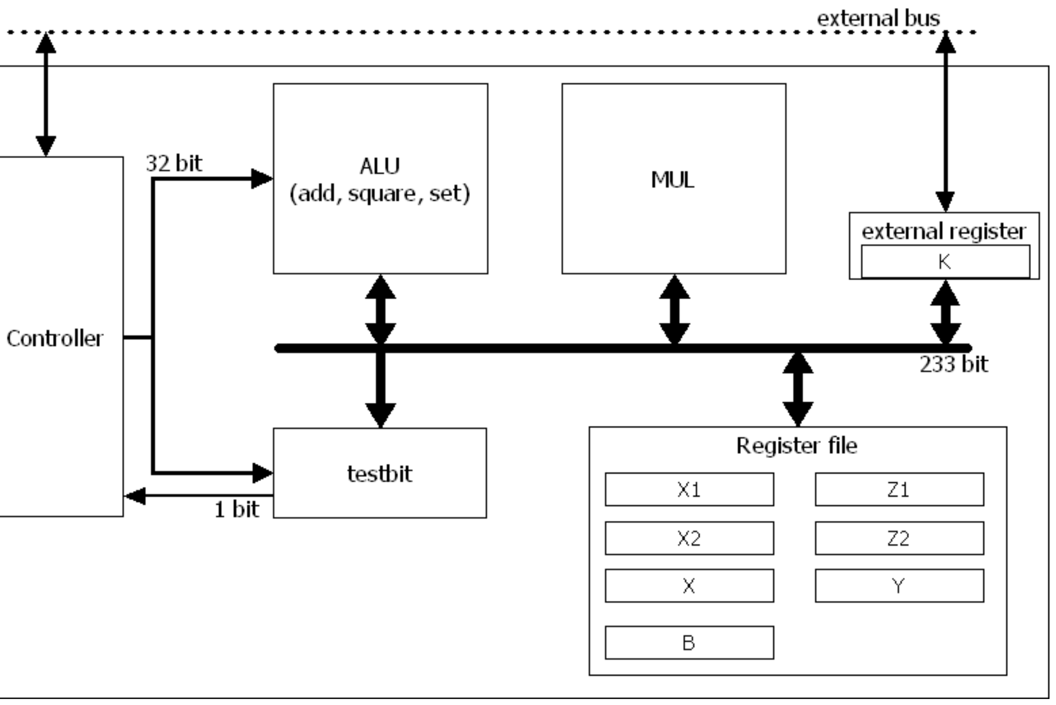


Figure 6.2.: ECC 233 block diagram. Polynomial multiplier (MUL) and ALU, with functionality for adding, squaring and manually setting of data words, are the 'working horses' of the chip. The controller block, which can be externally accessed, controls the bus access. The testbit block checks whether specific bits in the data word on the bus are set. The design has eight separate registers, seven in the internal register file and one that can additionally accessed over an external bus.

6.2. Design of a ECC 233 coprocessor

In the last section the plausible design space could be reduced to three remaining setups. The difference between these setups is the speed of the field multiplier. In the following, these systems are assembled and implemented to final ECC accelerator chips. These designs can finally be simulated for a silicon technology based library and implemented and tested on an FPGA.

Figure 6.2 presents the block diagram for the final design of the 233 bit ECC. It shows the FUs, all registers and data paths. The FUs are connected by a 233 bit bus. Worth mentioning are the two other connections:

- The 32 bit connection transports data and addresses from the Controller to the FUs that require additional data.
- The one bit connection transports the result of the comparison to the Controller.

In the design, these two connections are driven by one block each. Hence, they are called connections rather than buses. In general, these connections were conceived as additional

parallel buses that transport control informations.
In the following, the FUs are described in detail. Since every function is triggered by a bus operation, this notation is used:

$OUTPUT \Rightarrow\ INPUT$: A move operation over the system bus from OUTPUT to INPUT.
$x \overset{32}{\rightarrow}\ INPUT$: x is written on the 32 bit connection to the specified INPUT.
$OUTPUT \overset{1}{\rightarrow}\ Controller$: OUTPUT writes on the one bit connection to the Controller.

A star (*) is used when any assignment or value is allowed.

ALU

The ALU combines the functions of addition, squaring and manually setting the accumulator register. The accumulator register (AX) is the only register inside the ALU and all operations are working on it. Concretely the functions are:

$* \Rightarrow\ AX$: Setting the value of AX from the system bus (AX:=in).
$* \Rightarrow\ xAX$: Adding the value from the system bus to AX and writing the result back into AX (AX:=AX+in).
$* \Rightarrow\ SQ$: Square the input from the system bus, reduce it and write the result into AX (AX:=in^2).
$* \overset{32}{\rightarrow} AX(0..7)$: The value from the 32 bit input is written into the specified position of AX. The 233 bit AX is subdivided into seven 32 bit segments and one nine bit segment which can be set separately.
$AX \Rightarrow\ *$: Write the value of AX on the system bus.

With the ALU it is examplary possible to set $Z1$ to 1 as it is required in the initialization process of the MPM algorithm:

$$AX \Rightarrow\ xAX \qquad \text{// set AX to 0, since } AX + AX = 0$$
$$1 \overset{32}{\rightarrow}\ AX[0] \qquad \text{// set lowest bit of AX to 1}$$
$$AX \Rightarrow\ Z1 \qquad \text{// write AX into register Z1}$$

Multiplier

The multiplication unit is comprehensive explained in Section 5.4. The unit includes two input registers and one output register. The operations on the multiplier are:

$* \Rightarrow\ MUL_A$: Set the first factor of the multiplication to the value from the system bus.
$* \Rightarrow\ MUL_B$: Set the second factor and start the multiplication.

$MUL_R \Rightarrow\ *$: Write the reduced product on the system bus.

The factors MUL_A and MUL_B can be set while the previous multiplication is still performed. The Result MUL_R can be fetched during the first calculation cycle of the following multiplication. This way no additional setup or transport cycles are required.
In the current design, the multiplication includes the reduction logic.

Testbit

The testbit unit checks whether a specified bit of the word on system bus is set. This is the port from the system bus to the controller, which can perform branch decisions based on this input. The result of the comparison is written to the one bit output. The address of the examined bit is fetched from the 32 bit input. Hence, the supported operations of the testbit unit are:

$* \Rightarrow\ TB_W$: The data word on the system bus is examined in the testbit unit.
$* \xrightarrow{32}\ TB_{ADDR}$: The address of the bit that should be examined is determined by this operation on the 32 bit bus. This operation must be performed concurrently to the assignment on the system bus.
$TB_R \xrightarrow{1}\ *$: The result of the comparison is written to the one bit output.

Registers

The registers are the eight registers that have been described in the last section. Worth mentioning is that the register of the variable K is not embedded into the internal register file but is an external register that can additionally be accessed through an external bus. This external register is the interface, external units can write and read data words. The names of the register are based on the names in the inner loop of the MPM algorithm. They can be used independent from this algorithm, since every register can be directly set from and written to the bus.

Controller

As described earlier, the control unit is the place where the bus access is managed and the main program is executed. In this block the point multiplication and also the determination of the multiplicative inverse is performed while the other FUs are applied.
In the current design, the programs that are performed by the control unit are hard wired. This means that the programs are VHDL state machines. The selection of the program that should be executed is done by a command word that is provided over the external bus.

Table 6.5.: Results for complete 233 bit ECC designs after synthesizing with different embedded polynomial multipliers. Beside the standard design running at 66 MHz a faster version is reported.

Mul Setup	Size [mm^2]	Size [kgates]	Speed [ns]	Needed clk cycles	Total time [ms]	Power [mW]	Energy [μWs]
233_8	1.20	42	15	38507	0.58	86	50.1
233_8	1.30	44	6	38507	0.23	222	51.0
233_4	1.42	49	15	13164	0.20	136	26.8
233_4	1.51	52	6	13164	0.08	277	27.3
233_2	2.05	72	15	7383	0.11	186	20.6
233_2	2.12	76	8	7383	0.06	390	21.6

Currently implemented programs are:

LOAD_X:	Copy the word stored in external register into the X-register.
LOAD_Y:	Copy the word stored in external register into the Y-register.
LOAD_B:	Copy the word stored in external register into the B-register.
GET_X:	Copy the word stored in the X-register into the external register.
GET_Y:	Copy the word stored in the Y-register into the external register.
MUL:	Perform a point multiplication kP, whereby he external register contains the factor k and in the registers x and y the coordinates of the point P are stored.
INVERT:	Compute the multiplicative inverse in the base field of the word that is stored in the external register. This program is also used by the point multiplication.

6.2.1. Results

The three designs (233_2, 233_4, 233_8) with different multiplier configurations were implemented and synthesized as IC in order to determine the parameters. The required area, number of gates and clock speed were determined for a design that runs at 66 MHz for default and an alternative design with the possibly fastest clock frequency. Additionally, for every design the power consumption and required time were determined in a simulation environment by performing and measuring exemplary point multiplications. The results of these investigation are shown in Table 6.5.

The 233_8 design is about 20% smaller than 233_4, but is also much slower. The time for an execution of an operation on 233_8 takes about the threefold, compared to 233_4. This is the reason why the total energy for a full ECPM is the half for the four segment version compared to the implementation to the eight segment multiplier, even though the average power consumption is less. Thus, the 233_8 design is not very beneficial. It is slightly smaller but remarkably slower and has a higher total energy consumption compared to the four segment version.

Comparing the design with the four segment multiplier to the one using the two segment

Table 6.6.: Results for 233 bit ECC design on an FPGA

Setup	Size [4LUTs]	Size [FF]	speed [ns]	Needed clk cycles	Total time [ms]
233_8	6424	3192	9.8	38507	0.38
233_4	7767	3249	10.8	13164	0.14
233_2	11775	3520	11.4	7383	0.08

implementation, a more than 30% larger area is needed by the latter, whereby the required time is not even the half. Since the two segment multiplier is very complex with long paths, the maximum clock frequency is lower. Hence, the speed advantage, compared to 233_4, decreases to less than 30% (60μs to 80μs). Despite this and the higher average power consumption for 233_2, its total energy consumption for a point multiplication is more than 20% less than for 233_4.

It is a classical trade-off between time and area that is shown by the three designs. Faster execution time implies more required area. The 233_4 solution has the most convenient parameters, since it is not as large as 233_2, but much faster than 233_8. The choice, which design should be used in practice, eventually depends on the application area, in particular on the questions whether higher performance is worth more silicon.

We also implemented the designs on an Xilinx XC2VP70-7 FPGA and determined the area usage and the timing for this device. This FPGA implementation made it possible to test the design outside of simulations. The numbers for the three designs are shown in Table 6.6.

The data determined on the FPGA approve the conclusions made for the ICs. Compared to the four segment design, the eight segment version is only slightly smaller but much slower, whereby the two segment implementation is faster but also much larger.

Results of ECC designs with other bit sizes

Corresponding to the 233 bit implementation, designs with other bit sizes have been synthesized. Exemplary, the results for the five NIST curves are shown as Table 6.7. Hereby, the impact of the polynomial multiplication units with different configurations are additionally considered. The designs were synthesized with medium effort. This is why in particular the clock frequency does not reach the maximum possible values. Despite this, the numbers in the Table 6.7 and Figures 6.3 and 6.4 allow the following conclusions:

The bit size, execution time and the required silicon area are related about linearly. This means, double bit size leads to doubled execution time and a doubled silicon area. The total energy required for a point multiplication is about proportional to the square of the bit length. This implies that a doubled key length leads to fourfold energy consumption. This is reasonable,

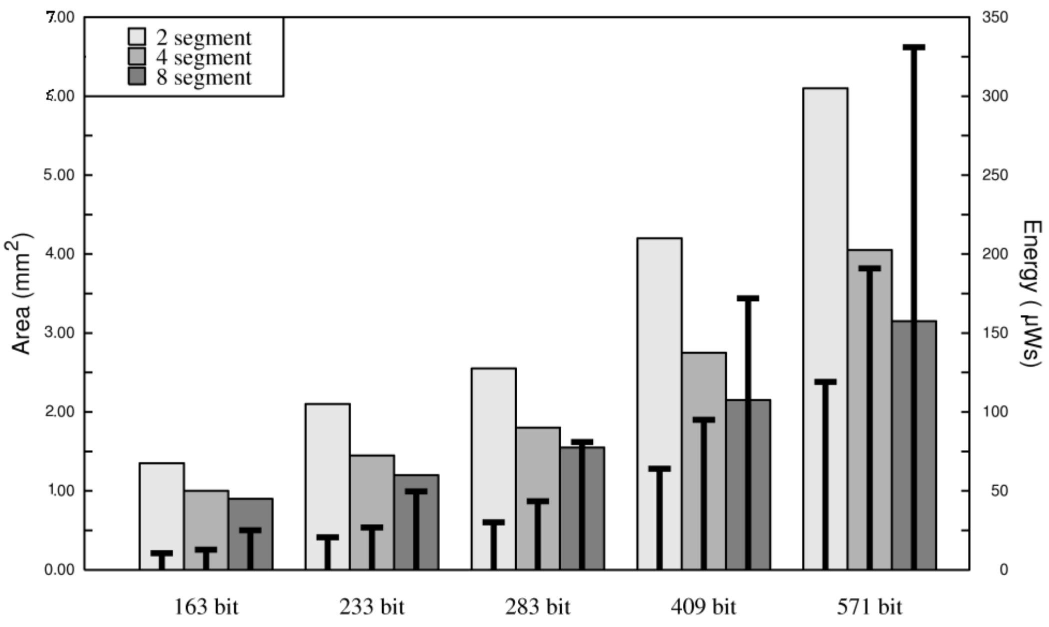


Figure 6.3.: The bars represent the silicon area and the lines show the corresponding energy for one complete ECPM. Slower designs (dark 8 segment solutions) are smaller, but due to slower execution time their total energy consumption is much higher than for the faster implementations.

