High Speed Elliptic Curve Processor

An Implementation of the Elliptic Curve Digital Signature Algorithm (ECDSA) over GF(p) in hardware on an FPGA PCI-Board

Christian Pühringer

High Speed Elliptic Curve Processor

An implementation of the Elliptic Curve Digital Signature Algorithm (ECDSA) over GF(p) in hardware
on an FPGA PCI-Board

Master's Thesis

at

Graz University of Technology

submitted by

Christian Pühringer

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology
A-8010 Graz, Austria

9th October 2005

Advisor: Ao.Univ.-Prof. Dr. Karl Christian Posch
Co-Advisor: Dr. Johannes Wolkerstorfer

Hochgeschwindigkeits-Elliptische-Kurven-Prozessor

Eine Implementierung des Elliptic-Curve-Digital-Signature-Algorithm (ECDSA) über GF(p)
in Hardware auf einem FPGA-PCI-Board

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Christian Pühringer

Institute für Angewandte Informationsverarbeit und Kommunikationstechnologie (IAIK),
Technische Universität Graz
A-8010 Graz

9. Oktober 2005

Diese Arbeit ist in englischer Sprache verfasst.

Betreuer:	Ao.Univ.-Prof. Dr. Karl Christian Posch
Mitbetreuender Assistent:	Dr. Johannes Wolkerstorfer

Abstract

This thesis presents an elliptic curve cryptography processor implemented on an FPGA PCI-board. It serves as a coprocessor to accelerate cryptographic operations, like signature generation and verification. It is intended to be used on systems with a high load of such operations, like web servers for e-Government applications. On a Xilinx Spartan-3 1500 2000 point multiplications per second over a 192-bit prime field are achieved. On a Xilinx Virtex-4 LX200 more than 10000 point multiplication should be possible and the processor can easily compete with the fastest reported ASIC implementations. The implementation on the smaller FPGA is still significantly faster than any other FPGA-implementation of elliptic curve cryptography over prime fields known to the author.

Kurzfassung

Diese Diplomarbeit präsentiert einen Elliptic-Curve-Cryptography-Prozessor der auf einem FPGA implementiert wurde. Er dient als Koprozessor zur Beschleunigung von kryptographischen Operationen, wie z.B. Signaturerzeugung und Verifikation. Der Prozessor ist für den Einsatz in System mit einem hohen Aufkommen von solchen Operationen gedacht, z.B. Webserver für E-Government Anwendungen. Bei Verwendung eines 192-bit Primkörpers werden auf einem Xilinx Spartan-3 1500 2000 Punktmultiplikationen pro Sekunde erreicht. Auf einem Xilinx Virtex-4 LX2000 wären sogar mehr als 10000 Punktmultiplikationen möglich. Damit kann der Prozessor leicht mit den schnellsten dem Autor bekannten ASIC-Implementierungen mithalten. Selbst die Version auf dem kleineren FPGA ist deutlich schneller als alle anderen dem Autor bekannten FPGA-Implementierungen von Elliptischer-Kurven-Krytographie über Primkörper.

I hereby certify that the work presented in this thesis is my own and that work performed by others is appropriately cited.

Ich versichere hiermit, diese Arbeit selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient zu haben.

Acknowledgements

I am indebted to my colleagues at the IAIK who have provided invaluable help and feedback during the course of my work. I especially wish to thank my advisor, Johannes Wolkerstorfer, for his immediate attention to my questions and endless hours of toil in correcting draft versions of this thesis. Furthermore I want to thank Johannes Großschädel and Martin Schläffer for valuable ideas regarding ECC-algorithms which helped to improve the performance of the processor significantly.

Last but not least, without the support and understanding of my family, my girlfriend, and my friends, this thesis would not have been possible.

Christian Pühringer
Graz, Austria, September 2005

Credits

- This thesis was written using Keith Andrews' skeleton thesis [3].

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Internet applications with high security requirements gain more and more importance. For applications, like e-Government or online banking, security and the trust of the users in the security is an important factor for their success. To achieve this high level of security encryption and authentication is used. There are several algorithms which can be used but all share the property that they require large amounts of computing power. As many devices on the client side, like smart cards or mobile phones, cannot provide this computing power easily many hardware implementations of cryptographic algorithms where proposed and are in use. However, server systems do have enough computing power to handle the cryptographic algorithms, therefore much less hardware acceleration solutions exist. However, while the client devices normally only have to calculate a single cryptographic operation, for example the digital signature for a document, the server systems have to handle many cryptographic operations in every second. For instance, an e-Government server may have to verify signatures of thousands digitally submitted documents every second. Therefore, a high load is generated on the system and it clearly makes sense to use a cryptographic hardware acceleration device, which can process the cryptographic algorithms much more efficiently than a general purpose processor. Thus, a smaller server with hardware acceleration can be used to handle the same amount of users and so the total costs of the system could be lowered significantly.

For the widely used *RSA* encryption system some server side solutions, as presented in Wöckinger [35] exist. However, RSA has the disadvantage that it requires large keys, at least 1024 bits, to achieve a basic level of security. This requires large and expensive hardware both at the client side and at the server side. *Elliptic curve cryptography (ECC)* is a very good alternative, as it allows much smaller key sizes, and therefore much cheaper hardware, than RSA. According to Hankerson et al. [14, chap. 1.3], a 160-bit ECC key offers the same level of security like a 1024-bit RSA key. As for ECC unlike for RSA no sub-exponential time algorithms for breaking the cryptographic system are known. For higher security levels the advantage of ECC is even higher. Particularly interesting is the security level equivalent to the symmetric cryptography standard AES with 128-bit keys. To achieve this level for ECC 256-bit keys suffice, while RSA requires 3072-bit. Although RSA is currently more widespread than ECC, more and more applications take advantage of ECC. For example a large share of the smart cards used in the Austrian e-Government system, called *Bürgerkarte* [2], make use of ECC over 192 bit NIST prime fields.

In this thesis, a multi-core ECC processor architecture for computing ECC over prime fields is proposed, which fits into a mid-range FPGA (Spartan-3 1500) on a PCI board. Support for the ECCP was added to the Java cryptography library JCE. It accelerates ECC operations on server side systems.

Benchmark results show that this system can process up to 2100 EC point multiplications per second over a 192-bit NIST prime field. As far as known to the author it is the fastest ECC processor for FPGAs for prime fields. If a larger Xilinx Virtex-4 LX200 is used, estimates indicate that even more than 10000 EC point multiplications per second can be achieved easily.

1.2 Outline of the Work

In **chapter 2** a summary of other prime field ECC processors is given. Properties like estimated performance, required hardware resource, and practical usability are analyzed and compared.

In **chapter 3** an overview of the theoretical basics used in this thesis are described. A short introduction to cryptography in general is given. Then elliptic curve mathematics and elliptic curve cryptography algorithms are explained. The next section is dedicated to finite field mathematical basics. The second topic in this chapter is an introduction to common hardware arithmetic units like adders or multipliers.

In **chapter 4** the design methodology for the development of complex digital circuits is presented. After a short introduction to design criteria the theoretical top-down design flow is described. In the next section the design flow is presented which was used to develop the ECC-processor. Finally, possible target technologies for implementing an integrated circuit are presented.

In **chapter 5** our system architecture is presented. Firstly the hardware is described in detail. The functionality of the various hardware modules which compose the ECC-processor are explained. The second part is about the software implementation. Basics of driver programming for Linux are described. Then the driver architecture for the ECCP is explained. Finally, the integration in the Java JCE library is addressed.

In **chapter 6** performance results of the ECCP are compared on various implementation levels with other hardware solutions and a software implementation. Firstly simulation results are presented, then the pure point multiplication performance of the hardware is given. After that the performance of the elliptic curve digital signature algorithm is analyzed and compared with the pure software Java JCE implementation. Finally, the performance is compared with other elliptic curve cryptography processors.

In **chapter 7** the achievements of this work are summarized. A short analysis of chances to use the hardware in real application is given. Possible improvements which could be done in future works are outlined.

Chapter 2

Related Work

While there are many server side acceleration processors for RSA, there are much less for ECC. Most of these focus on elliptic curves defined over binary extension fields $GF(2^m)$ because these are computational easier and so a higher performance is achievable as when prime fields $GF(p)$ are used. Section 3.6 discusses what difficulties arise when $GF(p)$-fields are used. However, when in an application prime field support is needed, for example in the Austrian e-Government system, these advantages cannot be used and the hardware must support $GF(p)$ fields. An ECC processor for $GF(2^m)$ is presented in Wolkerstorfer and Bauer [34]. The processor presented in this paper is based on this one, and therefore the basic architecture is similar. However, the support for $GF(p)$ fields complicates the architecture considerable.

One of the fastest ECC-processors for $GF(p)$ is presented in Eberle et al. [11]. It is a very powerful processor, which can also perform calculations for $GF(2^m)$ and RSA. Using a 64-bit multiplier, it achieves a very high performance, for example 6000 ECC-operations per second for a $GF(p_{224})$-field. However, these figures are for a hypothetical implementation in state of the art CMOS technology featuring a clock frequency of 1.5 GHz. It is very questionable whether it is feasible to use such powerful and therfore expensive technologies for a cryptographic processor. They also implemented a prototype using a Xilinx Virtex-2 V6000 FPGA running at 66 MHz. Unfortunately no performance figures are given for this implementation. Assuming that it could run at 100 MHz it would be about 15 times slower than their standard cell implementation. Therefore it can be estimated that even for the smaller $GF(p_{192})$-fields less than 1000 ECC-operations can be performed per second.

Satoh and Takano [31] present an ASIC elliptic curve processor which supports both $GF(2^m)$ and $GF(p)$-fields with arbitrary primes and arbitrary reduction polynomials. The high-speed implementation uses a 64x64-bit multiplier running at a clock rate of 137.7 MHz. It takes 1.44 ms for a point multiplication over a 192-bit prime field. This corresponds to nearly 700 point multiplications per second.

In Crowe et al. [8] an arithmetic unit is proposed which can handle both RSA and ECC over $GF(p)$. To handle the high difference of typically used field sizes multiple arithmetic units are used, which are pipelined for RSA operations and work in parallel for ECC operations. The architecture uses carry propagate adders, which can be a reason for the relative low clock frequency obtained on the target Xilinx Virtex-2 2000 FPGA of less than 50 MHz for 256-bit prime fields. A 256-bit field multiplication takes $5.75\ us$. Estimating the performance for point multiplication cannot be done easily as the multipliers operate in parallel and so performance would depend very much on how well the point multiplication can be parallelized.

[27] present a processor for $GF(p)$ which has a quite unique architecture using Montgomery multiplication with booth encoding and pre-computation. A prototype was implemented in a Xilinx

Virtex-1000E FPGA. It uses about 11416 LUTs and due to the pre-computation about 5700 flipflops and 35 block RAMs. It can be clocked at 40 MHz. Only a raw estimate for the performance is given, which is $3\ ms$ per point multiplication, which corresponds to roughly 330 point multiplications per second.

The ECC processor of [41] uses Montgomery multiplication utilizing a systolic array multiplier. Implemented in a Xilinx Virtex E-1000 6000 slices are occupied and a maximum clock frequency of over 90 MHz is possible. 160-bit point multiplication time is 14.14 ms, which corresponds to 70 point multiplications per second.

Chapter 3

Theoretical Background

3.1 Cryptography

Cryptography uses mathematical methods to allow secure communication over a non secure channel. The main goals of cryptography are

1. Confidentiality

   A third party should not be able to read the transmitted information.

2. Authenticity

   Participants of the communication should be able to verify who has sent the data.

3. Integrity

   It should be detected when the transmitted data was modified by a third party

4. Non-repudiation.

   The participants should no be able to deny that they sent some information after the other participant has received it.

To achieve these goals two principal types of cryptography exist. In *Symmetric cryptography* a secret is shared between communication partners. This is not the case in *asymmetric cryptography*.

3.1.1 Symmetric Cryptography

Clearly, symmetric cryptography is the simpler type of cryptography. Its history goes back to ancient Egypt (about 1900 b.c.) where novel hieroglyphics not known to the general public where used to allow private communication. Subsequent important methods include the *Caesar cipher* which just shifts the alphabet to encrypt the message and the *Vignere cipher* which was the first polyalphabetic substitution cipher which means that a passphrase is used for encryption. A special case of the latter is the *one-time pad* where the pass phrase has the same length as the message and is used only a single time. This allows perfect security but its usability is severely restricted.

In the 19. century the *Kerckhoff principle* was formulated which says that the security of a cryptosystem must only rely on the secrecy of the keys—the shared secrets—and must never rely on the secrecy of the used algorithm. This principle is still of great importance but is still not always respected sufficiently which often results in weak security. Examples of algorithms kept secret which

finally had severe security problems are the GSM encryption algorithm A5 and the DVD content scrambling system (CSS). Especially in the case of CSS the intention to keep the algorithm secret was somewhat absurd. After all the algorithm was implemented even in software players which made unveiling the algorithm merely a matter of time.

Currently the most important algorithm is probably the *Advanced Encryption Standard (AES)*[19] which was chosen by the *US National Institute Of Standard and Technology (NIST)* to replace the older DES algorithm. Both algorithms are *block ciphers*, which means that they operate on blocks of data in difference to *stream ciphers*, like RC4, A5/1, or A5/2, which encode each bit or character of the message subsequently.

Despite of their long history and the usability-drawback that the secret must be known to all communication participants symmetric encryption still plays an important role. So asymmetric cryptography which copes with this problem is usually only used to establish a secure communication channel by sharing a secret key and the real communication is encrypted by symmetric techniques because of the much higher performance. Additionally symmetric encryption is used in quantum cryptography where the key is exchanged over a photon beam, which is secure because of quantum physic laws. The message is encrypted by a one time pad. Therefore this approach is provably secure.

3.1.2 Asymmetric Cryptography

Asymmetric cryptography allows to get rid of the requirement to share a secret between the communication participants. This method is mathematically more complex than symmetric cryptography, which is probably the reason that it was discovered late in history. The *National Securiy Agency (NSA)* claims that it has invented asymmetric encryption in the 60s of the last century but do provide little evidence for this. Clifford Cocks invented a public key algorithm in the 1970s for the *Government Communications Headquarter (GCHQ)*. However, this was kept secret until 1997 and so usually its invention is attributed both to Diffie and Hellman for their key exchange [10] developed in 1976 and to Rivest, Shamir, and Adleman for RSA [30], which was invented in 1977.

The basic novelty of asymmetric cryptography is that instead of a shared key for all participants, each of them has a *key pair*. The *public key* which is available to the public and the *private key* which is kept secret. The public key can be used by anyone to encrypt a message for its owner. The asymmetric crypto-system now ensures that the owner of the corresponding private key—and nobody else—can decrypt the message and read it.

For asymmetric cryptography mathematical functions are used which are assumed to be very time consuming to invert but which are easy to invert when an additional secret is known. Such a function is called a *trapdoor one-way function*. This allows encryption by applying the function to the message and decryption by computing the inverse with the help of the secret. A third party should no be able to unveil the message because without knowing the secret the inversion is assumed to take too much time to be feasible. Currently three classes of mathematical problems are used as basic for asymmetric crypto-systems.

1. The *integer factorization problem* in RSA [30]
2. The *discrete logarithm problem* in ElGamal [12]
3. The *elliptic curve discrete logarithm problem* in elliptic curve crypto-systems like ECDSA [6]

This work is based only on the elliptic curve crypto-system. This is described in detail in section 3.3. However, because of the importance and proliferation of RSA and to justify the use of elliptic curve crypto-systems instead of it firstly the basics of RSA are given.

3.2 RSA

Algorithm 3.1: RSA key pair generation [14, page 7]

Require: Security parameter l
Ensure: public key (n, e) and private key d
1: Select two random primes p and q of bit length $l/2$
2: $n \Leftarrow p\ q$
3: $\Phi \Leftarrow (p-1)(q-1)$
4: Select an integer e with $1 < e < \Phi$ and $gcd(e, \Phi) = 1$
5: $ed \equiv 1 \pmod{\Phi}$ and $1 < d < \Phi$
6: **return** n, e, d

Algorithm 3.2: RSA encryption [14, page 7]

Require: RSA public key (n, e), plaintext $m \in [0, n-1]$
Ensure: c is ciphertext
1: $c \Leftarrow m^e \pmod{n}$
2: **return** c

Algorithm 3.3: RSA decryption [14, page 7]

Require: RSA public key (n, e), RSA private key d, ciphertext c
Ensure: m is plaintext
1: $m \Leftarrow c^d \pmod{n}$
2: **return** m

The security of the RSA algorithm [30] is based on the assumed computational difficulty to determine the prime factors of large numbers. In the key pair generation (see algorithm 3.1) two prime numbers p, q are multiplied to obtain n, which is part of the public key. It can be proved that the security of RSA depends on the difficulty to factorize n, although the factors p, q are not directly used as key. Both the encryption (see algorithm 3.2) and the decryption (see algorithm 3.3) are just modular exponentiation with either the public key d or the private key e as exponent. Decrypting a RSA ciphertext directly would require to compute the inverse of this modular exponentiation. This corresponds to the *discrete logarithm problem*. This has the same computational complexity as the integer factorization problem. It is therefore of less interest because computing the inverse would only break a single message while the factorization would unveil the private key. *RSA Laboratories* holds challenges about factorizing the RSA modulus n. Currently the largest modulus which was factored has 576 bits, while the lowest recommended number of bits to use for the RSA modulus is 1024.

3.3 Elliptic Curve Cryptography

The use of elliptic curves for cryptographic systems was independently proposed by Koblitz [21] and by Miller [23]. Later a couple of cryptographic primitives and protocols were defined. Among the most important ones is the *Elliptic Curve Digital Signature Algorithm (ECDSA)* which was standardized by various standardization organizations. It is a public-key signature scheme that can be used to sign digital documents with the private key of the signer, while everybody can verify the authenticity of the document by using the public key of the signer. ECDSA plays a key role in the target applications of this work, therefore the hardware was designed with this algorithm in mind. However, the base operation of ECDSA, the *point multiplication ($k \cdot P$)* is also the base operation in all other elliptic curve cryptographic primitives, for example in the encryption scheme *Provably Secure Encryption Curve scheme (PSEC)*. Therefore support for these can be easily added by changing only the software.

The Elliptic Curve Discrete Logarithm Problem (ECDLP)

The security of elliptic curve cryptography is based on the difficulty to solve the *Elliptic Curve Discrete Logarithm Problem (ECDLP)*, which is to calculate l in the equation

$$Q = l \cdot P$$

. Q and P are points on the elliptic curve E, which is a group defined over a finite field F_q, and l is an integer. The operation in $l \cdot P$ is called *point multiplication*.

No algorithms with sub-exponential run time are known to solve the ECDLP. The best known generic algorithms like *Pollard-ρ* or *Pollard-λ* [28] are of complexity $\sqrt{n}$, where n is the order of P and also the number of possible values of l. For some classes of elliptic curves more efficient algorithms exist to calculate the ECDLP. Therefore these type of curves should not be used for cryptographic applications. For example they are prohibited in the standard for the ECDSA. Therefore if sufficiently large fields (for example 192-bit) it is infeasible to solve the ECDLP and so elliptic curve cryptography can be assumed to be secure.

The calculation of the point multiplication over the curve $E(F_q)$ involves calculation in two different domains: Elliptic curve arithmetic operations, and in turn arithmetic operations in the underlying field.

3.4 Elliptic Curve Arithmetic

3.4.1 Weierstrass Equation

$E : y^2 + a_1xy + a_3y + a_2x^2 + a_4x + a_6$

is the general equation to define a elliptic curve over a field K. The coefficients a_i must fulfill some conditions. These are not given here to avoid too many details. An elliptic curve point P is a tuple of integers (x_1, y_1) which fullfills the curve equation and so lies on the elliptic curve.

The Weierstrass equation can be simplified for fields usually used in ECC. For prime fields $GF(p)$ or more exactly for all fields with a characteristic different of two or three, the following form is obtained:

$E : y^2 = x^3 + ax + b$

where $4a^3 + 27b^2$ must not be 0.

For finite fields with a characteristic of two the simplified equation is:

$E : y^2 + xy = x^3 + ax^2 + b$

It defines a non-supersingular elliptic curve over a binary extension field F_{2^m}.

Fields with characteristic three or higher are less important in ECC and so are not mentioned here.

The points on an elliptic curve form a group. The group operations is the point addition. This operation and point doubling, which is the point addition for the case that both operands are the same curve point, are described in the two following sections. The neutral element of the group is the point at infinity $\mathcal{O}$. It has no representation in affine coordinates, while in projective coordinates (see section 3.4.4) such a representation exists. The inverse element of a point P is $-P$, which equals $-(x, y) = (x, -y)$ in affine coordinates.

3.4.2 Point Addition

When the elliptic curve is based on a field of reals, addition $P_3 = P_1 + P_2$ is defined geometrically: The intersection point of the line connecting the two points P_1, P_2 is determined. P_3 is the reflection of this intersection point in the x-axis.

While in finite fields additions, which are used in cryptographic algorithms, geometric addition cannot be performed, the algebraic formula derived from the geometric definition can be used both in finite and infinite fields.

For prime fields $GF(p)$ the equation is:

$$x_3 = (\frac{y_2-y_1}{x_2-x_1})^2 - x_1 - x_2$$
$$y_3 = \frac{y_2-y_1}{x_2-x_1}(x_1 - x_3) - y_1$$

The formula for $GF2^m$ is:

$$\lambda = \frac{y_1+y_2}{x_1+x_2}$$
$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$$
$$y_3 = \lambda(x_1 + x_3) + y_1$$

This formulas cannot be used if x_1 equals x_2. If $y_1 = y_2$, that is $P_1 = P_2$, the point doubling formula presented in the next section must be used. The sole other possibility is, that P_2 is the inverse of P_1. This means that $y_2 = -y_1$ for $GF(p)$ and $y_2 = x_1 + y_1$ for $GF(2^m)$. Then the result of the addition is the point at infinity $\mathcal{O}$. This is the neutral element of the elliptic curve group which has no representation as coordinate.

3.4.3 Point Doubling

In field of reals point doubling $P_3 = 2P_1$ is defined geometrically: The intersection point between the elliptic curve and the tangent in of the curve in A is determined. C is the reflection of this intersection point in the x-axis.

The equation for prime fields is

$$x_3 = (\frac{3x_1^2+a}{2y_1})^2 - 2x_1$$
$$y_3 = \frac{3x_1^2+a}{2y_1}(x_1 - x_3) - y_1$$

The algebraic formula for $GF2^m$ fields

Coordinates	$2 \cdot P$	$P_1 + P_2$	$P_1 + P_2$ (mixed)
Affine	1I,2M,2S	1I,2M,1S	-
Standard Projective	7M, 3S	12M,12S	-
Jacobian Projective	4M, 3S	12M,4S	8M, 3S

Table 3.1: Comparison $GF(p)$ field operations with different coordinate systems [33, page 23]

$$\lambda = x_1 + \frac{y_1}{x_1}$$
$$x_3 = \lambda^2 + \lambda + a$$
$$y_3 = x_1^2 + (\lambda + 1)x_3$$

If $P_1 = -P_1$ then the result is the point at infinity $\mathcal{O}$ in both cases.