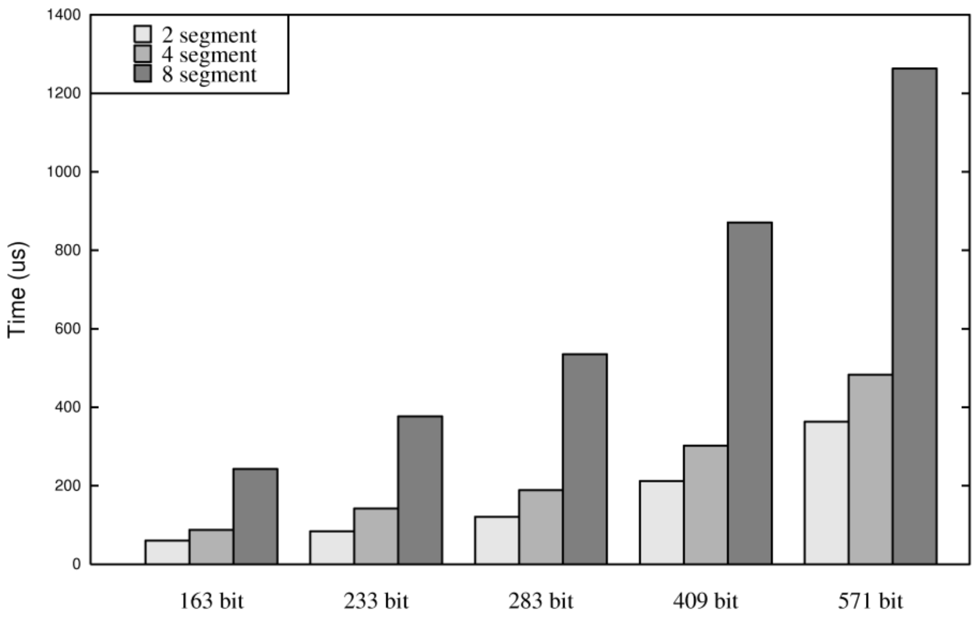


Figure 6.4.: Required time for one ECPM. The difference of time between (the brighter) 2 and 4 segment solutions is not as high as expected. The 8 segment versions are much slower.

Table 6.7.: Results for single curve ECC designs of different configurations.

Field size [bit]	Mul setup	Area [mm^2]	Total clk cycles	Speed [ns]	Total time [μs]	Power [mW]	Energy [μWs]
163	2	1.36	5344	11.3	60	66	10.6
163	4	1.02	9251	9.5	88	46	12.8
163	8	0.90	26999	9.0	243	31	25.1
233	2	2.05	7383	11.4	84	93	20.6
233	4	1.41	13164	10.8	142	68	26.9
233	8	1.20	38507	9.8	377	43	49.7
283	2	2.49	9215	13.2	122	109	30.1
283	4	1.87	15922	11.9	189	91	43.5
283	8	1.52	46567	11.5	536	58	81.0
409	2	4.20	13363	15.9	212	160	64.1
409	4	2.71	23080	13.1	302	138	95.6
409	8	2.11	67522	12.9	871	85	172.2
571	2	6.00	18655	19.5	364	230	128.7
571	4	3.92	32275	15.0	484	198	191.7
571	8	3.10	94321	13.4	1264	117	331.1

since in this case power consumption can be assumed to be directly related to the silicon area, which, as the execution time, has already been described as proportional to the bit size.
The results of the different multiplier configurations approve the conclusions made for the 233 bit design. Designs with faster multipliers require less energy for one ECPM than designs with smaller but slower multiplication units. Designs with two-segment multipliers are in average 30% larger than the four-segment versions, but do not provide the expected acceleration, due to the bus bottleneck and the fact that the large multiplier design cannot reach high clock frequencies. Compared to the eighth-segment designs, the four-segment versions are about three times faster but the increase of silicon area is not so high. Hence, designs with embedded mulitplier units that require nine clock cycles for one field mulitplication provide the best combination of performance, energy consumption and required area. Despite this, the other configurations can be preferred when time, area, or energy for an ECPM are constrainted.

6.3. Integration in a system on chip

In this section an exemplary real world application of the 233 bit ECC processor is described. The ECC block has been implemented in a system on chip (SoC) for network communication. This chip, which is really produced in silicon, contains a MIPS 32 bit RISC processor, SRAM memory, access blocks to a PC (Cardbus) and other periphery, several blocks for network processing and acceleration and cryptographic blocks for hash determination (MD5), symmetric stream cryptography (AES) and asymmetric cryptography (ECC). The blocks are

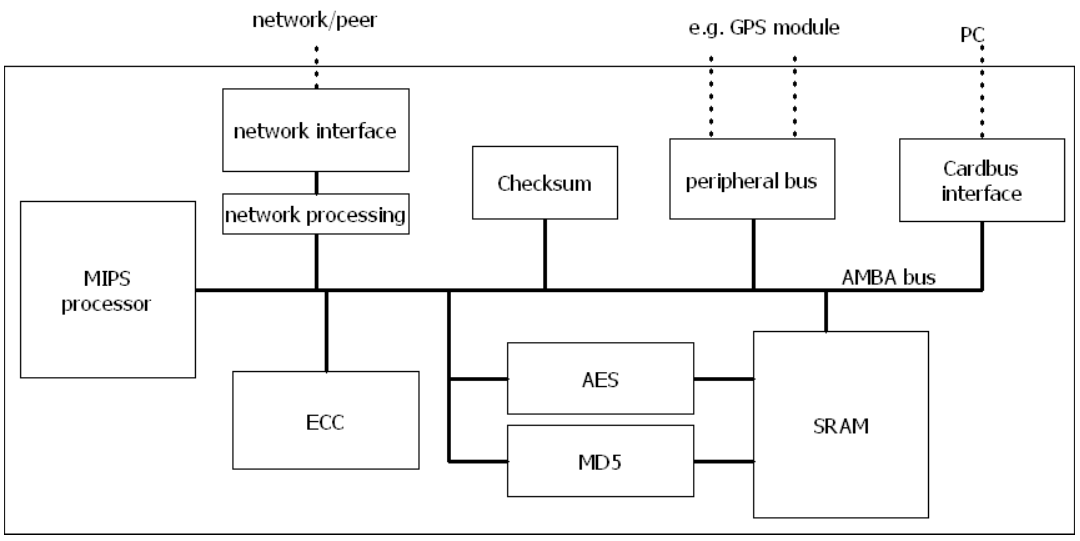


Figure 6.5.: Block diagram of the SoC. The ECC block is one part of the cryptographic engine. Processing of network protocols is another major focus of this chip. All blocks are interconnected using a 32 bit AMBA bus.

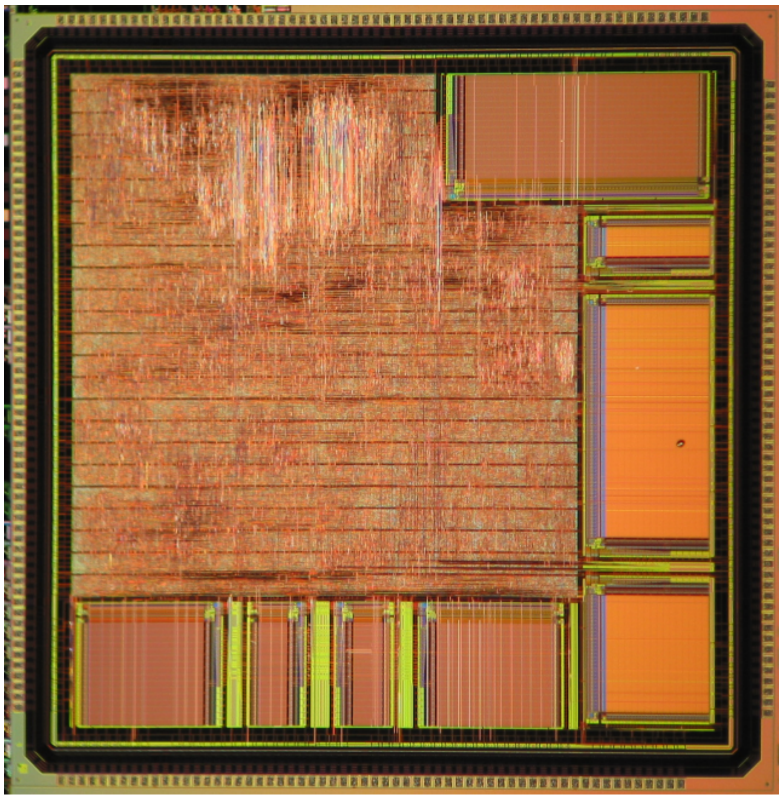


Figure 6.6.: The final system on chip. The ECC block is the top left and top middle in the picture. The metal routing in this area is noticeable due to the 233 bit internal bus. The rectangle blocks are memories.

```
//set factor k and start EC point multiplication
mm_write (ECC_REG0, 0x00000093); mm_write (ECC_REG1, 0x919255fd);
mm_write (ECC_REG2, 0x4359f4c2); mm_write (ECC_REG3, 0xb67dea45);
mm_write (ECC_REG4, 0x6ef70a54); mm_write (ECC_REG5, 0x5a9c44d4);
mm_write (ECC_REG6, 0x6f7f409f); mm_write (ECC_REG7, 0x96cb52cc);
mm_write (ECC_CMD , C_BUSY + ECC_C_MUL);
do mm_read(ECC_CMD, &v); while (v & C_BUSY);

//multiplication is done, get x coordinate of result
mm_write (ECC_CMD , C_BUSY + ECC_C_GET_X);
do mm_read(ECC_CMD, &v); while (v & C_BUSY);
mm_read (ECC_REG0, &res_x[0]); mm_read (ECC_REG1, &res_x[1]);
mm_read (ECC_REG2, &res_x[2]); mm_read (ECC_REG3, &res_x[3]);
mm_read (ECC_REG4, &res_x[4]); mm_read (ECC_REG5, &res_x[5]);
mm_read (ECC_REG6, &res_x[6]); mm_read (ECC_REG7, &res_x[7]);
```

Listing 6.1: Exemplary C-program that sets factor k, starts the ECPM, waits for termination, and reads the x-coordinate of the result.

connected by a 32 bit AMBA bus [25]. The block diagram of the chip is depicted as Figure 6.5, and a photo of the final chip is shown as Figure 6.6.

The 233 bit ECC design with the four segment multiplier was chosen as the ECC block. The implemented ECC block corresponds exactly to the design depicted in Figure 6.2. The external bus is an AMBA 2.0 high performance bus (AHB). This is an arbitrated and address based bus architecture. The ECC block has nine addresses in this bus system. One address for the command word and eight for the 233 bit external register. When the ECC block is accessed for a write operation and the address of command word is matching, the internal controller unit interprets the command word and starts an internal program. When the program is finished, a busy flag in the command word will be taken down. This is the signal for the system that the operation is done.
The ECC block can be accessed by every block that is able to write on the bus. In the system for example the MIPS and the Cardbus interface can access to the ECC. This allows to program the ECC block within the chip (MIPS) or through an external computer. An example how a MIPS C-program can access the ECC block is shown in Listing 6.1. It is assumed that Parameter B and the coordinates of the base point have already been written. Hence, after the factor k is set, the point multiplication starts. When the multiplication is done, the x-coordinate of the resulting point is read. In this example the MIPS program waits active by polling the busy bit until it is taken down. An alternative passive wait is also implemented into the system. Hereby, the MIPS goes into sleep mode after the EC multiplication is set up, and is woken up when the operation is done.

Table 6.8.: Comparison of software implementations of 163 bit ECs on different CPUs.

CPU	Time [ms]	Power [mW]	Energy [mWs]	Size kByte	Comment
MIPS R4000 33MHz	200	35	7.0	48	based on MIRACL[45]
MIPS R4000 33MHz	900	35	31.5	14	
Atmel ATmega128 7.4MHz	6000	20	120.0	12	time for 160 bit signature part of the TinyECC[26]
Motorola Dragonball 16Mhz	2700	50	135.0		described in [54]
StrongARM 206MHz	9	400	3.6		described in [39]
Pentium II 400MHz	3.2	28000	90.0		described in [13]

6.4. Conclusions

6.4.1. Comparison with previous software solutions

In the last section the ECC integration into a SoC with a RISC processor was described. A software implementation of the ECC operations is a reasonable alternative, since the processor is part of the system anyway. Therefore, an ECC software implementation based on the MIRACL library [45] was tested. On the system with the MIPS processor running at 33MHz, a point multiplication on the 163 bit NIST curve requires about 200ms, performed by the MIRACL library. A point multiplication on the curve B-233 takes 13 million clock cycles which equals 400ms. The code size for this implementation is 48 kilobytes. An alternative implementation requires only 14 kilobytes but is with 900ms for a point multiplication much slower. The code size must be considered when memory is an issue, as it is for many small mobile devices.

Table 6.8 shows a brief comparison of ECC software implementations for different CPUs with emphasis on solutions for mobile solutions. Due to the different architectures of the CPUs and the different technologies, this overview provides only rough hints. Beside the required time for a point multiplication on a 163 bit curve, the energy consumption for the operation is estimated.

Depending on the selected hardware and the effort on the optimization of the software, the time for a 163 bit point multiplication varies from few milliseconds to few seconds. The total energy consumption for the operation varies from 3.6mWs on the very efficient StrongARM implementation to 135mWs on the Dragonball processor. The comparison shows that the

Table 6.9.: Comparison of the 233 bit ECC hardware design to a MIPS solution on the SoC.

	Time[ms]	Power[mW]	Energy[mWs]
Software	410.2	40.2	16.490
Hardware	0.4	75.6	0.030

```
MIRACL: 233 bit GF(2^m) Elliptic Curve....
done!
cycles = 13535688
410,172 ms
result=
AC84A5BF1C94F9BA29D74A39D72D33E42549BC2A2D2F8D89145A46B438

HW: 233 bit GF(2^m) Elliptic Curve....
done!
cycles = 13362
0,405 ms
result=
AC84A5BF1C94F9BA29D74A39D72D33E42549BC2A2D2F8D89145A46B438
```

Figure 6.7.: Output of a program, running on the MIPS on the SoC, that utilizes the MIRACL library for the computation of a 233 bit ECPM, followed by the same operation on the ECC block.

MIRACL based MIPS implementation is a very reasonable software implementation.

Table 6.9 shows a comparison of the 233 bit MIRACL/MIPS software implementation to the ECC hardware design as both are implemented and running on the communication SoC. All the data were measured in the simulation environment (NC-Sim[14], PrimePower[49]) for one 233 bit point multiplication at a speed of 33MHz. The correctness of the operation as well as the practical timing were also verified on an FPGA. An a test of the SoC on the FPGA, a MIPS program that performs a the MIRACL software ECPM and afterward the ECPM on the 233 bit ECC block, produces the output shown as Figure 6.7. It practically approves the correctness of the data in Table 6.9, which were obtained in simulations.

In the table, one can see that the hardware solution is 1000 times faster and consumes 550 times less energy in comparison to the software implementation.

6.4.2. Comparison with previous hardware approaches

Table 6.10 shows a comparison of previous hardware implementations for the acceleration of EC scalar multiplications. High performance implementations with field sizes around 163 bit were chosen, because these field sizes are supported in the most implementations that have been presented. Due to different hardware configurations and different amount of functionality the numbers cannot be compared directly.

The design presented in [44] supports not only ECs based on binary extension fields but also curves on prime fields $GF(p)$. This renders the design that is presented as ASIC running at 510Mhz to the most configurable EC coprocessor. The hardware proposed in [12] also supports not only one curve but all ECs based on binary extension fields $GF(2^m)$ up to a size of $m = 256$. The design described in [32] is a very area efficient implementation of a EC based on $GF(2^{167})$.

Table 6.10.: Comparison of $GF(2^m)$ ECPM hardware designs.

Ref	field	Platform	ECPM time	size
our	GF(2^{163})	0.25 μm ASIC	0.06	1.36mm^2, 47 Kgates
our	GF(2^{163})	Xilinx XC2VP70	0.06	7400 LUTs
[44]	GF(2^{163})	0.13μm ASIC	0.19	117.5 Kgates
[12]	GF(2^{163})	Xilinx XCV2000E	0.14	19508 LUTs
[32]	GF(2^{167})	Xilinx XCV400E	0.21	3002 LUTs
[43]	GF(2^{191})	Xilinx XCV3200E	0.06	18314 CLB slices est. $\approx$30000 LUTs
[55]	GF(2^{191})	0.35 μm ASIC	6.21	1.31mm^2
[8]	GF(2^{233})	0.13 μm ASIC	6.68	71 Kgates

It does not reach the speed of the fastest designs but is very small. With a LUT number of 3002 it requires about the half of the area of our corresponding design on the FPGA. In contrast, there is a threefold increase in time compared to our implementation. The fastest known implementation has been reported in [43]. This design performs a ECPM on $GF(2^{191})$ within less than 60μs. On the downside this single curve implementation requires a huge amount of area. For comparison reasons we estimated the reported 18314 CLB slices correspond to 30000 LUTs. Our fastest implemented design of a corresponding curve is slightly slower but requires merely a quarter of the area. The ASIC design presented in [55] is another very small design. The ASIC manufactured in a 0.35 μm technology has a size of 1.31mm^2 and supports two fields, but requires more than 6 ms for an ECPM. The power consumption reported for this design is 213μWs for an ECPM in $GF(2^{191})$. It is, due to the poor performance, about the tenfold of our design. The commercially offered design [8] is the only other design that reports the power consumption. A 233 bit operation requires on the 50 MHz design, manufactured as 0.13 μm ASIC, 6 ms and a total energy of 140μWs.

In [43] and [44] comprehensive overviews of other previous designs are given, which were not considered relevant in this brief comparison.

7. Flexible hardware designs for ECC

In this chapter possible adaptations of the presented ECC hardware designs that provide flexibility are investigated. First, the meaning of flexibility in terms of ECC is specified. Afterward, the description of implications for ECC acceleration hardware precedes the discussion of possible adaptations of the field operation units. The actual evaluation and implementation of the flexible ECC designs is concluded by a comparison and discussion of the results.

7.1. Flexibility

ECC is working on a selected subgroup of the group of points on an elliptic curve. These points are elements of a large finite field. This field can be parametrized as well as the EC and the subfield. The possible parameters, whose adjustability is the key for flexibility, are described below and represented in Figure 7.1:

Subgroup: The subgroup is characterized by the underlying elliptic curve and a base point on this curve. Flexibility on this layer means that it is possible to calculate with different base points. Other base points supply other subgroups with other subgroup orders. It is conceivable that particular subgroups have more cryptographic strength than other ones. This is why the flexibility of selecting the base point, and therefore the subgroup, is a recommended property of ECC designs.
Actually, implementations that are fixed to one special base point, are not very common, because the limitation to one base point does not have improving attributes. It is sometimes considered for software implementations where very large lookup tables are used to accelerate the operations for one subgroup.
Our current implementation is not fixed to one subgroup. Hence, the flexibility of the subgroup is given, anyway.

Elliptic Curve: The equation $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ defines all elliptic curves. In practice only special cases of this equation are used. On prime fields usually EC $y^2 = x^3 + ax + b$ is applied, on binary extension fields $y^2 + xy = x^3 + ax^2 + b$ is used. For binary Koblitz curves b is always set to 1 and for the termed pseudo-random curves such as the recommended NIST curves, which we are steadily referring to, a is always set to 1. These specific equations are connected with tailored point operations. In particular point multiplication, point addition and doubling are affected by the choice of the curve. For hardware implementations, flexibility is determined by the way the point operations are realized. If the algorithms of the point operations are programmable, the design is able to handle different curves.

In our current design, point operations are determined by the control logic in the control block. Since this control logic is hard wired, it is currently not possible to change the curve at runtime. Beside the control logic and possibly the number of registers, there are no constraints or for special curves in the current design.

Finite field: The coordinates of points on elliptic curves are members of a finite field. Parameters of this field are the size and the irreducible polynomial that describes the field.

For hardware implementations, the maximum field size is a natural limitation. Flexibility on this layer means that all smaller fields can be applied. The main problem is that not only the field size varies but the irreducible polynomials that also define the field operations are specific for every field. This interferes with the reduction step, which must be performed after every multiplication. Both software and hardware designs usually provide very tailored implementations for this operation, in order to obtain high performance.

Our current designs are also optimized for exactly one field of one size and one reduction polynomial each.

Base of the finite field: All finite fields are of the structure $GF(p^m)$. Usually only prime fields $GF(p)$ and binary fields $GF(2^m)$ are used for ECC in practice. Both kinds of fields have very special algorithms and algebra. This is why implementations that support both field types are not common. Total flexibility on this layer would mean that all fields $GF(p^m)$ are supported.

Our implementations only apply binary fields $GF(2^m)$.

Independent from the layer, flexibility generates costs. These costs are:

Performance: Applying special properties of specific fields or curves leads to very efficient implementations with high performance. Algorithms that work on all curves are not as efficient as tailored algorithms.

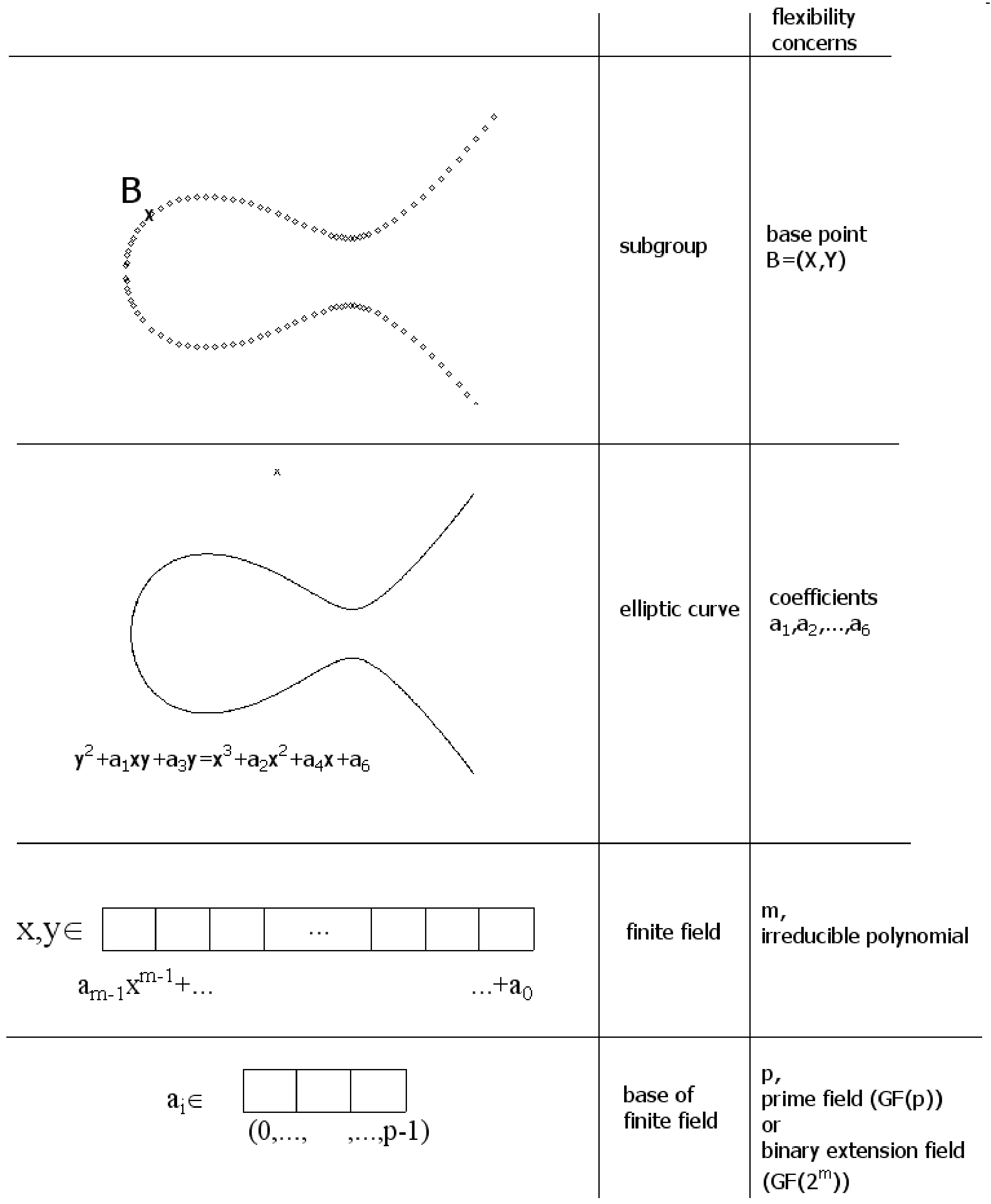

flexibility concerns
B
subgroup
base point B=(X,Y)
$y^2+a_1xy+a_3y=x^3+a_2x^2+a_4x+a_6$
elliptic curve
coefficients $a_1, a_2, \ldots, a_6$
$x,y \in$
$a_{m-1}x^{m-1}+\ldots$
$\ldots+a_0$
finite field
m, irreducible polynomial
$a_i \in$
$(0,\ldots, \quad ,\ldots,p-1)$
base of finite field
p, prime field (GF(p)) or binary extension field ($GF(2^m)$)

Figure 7.1.: Representation of the four layer of flexibility for ECC.