3.4.4 Projective Coordinate Systems

A uncomfortable property of the elliptic curve group operation in affine coordinates, which were presented in the last section, is that they require a field inversion. Field inversion is a very expensive operation. For example Fermat's method explained in section 3.5.1 uses in average $\frac{3}{2}m$ field multiplications to perform one field inversion. m is here and in the following sections the field size in bits of the finite field underyling an elliptic curve group. More efficient algorithms like 3.17 exists but they cannot prevent that inversion is an expensive operation.

Fortunately a method exist to reduce the number of required inversions drastically. The elliptic curve points are transformed to an alternative coordinate system, the *projective coordinates*, which uses three instead of two coordinates to represent a point on the curve. The transformation from affine to projective coordinates is simple. The back-transformation requires an inversion. However, the point addition and point doubling formulas do not use inversion anymore, while the number of other field operations used, such as multiplications, is increased. In the basic operation for elliptic curve cryptography, the point multiplication, about $\frac{3}{2} \cdot m$ group operations are used, and the use of projective coordinates reduces the number of inversions from this number to only one for the final back-transformation to affine coordinates. The costs of inversion are usually high and so—despite of the increased number of simple field operations—the total complexity is reduced by far. Table 3.1 shows the count of the different field operations in various coordinate systems. Only multiplications (M), squarings (S) and inversions (I) are considered because the simpler operations like addition account only for a insignificant share to the total complexity. The column "$P_1 + P_2$ (mixed)" is for the case that one point is in affine coordinates and the second point is in projective coordinates. This is possible in the point multiplication and is because advantageous of the reduced number of field operations.

The different projective coordinate systems are derived by substituting the affine coordinates (x, y) by $(\frac{x}{z^c}, \frac{y}{z^d})$ with d, c being constants. $c = d = 1$ defines standard projective coordinates. $c = 2, d = 3$ lead to the Jacobian projective coordinates. These are the most important coordinate systems over prime fields. Certainly infinitely more coordinate systems can be defined, for example $c = 2, d = 3$ are Lopez-Dahab projective coordinates which are very efficient for $GF(2^m)$ fields.

The group operations equations can be determined by substituting the coordinates in the equations for affine coordinates by the new coordinates. Thus for Jacobean-projective coordinates (x, y) has to be substituted by $(\frac{x}{z^2}, \frac{y}{z^3})$ to derive the new point addition and point doubling formulas.

$$P_1 + P_2 = (x_1, y_1, z_1) + (x_2, y_2, z_2) = (x_3, y_3, z_3) = P_3$$
$$x_3 = (y_2 z_1^3 - y_1)^2 - (x_2 z_1^2 - x_1)^2 (x_1 + x_2 z_1^2)$$
$$y_3 = (y_2 z_1^3 - y_1)(x_1 (x_2 z_1^2 - x_1)^2 - x_3) - y_1 (x_2 z_1^2 - x_1)^3$$
$$z_3 = (x_2 z_1^2 - x_1) z_1$$

$$2 \cdot P = 2 \cdot (x_1, y_1, z_1) = (x_3, y_3, z_3) = P_3$$
$$x_3 = (3x_1^2 + a z_1^4)^2 - 8 x_1 y_1^2$$
$$y_3 = (3x_1^2 + a z_1^4)(4 x_1 y_1^2 - x_3) - 8 y_1^4$$
$$z_3 = 2 y_1 z_1$$

3.4.5 Point Multiplication

Algorithm 3.4: Double-and-add algorithm for point multiplication

Require: scalar $k = \sum_{i=0}^{m-1} k_i \, 2^i$, EC Point P
Ensure: $k \cdot P$
1: $C \Leftarrow \mathcal{O}$
2: **for** $i = m - 1$ downto 0 **do**
3: $C = 2 \cdot C$
4: **if** $k_i = 1$ **then**
5: $C \Leftarrow C + P$
6: **end if**
7: **end for**
8: **return** C

The basic algorithm to calculate the point multiplication, $k \cdot P$, is the *double-and-add algorithm* (see algorithm 3.4). It works analog to the integer square-and-multiply exponentiation algorithm. The scalar k is scanned bitwise, and depending if the current bit is set, the intermediate result C is only doubled, or doubled and added to the base point P. If k is chosen randomly, on average m point doublings and $m/2$ point additions are required to calculate the point multiplication, where m is the number of bits of k.

Various optimizations can be done to reduce the number of elliptic curve operations performed in the point multiplication. Frequently used are *window methods* that pre-calculate a certain number of point multiples. This is in particular efficient if the base point for many elliptic curve computations is constant. However, for hardware implementations window methods are less appropriate as they use a lot of memory, which is costly in hardware, in particular in standard-cell designs. Additionally, control is more complicated.

Another approach, which can also be combined with the window methods, is to convert the scalar k in a special form, which has less bits set, and therefore reduces the number of elliptic curve point additions. Widely used is the *non-adjacent form (NAF)*. Here a digit of *k* can also be negative, which allows reducing the number of set bits to about $m/3$. Using negative numbers is possible, as the point subtraction is as efficient as point addition. The overall performance gain of this measure is not very impressive. However the hardware architecture required for the simultaneous point multiplication can be reused, and therefore despite of this it makes sense to use NAF for our processor architecture.

Algorithm 3.5: Double-add-and-subtract algorithm for point multiplication [16]

Require: scalar $k = \sum_{i=0}^{m-1} k_i \, 2^i$. EC Point P
Ensure: $k \cdot P$
1: $C \Leftarrow P$
2: $h \Leftarrow 3 \cdot k$
3: **for** $i = m - 1$ downto 1 **do**
4: $C = 2 \cdot C$
5: **if** $h_i = 1$ and $k_i = 0$ **then**
6: $C \Leftarrow C + P$
7: **else if** $h_i = 0$ and $k_i = 1$ **then**
8: $C \Leftarrow C - P$
9: **end if**
10: **end for**
11: **return** C

Algorithm 3.5 shows a variant of a NAF algorithm with a very simple pre-computation to obtain the NAF of k.

Algorithm 3.6: Double-and-add algorithm for simultaneous point multiplication

Require: scalars $k = \sum_{i=0}^{m-1} k_i \, 2^i$, $l = \sum_{i=0}^{m-1} l_i \, 2^i$. EC Points P, Q
Ensure: $k \cdot P + l \cdot Q$
1: $C \Leftarrow \mathcal{O}$
2: $PQ \Leftarrow P + Q$
3: **for** $i = m - 1$ downto 0 **do**
4: $C = 2 \cdot C$
5: **if** $k_i = 1$ and $l_i = 1$ **then**
6: $C \Leftarrow C + PQ$
7: **else if** $k_i = 1$ and $l_i = 0$ **then**
8: $C \Leftarrow C + P$
9: **else if** $k_i = 0$ and $l_i = 1$ **then**
10: $C \Leftarrow C + Q$
11: **end if**
12: **end for**
13: **return** C

The *simultaneous point multiplication* (see algorithm 3.6) is a special optimization for ECDSA. In the verification step it is necessary to calculate the sum of two point multiplications ($k \cdot P + r \cdot Q$). As the point multiplication is the dominating operation in the ECDSA this leads to a nearly doubling of the calculation time for the verification in comparison to the time required for signing. To avoid this an algorithm can be used which can do both point multiplications and the addition simultaneously. For this a single pre-calculation of $P + Q$ is required. The main loop of the algorithm is changed, so that it evaluates both scalar numbers k, r. When only one current bit of either *k* or *r* is set, the corresponding point is added, when both are set, the pre-calculated point $P + Q$ is added. Trading a

slightly more complicated control and additional memory for the two additional saved points, that is Q and $P + Q$, the runtime is reduced by far. The number of doublings is halved and the number of adds is reduced by one fourth in comparison to calculating both multiplications separately and adding the intermediate results.

As already mentioned another advantage is that the hardware for the simultaneous point multiplication can be reused for supporting scalars in NAF to calculate the single point multiplication $k \cdot P$, which is required for example for signing a message. However, the speed-up of the NAF variant is not very high. The average number of non-zero bits is reduced in the NAF from $\frac{1}{2}$ of the field length to about $\frac{1}{3}$. This means that the number of point additions is reduced by $\frac{1}{3}$ in the average case. The number of point doublings stays constant, namely equal to the number of bits. Thus the number of total operation is reduced only by $\frac{1}{9}$. In fact the influence on performance by the NAF is even less, because the final inversion is not considered in this numbers. Fortunately, if the base point of the multiplication is constant a more efficient method for single point multiplication can be used with the simultaneous point multiplication algorithm. The original method [22] was indented for exponentiation but can be used for point multiplication too. It relies on pre-computation, but in difference to the window-methods typically used in software implementations, only two points have to be pre-computed. The point multiplication can be transformed in the following way:

$$k \cdot P = k_1 \cdot 2^{m/2} \cdot P + k_0 \cdot P$$

with k_0 the $m/2$ least significant bits, and k_1 the remaining $m/2$ most significant bits. If P is known $2^{m/2} \cdot P$ can be pre-computed. The result can be used as point Q in the simultaneous point multiplication algorithm. Furthermore $Q + P$ has to be computed only once too. As the scalars k_1, k_0 are only of the half length of the original scalar k the number of point doubling is also reduced to $\frac{1}{2}$. The numbers of point additions is only reduced to $\frac{3}{8}$ because the density of zeros is reduced too. Therefore the total number of operations is reduced by $\frac{5}{12}$ in comparison to the operation count for the normal point multiplication at the cost of a point multiplication and addition for pre-computation.

In the case of multiplication of two points also optimizations using redundant codings of the scalar are possible. For example it is possible to recode both scalars k, l as NAF. A little more efficient is the *joint sparse form (JSF)*, which maximizes the number of zero pairs in the scalars. Unfortunately algorithm 3.8 cannot be used because $-P$ and $-Q$ would be required to compute the simultaneous point multiplication in NAF or JSF.

For elliptic curves over binary extension fields $GF(2^m)$ the *Montgomery-method* (see algorithm 3.7) is a more efficient algorithm to calculate the point multiplication. However, for prime fields $GF(p)$ it is less efficient than the double-and-add algorithm. Another advantage of the Montgomery method is that in every step both a point addition and a point doubling are performed. This gives a certain protection against *side channel attacks*, which use timing or power analysis to get information about the secret scalar k. However, in this work the security of the hardware must only be equal to the security of a pure software system, as in FPGA-technology a secure storage of the private key is not possible. For verification no additional measures are necessary because there is no secret involved. For signing only protection against timing attacks is required because direct access to the hardware which is required for power analysis would allow retrieving the secret key from the FPGA and also the server directly. The protection against timing analysis can be done by adding additional wait cycles in the driver or the ECDSA software.

Algorithm 3.8 is the version which is implemented in our elliptic curve processor. It does a pre-shifting of the scalars to save point doublings if the MSB-bits are not set. While this does not have a large effect in the normal case, it is important if a curve operation over a smaller field is executed on the hardware for a larger field. In addition optimized version of the basic point operations are used, which cannot handle the point at infinity. The preprocessing of the scalars does prevent this to

Algorithm 3.7: Montgomery's method for point multiplication

Require: scalar $k = \sum_{i=0}^{m-1} k_i \, 2^i > 0, k_{m-1} = 1$. EC point P
Ensure: $k \cdot P$
1: $P_1 \Leftarrow P$
2: $P_2 \Leftarrow 2 \cdot P$
3: **for** $i = m - 2$ downto 0 **do**
4: **if** $k_i = 1$ **then**
5: $P_1 \Leftarrow P_1 + P_2$
6: $P_2 \Leftarrow 2 \cdot P_2$
7: **else**
8: $P_2 \Leftarrow P_2 + P_1$
9: $P_1 \Leftarrow 2 \cdot P_1$
10: **end if**
11: **end for**
12: **return** P_1

Algorithm 3.8: Double-and-add algorithm for simultaneous and single point multiplication with pre-shifting

Require: scalars $k = \sum_{i=0}^{m-1} k_i \, 2^i, l = \sum_{i=0}^{m-1} l_i \, 2^i$. EC Points P, Q, R
Ensure: $k \cdot P + l \cdot Q$
1: $j \Leftarrow m - 1$
2: **while** $k_j = 0$ and $l_j = 0$ **do**
3: $j \Leftarrow j - 1$
4: **end while**
5: **if** $k_j = 1$ and $l_j = 1$ **then**
6: $C \Leftarrow R$
7: **else if** $k_j = 1$ and $l_j = 0$ **then**
8: $C \Leftarrow P$
9: **else if** $k_j = 0$ and $l_j = 1$ **then**
10: $C \Leftarrow Q$
11: **end if**
12: **for** $i = j$ downto 0 **do**
13: $C \Leftarrow 2 \cdot C$
14: **if** $k_i = 1$ and $l_i = 1$ **then**
15: $C \Leftarrow C + R$
16: **else if** $k_i = 1$ and $l_i = 0$ **then**
17: $C \Leftarrow C + P$
18: **else if** $k_i = 0$ and $l_i = 1$ **then**
19: $C \Leftarrow C + Q$
20: **end if**
21: **end for**
22: **return** C

occur. The main innovation and advantage of the algorithm is that it can be used both for single point multiplication—with or without NAF or the scalar length halving method— and for simultaneous multiple point multiplication. This is on the cost of an insignificant increase in cycles but saves considerable hardware resources in the control and eases implementation. To perform a simultaneous multiplication of two points R must be initialized with $P+Q$. To perform a single point multiplication without NAF the scalar l must be set to zero. To perform a single point multiplication R must be $-P$ and $l = 3 \cdot k$.

3.5 Finite Field Arithmetic

The finite field over that the elliptic curve is defined is normally of one of this two types: *Binary extension fields* $GF(2^m)$ or *prime fields* $GF(p)$. The latter is also used by RSA.

A finite field is defined with a finite set $\mathbb{F}$ and two operations $(+, \cdot)$ and complies the following arithmetic properties:

1. $(\mathbb{F}, +)$ is an abelian group with the neutral element 0
2. $(\mathbb{F} \backslash 0, \cdot)$ is an abelian group with the neutral element 1
3. The distributive law $(a + b) \cdot c = a \cdot c + b \cdot c \quad a, b, c \in \mathbb{F}$ holds true

Thus a finite field is defined such as the "normal" fields like $\mathbb{Z}$, or $\mathbb{R}$, the only difference is that the set is finite. The number of elements in field is called *order of the group*. Finite fields only exist if the order is a prime power p^k with p a prime number and $k \in \mathbb{N}$. If k is one, the field is a *prime field* and if k is larger than two, the constructed field is called a *extension field*. Extension fields with an order of 2^m are called *binary fields* or *characteristic-two finite fields*. This two field types are used in most ECC-cryptography applications.

Prime field elements can be represented by integers and the arithmetic is performed modulo the prime modulus p, thus they are easily comprehensible. For extension fields the situation is different. Here various representations exist. Commonly used for binary fields is the *polynomial basis representation*. The field elements are represented by binary polynomials of order $m - 1$, where the coefficients are either 0 or 1:

$GF(2^m) = \sum_{i=0}^{m-1} a_i \, x^i \quad (a_i \in \{1, 0\})$

An irreducible binary polynomial $f(z)$ of degree m is used to reduce the multiplication result of two field elements. The addition is performed such as normal polynomial addition where the coefficients are added modulo 2. Thus for addition no explicit reduction is necessary, the elements always stay in the field. The efficient addition is an advantage of the binary fields over prime fields. These advantages are discussed in section 3.6. The focus in this work is on prime fields, which are thus discussed in more detail in the following.

3.5.1 Prime Field GF(p) Operations

A prime field $GF(p)$ is a finite field with p elements. p must be a prime, otherwise the elements only define a ring instead of a field. The field operations addition, subtraction, multiplication, and inversion are performed modulo this prime p.

Modular Reduction

To ensure that the results stay in the range $[0; p-1]$ modular reduction has to be performed. This can be done either after each field operation, or at least sufficiently frequently that the result always is smaller than an upper bound, which for example is determined by the hardware size.

The trivial way to reduce an integer a by the modulus p is *quotient determination*, which is calculated by $a - \lfloor a/p \rfloor \cdot p$. This reduction is a costly operation because a division is needed to estimate the reduced result. In many cases it is acceptable to use not fully reduced numbers. Then *quotient estimation* can be used which allows a more efficient reduction. Instead of dividing by the modulus p a number of the same magnitude of p is used. Choosing a power of two allows replacing the division by a simple binary right-shift operation which is a very simple operation in computer systems. However, still a multiplication and a subtraction is required to compute the reduction result.

To simplify the reduction two approaches are widely used: *Montgomery multiplication* and *reduction for special primes*.

The Montgomery-multiplication [24] is—as the name suggests—not a pure modular reduction but a field multiplication which includes efficient reduction. Details can be found in the following section which deals with multiplication algorithms.

Algorithm 3.9: Fast reduction modulo $p_{192} = 2^{192} - 2^{64} - 1$ [14]

Require: An integer $c = (c_5, c_4, c_3, c_2, c_1, c_0)$ in base 2^{64} with $0 \leq c < {p_{192}}^2$
Ensure: $c \pmod{p_1 92}$
1: $s_1 \Leftarrow (c_2, c_1, c_0)$
2: $s_2 \Leftarrow (0, c_3, c_3)$
3: $s_3 \Leftarrow (c_4, c_4, 0)$
4: $s_4 \Leftarrow (c_5, c_5, c_5)$
5: **return** $(s_1 + s_2 + s_3 + s_4 \pmod{p_{192}})$

Algorithm 3.10: Fast reduction modulo $p_{224} = 2^{224} - 2^{96} + 1$ [14]

Require: An integer $c = (c_1 3, c_1 2, c_1 1, c_1 0, c_9, c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$ in base 2^{32} with $0 \leq c < {p_{224}}^2$
Ensure: $c \pmod{p_2 56}$
1: $s_1 \Leftarrow (c_6, c_5, c_4, c_3, c_2, c_1, c_0)$
2: $s_2 \Leftarrow (c_1 0, c_9, c_8, c_7, 0, 0, 0)$
3: $s_3 \Leftarrow (0, c_1 3, c_1 2, c_1 1, 0, 0, 0)$
4: $s_4 \Leftarrow (c_1 3, c_1 2, c_1 1, c_1 0, c_9, c_8, c_7)$
5: $s_5 \Leftarrow (0, 0, 0, 0, c_1 3, c_1 2, c_1 1)$
6: **return** $(s_1 + s_2 + s_3 - s_4 - s_5 \pmod{p_{224}})$

The second approach waives the support for arbitrary primes and only supports reduction for some specific primes, namely *generalized Mersenne primes*. For example the NIST primes recommended in the FIPS 186-2 standard are such primes. This is possible because many cryptographic applications are using exclusively these special primes. Therefore reduction for arbitrary primes is

Algorithm 3.11: Fast reduction modulo $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ [14]

Require: An integer $c = (c_15, c_14, c_13, c_12, c_11, c_10, c_9, c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$ in base 2^{32} with $0 \leq c < {p_{256}}^2$
Ensure: $c \pmod{p_256}$
1: $s_1 \Leftarrow (c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$
2: $s_2 \Leftarrow (c_15, c_14, c_13, c_12, c_11, 0, 0, 0)$
3: $s_3 \Leftarrow (0, c_15, c_14, c_13, c_12, 0, 0, 0)$
4: $s_4 \Leftarrow (c_15, c_14, 0, 0, 0, c_10, c_9, c_8)$
5: $s_5 \Leftarrow (c_8, c_13, c_15, c_14, c_13, c_11, c_10, c_9)$
6: $s_6 \Leftarrow (c_10, c_18, 0, 0, 0, c_13, c_12, c_11)$
7: $s_7 \Leftarrow (c_11, c_9, 0, 0, c_15, c_14, c_13, c_12)$
8: $s_8 \Leftarrow (c_12, 0, c_10, c_9, c_8, c_15, c_14, c_13)$
9: $s_9 \Leftarrow (c_13, 0, c_11, c_10, c_9, 0, c_15, c_14)$
10: **return** $(s_1 + 2s_2 + 2s_3 + 2s_4 + s_5 - s_6 - s_7 - s_8 - s_9 \pmod{p_{256}})$

Algorithm 3.12: Fast reduction modulo $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ [14]

Require: An integer $c = (c_23, ..., c_2, c_1, c_0)$ in base 2^{32} with $0 \leq c < {p_{384}}^2$
Ensure: $c \pmod{p_384}$
1: $s_1 \Leftarrow (c_11, c10, c_9, c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$
2: $s_2 \Leftarrow (0, 0, 0, 0, 0, c_23, c_22, c_21, 0, 0, 0, 0)$
3: $s_3 \Leftarrow (c_23, c_22, c_21, c_20, c_19, c_18, c_17, c_16, c_15, c_14, c_13, c_12)$
4: $s_4 \Leftarrow (c_20, c_19, c_18, c_17, c_16, c_15, c_14, c_13, c_12, c_23, c_22, c_21)$
5: $s_5 \Leftarrow (c_19, c_18, c_17, c_16, c_15, c_14, c_13, c_12, c_20, 0, c_23, 0)$
6: $s_6 \Leftarrow (0, 0, 0, 0, c_23, c_22, c_21, c_20, 0, 0, 0, 0)$
7: $s_7 \Leftarrow (0, 0, 0, 0, 0, 0, c_23, c_22, c_21, 0, 0, c_20)$
8: $s_8 \Leftarrow (c_22, c_21, c_20, c_19, c_18, c_17, c_16, c_15, c_14, c_13, c_12, c_23)$
9: $s_9 \Leftarrow (0, 0, 0, 0, 0, 0, 0, c_23, c_22, c_21, c_20, 0)$
10: $s_10 \Leftarrow (0, 0, 0, 0, 0, 0, 0, c_23, c_23, 0, 0, 0)$
11: **return** $(s_1 + 2s_2 + s_3 + 2s_4 + s_5 + s_6 + s_7 - s_8 - s_9 - s_10 \pmod{p_{384}})$

Algorithm 3.13: Fast reduction modulo $p_{521} = 2^{521} - 1$ [14]

Require: An integer $c = (c_1041, ..., c_2, c_1, c_0)$ in base 2 with $0 \leq c < {p_{521}}^2$
Ensure: $c \pmod{p_521}$
1: $s_1 \Leftarrow (c_1041, ..., c_523, c_522, c_521)$
2: $s_2 \Leftarrow (c_520, ..., , c_2, c_1, c_0)$
3: **return** $(s_1 + s_2 \pmod{p_{521}})$

not required. These primes allow very efficient reduction. Instead of a quotient estimation and subtraction only a few additions and subtractions are needed. For example in a $GF(p_{192})$-field only three 192-bit additions are required to perform the reduction (see also algorithm 3.9). In comparison to the Montgomery multiplication, the hardware size is significantly reduced, because only one radix multiplier is required instead of two. As this multiplier is the dominant part of the whole architecture, the hardware size is reduced significantly, which allows the use of either higher radices or using multiple cores, and so highly improves the overall performance.