Size: For hardware this means silicon area, for software it is code size. Flexibility means considering parameters, behavior and algorithms for more than one curve. Providing support for every parameter and every behavior requires additional size. Especially flexible implementations in software tend to include efficient algorithms for many different curves.

It follows that there is a trade-off between flexibility, performance and size concerning ECC-implementations. High flexibility leads to less performance and/or to more need of space. High performance implementations are very tailored and therefore inflexible. Very small implementations are usually not very fast and do not support many curves.
The contradiction of the three goals can be demonstrated by hardware implementations that have been reported previously. A brief summary of these designs is shown in Table 6.10. The fastest design has been reported in [43]. It is also the largest but not flexible, since it supports only one curve. The most flexible design is the cryptographic coprocessor reported in [44]. It works on all prime fields as well as on all binary extension fields. Even though the design is reported as running at 510Mhz on an ASIC, its performance is a third of the one of the fastest design. Small designs such as [55] and [32] present the least performance of the list and are bound to two and one curve, respectively.
A particular trade-off between performance and flexibility is the choice between software and hardware implementations. Hardware implementations obtain much more performance as it was shown in Table 6.9. This high performance must be paid with less flexibility than pure software. To close the gap between software and hardware, two approaches are known:

Hardware/software co-implementation: The hardware accelerator contains common functionality which can be used for different fields and curves. How this hardware accelerator is applied, is determined by a software program in a more complicated controller or even a microprocessor.

Reprogrammable hardware: Since in particular FPGAs obtain fast hardware designs which can be reprogrammed, they connect the speed of tailored hardware with the changeability of software.

The cryptographic coprocessor described in [44] is an example for a very flexible hardware/software co-solution. The design presented in [32] is an example for a hardware implementation that obtains flexibility by reconfigurable properties of FPGAs.
Our hardware designs, which were described in the previous chapter, can be classified as an entity of the designs that attain flexibility due to FPGA implementation. Little effort is

required to generate tailored implementations for every EC based on binary extension fields. Indeed, this kind of flexibility is futile for ASICs. Here, flexibility must imply a possibility to change the parameters at runtime.
An ASIC that supports the five NIST ECs B-163, B-233, B-283, B-409, and B-571 on an ASIC requires flexibility in the following parameters:

Subgroup: Since the recommended base point is different for every curve, this parameter must be adjustable.

Elliptic Curve: Every of the five ECs satisfy the equation $y^2 + xy = x^3 + ax^2 + b$, whereby a is always 1 and b is a specific parameter for each EC. Hence, b must be configurable. It is a parameter for the point operations, in our design in particular the Montgomery point multiplication.

Finite field: Each of the five ECs has its own field size and irreducible polynomial. The field size has an impact on every functional unit, registers, and bus width. The irreducible influences the reduction operation.

Base of the finite field: The five fields are binary extension fields $GF(2^m)$. This is why no flexibility on this layer is required.

Since the desired flexibility for the subgroup and the elliptic curve are already provided, the variable parameters of the finite field, the length m and the irreducible $r(x)$ are the only new parameters that must be considered. The flexible length issue can be solved while smaller fields fill the higher bit positions with zeros. The variable irreducible that influences the reduction, which must be performed after every multiplication and squaring operation, is much more a crucial problem. This is why in the following sections, where possible adaptations of the functional units are discussed, the main focus is placed on the reduction issue.

7.2. Reduction

As mentioned earlier, the reduction is the most challenging part of a flexible ECC implementation, because it is the only operation that directly uses the irreducible polynomial. Also, it is a very often applied operation, since it is executed after every field multiplication and every squaring. This is why a fast and efficient execution of this operation is indispensable.

7.2.1. Multiple hard-wired reduction

For single curves the best solution is a hard wired application of the fast reduction approach. This method allows to perform a reduction very fast, even within the same clock cycle the

Table 7.1.: Area consumption of combinatorial reduction blocks. Single is the area for a single curve reduction (see Figure 5.5), Accum. are the simply added areas of the single blocks. Combined is the actual area for a reduction block that can reduce the named fields up to the specific size.

Size	Area [mm^2]		
[bit]	Single	Accum.	combined
163	0.045	0.045	0.045
233	0.034	0.079	0.092
283	0.076	0.154	0.179
409	0.058	0.213	0.250
571	0.159	0.372	0.439

multiplier or square unit release their result. The disadvantage of this approach is that it exclusively works for one field. It is not possible to use such a reduction block of a larger field for smaller ones. But since other approaches are either very large or very slow, it is feasible to implement more than one of these blocks and select the one for the corresponding curve. In cases where the alternative curves are known, this method could result in a fast and competitive implementation. To examine this proposition, we implemented reduction blocks for the five NIST curves where each reduction block contains all smaller polynomials. Thus, the block of the 571-bit size includes reduction functionality for the 409, 283, 233 and 163 bit curves beside the 571 bit field. Figure 7.1 shows the resulting area consumptions of the combined reduction blocks. It also compares the area of the combined reduction blocks with the accumulated sizes of the standalone blocks. The results of this evaluation is that the overhead for the selection of the reduction block requires about 15% additional space. Larger blocks of reusable combinatiorial logic or intermediate results that appear in multiple reduction blocks, and therefore reduce the total effort, could not be observed.

7.2.2. Repeated multiplication reduction

In Section 3.4 the method of multiplicative reduction was described. The pure method with repeated multiplications was considered as slow and not efficient in the discussion of the single curve reduction. Despite this, it is the only reported approach for flexible hardware reduction for ECs in $GF(2^m)$ that is not bound to specific irreducible polynomials. In [44] every reduction is performed by repeated multiplications of the irreducible and the overlapping part of the word. [7] presents a combined approach that performs reduction in specific named fields by hard wired reduction blocks, other fields are reduced by the repeated multiplication method. The approach of repeated multiplication can be applied to every design that uses a polynomial multiplier. A possible data flow of such a design is depicted as Figure 7.2. In the block diagram it is presumed that the reduction can be performed within two reduction steps, which implies that the second highest power in the irreducible is less than $m/2$. Under these

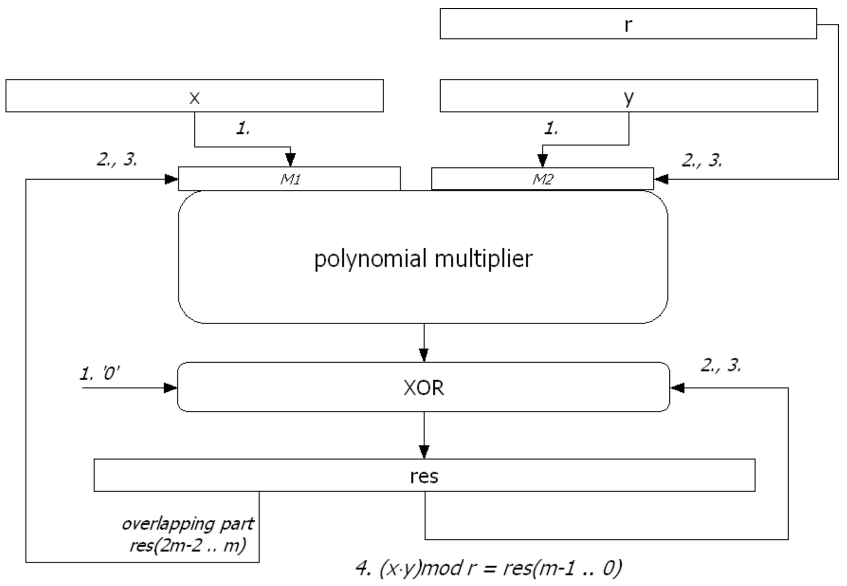


Figure 7.2.: Block diagram of a multiplicative reduction design that is independent from the reduction polynomial. The actual multiplication (1.) is followed by two multiplications of the overlapping part with the irreducible.

conditions, a multiplication with subsequent reduction can be performed within three steps:

```
1.    $res \leftarrow x \cdot y$                          //actual multiplication
2.    $res \leftarrow res[2m-2 .. m] \cdot r \oplus res$   //first reduction step
3.    $res \leftarrow res[2m-2 .. m] \cdot r \oplus res$   //second reduction step
(4.)  res contains the reduced result of the reduction
```

Hence, this approach requires two additional full polynomial multiplications. Since the multiplications are already the bottleneck in the ECC implementation, a significant decrease of the performance is expectable. Also, the area consumption increases significantly, because additional registers are required as well as a large logic for the selection of the overlapping parts in smaller fields. Since this reduction approach is directly connected with the multiplier, no discussion of area or speed impacts apart from the multiplication unit is reasonable. The effects of the method are evaluated in Section 7.3.

7.2.3. Flexible fast reduction

In 3.4 the following properties of commonly used reduction polynomials were described:

- Since the common used reduction polynomials are trinomials or pentanomials, a multiplication with these polynomials is not more than adding five shift results of the overlapping part.

- Since the second highest set power in the reduction polynomial is less than the half of the degree, only two successive multiplications are required for a complete reduction.

These properties, which are valid for all NIST curves, are a basis for a much more efficient implementation of the multiplicative reduction. In the following, a hardware design is described that realizes a flexible fast reduction unit. This means, a design that works for different reduction polynomials and for other field sizes, without losing as much performance as in case classic multiplicative reduction approaches are in use.

Figure 7.3 shows the approach that realizes a reduction for irreducible polynomials having the described properties. It is a method that reduces data words up to a specified length for every irreducible polynomial, as long as it is a trinomial or pentanomial. The maximum length is determined by the physical word size of the architecture. The picture shows the reduction process of a polynomial of full size in part a) and processing of a smaller polynomial in b). The full size reduction is quite obvious, since it is very similar to the basic idea of the fast reduction, described in Section 3.4.3. The overlapping part is shifted to the positions corresponding to the positions set in the reduction polynomial and is subsequently added to the word. After the first reduction step, more than the half of the overlapping part is zero. The positions of the data word that are set to zero are represented as shaded area in the picture. After the second step, which contains exactly the same operations as the first one, the complete reduction process is done.

Generally the scheme for smaller polynomials is the same as shown in part b) of Figure 7.3. The only difference is that the shorter word must be aligned, so that the least significant bit of the overlapping part has the same position as it does for the largest polynomial supported by the reduction unit. It corresponds to left shift of the input word by $(n - m)$, whereby n is the maximum supported field bit length of the hardware and m is the length of the calculated field. Thus, the initial word is padded with zeros at the left and the right end. Beside this, the reduction process is completely the same as the process for the full word.

For example, consider a reduction block for a polynomial size of 233 bit. In this block we want to reduce the recommended polynomial of the 163 bit NIST field. The reduction polynomial is $x^{163} + x^7 + x^6 + x^3 + 1$. For the alignment the input word is shifted left by the difference of field sizes, in the example 70 bits (233-163=70). This means that the shaded area of the input word of Figure 7.3 b) is 70 bits at the beginning and the end. The four shifting values are: $x3 = 163 - 7 = 156$, $x2 = 163 - 6 = 155$, $x2 = 163 - 3 = 160$, $x2 = 163 - 0 = 163$. After the second step, the reduced word must be shifted back by 70 to obtain the usual right aligned data word.

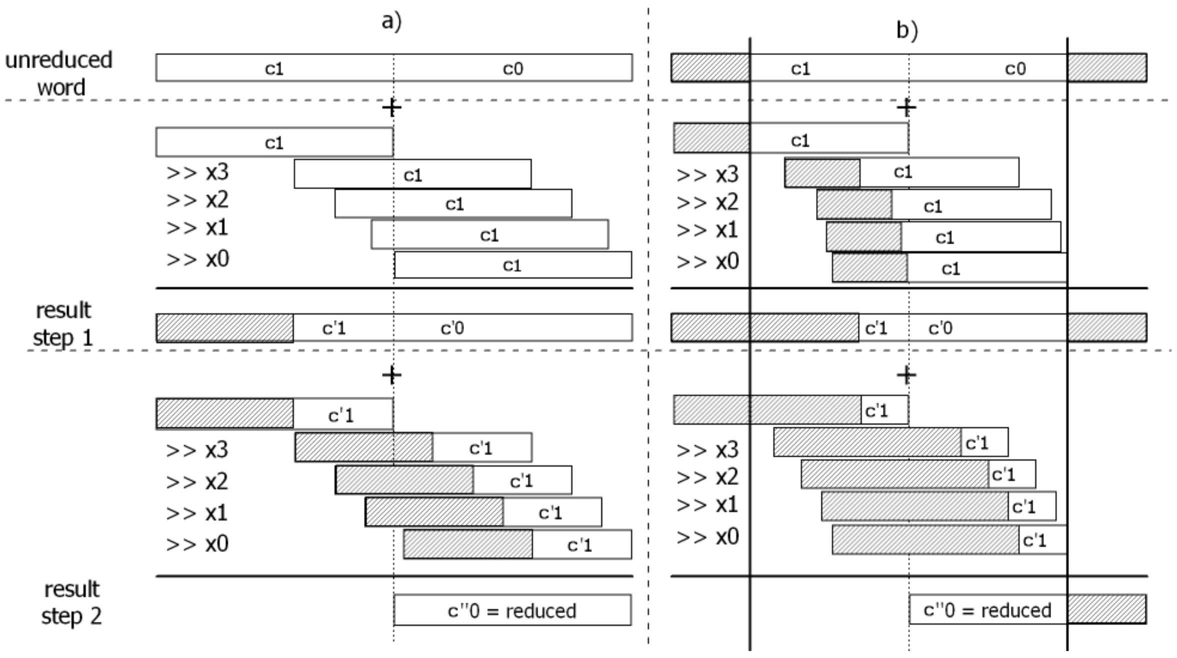


Figure 7.3.: Scheme of a flexible reduction method. The left part a) shows a full size reduction. The right part b) is a reduction for a smaller curve. When the smaller polynomials are initially aligned so that the overlapping part starts at the same position as long polynomials, the same reduction logic can be used. The shaded parts represent zeros. The configuration of the reduction is determined by the four shift values x1, x2, x3, x4.

After the approach was described in VHDL, the correctness of the idea could be verified in the behavioral simulation. But after synthesizing the design, it becomes apparent that especially the shifters are connected with a crucial problem. Flexible shifters with a size of hundreds of bits are very slow and large. A design that corresponds to the idea of Figure 7.3 requires:

- one left-shifter for the smaller input words
- four concurrent right-shifters for each of the two steps
- one right-shifter for the smaller output words

Therefore, ten large flexible shift operations are required for one reduction.
This problem can be reduced by minimizing the required number of shift operations. One approach would be the limitation of the irreducibles to trinomials, which decreases the total number of shift operation by four. On the downside, the elimination of the pentanomials would be a significant loss of functionality.
An approach that avoids one of the four concurrent shifters without losing functionality is depicted in Figure 7.4. All irreducible polynomials have the structure: $r(x) = x^m + ... + 1$. Hence, the terms x^m and 1 are part of every reduction polynomial. In the original scheme, the partial product that corresponds to x^m is not shifted, but the term that belongs to 1 is shifted right by $x0$. Since $x0$ is the difference of m and the lowest set power, which is always

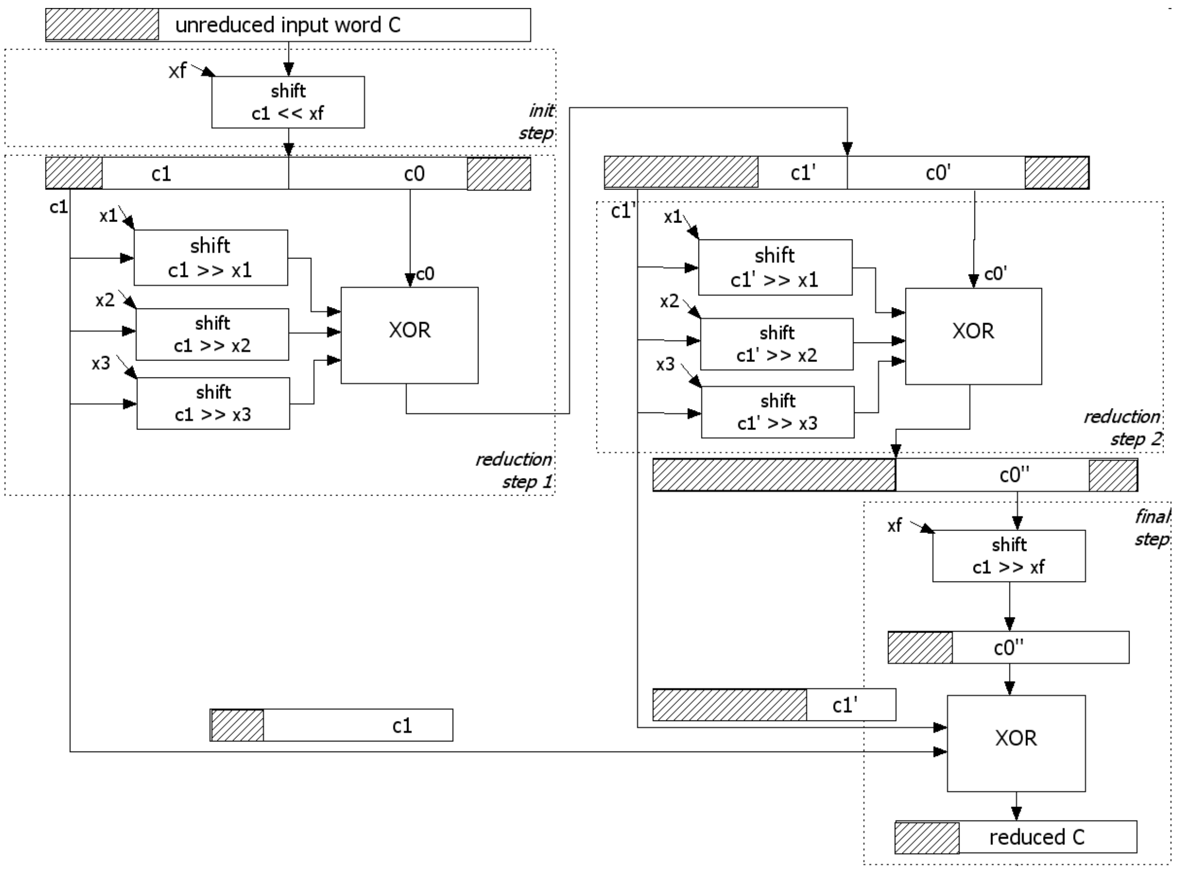


Figure 7.4.: Scheme of the optimized flexible shift reduction method. The input is shifted in the init step to obtain the middle alignment. Then the two reduction steps are performed with three shift operations each (x1,x2,x3). The fourth shift (x0) can be saved when it is forwarded to the final step and accumulated after the final right shift.

0 ($x^0 = 1$), $x0$ is alway equivalent to m ($m - 0$).

In the final step of the reduction, the intermediate result, which is still aligned to the middle, is shifted so that it is aligned to right. The amount of the final right shift is: $xf = n - m$, whereby n is the physical word length and m is the size of the calculated field.

Consider now, that the final result of the reduction is res and the final result without summarizing the $x0$-terms is res^*. The terms $c1$ and $c'1$ are the overlapping parts corresponding to Figure 7.3. Then

$$res = res^* + [(c1 >> x0) + (c'1 >> x0)] >> xf$$

Substituting $x0$ and xf one obtains:

$$res = res^* + [(c1 >> m) + (c'1 >> m)] >> (n - m)$$

which can be written as:

$$res = res^* + [(c1 + c'1) >> m] >> (n - m)$$
$$res = res^* + [c1 + c'1] >> n$$

Finally, the $x0$-shift can be substituted by a shift by n. This n depends on the hardware design and not on the calculated field. Therefore, this shift operation is not an expensive flexible shift operation but a cheap readdressing. With this improvement still eight shift operations for a reduction are required.

In spite of the issues connected with the large shifters, it is possible to build and synthesize a design that is corresponding to the scheme and that performs the reduction within one clock cycle. A 233 bit version of the reduction block requires about 1.3mm^2 in silicon after synthesizing at a frequency of 70 MHz. This design computes the polynomial reduction within one clock cycle in every field up to a degree of 233 as far as the reduction polynomial is a trinomial or pentanomial.

Experiments with sequential applications of this reduction process have shown that less silicon area and higher clock frequencies can be achieved. But the gain is not as high as expected. A design of this approach, taking two clock cycles, reduces the required area slightly to 1.2mm^2, and does not improve the clock frequency. A four clock cycle version still requires 1.1mm^2 with a clock frequency of 100 MHz.

The large area and relatively low clock speeds are a big disadvantage for this approach. Further investigations could improve the parameters. The presented design can reduce numbers of all field sizes up to a specific size, also e.g. $GF(2^3)$. A minimum field size of, for example, 150 bit will reduce the complexity of the shifter logic.

The benefit of this flexible shift approach is that it can perform a flexible reduction in very few clock cycles. In literature no comparable approach, neither the design nor the performance, could have been found.

7.3. Polynomial multiplication

Multiplication is the most important and most complex operation in the base fields of ECC. For the purpose of a flexible ECC implementation, the polynomial multiplication itself is not a big issue. In order to multiply a smaller polynomial in a larger field hardware it is only necessary to pad the small polynomial with zeros. The challenge is the followed reduction step, which is very depended from the field reduction polynomial.

In the last section flexible reduction hardware approaches were discussed. These flexible reduction blocks can substitute the embedded reduction blocks of the single field multiplication

Table 7.2.: Overview of flexible multiplication units supporting several fields in $GF(2^m)$.

Size	contains fields:						Area	Speed	
[bit]	163	233	283	409	571	all upto	[mm^2]	period[ns]	cycles
Multiple hard wired reduction (MHWR)									
233	X	X				-	0.67	10	9
283	X	X	X			-	0.94	11	9
409	X	X	X	X		-	1.54	12	9
571	X	X	X	X	X	-	2.37	15	9
Flexible shift reduction (FSR)									
233	X	X				233	1.72	20	9
255	X	X				255	1.94	21	9
283	X	X	X			283	2.30	23	9
Repeated multiplication reduction (RMR)									
233	X	X				233	1.22	11	30
233	X	X				233	1.70	13	12
255	X	X				255	2.85	12	30
255	X	X				255	1.40	14	12
283	X	X	X			283	1.60	13	30
283	X	X	X			283	2.18	15	12

designs as they were presented in Figure 5.11. For this functional extension, the multiplier obtains additional input signals for the selection of the reduction polynomial. It is premised that smaller polynomials are padded with zero to meet the full multiplication factor length. Table 7.2 shows an overview of multipliers for different fields applying the reduction methods as they were described in the last section. Concretely, the three considered approaches are:

- Multipliers with multiple hard wired reduction (MHWR) logic for several fields
- Multipliers that apply the flexible shift reduction (FSR) approach
- Multipliers that perform the reduction by the repeated multiplications reduction (RMR) method

The core multiplier is a four segment RAIK multiplier, except for the faster RMR designs, which apply the faster two segment RAIK to obtain a comparable speed to the other designs. One can see that the MHWR approach is very efficient. Both area and speed are barely affected by the additional reduction functionality. Compared to the single field multipliers (Table 5.8), merely ten percent additional area is required to support all smaller NIST ECC fields.
Contradictory results are observed for the FSR implementation. The area for these designs is more than double as large compared to the single field multipliers. Limiting the number of fields, for example setting the minimum field size to 150 bit, can reduce the complexity of the logic and accelerate clock frequency. Also inserting an extra clock cycle for the reduction at the end of the multiplication could be an option in further designs. For the four-segment IKM this would mean ten clock cycles instead of nine for one complete polynomial multiplication.

The RMR approach already requires additional clock cycles. With the same core multipliers this approach takes more than the threefold number of clock cycles. The next faster core multiplier, which is in the measurements the two segment RAIK, reduces the clock cycles but increase the area consumption to the level of the flexible shift multiplier. The benefit of this approach is that the possible clock frequency does not decrease as much as for the FSR multipliers.

The results show that flexibility for the Galois field multiplication is connected with extra costs. Additional time or area consumption is inevitable. The costs for the designs with the MHWR are low, whereby the flexibility is merely a selection between a number of curves. The additional costs for real flexibility are immense. Both approaches that allow all smaller fields, require much additional space and time.

7.3.1. Flexible iterative Karatsuba approach

The flexible polynomial multipliers discussed so far, took use of the fact that zeros can be padded to the higher bits when short polynomials should be multiplied. Thus, for example a 163-bit multiplication on a 571-bit hardware is always performed with 408 leading zeros. Obviously, this is not very efficient.

Consider for example the 8-segment 571-bit multiplier reported in Table 5.8. This multiplier needs 27 clock cycles for a polynomial multiplication. Internally it applies an 80-bit combinatorial multiplier. Thus 27 partial 80-bit multiplications are performed for one 571-bit multiplication. When executing a 233-bit operation with this hardware, the leading 338 bits are set to zero and again 27 partial 80-bit multiplications are performed. It is not surprising that at more than the half of the partial multiplications at least one factor is completely zero. The idea is now to skip these zero-multiplications. The separated control structure and data path of the multiplier, which was introduced in Section 5.4.2, supports this idea very well. Thus the extension of the control logic can be done without modifying the data path. Due to the skipped partial multiplications, the resulting design performs multiplications with smaller factor lengths significantly faster.

Concretely, when n is the length of the internal combinatorial multiplier, multiplications with a factor length up to n are done within one clock cycle. Multiplications with factor lengths up to $2n$ bit are done after three clock cycles, $4n$ bit factors take nine and $8n$ bit 27 clock cycles. In case of an 80 bit combinatorial multiplier, field multiplications of the NIST curves 163, 233 and 283 bit are performed in nine clock cycles and 409 and 571 bit multiplications take 27 cycles.

Table 7.3 shows some configurations of the large fields, the corresponding required clock cycles for the embedded smaller fields and the energy consumption for a full multiplication and a multiplication in the 163 bit field.