Multiplication

The obvious way to calculate the finite prime field multiplication

$$c = a \cdot b \pmod{p}$$

is to first compute the multiplication and afterwards reducing the result modulo p. This approach has two main drawbacks. Firstly, the complexity of the reduction is higher when larger numbers have to be reduced. This effect depends on the reduction algorithm used, for example for the specific prime reduction the gain in complexity is more or less linear with the size of the operand, for quotient determination or estimation the growth of complexity is very high. Secondly, both software and hardware solutions currently are no able to process a complete multiplication in one step for field sizes used in cryptography. They have to process at least one multiplication operand in smaller chunks. If the reduction is now performed after doing the complete multiplication resources and cycles are wasted for processing the higher bits which will not be used in the final result. To get rid of this drawbacks usually *interleaved reduction* is used. This means that reduction is performed in every step in the multiplication loop. In the following only algorithms which process one operand in full length in each step are described. These turned out to be advantageous for hardware implementations [33, page 105], although they have the disadvantage that the maximum size of the usable field is determined by the hardware. For software implementations on general purpose processors and for hardware which must not be limited in maximum field size algorithms must be used which process both operands at word level. This leads to a runtime quadratic dependent on operand size, while the full-precision approach leads to a linear dependency.

Algorithm 3.14: Left-to-right bit serial prime field multiplication with interleaved reduction

Require: $a, b = \sum_{i=0}^{m-1} b_i \cdot 2^i, GF(p)$
Ensure: $a \cdot b \pmod{p}$
1: **for** $i = m - 1$ downto 0 **do**
2: $c \Leftarrow 2 \cdot c + a \cdot b_i$
3: $c \Leftarrow c \pmod{p}$
4: **end for**
5: **return** c

The basic interleaved reduction multiplication algorithm 3.14 processes the second operand b on a per bit basis and so the multiplication uses m iterations with m the number of bits of the operand b. For the interleaved reduction the usual methods can be used but some optimizations can be done because the intermediate result is at most two bits longer than the modulus. For example quotient determination can be simplified to a single conditional subtraction which is performed when the m-th bit of the intermediate result is set.

Algorithm 3.15: Radix-k prime field multiplication with interleaved reduction

Require: $a, b = \sum_{i=0}^{m-1} b_i \cdot k^i$, $GF(p)$
Ensure: $a \cdot b \pmod p$
1: $c \Leftarrow 0$
2: **for** $i = m - 1$ downto 0 **do**
3: $c \Leftarrow c \ll k + a \cdot b_i$
4: $c \Leftarrow c \pmod p$
5: **end for**
6: **return** c

A higher performance at the cost of higher complexity and more hardware resource usage can be achieved by processing more than one bit of the operand b at one time. These number of bits is called *radix*. In difference to this definition in some literature radix is defined as 2^k with k the number of combined bits. The number of cycles for one multiplication is reduced to $m/radix$ with m the number of bits of the operand. Both the multiplication in the loop and the reduction is more complicated. While in the bit-serial case the multiplication is just a simple `and`-operation, now a $m \cdot radix$-bit-multiplier, which does this computation in a single step, is required. High-radix multipliers are very large in area. This fact limits the maximum radix which can be used. The drawbacks of quotient estimation respectively quotient determination reduction come into place too, as they require another multiplication of the same size. Therefore the use of the reduction for a specific prime is here very advantageous if possible. Because the algorithms for the reduction in this section are designed for a $2m$-bit input and here only $m + k$-bit—with k much smaller than m—inputs occurs they reduction can be optimized even more by limiting the maximum input to a smaller number which saves some subtractions and additions.

Algorithm 3.16: Radix-k Montgomery multiplication

Require: $m > 2$, $gcd(m, 2) = 1$
Require: k, n, $4 \cdot m < 2^{k \cdot n}$
Require: r, m', $2^{k \cdot n} r^{-1} \pmod m = 1$ and $-m \cdot m' \pmod{2^k} = 1$
Require: $0 \leq a \leq 2m$
Require: $b = \sum_{i=0}^{l-1} b_i \cdot 2^{ki}$, $0 \leq b \leq 2m$
Ensure: $a \cdot b \cdot r^{-1} \pmod m$, $0 \leq c \leq 2m$
1: $c \Leftarrow 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $q \Leftarrow ((c + b_i a) \pmod{2^k}) m') \pmod{2^k}$
4: $c \Leftarrow \frac{c + q \cdot m + b_i \cdot a}{2^k}$
5: **end for**
6: **return** c

Another approach is the *Montgomery multiplication* (see algorithm 3.16). It trades an additional transformation step, for replacing the trial division with a very easy one, usually a division by a power of two, which can be performed as a simple right shift. As in the point multiplication many consecutive field multiplications are performed, the time for transformation and the back-transformation,

both are themselves a Montgomery multiplications, is negligible, and the performance improvement is high.

Inversion

Algorithm 3.17: Extended euclidean algorithm for inversion in a prime field [14]

Require: a, $GF(p)$
Ensure: $a^{-1} \pmod{p}$
1: $u \Leftarrow a$
2: $v \Leftarrow p$
3: $x_1 \Leftarrow 1$
4: $x_2 \Leftarrow 0$
5: **while** $u \neq 1$ **do**
6: $q \Leftarrow \lfloor v/u \rfloor$
7: $r \Leftarrow v - qu$
8: $x \Leftarrow x_2 - qx_1$
9: $v \Leftarrow u$
10: $u \Leftarrow r$
11: $x_2 \Leftarrow x_1$
12: $x_1 \Leftarrow x$
13: **end while**
14: **return** $x_1 \pmod{p}$

Algorithm 3.18: Square-and-multiply algorithm for prime field exponentiation

Require: $a, x = \sum_{i=0}^{m-1} a_i \, 2^i$, $GF(p)$
Ensure: $a^x \pmod{p}$
1: $c \Leftarrow 1$
2: **for** $i = m - 1$ downto 0 **do**
3: $c \Leftarrow c^2 \pmod{p}$
4: **if** $x_i = 1$ **then**
5: $c \Leftarrow c \cdot a \pmod{p}$
6: **end if**
7: **end for**
8: **return** c

The field inversion $c = a^{-1} \pmod{p}$ can be implemented with the *extended Euclidean algorithm* (see algorithm 3.17) or by using the *theorem of Fermat*. The first method is usually faster but requires extra hardware, while the second approach can reuse the multiplier for calculating $a^{-1} = a^{2^p-2} \pmod{p}$. This exponentiation can be performed by the square-and-multiply algorithm (see algorithm 3.18). It takes about $2m$ multiplications, with m the length of the modulus p. That is because the primes commonly used do nearly all bits set. In this work the second approach was chosen.

3.5.2 Binary Polynomial Base Field GF(2^m) operations

The second widely used underlying fields in elliptic curve cryptography are the binary extension fields $GF(2^m)$.

Addition

$(a_{m-1}...a_1a_0) + (b_{m-1}...b_1b_0) = (c_{m-1}...c_1c_0)$
with $c_i = a_i \oplus b_i$

Multiplication

$(a_{m-1}...a_1a_0)(b_{m-1}...b_1b_0) = (r_{m-1}...r_1r_0)$
with $(r_{m-1}x^{m-1} + ... + r_1x + r_0)$ the remainder of
$(a_{m-1}x^{m-1} + ... + a_1x + a_0)(b_{m-1}x^{m-1} + ... + b_1x + b_0)/f(x)$
where $f(x)$ is the reduction polynomial.

3.6 Comparison of GF(2^m) and GF(p) Fields

This sections gives an overview of the difficulties that arise when $GF(p)$ fields are used instead of $GF(2^m)$ fields for hardware implementations. For software implementations on standard processors $GF(p)$ is better suited, because most processors support integer multiplication efficiently, while binary extension fields are not supported.

$GF(2^m)$ fields have the advantageous property that the field addition is just the *xor*-operation. Thus each bit of the result can be calculated independently from other bits of the input polynomials. This means that no carry propagation occurs. In $GF(p)$ fields this is not the case. Here the carry may propagate from the least significant bit up to the most significant bit of the result. As the numbers used in cryptography typically are very large, this carry propagation would lead to a very long critical path in the hardware design, and would result in very low maximal clock frequencies. To cope with this usually *Carry Save Adders (CSA)* are used. These calculate the sum and the carry of three input numbers separately. The three inputs can be for example one number in redundant representation, and another number in binary representation. A tree structure must be used when more than three numbers are added. This, together with the additional register used for saving the carry result, leads to a substantial larger hardware than when $GF(2^m)$ fields are used. This in particular occurs for high multiplication radices as in the radix multiplier for each additional bit in the radix an additional partial product must be generated and accumulated. When the binary result is needed, this is for example the case when the result of some sequential field operations must be saved in memory, or when the result is needed as an operand for a multiplication, the redundant result must be converted to its binary value by adding them together. This can either be done by a separate adder or by reusing the CSA-adder multiple times. The first approach requires large additional hardware, the second approach was used in this architecture. The drawback is that additional cycles are required for calculation. On average $ld(bits)$ cycles are used for converting the result. This time does not depend on the multiplication radix. Therefore the strategy to use higher radices for receiving more performance is inapplicable because the time for the conversion becomes soon the dominant factor. To cope with this two approaches were used. The first one is two use multiple cores instead of using very high radices. The second one is two use a mixture between CSAs and normal adders in the feedback path, where the width of the conventional adders depends on the radix of the multiplier, so that both paths are

balanced. This reduces the conversion time for higher multiplication radices, and therefore makes it feasible to use them. See section 5.1.1 for details.

Another advantage of $GF(2^m)$ fields is that squaring is much simpler than the multiplication of two different numbers, actually it can be done in a single cycle if the irreducible polynomial is fixed. For a ECC-point-multiplication over an $GF(2^{191})$-field about 2000 field multiplications and about 1300 field squarings are performed. The time required for a multiplication is about $191/radix$, for example 24 cycles for radix 8. Obviously the possibility to calculate the squaring in one cycle saves a lot calculation time, while in $GF(p)$ fields the squaring is not different to a multiplication.

The third and least important advantage is, that the subtraction operation is the same as the addition operation in $GF(2^m)$ fields. This implies that no negative numbers exist. This simplifies the hardware as no sign extension is required. To avoid sign extension, which is in particular unpleasant in the reduction circuit for $GF(p)$ fields a special negation is used which always adds a certain multiple of the prime p to the result. This assures that neither the sum nor the carry part of the result can become negative. See chapter 5 for details.

3.7 Digital Signature

A digital signature is used for signing documents. It allows checking the authenticity and identity of the signer of the document. Therefore it can bet determined whether a document with a valid signature was signed by the person who pretends to have signed it, and the document was not changed after that signing. To achieve this *Public Key Cryptography* is used. In difference to conventional symmetric cryptography two keys are used: The *private key* which is only known by its owner and must be kept secret, and the *public key*. For signing a document it is processed by a digital signature algorithm which uses the private key which is only known to the signer to calculate the *signature*. The signature can now be sent together with the document, and the receiver can use it to check the validity and authenticity of this document. To do this he uses the public key of the signer. This key is in some way, depending on the algorithm used, related to the public key, and so allows verifying the signature. However, it must certainly be impossible to calculate the private key in reasonable time out of the public key, as this would break the signature and documents could be forged. An algorithm with these properties is the ECDSA, which is based on elliptic curves. It provides high security with relatively short keys, and is therefore very well suited for systems with limited resources like smart cards or embedded systems.

3.8 Elliptic Curve Digital Signature Algorithm

The algorithm is based on the perceived difficulty to solve the discrete logarithm problem over the points on an elliptic curve which is associated with a finite field F_q. With the standardized ECDSA algorithm q is a prime p or a power of 2. The basic difference to common digital signature algorithms, like DSA [5] or ElGamal [12], is that these are based on the discrete logarithm problem directly in finite Fields $GF(q)$, while ECDSA is based on DLP in elliptic curves constructed over such fields. The advantage of the elliptic curve approach is that only fully exponential time algorithms are currently known for solving the discrete logarithm problem, while in finite fields sub-exponential time algorithms exist. This leads to a better theoretical security of the ECDSA, and allows shorter keys.

The finite field $GF(q)$ underlying the elliptic curve influences the performance of the algorithm as the efficiency of mathematical operations on the elliptic curve points depends on the efficiency of operations in the field. In this work $GF(p)$ fields with a fixed reduction prime are used.

The standard [6] defines the four principal steps of ECDSA: Firstly, the domain parameters, which principally define which elliptic curves may be used for the algorithm. Secondly, the generation of the key pair. Finally, the algorithms for signature generation and signature verification itself. These steps are described in more detail in the following sections.

Algorithm 3.19: ECDSA signature generation [14, page 184]

Require: Domain parameters: h, n, P, a, b, FR, S, q, message m, private key d
Ensure: (r, s) signature of message m
1: choose $k \in [1, n-1]$
2: $(x_1, y_1) \Leftarrow k \cdot P$
3: $r \Leftarrow x_1 \pmod{n}$
4: $e \Leftarrow H(m)$
5: $s \Leftarrow k^{-1}(e + dr) \pmod{n}$
6: **return** (r, s)

Algorithm 3.20: ECDSA signature verification [14, page 184]

Require: Domain parameters h, n, P, a, b, FR, S, q, message m, public key Q, signature (r, s)
Ensure: Accept or reject signature
1: **if** $r, s \ni [1, n-1]$ **then**
2: **return** Reject Signature
3: **end if**
4: $e \Leftarrow H(m)$
5: $s \Leftarrow s^{-1} \pmod{n}$
6: $u_1 \Leftarrow rw \pmod{n}$
7: $u_2 \Leftarrow ew \pmod{n}$
8: $X = (x_1, y_1) \Leftarrow u_1 \cdot Pu_2 \cdot Q$
9: **if** $X = \mathcal{O}$ **then**
10: $v \Leftarrow x_1 \pmod{n}$
11: **end if**
12: **if** $v = r$ **then**
13: **return** Accept Signature
14: **else**
15: **return** Reject Signature
16: **end if**

3.8.1 Domain Parameters

Clearly, for a working elliptic curve crypto-system the participating parties have to share the same elliptic curve parameters, which are also called domain parameters. Not all elliptic curves allow the same level of security, for some parameter choices efficient attacks are known. This is taken into account in the ECDSA standard, which defines the parameters that can be used.

- The order of the finite field q, which is the number of finite field elements. In the case of prime fields this equals the reduction prime p. For binary extension fields an estimation of the order is given by the *Hasse Theorem*: $q + 1 - 2 \cdot q^{1/2} <= \#E(F_q) <= q + 1 + 2 * q^{1/2}$
- The field element representation FR is unambiguous for prime fields but not for extension fields, thus it must also be chosen as parameter.
- The coefficients of the elliptic curve equation a, b.
- The ECDSA standard allows—besides using predefined parameters—the generation of random parameters and provides algorithms for this task. If this is done the random seed S is part of the domain parameters which allows verification that the elliptic curve parameters have been indeed generated randomly.
- The base point P which is a point on the elliptic curve of prime order.
- The order of the base point n. This is the number of elliptic curve points and so is the main security parameter of the algorithm.
- The cofactor h, which is the relation between the number of elliptic curve points and the number of field elements. $h = \#E(F_q)/n$.

Although the ECDSA standard defines various choices and allows random generation of the parameters it is in many cases advantageous to use fixed parameters. Besides simplifying the implementation the good choice of parameters also allows higher performance of the algorithm, an example is the fast reduction for NIST-primes explained in section 3.5.1. In the case of finite field parameters there should not be any security problems. However, if the elliptic curve parameters a, b, P are shared by k users, finding all their public key only takes $\sqrt{k}$ times the effort to break one key. As the standard relies on that it is infeasible to break a single key, this is not of to much concern. However, another security issue with using constant elliptic curve parameters is, that it is possible that attacks are found on specific classes of curves, as this has happened in the past. When keys are shared and are using such a class of curves all keys are weakened, while if they use random elliptic curve parameters only a part of them is affected.

If randomly generated parameters are used the parameters must be validated. Details are of minor importance for understanding the ECDSA, but it should be mentioned that additionally to checking whether the parameters are valid, it is also checked whether the parameters are not special cases which lead to reduced security: The *MOV* and *Anomalous* Condition must be fulfilled.

3.8.2 Key Generation and Verification

The key pair is generated in the following way: The private key d is a (pseudo) random number between 1 and $n - 1$. The public key is the point $Q = dP$. The standard also describes how the validity of such a key pair can be checked.

3.8.3 Signature Generation and Verification

Algorithm 3.19 serves for computing the signature of a message, while algorithm 3.20 allows verifying the signature. The elliptic curve point multiplication is the most complex and performance relevant operation in both algorithms. In the case of verification even two point multiplications must be performed. If the trivial approach is used verification takes nearly the double time than signature generation, which is a significant drawback of the ECDSA for many application, because usually

a single signature is verified multiple times. Furthermore, the second point is variable even when the base point is fixed—it is the public key. This prevents or a least reduces the advantage of point pre-computation for the verification which leads to a even worse relation of verification and signature generation performance. Therefore the use of simultaneous point multiplication is essential to reduce the disproportion between signature generation and verification.

Although the point multiplication speed is the main performance bottle neck, the other operations should not be underestimated. As hash function $H(m)$ SHA-1[4] is chosen by the ECDSA standard. In this thesis a library was used to perform the SHA-1 hashing, thus no details are given here. The second group of important additional function are the reduction and modulo inverse with the base point order n as modulus. This reduction modulus is not the same as the prime field modulus and is also required if extension fields are used as base for the elliptic curve. In particular on small devices this is a significant drawback because the requirement for this additional reduction minimizes the advantage of using prime fields with specific reduction primes and of using extension fields.

3.9 Advantages of Elliptic Curve Cryptography

RSA has proved itself over the years. It is widely used in current applications. The RSA patent expired in 2000 and so it can be used free of license fees. Therefore the question comes up why elliptic curve cryptography should be prefered over RSA. Clearly that it is not functionality, as it is the same as for RSA. An additional disadvantage of ECC is that many parts are patented, at least in the United States.[1] However, ECC has a large advantage in security, or more exact in the minimum key-length to reach a certain level of security. Both crypto-systems are not provable secure, they rely on the fact that they were investigated thoroughly by the scientific community and that no shortcuts were found to compute their mathematical basic function much more efficiently than trying all possible values. The latter is called brute force attack and is possible with all crypto-systems, which allow detecting whether a decrypted data string is the correct plain text message. The best case is that the running time of an attack is exponential to the key length. This means that adding a bit to the key approximately doubles the runtime to break the key. This allows keeping a crypto-system secure against the computing power available in the future, which is expected to grow also exponentially. That is because a linear growth of the key length makes attacks infeasible again. Only computers based on quantum physics could break out of this circle. However, it will probably take much time until this systems are available, and quantum physics allows absolutely secure cryptography which then could replace current crypto-systems. A crypto-system which has not the property of exponential growing security has the disadvantage that its efficiency diminishes over the time, because with the growing computer power available runtime of the algorithm grows faster than the time to break it. RSA is such a system. The best known algorithm to factorize a number is the *number field sieve (NFS)* [14], which has a runtime of $O(e^{(C+o(1))n^{1/3}(\lg n)^{2/3}})$ with n the length of the number. It was firstly proposed by Pollard [29]. He also proposed the fastest algorithm known to solve the ECDL-problem which is the Pollard-rho method [28], which has a runtime of $\sqrt{\pi n}/2$ with n the prime of the prime field underlying the elliptic curve. With this runtime a comparison of key length of both crypto-systems can be given. The correctness of the results 3.2 given by [14, page 19] rely on the assumption that the mentioned algorithms are really the most efficient algorithms. They only consider the run time, but not additional limiting factors like memory requirements or parallelization abilities. As only NFS is limited by these factors, ECC is favored by this method of comparison. The results show that the EC-crypto-system is superior to RSA. Already for the currently required security levels, for example the Austrian Bürgerkarte uses 192-bit ECC, RSA requires about seven times longer keys.

[1] Certicom claims do hold patents of many algorithms used in ECC. See `http://www.certicom.com/index.php?action=ip,patents` for details

	Security level (bits)				
	80	112	128	192	256
	(SKIPJACK)	(Triple-DES)	(AES-small)	(AES-medium)	(AES-large)
EC parameter n	160	224	256	384	512
RSA modulus n	1024	2048	3072	8192	15360

Table 3.2: Security comparison RSA vs. ECC [14, page 19]

The key length to achieve the security level of the already widely used symmetric algorithm AES with 128-bit key is twelve times larger for RSA than for ECC. For current high security requirements which will become standard security level in the future RSA already requires thirty times longer keys than ECC.

Because of the difference of the algorithms the runtime of cryptographic operations like signing or verification is not directly comparable by looking at the key length. Each step in EC computations is more complicated because calculations of EC points are performed which requires various computations in the underlying finite field, while RSA performs these directly in the finite field. However, the field is much smaller in the case of ECC. Therefore the advantages of ECC on the lowest level, the finite field operands are much smaller, and on the highest level, the number of bits in the operand for the point multiplication is much smaller than for the exponentiation in RSA, should easily overrule the disadvantage of the additional computation effort introduced by the additional level. The type of computations at this higher level are surprisingly similar. RSA performs square and multiply calculations, while ECC does double and add calculation, which is the equivalent operation in elliptic curve groups.

RSA has the advantage that it posses a basically symmetric run time for signing and verification[2] while the ECC signature algorithm ECSA takes more time for verification than for signing. This is on the one hand because of the fact that two instead of one EC point multiplications are required in verification. On the other hand this is caused because of the possibility to accelerate signing by doing pre-calculations. This increases efficiency of EC-signing by trading computations for memory. However, for verification pre-computation is not possible because the base point is not predefined. This leads to higher runtime for verification in the order of ten. In most applications a signature is more often verified than generated, therefore this runtime difference is quite unsatisfactory. Fortunately, in hardware implementations this disadvantage does not exist because there memory resources are low, and therefore usually no pre-computations are performed. The second cause of worse runtime for verification, the doubled amount of point multiplications can be reduced by using algorithms for simultaneous point multiplications. Verification using the hardware designed in this work is only about 10% slower than signing. Also hardware can benefit more from the smaller key length. Also for software implementations the influence of key length is large, because the runtime of field multiplication is quadratic to the operand length. However, efficient hardware solutions usually perform a multiplication at the full length of the operand. Therefore larger key-length results in larger hardware. This leads to higher costs or even prevents implementing crypto-systems requiring large key length like RSA in currently available technology.

[2] Choosing a small exponent d even allows significantly faster verification than signing

3.10 Hardware Solutions

The hardware architecture of the presented ECC processor is described in chapter 5. Furthermore hardware optimizations are presented in section 4.3. However, some solutions used are common knowledge in hardware design and therefore will be described here in the background chapter. For two areas of the hardware lots of solutions were proposed. Firstly the high radix multiplier, which computes the product of two numbers in a single cycle. Secondly, optimized addition methods using redundant number systems. These play a role both in the high radix multiplier and in the remaining algorithmic units.

3.10.1 High Radix Multiplier

Basically, a high radix multiplier consists of two parts: Firstly the *partial product generation* and secondly adders, usually in the form of an *adder tree*, to accumulate the partial products. The partial product generation works by multiplying the operand a for each bit set in operand b with its place value. This is nothing more than a left shift according to the place value and **and**-operation. Although the complexity and therefore the resource requirement of this operation is low, the total resource usage is considerable, so for example doubling the radix doubles the resource requirements. Several approaches to reduce the area exist.

Rather exotic is the use of *pre-computation*. It is mentioned here because one of the few reported prime field elliptic curve processors uses it, namely Orlando and Paar [27]. The idea is to calculate before computing the real (modular) multiplication 2^r, with r the radix, multiples of the operand. Then to perform the radix multiplication it is only necessary to select the result corresponding to the operand b. Thus the radix multiplier is not much more than a memory, the radix multiplication can be performed directly in the full length multiplication algorithm. The substantial disadvantage of the algorithm is the high memory usage. Therefore it is only feasible for FPGAs (see section 4.3.6) and even then only for small radices. For example for a radix of eight, 256 multiplies of the operand have to be saved. In the case of a $GF(p_{192})$ field each multiple is $192 + 8$ bits long. Thus over 50000 bits of memory are used. A 32-bit radix, which is easily usable with the standard radix multiplier architecture, would require 112 GByte of memory, which is certainly not possible to use in an FPGA. In addition the higher share of the pre-computation of the multiplication runtime grows, which also suggests the use of small radices. Another drawback of this solution is that control is complicated because of the pre-computation. In Orlando and Paar [27] *Booth's recoding* is used to minimize the memory requirement by half. This method is also widely used in standard high radix multipliers, where the number of partial production generators is halved.

Booth's recoding works by representing an operand by a redundant number system. It reduces the number of partial products by allowing subtracting of these. It is a generalization of the pen-and-paper method to calculate for example $994 \cdot a$ by computing $1000 \cdot a - 6 \cdot a$. A truth table is used to recode $k + 1$-bits of the multiplicand for Booth-k, where one bit overlaps. The most simple case is Booth-1. If two consecutive bits are equal to zero is put into the recoded string. If the first is one and the second zero, a one, and finally if the first is zero and the second is one, a minus one is the result. For example:

$$\begin{array}{ll} 1\,0\,1\,1\,1\,0\,0\,1 & [185] \\ 1\,-\,1\,1\,0\,0\,-\,1\,0\,1\,-\,1 & [256 - 128 + 64 - 8 + 2 - 1] \end{array}$$

Booth-1 does not provide any advantage in the average case. However, for numbers with many consecutive ones or zeros it reduces the number of partial products. This advantage can only be taken by a bit-serial multiplier, because it is not known in advance which position will be zero. Booth-2 is

Multiplicand bits	Booth's recoding
000	0
001	+1
010	+1
011	+2
100	-2
101	-1
110	-1
111	0

Table 3.3: Booth-2 recoding

the most common variant of the method. Three bits of the multiplicand, one of them the overlapping bit, are recoded according to table 3.3. To generate the partial product the second multiplicand is multiplied with the recoded value. This is performed by a conditional shift and a two's complement unit, which is a negate followed by an addition of one. The recoding of two bits and one overlapping bit reduces the number of partial products from n to $\lfloor \frac{n+2}{2} \rfloor$.

Booth's recoding can be used both in the parallel radix multiplier, or in the higher level digit serial multiplication. Because the number of partial products is only nearly halved, Booth's recoding is not useful for parallel multipliers with a small radix. For instance for a radix-4 multiplier the number of partial products is only reduced from 4 to 3, while the additional hardware resource usage is linear related with the radix. In the case of the digit serial multiplier the drawback of Booth's recoding is the bad scalability for high radices. With the number of recoded bits the number of hard multiplies increases exponentially. For example for Booth-3, where 3 bits of the multiplicand are recoded, the hard multiply is $\cdot 3$, which cannot be computed by a single shift operation, also an addition is required. In the case of Booth-4 multiplications already three hard multiplications are necessary: $\cdot 3$, $\cdot 5$, and $\cdot 7$. Therefore in common multiplier architectures Booth's recoding is less advantageous than for the exotic pre-computation architecture, where the multiples are saved in a table, and so the hard multiples are not more difficult to calculate than the other multiples.

The partial product adder, the second stage of the high radix multiplier, allows the following design decisions. On the one hand the adder architecture must be chosen. This is described in the next section. On the other hand the structure of the adders has also a large influence on efficiency. The obvious way is to add up the partial products in a linear structure. Then the complete multiplier is an *array multiplier*. The disadvantage of this approach is the long critical path. This can be optimized by using tree structures. Widely used in multipliers is the *Wallace tree*. It performs a 3:2 compression in each tree node. Thus the final result consists of the carry and sum part resulting from the addition of all partial products. If the final result is required, the two value have to be summed up. If a carry propagation adder is used for this addition, the total delay is the same as the delay in the array multiplier. Thus, the Wallace tree architecture is only advantageous if a faster and therefore larger final adder is used or if the carry and sum results can be used directly by logic connected to the multiplier. This redundant number form is called carry-save form and is explained in the next section. A disadvantage of Wallace trees is their irregularity. This leads to a more complicated routing that can cause a worse performance than when a more regular architecture with a worse logic delay is used.

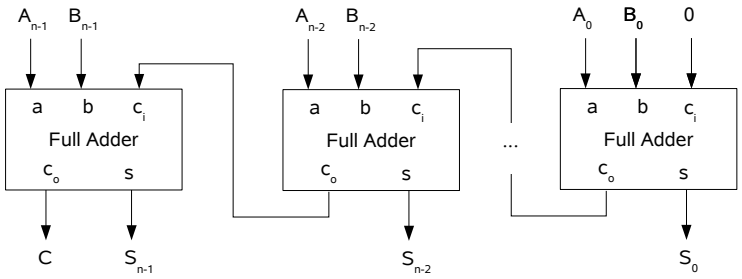

Figure 3.1: Ripple carry adder

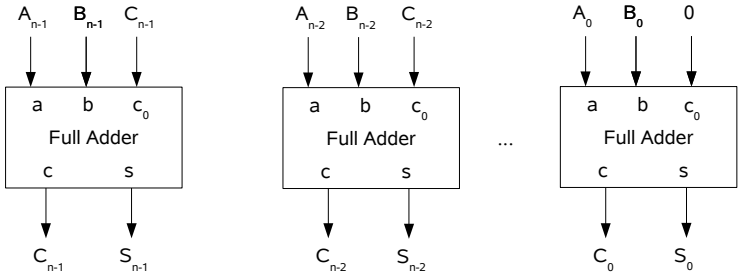

Figure 3.2: Carry save adder

3.10.2 Adder Architectures

The most simple adder architecture is a ripple carry adder (see figure 3.1). It uses a series of full adders to add two bits of the operands and the carry result of the addition of the lower bits. This leads to a long carry propagation chain, the carry of the least significant bit full adder can influence the result up to the most significant bit. Thus the adder delay is linear to the bit-length of the operands, which lengthens the critical path and therefore reduces the maximum clock rate of the circuit. This is not feasible for most applications, but it is in particular bad for the large operands used in ECC.

A very powerful but simple optimization is the use of *carry save addders (CSA)* (see figure 3.2). Instead of adding the carry to the higher bits the carry and sum of each bit is the output of the adder. This means that each output bit only depends of three input bits and so no carry propagation occurs. However, instead of one sum the result now consists of a carry and sum part. At the first glance this does not seem to useful because usually a non-redundant result is required. To achieve this the carry-save result basically has to be added by a carry propagation adder. However, the advantage of a CSA is that it can add to a carry-save value a binary value. Thus when a series of additions has to be done the results can stay in carry-save form and long critical path caused by carry propagation does not occur. Only one slow carry propagation addition is required after the last carry-save addition was performed. In the case of the radix multiplier the addition of the partial products can be carried out by carry-save adders. If the carry and sum of the result are reduced and saved separately, the complete finite field multiplication can be performed in carry save form, with the need of only a

single carry propagation addition. The requirement to have an additional carry propagation adder, which is in use very infrequently is inconvenient. In Wolkerstorfer [33] a solution is proposed to get rid of this. Namely, setting the third CSA input and adding the carry and sum values repeatedly allows a conversion to a binary number because the carry disappears over the time. Although the number of iterations is n in the worst case, in typical cases the carry disappears after $\log_2 n$ iterations. The only additional hardware required is a circuit to detect whether the result is fully reduced. This is the case when the carry is zero.

Chapter 4

Design Methodology

This chapter explains the *design metholodogy* and the *design flow* used to implement the elliptic curve cryptography processor. At first the abstract design flow, which is a *top down* approach, is described. In the second part the actual used tools for each of the level of the abstract flow are presented. Finally the two target technologies are described.

4.1 Design Criteria

Before designing a system architecture it is necessary to know what are the *design criteria* of the system. These depend on the desired application. Although this may sound trivial it seems that in particular in the field of cryptographic hardware this very basic requirement is not honored sufficiently or at least not sufficiently argued. For instance many implementations for server-side acceleration (see chapter 2) cover a very wide field of cryptographic algorithms and parameters without explaining what is the benefit for the application which justifies the higher complexity and therefore higher cost or lower performance of the systems.

General design criteria for hardware engineering are:

1. (Correct) functionality

   This means that the hardware should do what it is supposed to do. It also includes the already mentioned requirement, what the hardware should be able to do.

2. Throughput

   A performance constraint which says how many operations per second can be done.

3. Latency

   A performance constraint which defines how long a single operation takes.

4. Power consumption

5. Resource requirements

   Defines the cost of the system, includes target system/process and area.

6. Development cost/effort

   An efficent way to lower these is reusing existing components.

4.1.1 Design Criteria for the elliptic curve processor

An aspect of functionality which is obviously very important in cryptographic applications is security. If it is required that the key cannot be obtained from the hardware, targeting an FPGA is out of question because it cannot be prevented that the data, and so also the key, can be read from the FPGA. On ASICs measures must be taken to prevent attacks by physically intrusion or side channel attacks. The latter attack tries to get information about the key by analyzing measurements of side channels, like timings or power consumption. In the case that this level of security is not needed, it is safe to use an FPGA as target technology. For example in this work, the processor extends a server system. Therefore no higher security is required to be provided by the cryptographic processor. It must only account that like for the software version remote *differential timing attacks (DTA)* are prevented. This means an attacker measures many times the time the server takes to decrypt or sign a message and retrieves information about the private key with statistical analysis of the timing measurements. This can be prevented by either using algorithms which have data independent runtime, or by waiting after the calculation until the worst case runtime is reached. The processor in this work is designed to perform the elliptic curve part of the ECDSA. In this case the private key is never used in the hardware, but a random number, therefore the hardware itself is not affected by the timing attack.

As the ECCP is intented to be used to acclerate server systems the *performance criteria* plays here an important role. Latency is not a real issue in a server system, even a simple software implementation should take less than $10\ ms$ to compute a cryptographic operation, which is easily sustainable for the user. The important performance criteria is througput, that means the number of cryptographic operations which can be computed per second should be high. This increases the number of possible performance optimization, for instance pipelining or parallel calculation can be used. As the ECCP principally computes point multiplications it is self-evident to use the point multiplications per second as benchmark for the throughput.

The *resource requirement criteria* is quite obvious and strict: ECCP must fit into the Xilinx Spartan-3 1500 FPGA on the target PCI board [7]. Additionally, two other properties must be considered. Firstly, it should be possible to influence the resource requirements so that the ECCP can also fit into a smaller FPGA and that it can take advantage of a larger FPGA. This leads to the requirement that the ECCP must be parameterizable. Secondly, the possibility to use standard cells as target technology should be kept in mind. Therefore no optimizations for the FPGA target should be taken which have a negative impact on resource requirements or performance on standard cell technology. This principally leads to the requirement that memory resource like registers should not be used to generous because these although cheap in FPGA technology are expensive in standard cell technology.

The *power consumption criteria* is not a big concern for the ECCP implementation. The targeted server system can supply a lot of power, and it is save to assume that the hardware implementation will always be much more power efficient than a software implementation on the general purpose processor of the server. This can be a slight additional advantage of the ECCP because it can reduce the overall power consumption of the server system. Although this not essential like for mobile systems, it nevertheless reduces costs a little.

The *Development cost/effort* cost criteria is here because of the nature of a thesis quite strict: It must be possible to implement the system in four to five months. To comply this goal it is useful to divide the complete system in tasks, with basic tasks required to implement a working system, and advanced optional tasks and to use a good design methodology like the top down design presented in the next section.

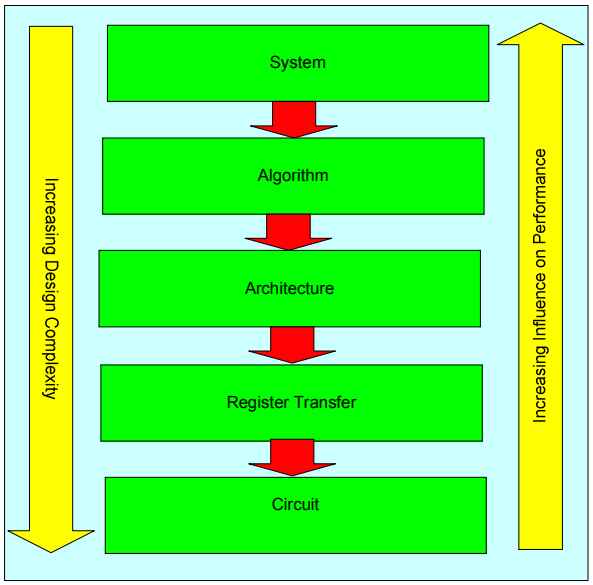

Figure 4.1: Top-down design flow abstraction levels

4.2 Top Down Design Flow

Clearly it is much easier to implement a system on a high level of abstraction, for instance as a Java program, than to implement it on a low level of abstraction. In the case of hardware design a low abstraction level can be for example a gate level representation. This advises to use a top down design methodology for implementing complex systems: The design process starts with a high level model of the system, which is continuously refined until the circuit level is reached which can depending on the target technology be programmed on an FPGA or be produced as an ASIC. These abstraction levels are shown in figure 4.1. In each level verification has to be done to ensure that the model works correctly. The favored method would be that the refinement could be done automatically by software tools. While this is not yet possible for the high abstraction levels, it can be done for the lower levels, currently between architectural level and RTL representation. Besides that the top down approach basically allows the implementation of complex systems it also accommodates that the influence on the quality aspects like performance is much higher on the higher abstract levels than on the lower ones. For example an algorithm chosen appropriately for the chosen quality aspects has much more influence than optimizations on the gate level or optimizations specific for a FPGA target hardware have.

4.2.1 System Level

The system is the highest level in the abstraction hierarchy. It defines the functionality of the system and defines system constraints and design criteria, while its hiding the implementation details.

The system level description is commonly verbal, but also suitable programming languages like SystemC can be used.

4.2.2 Algorithmic Level

After establishing the system design the algorithms to use must be developed or chosen. Depending on the algorithm domain programming languages like MATLAB, C++, or Java are used to implement algorithms. Another possible choice is SystemC, which is especially beneficial, if already the system level description was written in this language. The use of this high level languages allow the comparison of various algorithms and choosing the most appropriate to comply the system constraints. This also includes considerations regarding the target technology. For instance for many applications—including ECC—optimized algorithms exist which trade higher speed for using more memory. While this is a good strategy when the algorithm is implemented in software, it is normally not appropriate for hardware implementations. The algorithmic level implementation is usually used to retrieve the first performance estimates

4.2.3 Architectural Level

While the algorithmic level description only defines the algorithm, in the architectural level the partitioning of this algorithm into modules is performed. *Hardware description languages (HDL)* like Verilog[17] or VHDL[18], or again SystemC are usually used for the description at this level. The description is both structural—if the functionality is provided by submodules—and behavioral. However to ease the design process it is often not a bad idea to refine the high level language algorithmic implementation down to the architectural level or below, and to implement the HDL description based on this low-level model.

The architectural description represents the inherent parallelism of the hardware, by using modules working in parallel. Parts of the algorithm which cannot be processed in parallel or to save hardware resources modules can be reused to process different parts of the algorithm at different times. In this stage also the partitioning between control and data path is performed.

4.2.4 Register-Transfer Level

In the register-transfer level the architectural description is refined by adding the clock and therefore cycle accuracy. Hardware design tools can automatically synthesize this description written in hardware description languages. Unfortunately, these languages were designed without this possibility in mind. Therefore for a long time no common standard existed to define what language constructions are synthesizeable, and still incompatibilities between various synthesis tools persist.

4.2.5 Circuit Level

The description at this level is the physical representation of the circuit, which is generated automatically by the synthesis tools. This is done by first analyzing and synthesizing the HDL representation. The result is a netlist, which is mapped onto the target technology. The last step is the placing and routing of the circuit to get a physical representation. For standard cell target technology this routed version must be verified by a *Design Rule Checking (DRC)*, and *Layout versus Schematic (LVS)* check and simulation before it is produced. This ensures that the functionality of the layout representation is the same as of the gate level representation. For programmable targets like *field programmable gate arrays (FPGA)* this is usually not necessary, the configuration file resulting from routing can be downloaded directly in the hardware.

4.3 Applied Design Flow

While in the last section 4.2 the theoretical abstract design flow was explained in this section the actual used tools to implement the ECCP are presented.

4.3.1 System Level Considerations

In the case of the ECCP the system description is the task definition of the thesis itself. There the decision was taken to use a processor accelerating ECC to implement a server system with a high load of signing and verification of documents like an e-Government application server. Due to the fact that a thesis is a scientific work and not mainly aimed at designing an useable product the system level analysis was not as complete as probable required for product design case. For example, examinations of the share and total number of ECC operations in a real world e-Government server system where out of scope of this analysis. However, if a useable product is the objective of the design process this will be important to justify the use of special acceleration device in the server. Nevertheless in comparison to other works in the field of cryptographic hardware acceleration the consideration of constraints due to the target application played a much more important role. A great deal of the publications in this field does not give any justification for the system architecture which was chosen.

4.3.2 High Level Model

For high level modelling Java was used. Reasons which leaded to this decision were that the Java model used by the EC processor for wireless applications [see 33, chapter 6] could be reused which eased modelling a bit although the architecture of this processor is quite different. Another reason is that Java provides powerful support for large numbers, which is essential for cryptographic applications. The Java model was continuously refined and finally covered the abstraction levels from algorithmic level down to register transfer level for the data path. The control was only modelled on the algorithmic level because the implementation of control was expected to be easier. The high efforts at this level payed off when implementing the HDL model. Implementation and verification of the ALU only took a few days, while implementing the control was much more time consuming.

The Java RTL model uses an object orientated approach to realize a hierarchical, regular, and modular representation. That means modules are represented by Java classes, where their submodules are again represented by classes. For the connections of the modules no classes where used, they are represented by variables, which are used in methods to interconnect the submodules. These methods implement the functionality of the module. Figure 4.2 shows the code for the method representing the top level of the ALU (figure 5.2).

The Java model also served for obtaining first performance estimates. A clock counter variable which is increased each time the ALU top level method `alu_do` is called. Other counters gather information about the cycles used by specific operations, like field multiplication or register loading. For an example of the Java simulation results see figure 4.3. A Excel spreadsheet was used to analyze timing results of different simulation runs, and so information about the optimal choice of parameters was obtained. It also showed in an early stage of the design process that a system architecture using parallel cores must be used to allow a acceptable scalability of the circuit. Although the Java timing results played an essential role in the design process, it should be mentioned that the absolute accuracy of the Java results was worse than expected. It underestimated the number of cycles by about 20% in comparison to the HDL simulation results. The main reason for the performance loss is probably the pipelining of the control which was not modeled in Java.

```
 void alu_do(BigInteger a_in, int mul_mux_select, int b_reg_mux_select,...
    ...int adder_mux_select)

//wires
BigInteger adder_a_in,adder_s_in,adder_c_in;
boolean carry_in;  //carry in for csa, used for subtracting
BigInteger multiplier_b_in;
BigInteger bi_wire; //radix MSB of rb
BigInteger b_wire;
BigInteger b_reg_mux_out;
BigInteger s_wire;
BigInteger c_wire;

reduce.reduce_CSA(s_reg.getValue(),c_reg.getValue(),m);
s_wire=reduce.getSum();
c_wire=reduce.getCarry();

bi_wire=b_reg.getValueOfRange(hw_bits-1,hw_bits-radix); // msb of b (radix bi
b_wire=b_reg.getValueOfRange(hw_bits-radix-1,0);
multiplier_b_in=mul_mux.evalute(mul_mux_select,BigInteger.valueOf(0),...
  ...BigInteger.valueOf(1),bi_wire);
radix_multiplier.multiply(a_in,multiplier_b_in);
adder_a_in=radix_multiplier.getSum();
subtractor.subtract(s_wire,c_wire,m);
adder_s_in=adder_mux.evalute(adder_mux_select,BigInteger.valueOf(0),s_wire,..
  ...s_wire.shiftLeft(radix).mod(BigInteger.valueOf(2).pow(adder_mux.width)),
  ...s_wire.shiftLeft(1).mod(BigInteger.valueOf(2).pow(adder_mux.width)),...
  ...subtractor.getSum());
adder_c_in=adder_mux.evalute(adder_mux_select,BigInteger.valueOf(0),c_wire,..
  ...c_wire.shiftLeft(radix).mod(BigInteger.valueOf(2).pow(adder_mux.width)),
  ...c_wire.shiftLeft(1).mod(BigInteger.valueOf(2).pow(adder_mux.width)),...
  ...subtractor.getCarry());
csa1.add(adder_a_in,adder_s_in,adder_c_in,carry_in);
csa2.add(radix_multiplier.getCarry(),csa1.getSum(),csa1.getCarry(),carry_in);
b_reg_mux_out=b_reg_mux.evalute(b_reg_mux_select,b_wire.shiftLeft(radix),...
  ...s_reg.getValueOfRange(hw_bits-1,0));
ccycle++;
b_reg.load(b_reg_mux_out);
s_reg.load(csa2.getSum());
c_reg.load(csa2.getCarry());
}
```

Figure 4.2: RTL level Java model

```
 Cycle   |Inv.   |MulInv |Mult.  |Squar  |Add.   |Neg.   |Sub.
         | gfp   | gfp   | gfp   | gfp   | gfp   | gfp   | gfp
         --------+-------+-------+-------+-------+-------+-------
  112340 | 1     | 380   | 3157  | 0     |  0    | 1025  | 0

         |Sub.r |Shift  |Shift  |Hold   |Nop    |Load_a |Load_m
         | gfp  | left  | right |       |
         +------+-------+-------+-------+-------+-------+------
         | 1937 | 1146  | 0     | 14409 | 3266  | 2131  | 1

        Mult. (gfp) doesn't include multiplications for
        inversion (mulinv)  Pointmultiplication Statistics

        191 doublings 61694 cycles 323 cycles/point doubling
        113 adds 39717 cycles 351 cycles/point addition


INFO:   TEST PASSED, result fully reduced
```

Figure 4.3: Timing results JAVA (point multiplication over $GF(p_{192})$ with multiplication radix 8)

The Java model was also used to generate test vectors for the verification of the *Hardware Description Language (HDL) Model*. This test vectors were not used for testing the hardware implementation, instead functionality was verified by performing large amounts of elliptic curve computations using the JCE test program.