A comparison shows that the power consumption for the flexible multipliers is significantly higher than the consumption for the corresponding single field implementation. The increase of the energy consumption for a multiplication with full utilization of the hardware is in average 33%. This raise is caused by the more complex combinatorial reduction logic. Actually, in the 571-bit mulitplier all supported field reductions are performed, and a multiplexer finally selects the needed result. This approach is far away from a sophisticated low power design and therefore the 33% raise is not surprising. Another evaluation showed that in the current flexible design, the reduction takes about 40% of the total energy of a mulitplication. For single curve implementations the amount is less than 10%.
Furthermore the huge increase in power consumption of the 163-bit multiplication within the large multipliers is noticeable. Indeed, the needed energy in smaller fields is much less than for the full field operation, which is caused by the flexible IKM approach. But compared to the 163-bit single curve implementation with equivalent speed, the power consumption of a 163-bit multiplication within the flexible 571-bit hardware is more than doubled. The first thought, that again the reduction is the reason for this increase, is wrong. Since a large portion of the data words is always zero, large parts of the reduction logic are not changing and do not consume additional power. An important reason for the high power consumption is the high number of additional, actually not used flip flops. Concretely, a flexible 571-bit multiplier contains 2055 flip flops. In contrast, the 163-bit single field multiplier with the same speed only contains 703 flip flops. Another reason is the fact that the size of the internal combinatorial multiplier is not fitted to the size of smaller field multiplication. For example, the 163-bit multiplication in the 571-bit hardware with a 160-bit combinatorial multiplier is kind of a worst case. It requires three partial 160-bit multiplications to perform one 163-bit field operation. An 82-bit partial multiplier is sufficient to execute this operation in the same time.

Table 7.3.: Results of large flexible MHWR multipliers. Small field multiplications are performed faster. The relative energy concerns corresponding single field designs.

Size	internal	clock cycles					area	energy			
		163	233	283	409	571		full size		163 bit	
[bit]	[bit]						[mm^2]	[nWs]	rel	[nWs]	rel
409	112	3	9	9	9	-	1.53	28.3	+28%	7.8	+82%
409	56	9	27	27	27	-	1.04	62.0	+39%	12.7	+49%
571	80	9	9	9	27	27	1.55	91.5	+36%	17.8	+109%
571	160	3	3	3	9	9	2.36	49.4	+29%	12.1	+180%

In spite of the presented properties, which show weak spots of our flexible polynomial multiplier design, it is very competitive. The performance is not negatively affected, and the increased amount of energy in most cases is acceptable and an expected price for the flexibility. The properties can be improved in further designs that should address the following pending problems:

- The reduction logic, which is not required for the current field, should be kept constant to reduce the power consumption.
- Not used flip flops should be disabled. For example for a 163-bit multiplication in a 571-bit hardware about 65% of the flip flops are always set to zero.
- Find optimal partial multiplier sizes. Even though this question depends the selected fields, there should be better solutions for standard curves. To apply a 163-bit multiplication on a 160-bit partial multiplier is an example for a bad combination.

Finally, based on the evaluations in this section, selecting the size of the internal multiplier and the supported curves, it is possible to provide a polynomial multiplier with a specified profile of energy consumption, area and speed. The optimal combination eventually depends on the practical application. Which curves are needed, at which speed and how much area and energy is available.

7.4. Polynomial squaring

For the single field square implementation (Section 5.5) a modified reduction block has been described as best solution. The combined square-and-reduce block is in average 60% smaller than a stand alone reduction block. The disadvantage of this approach is the inflexiblity, since it is only working for one single field. In this section the squaring functionality is combined with flexible reduction blocks. The general design hereby correponds to the idea shown in Figure 5.14

As first approach, the single field reduction inside the square-and-reduce units is substituted

Table 7.4.: Area consumption of flexible square units using MHWR blocks. The area of the combined square-and-reduce units is more than 50% smaller than corresponding stand alone reduction blocks.

Size	Area [mm^2]		
[bit]	Square &reduce	detached reduction	Relative
233	0.041	0.093	-56%
283	0.084	0.179	-54%
409	0.109	0.250	-56%
571	0.212	0.439	-52%

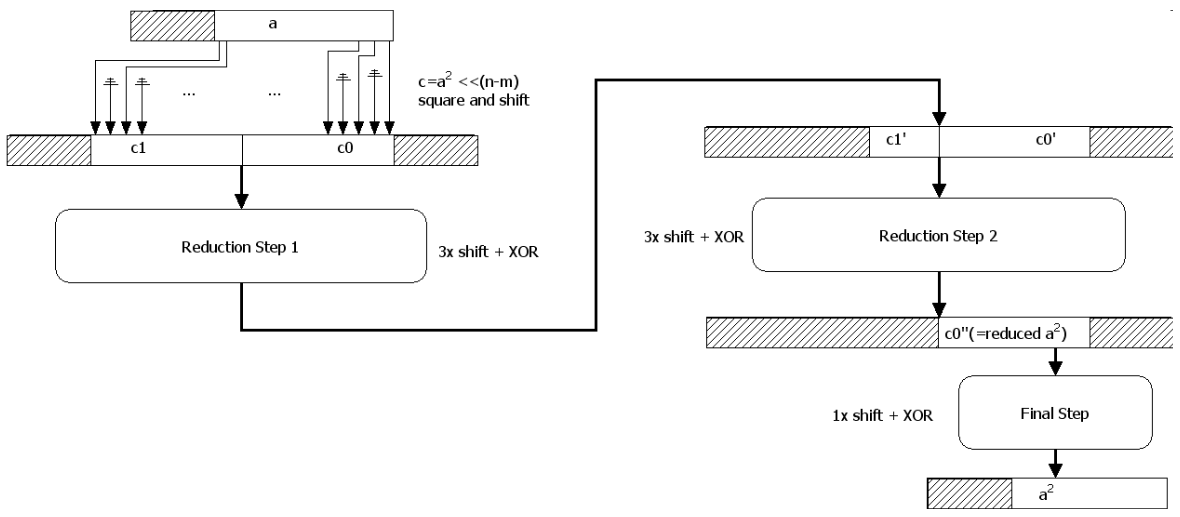


Figure 7.5.: Block diagram of a flexible combined square-and-reduce block. The initial squaring operation is followed by a complete flexible shift reduction (see Figure 7.4).

by the MHWR blocks presented in Table 7.1. The parameters of the new combined units are shown in Table 7.4. As for the single field solutions, the combined square-and-reduce units are smaller than the corresponding stand alone reduction. The complexity of the combined blocks is less than the half of the area for the detached reduction. A square unit that supports all five NIST fields has an area of 0.21mm^2, and is therefore quite small.

These designs work only for a specific number of fields. With FSR and RMR, two reduction methods have been presented that provide full flexibility and can replace the limited reduction block. RMR thereby is not considered as possible reduction block for the squaring operation. Separate squaring units are feasible if they significantly accelerate the ECC processing. With two full polynomial multiplication after the actual squaring it is not the case for the RMR. This is why squaring operations on our RMR designs are preferably performed as normal polynomial multiplication.

In contrast, FSR can improve the performance of the squaring operation as it has been for the single curve designs. The scheme for the a combined square-and-FSR unit is depicted in Figure 7.5. Generally, the reduction part is the same as described in Figure 7.3. Merely, the initial left shifter is modified in a way that it insert a zero between every input bit to apply the square functionality. Thus, it is expectable that the initial left shifter is less complex, because it sets every second bit to zero for sure. With the knowledge that every second input bit is zero, one can expect that also the first reduction step is less complex. For further steps no optimizations are expected. These considerations in mind it is not surprising that measurements show a reduction in silicon area of 16%. Concretly, the 233-bit square and reduce unit that can perform this operation in every field up to 233 bit requires 0.093mm^2.

7.5. Modular multiplicative inversion

For single field ECC designs the Itoh-Tsujii method based inversion is the preferred algorithm. This inversion algorithm is part of the program that is executed in the controller block. It uses the field multiplier, the squaring unit, and two registers. Thus, no additional hardware is required. But this approach relies on fast execution of multiplications and squaring operations in particular. These properties are not necessarily provided in a design that should support many ECs. Consider a design that performs the reduction with repeated field multiplications, as it has been described in the previous sections. Then a complete field multiplication or squaring operation implies at least two additional polynomial multiplications each. An inversion in $GF(2^{233})$ as shown in Algorithm 3.6, which applies ten field multiplications and 232 squaring operations, therefore requires at least 494 multiplications.
For flexible designs, the Shantz-division is a reasonable alternative. Independent from the other units, this extra division-FU requires $2m$ clock cycles for the execution of an inversion. The algorithm can perform all divisions up to a field size that is determined by the physical boundaries of the division block. Hence, it is suitable for flexible ECC designs. The main disadvantage is the additionally required silicon area as it was shown in Table 5.10.
Finally, it depends on the complete design whether Itoh-Tsujii inversion or Shantz division is the better choice. On a design with fast multipliers and square units there is no reason for an additional division block. On slower designs a division block implies an acceleration. According to Table 5.10 the single inversion that is required for a MPM needs less than 5% of the total time. Even in a slow design, as it was mentioned earlier in this section, requiring 494 mulitplications for a inversion in $GF(2^{232})$, it is merely 7% of the total effort for a MPM, since also all other algorithms are performed slower.
We are considering that a small improvement of an operation that requires less than 10% of the total computation time is not worth the additional silicon. This is why we will not apply the Shantz division algorithm for full ECC designs.

7.6. Flexible ECC designs

The functional units discussed previously shall now be combined to full flexible ECC hardware designs. Therefore, the single curve design which was presented in Figure 6.2 will be taken as starting point. Based on the attributes of the reduction blocks (see Section 7.2), two general types of flexibility can be distinguished.

- Designs that support a specific selection of ECs (MHWR)
- Designs that support all ECs up to a specific size (FSR, RMR)

7.6.1. ECC designs for a specific number of ECs

Hardware designs for the acceleration of ECPMs that support a specific selection of ECs, work with multiple hard wired reduction (MHWR) blocks as they were described Section 7.2.1. This approach has the benefit that the adaptations of the existing system is quite easy. The timing is not interfered too badly, since only a multiplexer is attached to the actual reduction functionality, which is hard wired as for the single curve implementations. The corresponding multiplier and squaring units have already been discussed in Section 7.3 and 7.4, respectively. Hence, if both functional units are substituted in the single curve design and the control program is adapted, the ECC accelerator that supports a specific number of ECs can be synthesized, tested, and measured.
We decided to implement two designs: one that includes the NIST curve B-163, B-233, and B-283 and another one that additionally comprises curves B-409 and B-571.

ECC 283, 233, 163 This design that supports three fields, is a standard single curve 283 bit design where the reduction blocks in multiplier and square unit have been extended and the program in the controller unit was adapted. The core multiplier has a factor size of 80 bit, and a multiplication is computed within nine clock cycles.

ECC 571, 409, 283, 233, 163: This five field design is the single curve 571 design with extended reduction blocks, adapted program and improved multiplication controll flow. Hence, on the 160 bit core multiplier, a multiplication in B-409 and B-571 requires nine clock cycles while multiplications on smaller fields are three times faster.

The results of these two flexible designs, together with a comparison to the single curve designs of 163, 283, and 571 bit, is shown as Table 7.5. In the table, parameters for timing, area and energy are reported. An evaluation of these results indicates the following conclusions

Area: The silicon area is in both considered designs less than 10% more than a single curve full size design. Therefore, the price for the support of additional smaller curves is not very high.

Speed: The number of clock cycles, required for an ECPM, is equal to the corresponding single curve implementations. An exception is the execution of small curves on the large 571 bit design. Here, the efficient application of the large internal multiplier reduces the number of clock cycles.
The maximum clock frequency is slightly below the frequency for the single curve full size design. Indeed, a small 163 bit ECPM cannot be performed with a faster clock frequency than the 571 bit operation on the same chip. This is why the execution time

Table 7.5.: Comparison of parameters of the two MHWR ECC blocks and corresponding single curves.

	ECPM bit size	single curve	ECC 283, 233, 163	ECC 571, 409, 283 233, 163
Period [ns]	163	9	11	16
	283	11	11	16
	571	15	-	16
clk cycles	163	9251	9251	7383
	283	15922	15922	9215
	571	32275	-	32275
Time [μs]	163	83	102	118
	283	175	175	147
	571	484	-	517
Area [mm^2]	163	1.0	2.0	4.3
	283	1.9	2.0	4.3
	571	3.9	-	4.3
Energy [μWs]	163	12.8	25.8	23.2
	283	43.5	50.6	48.1
	571	191.7	-	248.8

of the smaller curves is negatively affected, unless larger core multiplication units allow to reduce the number of clock cycles.

Energy consumption: Figure 7.6 compares the required total energy of one ECPM for single and multiple curve designs. The energy consumption is increasing significantly, in particular for the larger word sizes. Interestingly, for the three-curve design, the needed energy is higher than for the five-curve implementation. This is due to the improved faster execution of field multiplications on the larger hardware.

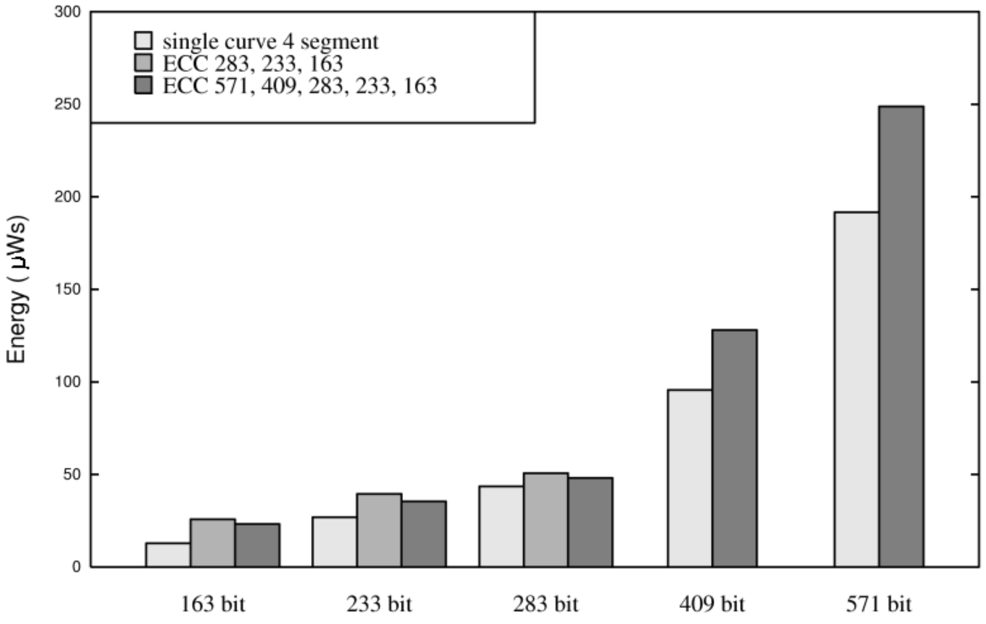
The result of this evaluation is that the support of additional smaller fields is perfectly considerable. The amount of additionally required area, time, and energy is very low, compared to a corresponding single curve design. For example, the ECPM on the 283 bit single curve design has a size of 1.9mm^2, the operation requires 175μs and 43.5μWs, a design that additionally supports curves B-233 and B-163 has a size of 2.0mm^2, and the 283 bit ECPM needs 175μs and 50.6μWs.

The flexible five field 571 bit ECC processor thereby is the first reported ECC processor that supports all five pseudo random NIST fields in $GF(2^m)$.

7.6.2. Full flexible ECC designs

The multiplier and square units with embedded FSR or RMR allow to implement ECC designs that work with all possible curves up to a specific bit length. Both approaches are connected with a considerable increase of area or required time. Actually, both methods can handle field sizes up to 571 bit. Such designs can also be described in hardware, but simulating,

Figure 7.6.: Energy consumption for an ECPM on single curve designs and on implementation for multiple ECs



synthesizing, and measuring the designs would take too much time. This is why in the following designs with a maximum field size of 283 bit are discussed:

FSR 283: It is the standard 283 bit ECC design with a four segment 283 bit multiplier unit with embedded FSR. Also, the squaring unit comprises an FSR. The multiplier and square unit were described in the preceding sections. Beside the additional registers for the configuration of the reduction polynomial, the rest of the design stays the same as for the single curve.

RMR 283: It is a 283 bit ECC design where the multiplier was replaced by an RMR block that corresponds to the design in Figure 7.2. The core multiplication unit is a two-segment multiplier. Therefore a complete field multiplication with reduction requires twelve clock cycles. The design does not contain a squaring unit. Due to the squaring operations that are performed in the multiplier and the changed timing, also the program in the controller unit must be adapted. In addition, a 233 bit register to store of the reduction polynomial is necessary.

Table 7.6 shows a comparison of both approaches. Figure 7.7 and 7.8 depict the required time and energy, respectively, for the complete ECPM of different sizes. Based on the data, we conclude:

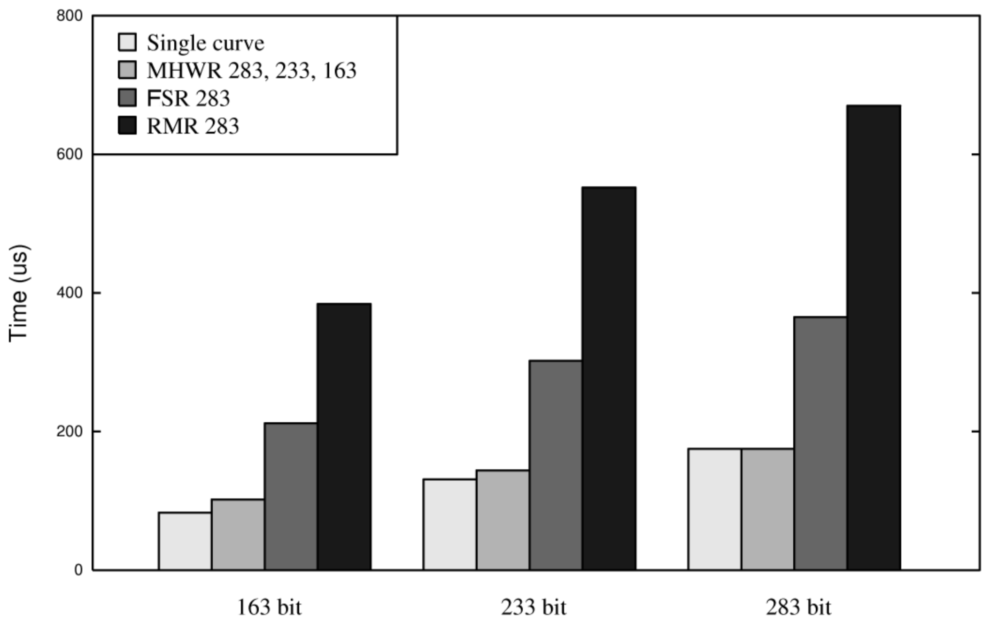


Figure 7.7.: Required total time for an ECPM on three ECs. The single curve designs and the MHWR implementation that supports exactly the three curves are much faster than both full flexible designs. The FSR implementation is nearly twice as fast as the RMR design.

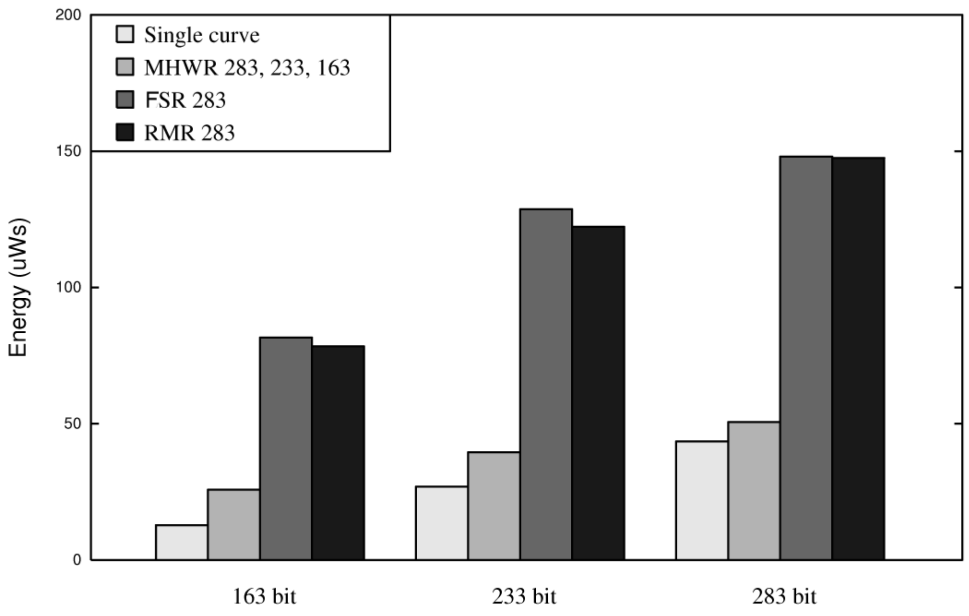


Figure 7.8.: Required total energy for an ECPM on three ECs. Both full flexible designs are on a same level but require at least the threefold energy than the single curve or MHWR implementations.

Table 7.6.: Comparison of parameters of the full 283 bit FSR and RMR designs and the corresponding single curves.

	ECPM bit size	single curve	ECC FSR 283	ECC RMR 283
Clock period [ns]	163	9	23	16
	233	10	23	16
	283	11	23	16
Clock cycles	163	9251	9251	23990
	233	13164	13164	34500
	283	15922	15922	41900
Time [μs]	163	83	212	384
	233	131	302	552
	283	175	365	670
Area [mm^2]	163	1.0	4.5	3.2
	233	1.4	4.5	3.2
	283	1.9	4.5	3.2
Energy [μWs]	163	12.8	81.6	78.4
	233	26.9	128.7	122.3
	283	43.5	148.1	147.5

Area: The FSR design requires more than the doubled area than a single curve design for the same word length. The RMR design is about 30% smaller than the version with FSR, but requires with 3.2mm^2 still much more area than the 283 bit single curve design, which has 1.9mm^2.

Speed: The number of clock cycles, is not negatively affected by the FSR approach. In contrast, the RMR design needs more than double the number of clock cycles.

The clock frequency is negatively affected for both approaches. While the clock frequency for the RMR design is above 60 MHz, FSR does not even reach 50 MHz.

Hence, also the total time, which is the product of clock period and clock cycles, is worse, compared to the single curve designs. Figure 7.7 shows the comparison of the two full flexible designs, the 283 bit MHWR with three curves, and the corresponding single curve implementation. One can see that the FSR design, due to the slow clock, requires roughly the doubled time of the single curve operation, but is about twice as fast as the RMR implementation. In contrast, MHWR with three supported curves is only marginally slower than the 283 bit single curve design.

Energy consumption: Figure 7.8 compares the required total energy for one ECPM on the corresponding single curve and flexible designs. Both designs with full flexibility require about the same energy for the ECPM. But this energy is at least the threefold compared to the corresponding single curve designs. For small 163 bit ECPM the energy consumption is even increased by the factor of five. The MHWR design is hereby much more energy efficient.

7.6.3. Conclusions

In this section approaches have been presented that provide the flexibility to select the size m of base field $GF(2^m)$. The two general approaches are designs that:

- Support of a specific selection of ECs
- Support of all ECs up to a specific size

Designs that support a specific selection of ECs, as compared in Table 7.5 are a very efficient way for providing limited flexibility. The integration of additional reduction blocks for smaller fields in a large EC design is connected with little additional area, marginal slower clock frequencies, but about 20% additional energy. The increase of the energy is caused by the fact that in the current designs every integrated reduction block is applied and afterwards the valid signal selected. For smaller ECPMs the increased energy consumption compared to the single curve design is even worse. For example the 163 bit ECPM on a 283 bit MHWR design requires the double energy. This is caused by additional flip flops and larger logic that are not part of the 163 bit hardware.

For approaches that support all ECs up to a specific size, the increase of needed energy for an ECPM is much higher. Figure 7.8 shows the energy consumption for the single curve and the flexible designs. It is obvious that both discussed flexible approaches, FSR and RMR, need much more energy than single curve or the MHWR design. RMR requires the additional energy because the number of polynomial multiplications, and therefore the total time, increases at least by the factor of three. It is questionable whether this number can be reduced in future designs, since the three repeated multiplications are integral part of the RMR algorithm.

The increase of energy needed by the FSR implementation is due to large combinatorial shift logic. We assume that further investigations of this approach will detect further improvements. For example a limitation of the field sizes into an interval from 163 to 283 bit can reduce the complexity of the shift logic. This should also reduce the required area, which is currently more than double than for the corresponding single curve design, and increase the clock frequency.

8. Conclusions

In this thesis design alternatives for hardware solutions that accelerate flexible elliptic curve cryptography in $GF(2^m)$ are evaluated. Elliptic curve cryptography has turned out to be the most efficient approach in the field of public key cryptography. Today, public key cryptography is inevitable for digital signatures, mutual authentication and key exchange, but in particular small mobile devices suffer from the very complicated and expensive operations, required for the encryption and decryption. Hardware accelerators for the EC-operation are a common solution. But these solutions are usually fixed to one field size. This restriction is due to the reduction operation, which is required after every polynomial multiplication, and must be tailored to the specific field to obtain efficient results.

Before considering possible flexible solutions, in this thesis efficient designs tailored to single elliptic curves are investigated first. Hereby it turned out that the most important issue that influences the performance is the speed of the polynomial multiplication unit. In ECC hardware designs reported in literature, the multiplication unit requires at least the half of the design area and usually limits the speed of the full cryptographic processor. This is why in this thesis (already in Chapter 3, which discusses the theoretical algorithms) special emphasis is placed on efficient multiplication algorithms. Based on the iterative Karatsuba multiplication approach, an improved multiplication method is introduced. The recursively applied iterative Karatsuba multiplication approach reduces the the number of single bit operations for a 256 bit multiplier from 49439 to 36313. The theoretical results are applied in Section 5.4, where combinatorial polynomial units are designed that are 30% smaller than multipliers using the classic Karatsuba methods. These efficient combinatorial multipliers are the basis for polynomial multiplication units that iteratively compute larger products. A 233 bit multiplication can be performed either by three 128 bit, nine 64 bit, or 27 32 bit multiplications. The results show a trade-off between area and speed, since the smaller multipliers require more clock cycles.