4.3.3 Hardware Description Language (HDL) Model

Most modules of the ECCP system were implemented in VHDL [18]. Figure 4.4 shows a portion the ALU top module, namely the result registers, and the second CSA adder. The much longer code in comparison to the Java model is principally due to strictly enforcement of modular design by VHDL. Only a small share is because the model is on a less abstract level. The clock is now explicitly used.

There exist various HDLs. Widely used are Verilog, VHDL and SystemC. SystemC is more appropriate for design on system level, and thus it is more a *system description language* than a hardware description language. The real HDLs like Verilog and VHDL have a different syntax but the basic language elements are similar. In the following they are described regarding VHDL:

- Interface definitions

  The definition of the interface of a module is strictly separated from the module implementation. `entity` defines this interface with ports and generic declarations. Ports are used to connect input and output of a module with another module, or to define the pins of the chip. Generic declarations allow the parameterization of the hardware. For example an adder which defines a generic width, can be instantiated various times with various widths.

```
architecture structure of gfp_multiplier is
  .
  .
  component carry_save_full_adder
   generic (
    width      :     integer := 194);
   port (
    a, b, c    : in  std_ulogic_vector(width-1 downto 0);
    carry_in   : in  std_ulogic;
    sum, carry : out std_ulogic_vector(width-1 downto 0));
  end component;
  .
  .
  signal  s_reg, c_reg   : std_ulogic_vector(csa1width+1-1 downto 0);

  signal csa2_sum_out : std_ulogic_vector(csa1width+1-1 downto 0);
  signal csa2_carry_out : std_ulogic_vector(csa1width+1-1 downto 0);
  .
  .
  begin
  .
  .
 csa2 : carry_save_full_adder
   generic map (
      width => csa1width+1)
   port map (
     a     => csa2_a_in, --0&csa1_sum_out
     b     => csa2_b_in,               --0&csa1_carry_out
     c     => csa2_c_in,
     carry_in => '0',
     sum   => csa2_sum_out,
     carry => csa2_carry_out);
  .
  .
  .
  sync: process(clk, reset)
  begin
    if (reset = '1') then
      b_reg <= STD_ULOGIC_VECTOR(TO_UNSIGNED(0,b_reg'LENGTH));
      c_reg <= STD_ULOGIC_VECTOR(TO_UNSIGNED(0,c_reg'LENGTH));
      s_reg <= STD_ULOGIC_VECTOR(TO_UNSIGNED(0,s_reg'LENGTH));
    elsif (clk'event and clk='1') then
      b_reg <= b_reg_mux_out;
      c_reg <= csa2_carry_out;
      s_reg <= csa2_sum_out;
    end if;
  end process;
end structure;
```

Figure 4.4: RTL level VHDL

- Implementation

  The keyword `architecture` starts the implementation of a module. It is possible to have various implementations of a single module for example a behavioral and a RTL version.

- Component

  The `component`-keyword is used in the architectural body of a module and declares external modules which are used.

- Module instantiation

  The declared components can be instantiated. Then port and generic mapping has to be done. This assigns local signals and generic values to the instantiated module, thus it connects the two modules.

- Combinational processes

  A combinational process is declared by the keyword process and a sensitivity list. This list must contain all signals used in the process. The logic functions must generate an definite output for all possible input values. If this is not the case implicitly registered logic is created. All processes are executed in parallel while each process contains sequential statements.

- Sequential processes

  This processes only have the reset and the clock signal in the sensitivity list. An `if` statement is used to model the asynchronous reset, where the registered values are set to their initial value, and the synchronous assignment of values to the registers is done when the reset is not active and a clock edge occurs.

4.3.4 Design for High Clock Frequencies

Although the system level and algorithmic level design decisions have the most influence on the total speed, the influence of architectural level design decisions must be taken into account. While the influence on speed by choosing between parallelizing modules and reusing them was mentioned in 4.2.3 now design measures to increase the maximum clock frequency are described.

The maximum clock frequency depends on the delay of the critical path, which is the path with the highest delay in the circuit. Unfortunately this delay is not only the sum of the delay of the logic elements in the path because the routing between the elements does also increase the delay. The routing is responsible for about 50% of the total delay, but it can also have a much larger share. This has to be kept in mind during the design process because very optimized structures which have fewer levels of logic can perform worse than a less optimized version, which has more levels of logic but is much more regular and therefore easier to route.

A possibility to reduce the critical path length is to put registers into it. This breaks down the path into a number of shorter paths. This clearly does not reduce the time data requires to travel through the data path, the latency, but its possible to begin processing new data although old data has not completely passed through. This way the throughput is increased, when the circuit is clocked at increased clock frequency. This measure is called *pipelining* and is commonly used in the design in digital circuits. For example the general purpose processors Intel Pentium-4 uses up to 31 pipeline stages. However, this design decision was to a large part due to marketing because at the time the processor was introduced to the market the clock frequency was by far the most important performance figure for many customers, while the real performance was much less considered. Because of the disadvantages of architectures optimized for very high clock frequencies, like high power dissipation,

current processor general purpose processors use again much shorter pipelines. So the maximum clock frequency decreases but the number of operations per clock cycle increases. For instance the AMD Athlon-64 uses a 12-stage for integer computations and a 17-stage pipeline for floating point computations. The Intel Pentium-M processor is in the same range.

In the ECCP datapath pipelining was not used. On the one hand because the datapath is already quite short, about six logic levels, on the other hand control would be complicated by far and therefore it was not feasible to implement pipelining in the given time-frame. However some pipelining is introduced by the control because all control signals are registered.

The registering of the control output signals is a very efficient measure to increase the maximum clock frequency. That is on the one hand because the logic for generating the control signal does not add to the data path, which could be the critical path. On the other hand this allows replicating the registers and therefore amplifying the signal before the register. This removes the amplification of the control signal from the critical path. In particular if a control signal is used to control many signals in the datapath the benefit is very high. This is the case with the ECCP because of the width of the data-path which is more than 192 bits, a single control signal drives 428 datapath signals. This introduces a delay of about $9.5\ ns$ on the Spartan-3 FPGA. The synthesis tool can recognize this path and replicates the register about fifty times, which reduces the amplification delay by far and the control signal is not anymore in the critical path.

Another fundamental measure is to move logic elements out of the critical path. In the ECCP for example the modular reduction was moved from its natural place between the arithmetic units and the register to the feedback loop (see figure 5.2) in front of the arithmetic units. So a higher clock frequency was traded for a little larger hardware, because the results have to be saved in non reduced form in the registers, and a low increase in number of clock cycles required for a computing an arithmetic operation.

4.3.5 Target Technologies

A hardware design can be either implemented in a *Field Programmable Gate Array (FPGA)* or it can be produced as an *Application Specific Integrated Circuit ASIC*, where usually standard cell design is used. To further optimize criteria like performance or power consumption full-custom or semi-custom designs can be used. FPGA have the advantage that the can be reprogrammed in a few seconds, while producing an ASIC usually takes month, and is very expensive. The principal disadvantage of an FPGA is that maximum performance and the maximum complexity of a design are much smaller than if it is implemented as an ASIC. Also the cost per unit is much higher if a high amount of chips are produced, for a small amount of chip to produce the contrary is true. Therefore a principal application of FPGA is testing and rapid prototyping. While its infeasible to produce an ASIC for first verification attempts, this can be done without problems by using an FPGA. However this is not the only application for FPGAs. If only a small number of chips is required, and the performance obtainable by an FPGA is sufficient, using the FPGA also for implementing the final version is usually the better alternative, because producing ASIC is very expensive. This is probably the case for the ECCP because it is quite possible that the market for server side cryptography acceleration is not large enough to justify the production of ASICs. Furthermore the performance in particular if high-end FPGA are used—for instance a Xilinx-4 LX200—should be easily sufficient for the intended applications.

4.3.6 FPGA

A *Field programmable gate array (FPGA)* is a programmable logic circuit. The basic elements of an FPGA are *Look-up tables (LUT)* which implement freely programmable logic functions. LUTs can also be used to implement memory. A number of LUTs form together with elements for register implementation a *slice*. A number of slices again forms another element, the so called *configurable logic block (CLB)*. These CLBs are arranged in an array structure and form together with the interconnection network the basic architecture of the FPGA (figure4.5). Modern FPGAs additionally provide a large number of other functional components. This includes from block rams, multipliers, or even general-purpose processors. The synthesis tools try to identify such functionality in the HDL code. The extracted functionality is subsequently mapped on the corresponding function block of the FPGA by the mapping tool.

As an example some basic figures of the Spartan-3 1500 FPGA[39], which is used in this work, are given.

1. 3328 CLBs
2. 13312 slices (4 per CLB)
3. 26624 LUTs (2 per slice)
4. 26624 Flip-flops (2 per slice)
5. 576 Kbit block RAM
6. 208 Kbit distributed RAM[1]

Each of its LUTs has four inputs and one output. This allows implementing any boolean logic function with four variables in one LUT.

Some principal properties of such an FPGA architecture are, that large amount of memory resources and flip-flops are provided, and that to use 100 percent of each LUT the logic functions should have the same number of logic variables as a LUT has inputs. If the latter is not the case, LUTs are wasted by not filling them up entirely. Clearly that it is hardly possible to obey this always or at least frequently but it is a good idea to keep this in mind.

The large amount of offered registers and memories permits a much more generous use of these resources than in the case of standard cells. However, because in many cases architectures should be useable both in FPGAs and ASICs it is often preferable do not make use of these. This rule was obeyed in this work. Because of that the ECCP with optimal parameters for the Spartan-3 1500 uses all but two slices, and 90% of the LUTs, while only 14% of the registers are used. The ECCP cores itself do not make use of block RAM but the PCI-bridge which is included in these numbers uses 10 of the total 32 BRAMs.

In the following some features of the Xilinx Spartan-3 FPGA which were used in this work are described in more detail. Although specific for this FPGA the situation should be similar for other FPGA families.

RAM

The Spartan-3 FPGA offers two RAM types. The first one is *block RAM (BRAM)*. Quite a few of this RAM blocks are available in an FPGA but each of them is relatively large. For instance the Spartan-3

[1] Uses LUTs to provide RAM. A LUT (in fact only one LUT per slice) can be configured as a 16-1-bit RAM

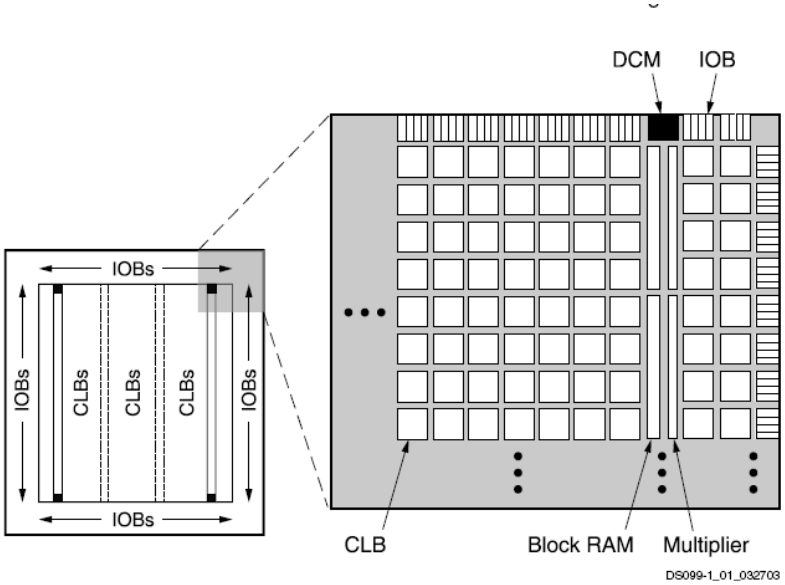

Figure 4.5: FPGA architectural overview (Spartan 3) [39]

1500 offers 32 BRAMs with 18 KBit each. More details can be found in [39, page 20]. The synthesis tool automatically infers BRAM when it finds memory elements in the HDL description when BRAM is suitable. This is the case when the memory is accessed synchronously. The main disadvantage of BRAM especially for the ECCP architecture is the layout of the BRAM in the FPGA. The maximum number of data bus bits is 36 per BRAM, which means that for a 192-bit ECCP 6 BRAMs are required. The BRAM locations are fixed on the FPGA therefore high routing efforts are required to connect the BRAM inputs and outputs to the data path. As routing in particular in FPGAs is responsible for are large share of the complete delay, for example in the critical path of the ECCP nearly 60% of the delay are due to routing, that is hardly acceptable and the use of another type of RAM must be forced by the using the corresponding constraint. In the ECCP only the PCI bridge [26] makes use of BRAMs. There they are used for the FIFOs which synchronize the two clock domains of the PCI bus and the WB bus to which the ECCP is connected.

The other type of RAM available is *distributed RAM* [40]. As the name implies, it is not organized in blocks but it is distributed on the FPGA area. To permit this LUTs are used to implement RAM. This is easily possible as a LUT is in fact a RAM which contains the truth table of the logic function which it is implementing. The disadvantage of the distributed RAM is that the used LUTs are not longer available to implement logic functions, while BRAM are additional elements which do not directly use LUT and slice resources. Therefore, distributed RAM is less appropriate for large memories which do not have a distributed nature. However, distributed RAM is quite efficient. Each CLB can save 64 bits, where each LUT has a 1-bit wide, 16-bit deep configuration. This fits very well the ECCP architecture. For a 192-bit configuration only 48 CLBs are used for the complete RAM when the PCI bridge is not taken into account. This is less than 1.5% of the total CLBs of the Spartan-3 1500. Each bit slice of the distributed RAM can be placed near to the its data input and output connection of the ALU, which allows much more efficient routing of the interconnect.

Digital Clock Manager (DCM)

Digital clock manager (DCM) [37] serve for two principal tasks. Firstly, they can be used to eliminate clock skew. Secondly, they can be used to multiply and divide the frequency of an input clock signal to generate a new clock signal with a different frequency.

Clock skew is the delay of the clock in different areas of the circuit. A special high performance network is used for the clock tree in the Spartan-3 to minimize the clock skew caused by clock distribution. However, clock skew can also occur for input and output ports. A DCM can be used to phase shift the clock by the skew and so eliminate the skew. The DCM predicts the phase shift necessary automatically with the help of a feedback loop. In the ECCP this skew is not an issue, therefore no DCM for phase shifting is used.

The frequency synthesis allows creation of new clock frequencies by multiplying and dividing an input clock. Both is done simultaneously so the number of possible factors is quite high. Therefore the clock can be adjusted to be near to the maximum frequency of the circuit which can lead to a significant performance boost, depending how much the maximum clock rate and the external clock rate differ. As the maximum clock rate of the ECCP is up to 100 MHz on the Spartan-3 and the clock rate provided by the FPGA board[7] is only 66 MHz the performance gain achieved by the use of a DCM is considerable.

FPGA Design Flow

Figure 4.6 shows the Xilinx design flow as an example for an FPGA design flow. The higher abstraction levels are independent from the target technology. Therefore the focus in this diagram is on the steps between the RTL-level and the real circuit. Firstly, the HDL description is synthesized. The Xilinx tool tries to recognize general functional units. These includes basic entities like flip-flops, multiplexors, and logic functions like xor, but also more complex units like RAM, ROM, adder, counter, state-machine, or even multiplier are inferred. On the one hand this allows faster synthesis and more efficient results because predefined optimized units, called *macros*, can be used for the detected functionality. On the other hand in FPGAs parts of the functionality are hardwired which is much more efficient than implementation by using LUTs and Flip-Flops. Thus the recognition is necessary to allow the use of these elements. To enforce the use of a specific FPGA functional element entities are defined in libraries supplied by the manufacturer which can be instantiated in HDL directly. This is in particular useful for FPGA specific functions like DCMs. However for normal functions implementing the functionality in HDL is preferable because only this way other FPGAs or even ASICs can be targeted easily. The FPGA manufacturers normally explain and give code samples how to ensure that a function is detected correctly as a macro by the synthesis tool, for example the manual from Xilinx is [38]. In the next step the circuit is optimized. This is can be already done under timing and area constraints, but at least the tool tries to optimize speed while the circuit must fit into the target device.

After the synthesis the design is mapped onto the target FPGA. This means that the circuit functionality is assigned to the various elements of the FPGA. Here it is decided whether the design fits the device. In the case of the Xilinx flow even synthesis results which do use more than 100% of the resource can fit because the mapping tool can put unrelated logic into one CLBs and so save resources, on the cost of a slower result. Despite of this very crowded designs are problematic because small changes in the design or various synthesis parameters can make the difference between a working result or a circuit not fitting the device.

The last step is place and route. Placing assigns the mapped resources to a location on the device. This is crucial for the achievable performance because it determines the effort required for routing.

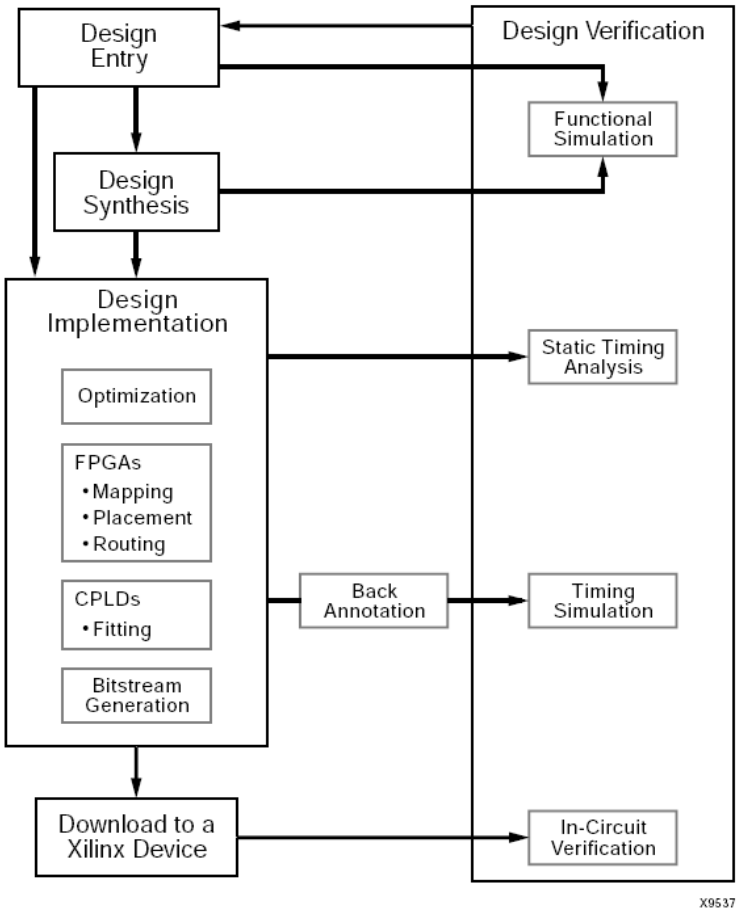

Figure 4.6: Xilinx FPGA design flow [36]

The routing adds the wiring to the circuit, it does the interconnects. A timing constraint is used to define the minimum clock frequency. Routing iterates until the constraint is met or it decides that this is not possible. After that timing analysis is performed to determine the achieved maximum delay in the critical path. Then a bitstream file is generate which can be used to configure the FPGA.

4.3.7 Application Specific Integrated Circuit (ASIC)

ASICs are circuits which are produced for a specific application. Its functionality is then fixed, it cannot be changed. This allows higher performance than with FPGAs. The total costs are very high, but for a high volume of chip the unit cost is much lower than for FPGAs.

The usual way to design and produce a digital ASIC is to use *standard cells*. The chip foundry supplies a standard cell library for the chosen process. This is used by the mapping tool to map functionality described in a HDL to these standard cells. The functionality provided by standard cells varies with the supplier and the process technology. Always cells for basic logic functions like `NOR` and for basic memories like `Flip-Flop` are provided. Additionally more complex function are included, for example adders in the UMC $0.13\ um$ process[32]. Cells for different fan in and fan outs are provided. This allows choosing more appropriate cells in the mapping process, as cells with higher fan in or fan out are larger, slower and consume more power. All standard cell of a library have the same height, while their width varies depending on the complexity of each cell. Power lines are included in the cells. The location is fixed over all cells, one rail is at the bottom the other at the top where V_{dd} and V_{gnd} are routed. Figure 4.7 shows as an example an inverter of the Austria Micro Systems $0.35\ um$ standard cell library. Every second rows of the layout is flipped to allow sharing of the power rail for two neighboring rows. Additionally to the layout the standard cell libraries include information about timing and power properties. These are used by the tools to estimate timing and power consumption, which is clearly much faster and less complex than doing this calculations on the transistor level.

ASIC design flow

The design flow is very similar to the FPGA design flow. However the following differences exist.

Firstly, the used tools are different. While the FPGA manufactures like Xilinx or Altera provide design tools for their products in the range of some 1000€ or even for free[2] the design tools for CMOS circuits are not provided by the chip foundries but by specialized companies like Cadence, Synopsys or Mentor. The tool chain is much less homogeneous than the FPGA tool chain. The tools for different design stages are usually sold separately and they can even be combined with tools of other manufacturers. The costs of the tools is much higher than for the FPGA tools.

The principal difference between the both targets is the importance of verification. If FPGAs are targeted it is sufficient to verify the HDL representation by simulation because the tools reliably produce a working FPGA configuration. Additionally if despite of this problems occur it takes only some minutes to synthesize a new configuration which can be loaded into the FPGA. This is not the case in an ASIC flow. On the one hand the tools are less reliable. It is possible that although all constraints are met the result will not work. On the other hand it takes weeks to months and a lot of money to produce new hardware if the first run was not working correctly. This leads to the *first time right* design paradigm. That means it must be ensured by means of extensive verification that the first produced chip is working correctly. One method to achieve this is to create FPGA prototypes. This allows much faster verification than with simulation. However it is necessary that the design is

[2]This tool chain usually begin with the synthesize tool. Simulation tools are mostly not included or of very basic functionality.

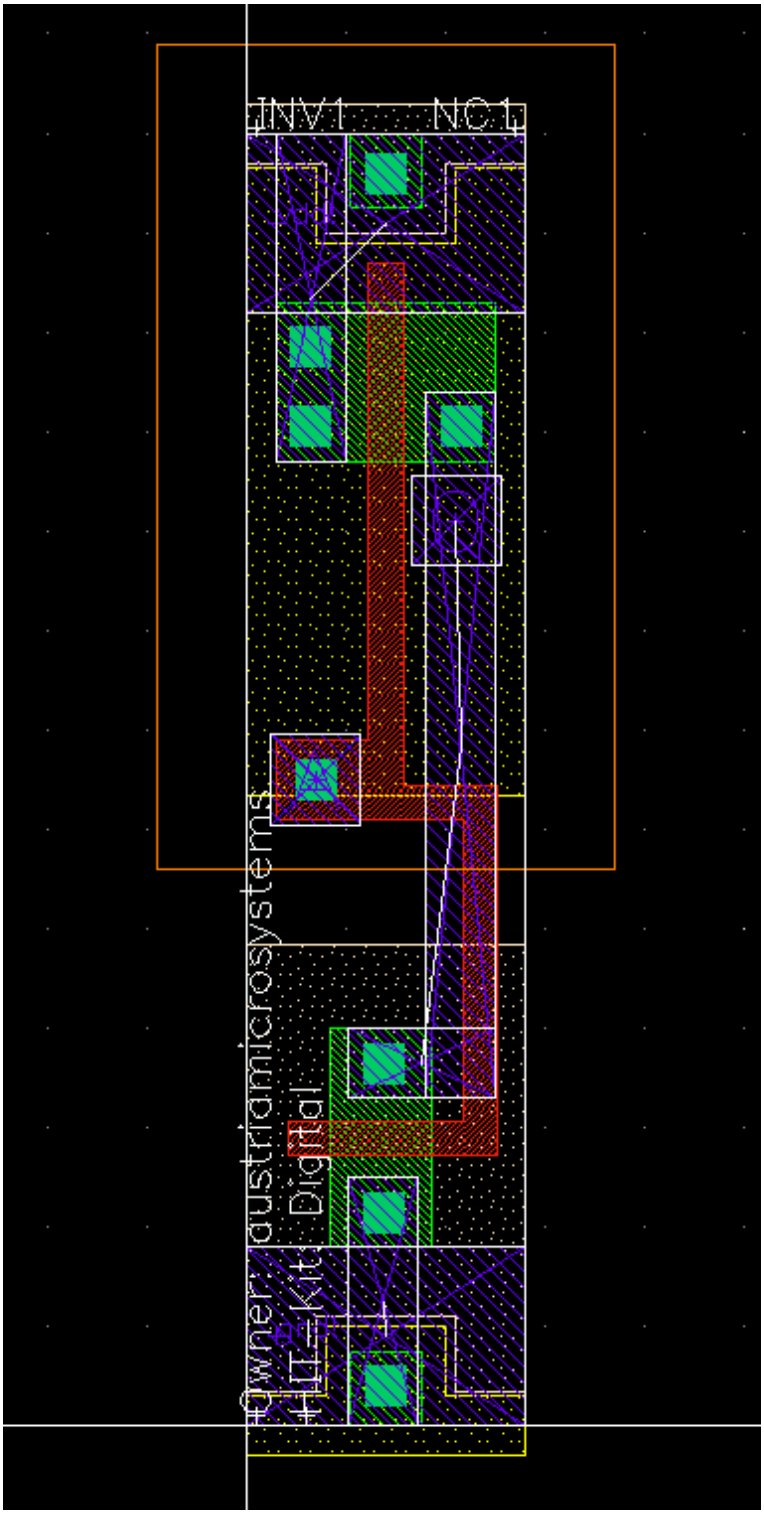

Figure 4.7: Austria Micro Systems $0.35\ um$ inverter standard cell

small enough to fit into an FPGA or at least that it can be divided into appropriate units. Additionally the FPGA prototyping is only an acceleration of the HDL level verification. Errors introduced by ASIC tools are not covered. To perform the verification of the placed and routed design a *design rule check* is performed. This ensures that the design rules of the used process technology—for instance geometric rules like minimum distances between transistors or wires—are accomplished. However the real effort lies in simulation. The final design is back-annotated to HDL code and a timing simulation is performed. This simulation takes a lot of time and therefore choosing a complete but as small as possible set of test vectors is essential.

In this work a design flow framework was used which allows changing between the two target technologies seamlessly. ASIC synthesis runs were only performed to get area and speed estimates. Therefore no additional verification was performed to ensure correctness of the circuit.

Chapter 5

Implementation

This chapter explains the details of the implementation. At first a short overview of the complete system is given. Then the hardware architecture is explained. Finally the software implementation is presented.

Figure 5.1 shows the operation layers of the elliptic curve processor system. The lowest level colored in red is the hardware, that is the processor itself and the PCI-bridge, both implemented in the FPGA on the PCI-board[7]. The yellow levels are the system levels, where the PCI-driver is already provided by standard Linux drivers. The device driver is implemented specific for our hardware. This system parts run in the privileged mode of the processor. The layers colored in blue are libraries to ease the access for native Linux program and to allow hardware access by Java programs. Finally the green layers are the applications itself, respectively the Java virtual machine running the JCE library for cryptographic applications, and the real application.

5.1 Hardware

For server applications throughput is the most important performance criteria. A higher throughput can be achieved by decreasing the latency of one point multiplication or by executing several operations concurrently. Java model benchmark results (see section 6.1) showed that the obvious way to decrease latency, namely using a faster multiplier with a higher multiplication radix, scales badly for higher radices. Therefore a multiple core approach was chosen to allow higher performance especially on large FPGAs. Each core computes a complete point multiplication independently of the other cores.

To calculate the point multiplication algorithm 3.8 is used. This novel approach supports both single point multiplication and simultaneous point multiplication of two points with a very small hardware overhead. No pre-computations are used as this would lead to a much more complex control and the resulting hardware architecture would be only usable in FPGAs.

The most performance critical finite field operation is the modular multiplication. An interleaved modular reduction with a fixed modulus is used and the operand B of the multiplication is processed by a MSB-to-LSB scheme (see algorithm 3.15). In hardware only the NIST reduction modulus for $GF(p_{192})$ is implemented, but the JAVA RTL-model also supports the NIST-primes for several larger fields and an implementation in hardware should be straight forward. In comparison to Montgomery multiplication (see algorithm 3.16) which is commonly used to implement modular multiplication in hardware, the fixed reduction prime approach has the advantage that much less hardware resources are required. So only one radix multiplier, which is the dominating part in the architecture, is used,

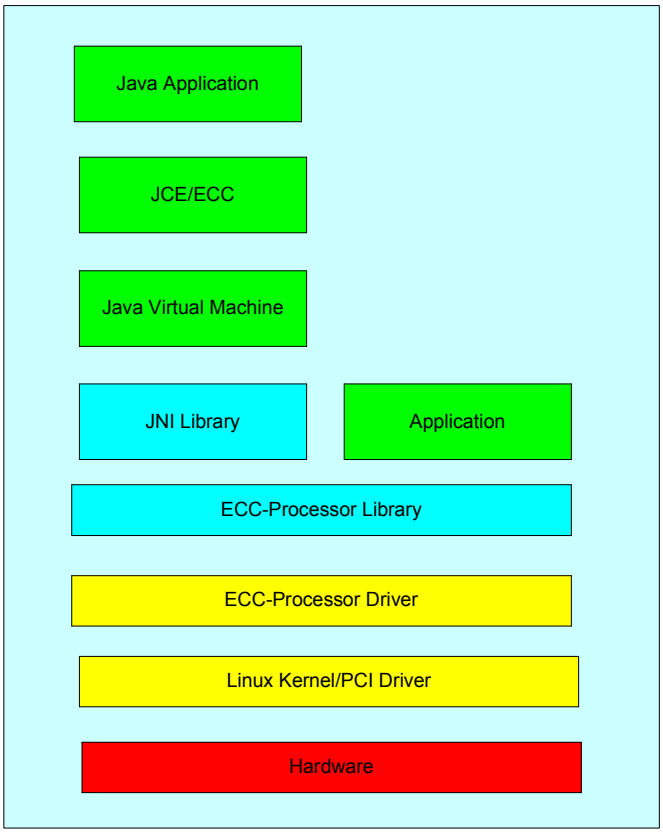

Figure 5.1: System layers of the ECC-processor system

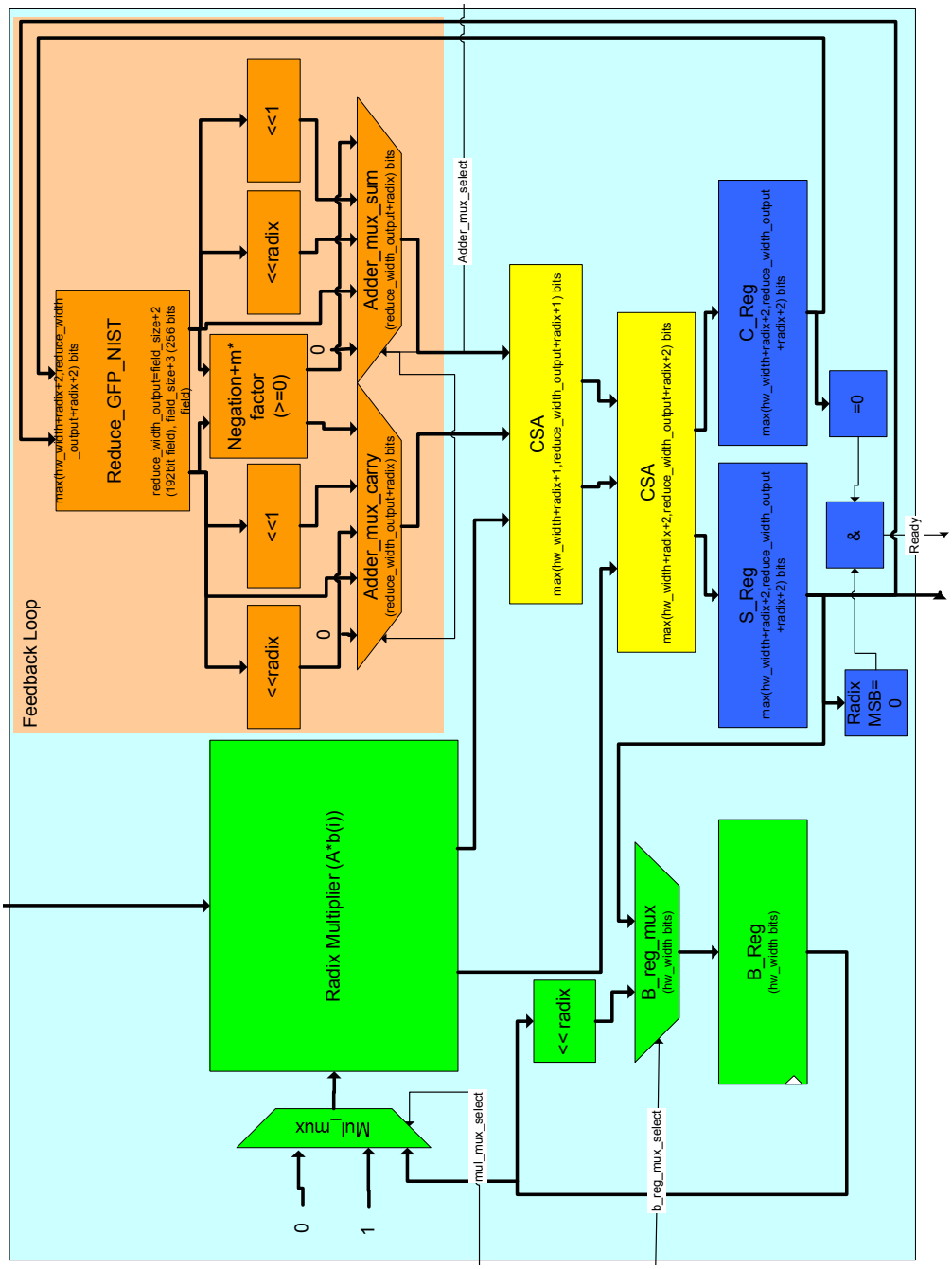

Figure 5.2: Elliptic curve processor ALU

while for Montgomery multiplication two radix multipliers are required.

Figure 5.3 shows the architecture of the elliptic curve processor core. The main part of each core is the *ALU*, which performs the finite field operations. The main task of the *control unit* is to carry out the point multiplication and it performs also the other elliptic curve operations like point addition. The *register file* is the memory for the operands and intermediate results but also supports the bus-IO-transfers, by supplying, additionally to the full word length, 32-bit-IO-ports with bank select. To reduce area requirements this circuit is reused for retrieving the current bit of the scalars k and l. For this task additionally a 32-to-1 multiplexor (KBit Mux) is used.

5.1.1 ALU

The *ALU* (see figure 5.2) operates on the full length word for one operand of the multiplication and for all other supported operations. The second multiplication operand is processed on a per digit basis with a parameterizable radix size, which can be between 1 bit and 32 bits for 192-bit prime fields, with the condition that the hardware width must be divisible by the radix. The maximum radix is limited by the reduction unit, which can easily be extended to support larger numbers. However, simulation results show that for high radices the redundant-to-binary conversion is dominating, and therefore using higher radices is little beneficial.

The ALU consists of two paths: The radix multiplier and the feedback loop, which besides doing the shifting for the multiplication, also performs the other finite field operations, and includes the reduction functionality. The results of both paths—four full precision values, because both the radix multiplier and the feedback loop produce numbers in carry save representation—are summed up with two carry save adders and saved in two registers. Locating the reduction unit in the feedback loop does have the drawback that an additional clock cycle is required to calculate the reduced result. However, it has the advantage that the critical path is shorter and that the paths are well balanced. This would allow the use of hybrids of CSA and carry propagation adders in the feedback path, where the width of the carry propagation adders is varied depending on the radix of the radix multiplier, such that both critical paths stay balanced. The benefit is that the carry-save-to-redundant conversion takes less cycles when higher radices are used, which will improve the scalability of the architecture for higher radices.

Radix multiplier

The *radix multiplier* (see figure5.4) is based on the one presented in Wöckinger [35]. The partial products are generated by shifting the operand by the position index of the corresponding radix bit and are multiplied by this bit by an and-function. The results of this partial product generators are added up by a *Wallace-tree* using carry save adders. This structure is optimized for a minimum propagation delay. However, it has the drawback that the structure is irregular, which leads to complicated routing. Particularly for FPGA implementations it is probable that using a structure which has a higher delay but is more regular, for example a carry-save adder array, performs better.

While the full length operand for the multiplication originates directly from the register file, the second operand, which has a length of radix bits, is generated and saved internally. An operation is defined which loads the content of the register `s_reg` which holds the sum to the register `b_reg`. Only the lowest bits of s_reg corresponding with the hardware width are used, therefore the register must be reduced and in binary form before beginning a multiplication. A feedback loop is used for shifting its content in each multiplication cycle to the left by radix bits, where the highest bits are used as operand for the multiplier. A multiplexor is required for selecting between the load and the shift operation. In addition another multiplexer is used to select between the radix bits of the operand, zero

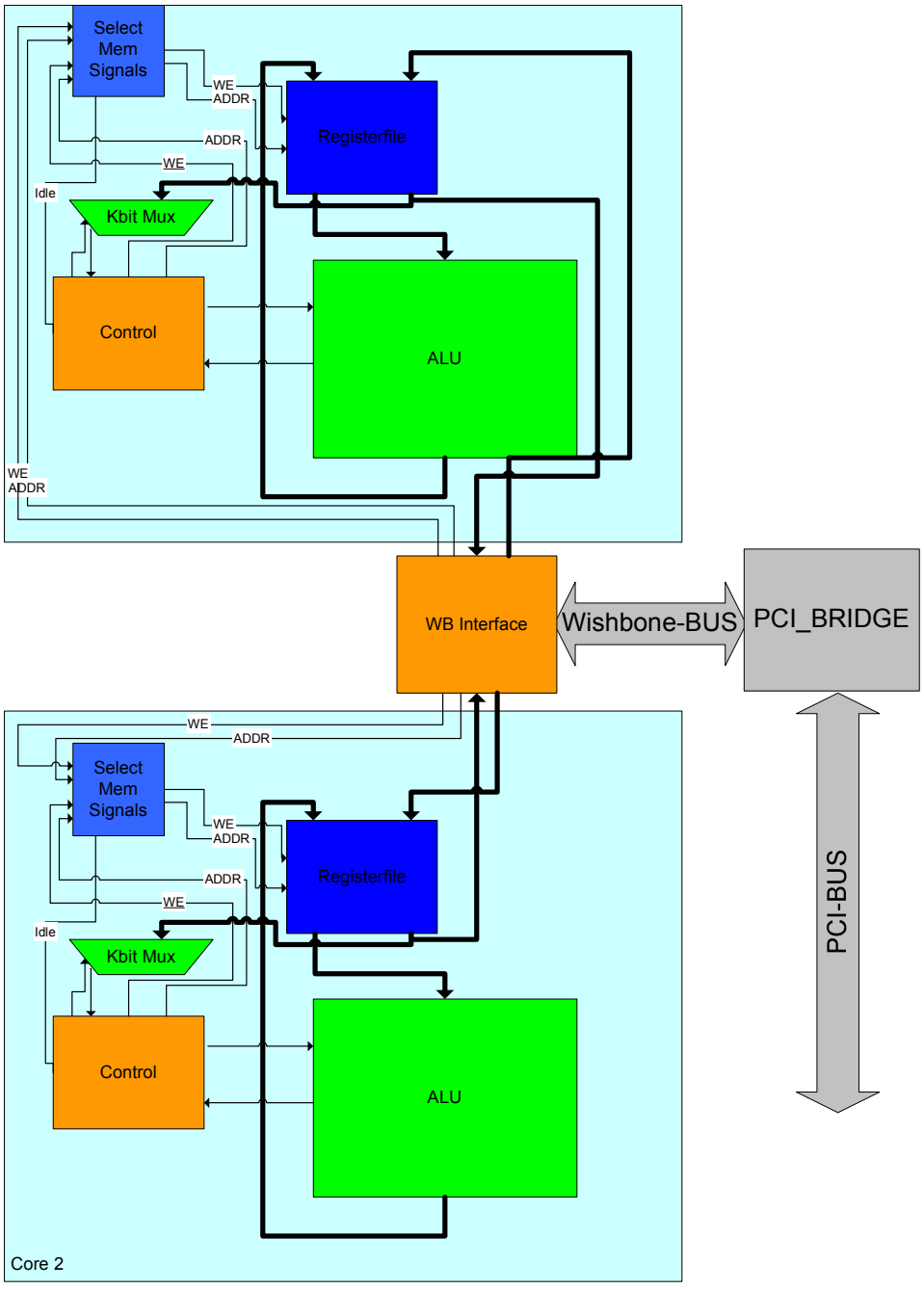

Figure 5.3: Elliptic curve processor architecture

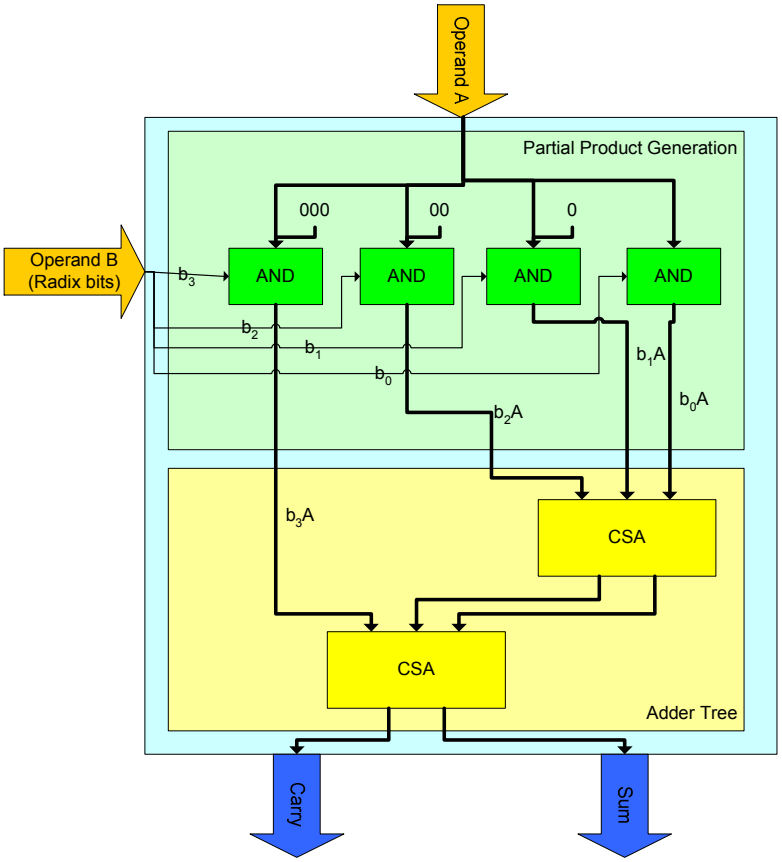

Figure 5.4: Parallel radix multiplier (4-bit radix)

and one. This is necessary to disable the multiplier when other arithmetic operations are performed, respectively for loading the register with a value from the register file because the only input to the ALU is over the radix multiplier.

Feedback loop

The basic function of the feedback loop is the shifting of the intermediate result for the modular multiplication. The aligned intermediate result is added to the new partial product generated by the radix multiplier. Because of the high complexity and the long critical path of the radix multiplier, it makes sense to perform the other arithmetic operations in the feedback loop and so reuse the final adders and balance the critical paths. To do this two multiplexors are used, one for the carry and one for the sum part, to switch between the shift left by radix operation required by the multiplication, shift left by one for the doubling, the result of the negation unit, which is used for the subtraction, and finally the direct loop back which can be used for addition and for redundant-to-binary conversion. In addition a zero input is used to allow loading of the registers. To perform arithmetic operations the radix multiplier operand_b input must be set to zero respectively to one to disable the radix multiplication. To balance the critical paths of the multiplier and the feedback loop even more also the modular reduction is performed here. This comes at the cost of an additional cycle for reducing the result. The shorter critical path allows higher maximum clock frequency that compensates the additional clock cycle by far. In the following the two more complex units used in the feedback loop are described in detail, the reduction unit and the inversion unit.

Reduction unit

The *reduction unit* reduces the intermediate result by using the equations for reduction with NIST primes recommended by the FIPS 186-2 standard. A shortened version of the equations was used which only support inputs up to $field\_size + k$ instead of $2 \cdot field\_size$ with $k = 64$ for the 192-bit prime field. This saves adder/subtracter steps at the cost of limiting the maximum radix for the multiplier. While the reduction with the 192-bit NIST prime only uses additions. Larger NIST-prime fields also require subtractions. To avoid the occurrence of signed numbers the same strategy is used as in the negation unit. A multiple of the NIST-prime is added, which is definitely larger than the possible most negative result. This ensures that both the carry and the sum part of the reduced result are always positive. As a drawback the reduction result can now be larger, for example one bit for 256-bit fields, which requires that the hardware width must be enlarged by this single bit, and the final result of the complete point multiplication has to be fully reduced in software, which is only a single conditional integer subtraction of the NIST-prime. It is possible to extend the ALU for supporting not only one single field, but also all smaller NIST-prime fields. This is a called a *maximum word length* architecture. To archive this a multiplexor can be added to allow switching between reductions unit for different fields.

Negation unit

The *negation unit* is used for the subtract operation. It calculates the two's complement of the intermediate result and adds a multiple of the reduction modulus to ensure that both the carry and the sum result for finite field inversion are always positive after the negation. This is required because sign extension, which is necessary for negative numbers, would complicate the hardware, and is incompatible with the redundant-to-binary conversion because of the reduction unit in the feedback loop.