A similar trade-off can be seen for the full ECC designs that perform an elliptic curve point multiplication, which is the key operation for ECC. The faster and larger the internal polynomial multiplier, the faster and larger is the whole ECC design. After an extensive analysis of the design space for the Montgomery point multiplication, which is the fastest

algorithm for ECPM in $GF(2^m)$, three practical designs for each field size are presented in Section 6.2. These designs differ in the speed of the embedded polynomial multiplier. Our fastest ECC design with polynomial multipliers that require three clock cycles for a field multiplication show a performance that is competitive with the fastest hardware designs reported in literature. The complexity of our design is less than a third of these designs. An 233 bit ECPM requires 84μs with a complexity of 76 kgates, which is equivalent to 2.0mm^2 after synthesizing for the IHP in-house 0.25μm CMOS technology. This operation has an energy consumption of 20.6μWs. It is a seventh of the best reported value found in literature. Alternative software implementations require three orders of magnitude more time and more than 500 times the energy.
The efficient single curve hardware designs are the starting point for the evaluations of flexible design approaches. Flexibility in terms of ECC means that support for elliptic curves of different bit sizes is provided. Two general approaches for obtaining flexibility are discussed in Section 7.6:

- Designs that support a specific selection of ECs
- Designs that support all ECs up to a specific size.

Designs with a specific number of selected curves are very efficient, since only those components are duplicated that are specific for every elliptic curve. Specifically, the reduction logic, which is part of the multiplication unit and the square unit, is integrated into the design separately for every curve. A 283 bit design that supports the curves B-283, B-233, and B-163 is merely 5% larger than the corresponding 283 bit single curve design, while timing is marginally affected and energy consumption 16% higher. A 571 bit design that aditionally supports curves B-409 and B-571 is the first reported hardware design that supports all five ECs in $GF(2^m)$ recommended by the NIST. After synthesizing it has a size of 4.3mm^2, a 571 bit ECPM requires 517 μs and 249μWs, a 283 bit ECPM on the same design needs 147 μs and 48μWs.
Full flexibility is provided by designs that support all curves up to a specific field size. The main issue hereby is the reduction functionality. Two approaches are discussed that can solve the problem: the repeated multiplication reduction (RMR) and the flexible shift reduction (FSR) approach. Both designs raise serious issues: compared to a single curve implementation, RMR needs more than three times the time, and FSR requires more than double the area. The energy consumption increases for both approaches by a factor of three, compared to a corresponding single curve implementation.
In particular the energy consumption shows the additionally required effort is the price for flexibility. For example, a 283 bit ECMP performed on a single curve design needs 43.5μWs. On a 283 bit design that additionally supports 163 and 233 bit operations the consumption

is 50.6μWs. Both discussed design approaches that provide full flexibility up to 283 bit need 148.1μWs or 147.5μWs. The proposed designs present efficient solutions for ECC of a wide spectrum of purposes, from the very fast single curve implementations up to ECC hardware designs that provide a high level of flexibility. The latter require more area, time, and energy than inflexible designs, but the exemplary performance for the 283 bit FSR design is more than 500 times faster and the energy consumption still about 200 times less than alternative software solutions. These hardware approaches demonstrate that flexible designs for ECC are feasible in general, and that they, even supporting more than one elliptic curve, are fully convincing alternatives to software implementations.

Bibliography

[1] Daniel V. Bailey and Christof Paar. Optimal extension fields for fast arithmetic in public-key algorithms. In *CRYPTO '98: Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, pages 472–485, London, UK, 1998. Springer-Verlag.

[2] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. NIST special publication 800-57 - recommendation for key management, 2005. Available from `http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html`.

[3] Whitfield Diffie and Martin Hellman. Multiuser cryptographic techniques. In *Proceedings of the AFIPS 1976 National Computer Conference*, pages 109–112, Montvale, New Jersey, 1976. AFIPS Press.

[4] Peter J. Downey, Benton L. Leong, and Ravi Sethi. Computing sequences with addition chains. *SIAM J. Comput.*, 10(3):638–646, 1981.

[5] Zoya Dyka and Peter Langendoerfer. Area efficient hardware implementation of elliptic curve cryptography by iteratively applying karatsuba's method. In *DATE*, pages 70–75, 2005.

[6] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (SHA1), 09 2001. RFC 3174, Informational.

[7] Hans Eberle, Nils Gura, and Sheueling Chang Shantz. A cryptograhpic processor for arbitrary elliptic curves over. In *ASAP*, pages 444–454, 2003.

[8] Elliptic semiconductor. *CLP-22 Elliptic Curve Point Multiplier Core*, 2006. Available from `http://www.ellipticsemi.com/CLP-22_60102.pdf`.

[9] K. Fong, D. Hankerson, J. López, and A. Menezes. Field inversion and point halving revisited. Technical report, Department of Combinatorics and Optimization, University of Waterloo, 2003. Available from `http://citeseer.ist.psu.edu/fong03field.html`.

[10] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, 1998.

[11] C. Grabbe, M. Bednara, J. Shokrollahi, J. Teich, and J. von zur Gathen. FPGA designs of parallel high performance $GF(2^{233})$ multipliers. In *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS-03)*, volume II, pages 268–271, Bangkok, Thailand, May 2003.

[12] N. Gura, S. Shantz, H. Eberle, D. Finchelstein, S. Gupta, V. Gupta, and D. Stebila. An end-to-end systems approach to elliptic curve cryptography. In *CHES '02: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, 2002.

[13] Darrel Hankerson, Julio López Hernandez, and Alfred Menezes. Software implementation of elliptic curve cryptography over binary fields. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–24, London, UK, 2000. Springer-Verlag.

[14] Cadence Inc. *Cadence HDL Simulator*, 2005. Available from Cadence website `http://www.cadence.com/products/functional_ver/simulation/index.aspx`.

[15] Cadence Inc. *SoC encounter*, 2005. Available from Cadence website `http://www.cadence.com/products/digital_ic/soc_encounter/index.aspx`.

[16] Innovations for High Performance microelectronics. *IHP microelectronics: technology*, 2006. `http://www.ihp-ffo.de/24.0.html`.

[17] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.

[18] M. Boehm J. Franke, F. Bahr and T. Kleinjung. rsa200, May 2005. Available at `http://www.rsasecurity.com/rsalabs/node.asp?id=2879`.

[19] J. Janssen. *Compiler strategies for transport triggered architectures.* PhD thesis, 09 2001.

[20] A. Joux and R. Lercier. Discrete logarithms in $GF(p)$ — 130 digits, June 2005. Available at `http://listserv.nodak.edu/archives/nmbrthry.html`.

[21] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii nauk SSSR*, 145:293–294, 1962.

[22] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[23] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 14(4):255–293, 2001.

[24] Rudolf Lidl and Harald Niederreiter. *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications.* Cambridge University Press, Cambridge, second edition, 1997.

[25] ARM Limited. AMBA specification, revision 2.0, 1999. Available from ARM website `http://www.arm.com`.

[26] An Liu and Peng Ning. TinyECC: Elliptic curve cryptography for sensor networks (version 0.1), 2005. Available from `http://discovery.csc.ncsu.edu/software/TinyECC/`.

[27] Julio López and Ricardo Dahab. Improved algorithms for elliptic curve arithmetic in $GF(2^m)$. In *Selected Areas in Cryptography*, pages 201–212, 1998.

[28] Julio López and Ricardo Dahab. Fast multiplication on elliptic curves over GF(2^m) without precomputation. In *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, pages 316–327, London, UK, 1999. Springer-Verlag.

[29] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1996. Available at `http://www.cacr.math.uwaterloo.ca/hac/`.

[30] V. S. Miller. Use of elliptic curves in cryptography. volume 218, pages 417–426, 1985.

[31] P. L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[32] Gerardo Orlando and Christof Paar. A high performance reconfigurable elliptic curve processor for $GF(2^m)$. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 41–56, London, UK, 2000. Springer-Verlag.

[33] Gerardo Orlando and Christof Paar. A scalable $GF(p)$ elliptic curve processor architecture for programmable hardware. In *CHES '01: Proceedings of the Third International*

Workshop on Cryptographic Hardware and Embedded Systems, pages 348–363, London, UK, 2001. Springer-Verlag.

[34] Krzysztof Piotrowski. Design and implementation of an off-line e-cash scheme. Master's thesis, University of Zielona Gora, 2004.

[35] John M. Pollard. A Monte Carlo method for factorization. *BIT*, 15:331–334, 1975. Available at `http://cr.yp.to/bib/entries.html#1975/pollard`.

[36] John M. Pollard. Factoring with cubic integers. In Arjen K. Lenstra and Jr. Hendrik W. Lenstra, editors, *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 4–10, Berlin, 1993. Springer-Verlag.

[37] Dick Pountain. Transport-Triggered architectures. *Byte Magazine*, 20(2):151, February 1995.

[38] B. Preneel. *Analysis and design of cryptographic hash functions.* PhD thesis, Katholieke Universiteit Leuven, 1993.

[39] Ingo Riedel. Security in ad-hoc networks: Protocols and elliptic curve cryptography on an embedded platform. Master's thesis, Ruhr-Universitaet Bochum, 2003.

[40] Vincent Rijmen and Paulo S. L. M. Barreto. The WHIRLPOOL hash function. World-Wide Web document, 2001. Available from `http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`.

[41] R. Rivest. The MD5 message-digest algorithm. Technical Report Internet RFC-1321, IETF, 1992. Available from `http://www.ietf.org/rfc/rfc1321.txt`.

[42] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[43] Nazar A. Saqib, Francisco Rodríguez-Henríquez, and Arturo Díaz-Pérez. A parallel architecture for fast computation of elliptic curve scalar multiplication over $GF(2^m)$. In *IPDPS*, 2004.

[44] Akashi Satoh and Kohji Takano. A scalable dual-field elliptic curve cryptographic processor. *IEEE Trans. Comput.*, 52(4):449–460, 2003.

[45] Michael Scott. *MIRACL—A Multiprecision Integer and Rational Arithmetic C/C++ Library, Version 5.0.* Shamus Software Ltd, Dublin, Ireland, 2005. Available at `http://indigo.ie/~mscott`.

[46] Sheueling Chang Shantz. From euclid's gcd to montgomery multiplication to the great divide. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2001.

[47] R. Shirey. RFC 2828: Internet security glossary, May 2000. Available from `ftp://ftp.math.utah.edu/pub/rfc/rfc2828.txt`.

[48] M. Swanson. NIST special publication 800-26 - computer security, 2001. Available at `http://csrc.nist.gov/publications/nistpubs/800-26/sp800-26.pdf`.

[49] Synopsys Inc. *PrimePower: Full-Chip Dynamic Power Analysis for Multimillion-Gate Designs*, 2005. Available from Synopsys website `http://www.synopsys.com/products/power/primepower_ds.pdf`.

[50] FIPS U.S. Department of Commerce/NIST. Digital Signature Standard (DSS), FIPS PUB 186-2, January 27, 2000. Available from `http://csrc.nist.gov/publications/fips/fips186-2/fips186-2.pdf`.

[51] FIPS U.S. Department of Commerce/NIST. Secure Hash Standard , FIPS PUB 180-2, August 1, 2002. Available from `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`.

[52] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *Proceedings of the CRYPTO Conference*, pages 17–36, 2005.

[53] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *Proceedings of Eurocrypt 2005*, pages 19–35, Aarhus, Denmark, May 22–26 2005.

[54] André Weimerskirch, Christof Paar, and Sheueling Chang Shantz. Elliptic curve cryptography on a palm os device. In *ACISP '01: Proceedings of the 6th Australasian Conference on Information Security and Privacy*, pages 502–513, London, UK, 2001. Springer-Verlag.

[55] Johannes Wolkerstorfer. Is elliptic-curve cryptography suitable to secure rfid tags? In *Workshop on RFID and Light-Weight Crypto*, 7 2005.

[56] Xilinx. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, June 2005. Available from Xilinx website `http://www.xilinx.com`.

[57] Xilinx Inc. *Xilinx ISE 6 Software Manuals and Help*, 2004. Available from Xilinx website `http://toolbox.xilinx.com/docsan/xilinx6/books/manuals.pdf`.

A. Used abbreviations

ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
CPM	**C**lassic **P**olynomial **M**ultipication
CKM	**C**lassic **K**aratsuba **M**ultipication
DL	**D**iscrete **L**ogarithm
DLP	**D**iscrete **L**ogarithm **P**roblem
EC	**E**lliptic **C**urve
ECC	**E**lliptic **C**urve **C**ryptography
ECDL	**E**lliptic **C**urve **D**iscrete **L**ogarithm
ECPM	**E**lliptic **C**urve **P**oint **M**ultipication
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSR	**F**lexible **S**hift **R**eduction
FU	**F**unctional **U**nit
GF	**G**alois **F**ield
IC	**I**ntegrated **C**ircuit
IKM	**I**terative **K**aratsuba **M**ultipication
MHWR	**M**ultiple **H**ard **W**ired **R**eduction
MPM	**M**ontgomery **P**oint **M**ultipication
RAIK	**R**ecursively **A**pplied **I**terative **K**aratsuba
RMR	**R**epeated **M**ultiplications **R**eduction