Final adders and registers

The results of the radix multiplier and the feedback loop are added up by a CSA-adder structure. Both paths are in redundant representation, therefore four inputs are required for the adder structure. This means that two CSA-adders are necessary. The logical place for the reduction unit would also be in or after this final addition step. However to reduce the length of the critical path the reduction was moved to the feedback loop. Therefore the width of the intermediate results is increased by the radix and a small value because of the adders in the path. This two parts of the result, carry and sum, are saved separately in the two registers `c_reg` and `s_reg`. As the hardware architecture has no carry propagate adder for redundant-to-binary conversion it uses the feedback path for adding up the sum and the carry repeatedly until the carry vanishes. Detection that carry equals zero is required. This is also kept out of the critical path by using the registered value for comparison. This increases the cycle count for the required conversion by one, but is probably the better solution, in particular because the conversion is not performed after all arithmetic operations. The same is true for the reduction moved to the feedback path, which also increases the cycle count by one. In fact the reduced and binary result is only required for performing a further multiplication or for saving it into memory.

5.1.2 Control

The control unit generates the control signals for the ALU. It is designed as a hybrid between microcode and hard-wired control. That is, the control signals are generated by a ROM based microcode program. However, no conditional instructions are supported and so the higher level control is performed by a hard-wired state machine. Viewed a little simplified the state machine control is responsible for the elliptic curve group computations while the finite field operations are carried out by the microcode part of the control. However, the field operations are not pure microcode, because waiting for multiplications which takes more than one cycle, the redundant-to-binary-conversion, and the complete inversion is performed by hard-wired functions.

The state machine for the elliptic curve group operations and for finite field inversion can be found in figure 5.5. It is simplified to allow a more comprehensible diagram. Firstly, the pre-shift state in the real state machine reuses the check scalar and copy point states to save resources. Secondly, the check scalar state in fact consists of three separate states, two for loading the scalar from memory, and one to do the real checking.

The main task of the state machine is the point multiplication. For this the algorithm for simultaneous point multiplication with pre-shifting (see algorithm 3.8) is used. It is in large parts directly implemented in the state machine. In the conditional point addition three different points are added to the intermediate result, which depends on the currently processed bits of the two scalars. As each micro code instruction is fixed to a memory address, and therefore uses a specific variable, the selection of the correct point has to be performed in the state machine entirely. The trivial solution to use three complete point addition micro code functions each fixed for a specific point has the disadvantage of wasting resources—the code for a single point addition is about 800 bits—and of complicated maintenance. Therefore another solution is used, that is the selected point is copied by the state machine to a fixed memory address. This only takes a little more clock cycles, but this is insignificant in comparison to the total number of clock cycles required for a single loop. The two target memory address temporary variables, which are required anyway for the computations, can be reused. Thus no additional memory in the register file is required.

The selection of the current bits of the scalars is a non-trivial task. Certainly, it is possible to simply use two additional shift registers or two multiplexors two retrieve the bits. However, the scalars are very long, they have at least 192 bits. Thus this approach would use considerable resources. The

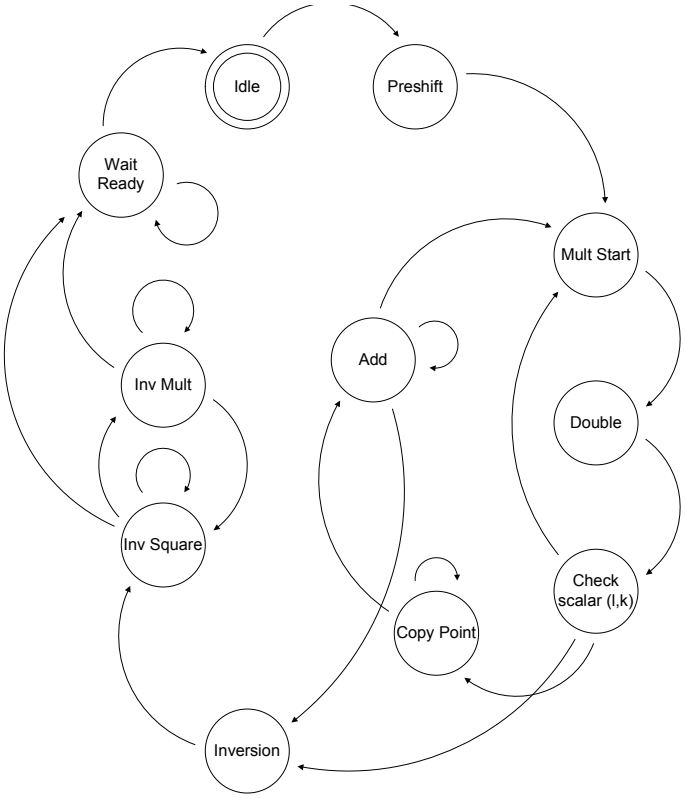

Figure 5.5: Control state machine

```
( cnt& whold & shift   & rd & ry ), -- Top.alu.shiftl()
( cnt& wnow  & hold    & rd & rz ), -- Top.alu.red2bin()
( cnt& wnow  & mult1st & rd & rz ),
( cnt& whold & mult    & rd & rz ),
          -- Top.alu.mulp(Top.memory.mem[Top.memory.mem_z])
( cnt& wnow  & hold    & wr & rz ),  --Top.alu.red2bin(),
          -- Top.memory.mem[Top.memory.mem_z] = Top.alu.get()
```

Figure 5.6: Example of microcode in the control unit

control of the EC processor for server applications reported in Wolkerstorfer [33]—which uses a quite similar architecture as our processor—solves this problem by using the ALU for shifting the scalar. For performing the multiplication the ALU possesses the ability to shift left by the radix. Thus it is only necessary to allow conditional disabling of the reduction unit, and to save the radix most significant bits in a register, where a small multiplexor can be used to retrieve the actual bit. Although this solution uses very few additional resources it has the disadvantage that it complicates the ALU with additional functionality which is not related with its real purpose. In addition, for disabling the reduction a large number of data wires has to be controlled. This large number of signals to control does not make a difference in the implementation in HDL. However, it probably leads to a considerable increased use of resources in the placed-and-routed design because the control signal has to be amplified for this high load. Thus in this work another approach is used. In fact the idea is similar, an existing component is reused to retrieve the most significant word of a scalar. This is saved to a register and the actual bit is selected by a multiplexor. The difference is that the register file is utilized instead of the ALU. This register file already has input and output capabilities with a 32-bit word length to allow bus transfers, and thus it is very well suited for the bit retrieval by using a relatively small multiplexor for retrieving the actual bit.

Finite field inversion is the second important computation performed by the control. It is performed with the theorem of Fermat (see section 3.5.1) and uses the square-and-multiply exponentiation algorithm (see algorithm 3.17). This exponentiation algorithm is very similar to the point multiplication, therefore their state machines are closely related. Only the microcode implementation operations differ, in the case of exponentiation it is much shorter. A design alternative which was considered is the hard coding of the exponent. This would ease control because no conditional operations are necessary. Furthermore an entry in the memory file would be saved, and it would relieve the software from the requirement to compute the exponent $m - 2$ and to transfer it to the hardware. However, this approach was discarded because it would complicate the addition of larger fields because inversion must be rewritten and especially the support for more fields in one core would be harder. In addition the selection of the current bit can be performed by reusing the bit selection for the point multiplication and is therefore easily possible.

The microcode programming, an example is shown in figure 5.6, is complicated by the fact that pipelining is used to allow higher clock frequencies. This means that the operation performed by a single instruction is not performed on the data respectively memory address included in the instruction. On the one hand this is caused by hardly avoidable reasons. For example reading a variable from the register file always has to be performed at least one cycle before using it. Here waiving pipelining would cause a considerable performance loss. On the other hand pipelining is necessary because of a less obvious reason. That is that control signals have to be registered. If this is not done time required for the generation of these signals adds up to the time for the operation controlled by

the signals. In our case the situation is even worse because single signals control hundreds of data signals, the *fan-out* is very high. To allow this control an amplification of the control signal has to be performed, and the timing delay caused by this can easily half the maximum clock rate of the complete circuit. Registering the control signals solves this problem. Firstly, the signal generation path is separated from the algorithmic path. Secondly, the register can be replicated by the synthesis tools, which allows performing the amplification in the control path, which usually is much shorter than the path in the ALU. For example the register which holds one of the signals controlling the feedback path multiplexor is replicated about 50 times to allow clock rates of 88 MHz on the Spartan-3. The disadvantage of this registering is that an additional cycle is required in the control, which demands the use of pipelining to prevent a performance drop. This registering is not only done for the control signals but also for the status signals of the ALU, namely the ready signal. As this signal is generated by the output of the register even one more register stage is introduced, which leads to a total delay of two cycles to detect that the result is fully reduced and in binary form.

5.1.3 Register file

The register file holds the variables and EC parameters for the elliptic curve point multiplication. It has the width of the ALU operands, for example 192 bits and is organized in banks of 32 bits. The depth, that is the number of entries of the register file, is 16. In detail it holds the two scalars l, k, the four coordinates of the two base points p and q, the two coordinates of the pre-computed point $p+q$, the elliptic curve parameter a[1], and the inversion exponent $p-2$. Two more entries are used for the projective result, one entry for the third coordinate of the result, and three entries for temporary variables. Fortunately, a depth of sixteen is very efficient in the target FPGA Spartan-3. Reducing the number of entries does not have any advantage as long as more than eight entries are required, and such a massive reduction is not possible.

The organization in RAM banks is required for the bus transfers. Two multiplexors, one for writing, the other for reading, allow selecting the bank, and separate outputs and inputs can be connected to a bus. The register file is parameterizable in width, depth, and bank width, and can therefore easily be adopted for other field sizes, and even bus widths. The IO output is also used for retrieving a specific bit for a variable by the control, for details see section 5.1.2.

5.1.4 Performing an arithmetic operation

The finite field operations which can be performed by the arithmetic unit are: *multiplication*, *addition*, *subtraction*, *inversion* and *doubling*. The elliptic curve group operation algorithms where changed slightly to allow weaving the shift right operation. This operation is unpleasant because it requires a conditional addition of the field prime if the least significant bit is set. The EC group algorithms also do not use the field addition operation, but it is implicitly available because of the architecture of the ALU.

To separate the datapath and control more cleanly and to reduce the number of inputs the ALU can not directly carry out the finite field multiplication, but the ALU instructions (see table 5.1) allow performing the field operations with little control overhead. For example to perform a finite field multiplication the control has to carry out the following operations:

1. load

   This loads the ALU input into the sum register `s_reg`. Because of the pipelined control this operand must be read from the register file one instruction before.

[1] The parameter b is not required for mathematical reasons

Operation	Adder Multiplexor	Multiplier Multiplexor	Operand B Multiplexor
mult_1st	Zero	Zero	Load b_reg
mult	Shift by Radix	Radix highest bits b_reg	Shift b_reg by radix
neg	Negation	Zero	-
sub_rev	Negation	One	-
shl	Shift Left by 1	Zero	-
hold	Feedback	Zero	-
load	Zero	One	-

Table 5.1: ALU operations

2. mult_1st

   Loads the sum register value into the operand B register `b_reg` and clears the sum and carry registers. Together with this instruction operand A has to be read from the register file, and has to be kept throughout the multiplication.

3. mult

   Performs a multiplication cycle. The ALU input is multiplied with highest radix bits of the operand B register `b_reg`. The result is added to the result of the last iteration, which is multiplied by 2^{radix} and reduced by p before. The redundant sum is stored in the result registers. In addition the operand B is shifted left by radix bits and saved into `b_reg` again. The mult instruction has to be repeated $width/radix$ times.

4. hold

   After performing the multiplication intermediately further arithmetic operation expect another multiplication can be performed. However, to retrieve the binary result, required to retrieve the result from the ALU, or for performing another multiplication the operation hold must be executed repeatedly until the `s_reg` is completely reduced and `c_reg` is zero, which is signaled by the ready output of the ALU. The hold instruction just loops the intermediate result over the feedback loop without performing an arithmetic operation. Thus the sum and carry part of the result are reduced and summed up. The carry disappears after some loops.

The other operations are carried out analog, one operand always has to be in ALU registers, the second, if there is one, must be fed into the ALU input. Interesting is the subtract operation. Here the register content is negated and the ALU input is added, therefore the first operand is subtracted from the second operand, which is the opposite of the normally expected behavior. If the algorithm cannot be modified to use only such subtractions, it is necessary to perform an negate instruction after the subtraction.

5.1.5 Interface

The connection of the ECCP to the PCI bus is provided by the PCI Opencores bridge. The bridge connects the PCI bus to a Wishbone bus. It is very flexible and can be configured to work as host or guest bridge, and supports in both modes master and slave devices on both buses. In our case the host bridge is part of the PC system. The Opencores bridge operates in guest mode and connects to the ECCP, which is a slave device on the Wishbone bus.

The ECCP top level module `gfpecc_wb` instantiates several instances of the core and connects them to the WB bus. The WB connection uses the registered mode. This means that the address and data signals are saved in registers and are not directly connected to the RAMs. Because of the high clock frequency of the ECCP cores it is quite probably that the critical path of the complete circuit would be in the WB interface, if the not registered method had been used. The disadvantage of the registering is that the duration of bus cycle is doubled. To get rid of this the WB standard defines burst transfers, where more than one data word is transfered in a single bus cycle. To allow this special bus signals are used which notify the device that it should not terminate the bus cycle immediately after sending or retrieving the first word. However, because the WB side is much faster clocked than the PCI side, which only operates at 33 MHz, this burst transfers on the WB side are probably not a real benefit, and therefore only the basic registered bus transfer is supported by the ECCP interface. Such a basic cycle is shown in figure 5.7. The bus master, here the Opencores bridge, signals with `CYC_O` and `STB_O` that a bus transfer takes place. Here `WE_O` is 0 thus a read cycle is performed. The bus master must present a valid address on `ADR_O` at the same time. The WB slave can fetch the corresponding data from the memory and put the data on `DAT_I` and assert `ACK_I` as soon as it is ready to signal the end of the cycle. The master fetches the data at the next clock edge when `ACK_I` is raised and the slave sets `ACK_I` to zero. The master can now start another transfer but the extra cycle for negating the acknowledge signal halves the maximum transfer rate in the classic bus cycle. Writing to the bus works in an analog way.

5.2 Software

Figure 5.1 shows the software layers which have to be implemented. In the following three sections, device driver, library, and application are explained in detail.

5.2.1 Device driver

The device driver implementation[2] is a Linux character device driver written in C. A character device driver allows accessing the hardware over file descriptors which are also used for physical files. The structure `file_operations` defines and assigns the file functions implemented in the driver. These operations only define the access of applications to the driver, in addition functions are defined which allow loading and unloading the driver module. For this purpose the `MODULE_DEVICE_TABLE` with functions for loading, probing, and unloading the driver are used.

Fortunately the PCI standard allows plug and play configuration of devices. Linux implements this and automatically assigns resources like memory regions and interrupts to the device by writing them to its configuration registers during boot. Command line tools likes `setpci` and `lspci` allow easy access to the configuration registers, which is useful for debugging purposes. Furthermore, it was used during the development process to save the configuration, which usually does not change unless additional hardware is added to the computer, and to load this configuration after reprogramming the FGPA. This eased hardware debugging by far because it saves a lot of reboots. Figure 5.8 shows a typical configuration of the elliptic curve processor implemented on the FPGA PCI board. Most of the parameters are defined by the PCI bridge, and handled by Linux, and thus do not have influence on the ECCP processor and its driver. However, despite of these some modifications are necessary in the Opencores bridge to make the hardware work. For example defines the PCI-Bridge the device as type “bridge”. This is even not modifiable by the normally used user constants. However, the used

[2] The sample implementation of a driver and an Opencores-PCI hardware by Fürbass and Bouvier is used as a basis, and proved to be very useful.

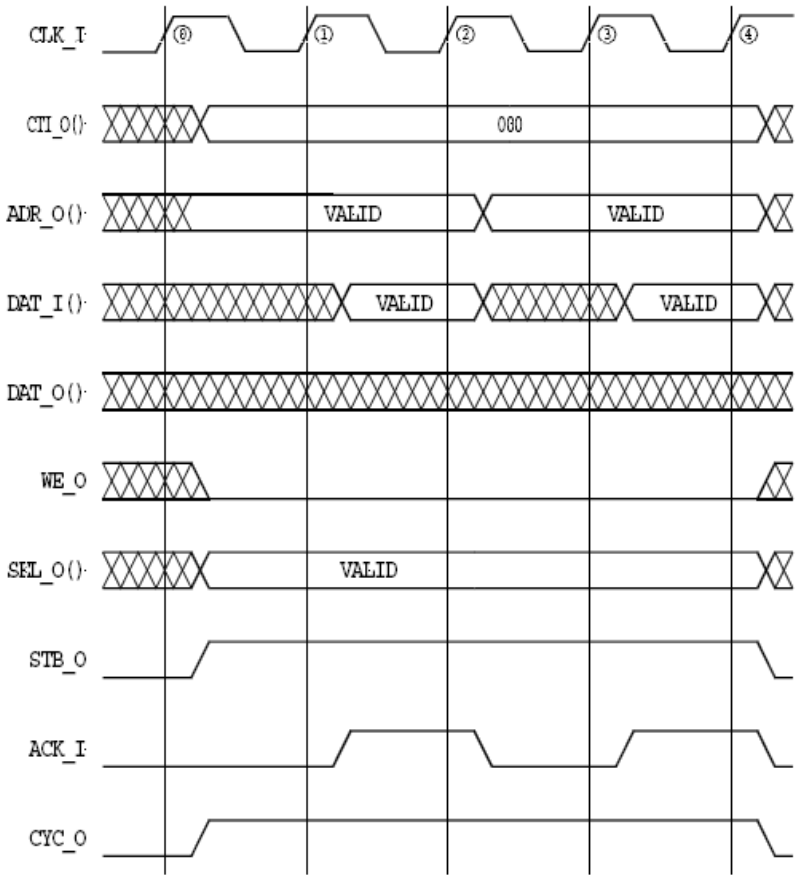

Figure 5.7: Wishbone registered feedback cycle [25]

```
0000:00:09.0 00ff: Unknown device 1895:0001 (rev 01)
prjsem02:/home/cpuehrin/diplomarbeit/tex/images# lspci -d 1895:1 -vv
0000:00:09.0 00ff: Unknown device 1895:0001 (rev 01)
        Subsystem: Unknown device 1895:0001
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop-
                  ParErr- Stepping- SERR- FastB2B-
        Status: Cap- 66MHz- UDF- FastB2B+ ParErr-  DEVSEL=medium
                   >TAbort- <TAbort- <MAbort- >SERR- <PERR-
        Latency: 32 (2000ns min, 6500ns max),
                   Cache Line Size: 0x08 (32 bytes)
        Interrupt: pin A routed to IRQ 11
        Region 0: Memory at da010000 (32-bit, non-prefetchable) [size=4K]
        Region 1: Memory at da000000 (32-bit, non-prefetchable) [size=64K]
```

Figure 5.8: Configuration of ECCP PCI card displayed by `lspci`

Linux version, proved itself incompatible with this setting and failed to assign a correct interrupt. A user constant to define the type was added, and was changed to “unknown”, which allows correct function of the interrupt. Directly important for the hardware and the driver are the assigned memory regions, and the interrupt number.

The PCI bridge defines the number of memory regions, their size, and their type. The type is used to differentiate between normal memory and IO memory. IO regions are non-prefetchable which means that the CPU must not cache data at this addresses because the data may be changed by the device. This IO regions are the preferred way of doing IO operations, furthermore access by ports is possible but not recommended. During the boot process the operating system reads these parameters and assigns them to a free physical memory address region, by writing the 32-bit start address to the configuration register of the device. The possibility to use various memory regions is only used by the ECCP device for having a separate configuration space—which is provided by the bridge—and a memory for the elliptic curve data. Because of the homogeneous nature of the multiple cores it is more simple and efficient to do the assignment of memory addresses directly in the processor than assigning different PCI memory regions. The latter would also limit the maximum number of cores to six.

Interrupt configuration is done automatically by the operating system. The PCI standard allows up to four interrupt pins per device. However, most devices use only a single interrupt pin, namely `INT_A`. The interrupt lines of the devices are routed to the interrupt controller. The assignment to the visible interrupt in the operating system is not unique because interrupts are a scarce resource, the interrupts are shared among various devices. Thus the interrupt handlers of the device drivers have always to verify whether the interrupt is from its device.

The ECCP driver or more general any Linux PCI character device driver can be divided in three parts. Initialization and unloading functions are called when the driver module is loaded respectively unloaded. The driver uses a variable of type `pci_device_id` to allow Linux to find the correct driver for each available PCI device. The initialization routine of driver maps the memory of the PCI device using `ioremap` to obtain access to this memory. Then it registers a character device with its `file_operations` structure assigned by using `register_chrdev`. This allows access by applications to the device using file functions. Optionally it also registers a device in the `/proc/` file system. The unloading of the device driver unregisters the devices and unmaps the memory regions.

The file functions are the second important part of the driver. It provides the basic functions `open`, `release`, `read`, and `write`. Of special importance are the memory mapping function `mmap` and `ioctl` which is used for device configuration. Here `open` is used to assign the interrupt handler, and to enable interrupt generation in the device PCI bridge. The `release` disables them and unregisters the handler. The `read` function plays a different role than in normal file operation. Although it could be used for reading, the data access is only performed over the mapped memory, and read only serves for blocking the calling thread until the processor finishes the computation. The `mmap` function maps the device memory to user space by using `remap_page_range`. The application can now access directly the device memory. This leads to higher efficiency, because no memory copies have to be performed, like in the file `read` and `write` functions. `ioctl` is used to allow software to retrieve device parameters like the number of cores or their status.

The application or the provided library performs an elliptic curve computation in the following way. It writes all the data, like the curve parameters, scalar and the base point, to the memory region assigned to a processor core, then writes the start command to the ECCP, and calls the `read` function. This blocks the application thread. When the processor core finishes computation it generates an interrupt, the interrupt handler wakes up all threads blocked in `read`. These check whether their core is ready, if not they return to sleep again. If their core is ready they leave `read` and the application reads the elliptic curve results from the ECCP memory.

The third part is the interrupt handler which is responsible for interrupt processing. Correctness of all functions in the driver is very important because of their privileged state. However, the interrupt handler is even more critical, sloppy programming not taking into account proper protection of the critical section can lead to disabling of the interrupt by the kernel or even a complete lock up of the system. Thus a spinlock is used to protect the interrupt handler of being called multiple times. The first task of the handler is to check whether the interrupt was generated by the ECCP by reading the interrupt status register of the PCI bridge. Then the interrupt status registers of all cores are reset to clear the interrupt. After that the bridge interrupt is cleared by writing to the interrupt control register. The order of this is relevant, if the device interrupt is not cleared before clearing the bridge interrupt malfunction can occur. The preferred way in Linux is to use tasklets to perform the real tasks of the handler. This is also called *bottom-half* processing, and allows higher efficiency because only the critical part is performed in the handler, while the non-critical, and usually more complex part is performed in a schedulable tasklet, which therefore does not block other processes. This was also done here although the only task is to call `wake_up_interruptible` which awakes the threads sleeping in the blocking `read` function.

This was in short the functionality of the driver. Perhaps noticeable is the fact that the scheduling is not done in the driver. This has to be done by the calling application, or more typical a library. The only required support in the driver is that in blocking function `read` depending on the address the status of the related core is checked, and only threads waiting for the really ready cores return from the function.

5.2.2 Library

Two libraries are part of the system. One serves to allow access to the ECCP on a slightly higher abstraction level. Principally the character device nature is hidden, because this is not a very logical approach for a device such as the ECCP. Thus functions are provided to encapsulate for example the `ioctl` function for retrieving the number of cores or the status of the cores.

The second library is a JNI library to allow access to the hardware from Java applications by supporting the JCE library. This is in particular useful because powerful elliptic curve cryptography is available for Java which eases developing of such applications by far. First, an interface to define the supported functions is declared in Java. The methods to implement in the C library a declared natively. The declaration can be used by the `javah` tool to generate the header files for the C library automatically. Special data types are defined by `jni.h` to represent Java data types in C. In addition to the parameters defined by the functional interface a pointer to `JNIEnv` and `jobject` is passed. This allows access to Java functions respectively the current object from within the C library. This could for example be used for synchronizing threads which are using the function concurrently. However, to allow the reuse of the library for native C library Linux system call are used for this purpose. Synchronization plays an important role in the library because it is responsible for assigning threads to free cores.

The JCE library defines three types of functions. The *initialization* initializes the ECCP hardware and creates the semaphores used for synchronization respectively critical section protection. Some *configuration* methods allow retrieving configuration information of the hardware from Java program. This allows transparent switching between software and hardware implementation depending on the curves supported by the hardware. The third group are the *arithmetic functions*. Currently two such functions are implemented, `mulpoint` which performs a single elliptic curve point multiplication, and `mulpoint_simultaneous` which computes the simultaneous two point multiplication efficiently used in signature verification. Both functions work nearly equal. They only differ in the number of parameters and in the values written to the elliptic curve memory. Thus, only the single

point multiplication is described here in algorithm 5.1.

Algorithm 5.1: JNI single point elliptic curve point multiplication

Require: Byte arrays: arr_px, arr_py, arr_d
Ensure: $(arr\_qx, arr\_qy) \Leftarrow arr\_d \cdot (arr\_px, arry\_py)$
1: core $\Leftarrow$ getfreecore_blocking()
2: Assign parameter arrays to core specific buffer
3: Write buffer to core memory
4: start(core)
5: waitready(core)
6: Copy result to buffer and assign to (arr_qx, arr_qy)
7: Release core
8: **return** (arr_qx, arr_qy)

The `getfreecore` function waits on a semophore initialized with the number of available cores. An array `core_busy` which is protected by a lock is used to find out which core is free and thus can be used. In `releasecore` the semaphore is released and the entry for the current core in `core_busy` is cleared.

Unfortunately the implementation of the JCE only allows access to large numbers as a byte array. Therefore the data arrays have to be parsed byte-wise and every four bytes are merged to a 32-bit word before they can be transfered to the ECCP memory. However, the share of bus transfer is low and therefore no significant performance loss should be caused.

To obtain a working JCE extension only two additional Java classes are required. Firstly, an implementation for elliptic curve, which does not much more than calling the JNI native functions. Secondly, a Factory which creates such a hardware curve for supported prime fields and a software curve for not supported fields.

5.2.3 Application

Certainly, the C libraries can be used to program applications for elliptic curve cryptography. However, although the hardware performs the computational most demanding functions, it is still a long way to implement a useful application, for example for signing and verifying documents. Fortunately, using the JCE[15] library eases this task by far. The most simple way, which is also taken in this work is changing an existing JCE application to support the hardware. In fact it is sufficient to add the line `ECGroupFactory.setDefaultFactory(new HWECGroupFactory());` to the code and to add the hardware acceleration library to the class path when executing the application. Then the hardware acceleration is used for supported curves, while other curves are transparently processed by the software implementation of the JCE. The application used in this work is a signing and verification demo, which is extended for benchmarking the various tasks of the system.

Chapter 6

Results

The performance results are given for three scenarios, respectively system levels. The first one is the result of the Java RTL level model. This comprehends only theoretical values which do not have a direct significance. Despite of this they are given because on the one hand the accuracy of the model can be discussed and on the other hand it allows performance estimations for configurations of the ECCP which do not fit into the target FPGA and of prime field sizes which were not implemented in HDL. The latter is in particular interesting because the reduction is more complicated for larger fields and therefore the performance does not scale linearly.

The VHDL simulation result would be more accurate but because of long run times it was infeasible for performance comparisons. Thus it is only used to verify the correctness and to assess the accuracy of the Java model.

The second set of benchmarks was run as a C program which uses the real ECCP on the FPGA board. This are the most significant results for determining the performance of the hardware design.

The last scenario uses the ECDSA algorithm running on Java with the JCE library. This are the most interesting results from the application point of view but because of the overhead are less suitable to determine the hardware performance and to compare it with other ECC processors. This comparison is the final section. It basically uses the results of the C++ benchmark but also uses estimates for the Virtex-4 LX200 FPGA and compares it with results or estimates of other ECC processors.

6.1 Java Model Results

The Java model timing results were obtained in an early stage of the design process. Therefore the played an important role to establish the ECCP architecture. They lead to the decision to use a multiple core architecture because the performance scales quite poorly with the area for higher radices. This caused by the share of cycles of operations which are not accelerated by higher radices increases with higher radices and finally dominates the total timing. This share is shown in figure 6.2. For example it turned out after implementation that the ECCP with three 8-bit radix cores performs 50% faster than a ECCP with one 24-bit radix core even if both are clocked at the same frequency while in reality the 8-bit core can be clocked at a higher frequency. At this stage of the design process this information in particular the exact area requirement was not available. Therefore the area-cycle product was used to estimate the optimal configuration. For this the area was roughly estimated by $field\_size(radix + 3.5)$. The result (see figure 6.1) suggests that the 8-bit radix processor is the optimum solution for the 192-bit field which turned out to be correct for the target board. However,

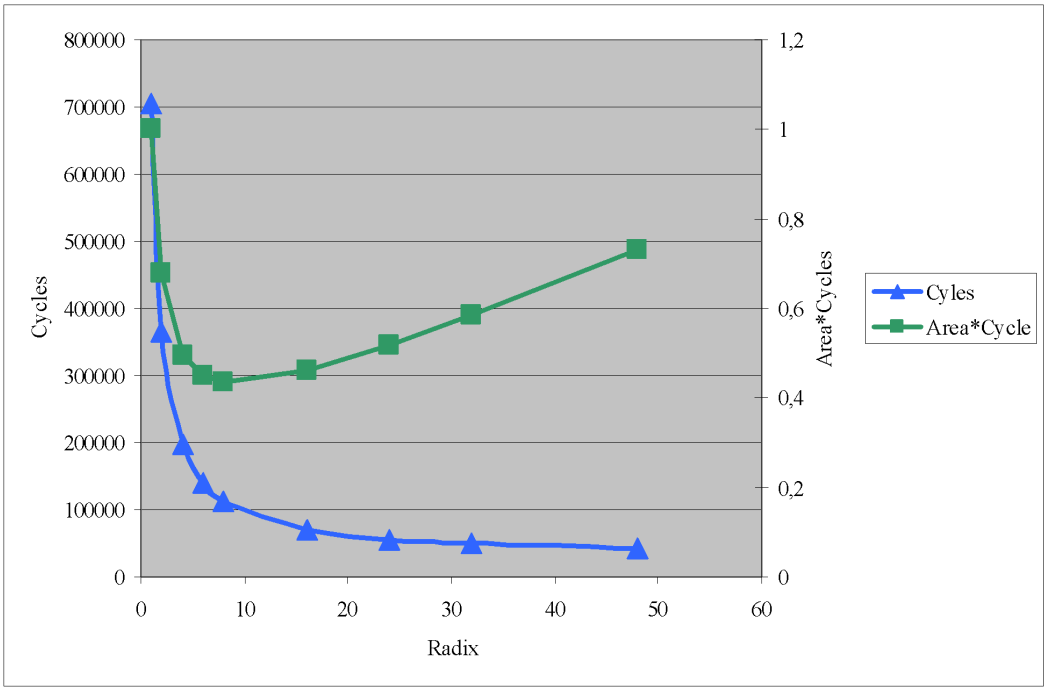

Figure 6.1: Area-cycle product for the 192-bit field

the comparison with the HDL simulation showed (Figure 6.1)[1] that the absolute estimates were too optimistic. The HDL performance estimations—which agree very good with the final results— are about 15% slower. Apparently the `HOLD` cycles were underestimated. This main reason for this to happen is the insufficient consideration of additional cycles caused by the pipelining introduced by the control.

6.2 Point Multiplication Performance on the FPGA Board

This benchmark was conducted with a simple C program under Linux. The functionality tested is similar to the Java simulation. This means that single point multiplications were tested. Advanced functions like simultaneous point multiplication were not tested at this stage because no library support is available for the required tasks like point addition and modular reduction. The benchmarks timings include the bus transfers required for writing the operands and parameters and reading the result. Verification of the results was only performed for a fixed set of operands because of the lack of library functions. Exhaustive verification was performed with the Java and JCE program. The measured performance is usually constant over more benchmark runs. However, sometimes the performance drops by about 10 point multiplications per second. This is apparently caused by background processes influencing the measurements and impaired because of the low performance of the system used. This was a PC with a 866 MHz Pentium II CPU and 256 MB RAM. Even when the system was idle nearly 10% of CPU time were used. Therefore additionally to performing 10000 multiplications

[1] The scalar k had about 60% of its bits set. Thus 113 point additions and 191 doubling where performed. Therefore all timings are slightly worse than the medium case and the share of the additions is increased.

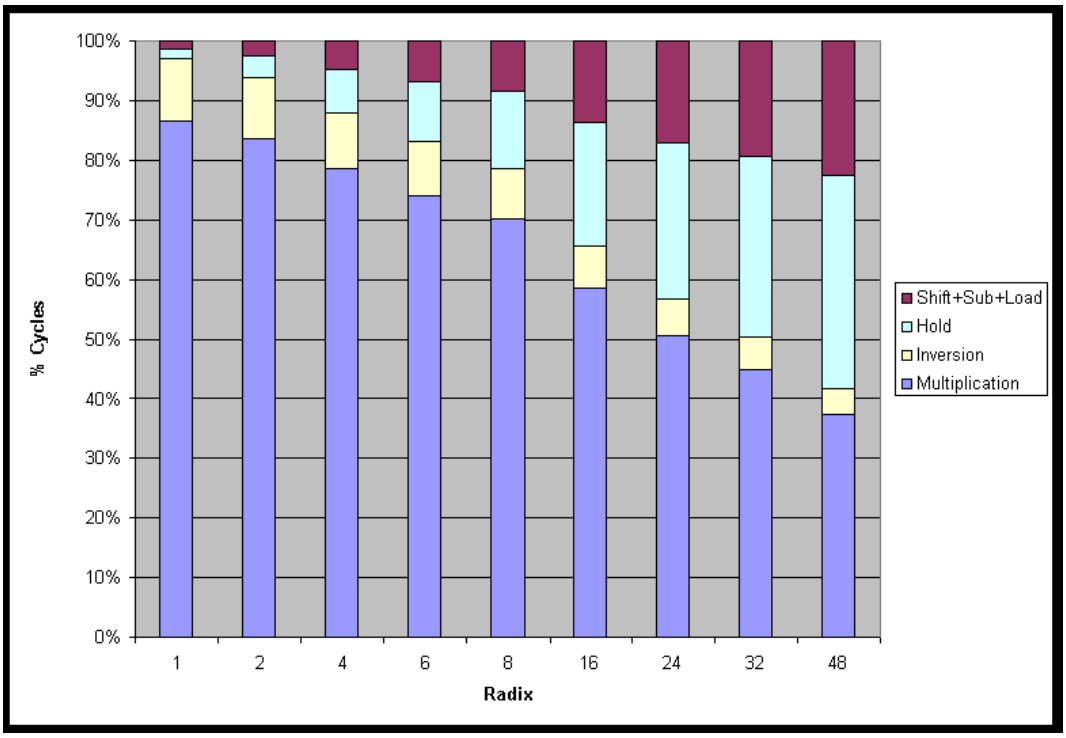

Figure 6.2: Share of various operations of a point multiplication over $GF(p_{192})$

Operation	Cycles HDL Simulation	Cycles Java Simulation	% Cycles
Inversion	13040	10940	119
Check Scalar	573	0	-
Copy Point	452	0	-
Preshift, Final Wait	1464	0	-
Other Overhead	191	0	-
Double+Add	114896	101395	113
Total	129400	112325	115

Table 6.1: Comparison HDL simulation vs. Java model ($k \cdot P$ over $GF(p_{192})$, 8-bit radix)

Radix	Cores	Clock Frequency	Area	Constant k	Average	Minimum
		MHz	% FPGA	kP/s	kP/s	kP/s
6	3	66	86	1235	1310	1000
8	1	66	40	500	530	430[1]
8	3	66	100	1485	1585	1210
16	1	66	60	735	780	600
24	1	66	87	865	910	705
4	4	92	92	1680	1763	1367
6	3	92	86	1700	1800	1385
8	1	88	40	660	700	540
8	*3*	*88*	*100*	*1960*	*2100*	*1600*
12	2	82.5	88	1560	1670	1270

Table 6.2: Point multiplication performance Spartan-3 1500 over $GF(p_{192})$

per measurement the benchmark was run a couple of times and outliers were not considered in the results. Another point to mind is that the maximum clock rate of the circuit synthesized placed and route by the Xilinx flow is very variable and unpredictable. Especially for the configurations which use nearly the complete area of the FPGA only changing the multiplier of the digital clock managers (DCM) can cause drops or rises of the maximum clock rate by tens of MHz. For instance while the maximum clock frequency for the 8-bit radix ECCP is 95 MHz when no DCM is used, it drops to 88 MHz when a DCM is added although this should not use more logic resources. Additionally this frequency was only reached by many trial synthesis, place and route runs with various parameters. If the DCM is just added without further measures the core even does not fit into the FGPA anymore because the synthesis tool is too generous and so uses too many resources. Because of this unpredictability of the maximum clock frequency a less extensive optimization was performed for the not optimal configurations. Therefore it can be expected that their clock rate could be a bit higher and thus they could perform a bit better.

The runtime of the double-and-add algorithm (see algorithm 3.4) used to compute the point multiplication in the ECCP is dependent on the scalar k. While it performs point doubling for every bit of k, point addition is only performed for bits of k that are set. To consider this two measurements were taken. The most significant for most applications is the one with a random scalar k averaged over 10000 multiplications. The worst case was measured by setting all bits of the scalar to one. This figure is especially interesting when countermeasures against timing side channel attacks have to be taken on a low level for example in the device driver. Then delaying the point multiplication result such that always the worst case time is reached impedes the attack. However, this wastes quite a bit performance therefore it is preferable to perform this time balancing on a high level, for example in the server application. The third measurement uses a constant scalar k with 113 bits set. Its only purpose is to allow comparison with the Java model results where this constant k is used to allow cycle accurate repetitions.

The results (see table 6.2) show that the ECCP can perform more than 2000 point multiplications per second on a mid-range FPGA. The first part of the table shows all configurations clocked at 66 MHz to allow comparison between them. The second part gives the results for the highest clocks obtained for various configurations. The maximum clock frequency decreases for higher radices,

[1] Estimated

Description	Sign	Verify	Keypair Generation
Spartan-3 1500, 3 cores, 8-bit radix, 88 MHz	0.74	0.82	0.72
Spartan-3 1500, 1 core, 24-bit radix, 66 MHz	1.61	1.83	1.53
Spartan-3 1500, 3 cores, 8-bit radix, 66 MHz	0.87	1.00	0.85
Spartan-3 1500, 1 core 8-bit radix, 66 MHz	2.40	2.76	2.34
Software, Pentium III, 866 MHz, 12-bit window	3.02	23.50	3.26
Software, Dual Pentium III, 1133 MHz, 12-bit window	1.22	9.80	1.4
Software, Dual Pentium III, 1133 MHz, no pre-computation	7.80	9.80	8.29
Software, Dual Pentium III, 1133 MHz, 15-bit window	0.93	9.80	-

Table 6.3: Java ECDSA performance (ms per operation)

Description	Sign	Verify	Keypair Generation
Spartan-3 1500, 3 cores, 8-bit radix, 88 MHz	1350	1220	1390
Spartan-3 1500, 1 core, 24-bit radix, 66 MHz	620	550	650
Spartan-3 1500, 3 cores, 8-bit radix, 66 MHz	1150	1000	1180
Spartan-3 1500, 1 core 8-bit radix, 66 MHz	420	360	430
Software, Pentium III, 866 MHz, 12-bit window	330	40	300
Software, Dual Pentium III, 1133 MHz, 12-bit window	820	100	710
Software, Dual Pentium III, 1133 MHz, no pre-computation	130	100	120
Software, Dual Pentium III, 1133 MHz, 15-bit window	1080	100	-

Table 6.4: Java ECDSA performance (operations per second)

which is mainly caused by the longer critical path in the radix multiplier adder tree. As expected by the area-cycle product of the Java model the radix 8 version is the most efficient configuration. As three 8-bit radix cores fit perfectly into the Spartan-3 1500 FPGA—100% of the slices are used—this configuration is even more favored because other choices of parameters which lead to a comparable area-cycle product waste resources because they do not fit as well into the FPGA. Clearly this can change when other FPGA models are targeted. Possibly then a configuration with a worse area-cycle product is the solution with the best total performance.

6.3 ECDSA Performance in Java with JCE

From an application point of view not the pure point multiplication performance is of main interest but the performance of a complete system. Therefore benchmarks where run using a Java ECDSA application with the hardware acceleration added to the IAIK JCE libarary[15]. Although some substantial performance loss occurs this allows a fair comparison with a pure software version and so gives an idea of the achievable benefit by using the hardware acceleration.

Three different measurements where taken. The first is the *key pair generation* which is merely a single point multiplication without much overhead. This mainly serves for estimating the performance overhead caused by the Java system. In a real system, for example for e-Government applications, the key pair generation will not have a significant share of the cryptographic operations.

The second task is the *signature generation*. The overhead is higher because operations have to

Description	Sign	Verify	Keypair Generation
Spartan-3 1500, 3 cores, 8-bit radix, 88 MHz	1350	2440	1390
Spartan-3 1500, 1 core, 24-bit radix, 66 MHz	620	1090	650
Spartan-3 1500, 3 cores, 8-bit radix, 66 MHz	1150	2000	1180
Spartan-3 1500, 1 core 8-bit radix, 66 MHz	420	730	430
Software, Pentium III, 866 MHz, 12-bit window	330	90	300
Software, Dual Pentium III, 1133 MHz, 12-bit window	820	200	710
Software, Dual Pentium III, 1133 MHz, no pre-computation	130	200	120
Software, Dual Pentium III, 1133 MHz, 15-bit window	1080	200	-

Table 6.5: Java ECDSA performance (point multiplications per second)

be performed which are no supported by hardware. This are mainly the hashing of the message with the SHA-1 algorithm and some modular operations over a not hardware supported prime field. This is the task where the pure software implementation is most competitive because the point multiplication performed uses a predefined base point and so it can benefit from the massive pre-computations, which have been carried out before. Furthermore the current ECCP system does not use the scalar length halving method by pre-computation (see section 3.5.1). Implementing this would only require to change the ECCP JCE library, and could give a significant speedup for signing. Depending on the application this task can have a large share of the total elliptic curve computations. However, in the typical target application the verification plays a much more prominent role, as a document is only signed only one time and this signature typically is verified a couple of times.

The last but most important task is the *signature verification*. The overhead is less compared with signing because two point multiplications have to be performed and so the additional operations have an smaller share on the complete execution time. This is the most advantageous task for the hardware accelerated version because it can perform the two point multiplications simultaneously. It is only slowed down a bit because of an additional point addition for pre-computation and because 50% more point additions are required throughout the multiplication. In addition the pure software version cannot make use of pre-computation because the second elliptic curve point is not predetermined and so loses much performance. In most applications a signature which is generated one time is verified more frequently. Therefore the property of the ECDSA that verification takes much more time than a signification is a significant disadvantage. The hardware acceleration can cope with this problem. The ECCP achieves a 12 times higher throughput than the software implementation, although it runs on a much slower system.

All benchmarks are performed by measuring the time for 10000 operations. Table 6.3 shows how many milliseconds one operation takes while table 6.4 shows how many operations can be performed per second. Table 6.5 gives the number of point multiplications performed per second. For the signature and key generation operation this is the same as in the last table, for verification the number is doubled because for each verification two point multiplications are required.

6.4 Comparision with Related Work

For performance comparison with other hardware architectures the point multiplication performance measured in section 6.2 is used. This is because the performance estimates of the other processors are also based on this level or even only on the HDL simulation results. Table 6.6 shows that the

Reference	$GF(p)$ field	Multiplier architecture	Target	Area	Freq. [MHz]	Cycles per kP	kP/s
This work	192 NIST	three 192x8	Spartan3-1500	13310 Slices[1]	88	125700	2100
This work	192 NIST	ten 192x16	Virtex4-LX200	70000 Slices	100	70000	14000
Satoh and Takano	192	64x64	$0.13um$ ASIC	118000 Gates	137.7	198288	694
Eberle et al.	224	64x64	ASIC[2]	?	1500	245330	6114
Eberle et al.	224	64x64	Virtex2-6000	?	66	245330	270[3]
Örs et al.	160	systolic array	VirtexE-1000	6055 Slices	91.3	1316160	69.3

Table 6.6: Comparison of ECC $GF(p)$ processors

processor performs very well in comparison with other implementations. It is s much faster than the reported FPGA results and competes very well with the ASIC implementations. Running on the high-end FPGA Virtex-4 estimates show that our solution performs about two times better than the fastest ASIC implementation. Furthermore this ASIC design is targeted on a very fast and therefore expensive technology and it is very questionable whether it is feasable to produce a such a hardware with a probably low production volume in such a technology. The ASIC by Satoh and Takano is much slower. Our processor is three times faster, even if it is implemented on the slower and smaller Spartan-3 FPGA. The advantage is even higher if only FPGA solutions are compared. Then the FPGA prototype for the ASIC by Eberle is the fastest alternative, which uses an FPGA which is roughly two times larger than the Spartan-3 1500 and three times smaller than the Virtex-4 LX200. Our processor is if running on the Spartan already over seven times faster. For ECDSA the performance lead of our work can be expected to be even higher, because the other processors do not support simultaneous multiplications of two points which allows a significant acceleration of signature verification.

The performance advantage of our work is mainly achieved by the restriction of only supporting the essential fields. Most other implementations focus on supporting arbitrary fields, which can be useful when they are used for small client devices. However, for server applications, which are targeted in this work, the benefit is arguable because usually virtually all ECC-operations to perform will be over NIST-prime-fields. Thus the few others can be performed by the main processor. The performance loss for this can be safely expected to be much less than the performance loss of the hardware caused by the requirement to support arbitrary prime fields.

[1] Includes area used by the Opencores PCI-bridge. (about 800 slices)
[2] Current processor technology which allows 1.5GHz when architecture is fully pipelined
[3] Author's estimate based on reference

Chapter 7

Conclusion and Outlook

7.1 Summary

Many processors for elliptic curve cryptography are reported in literature. Frequently novel architectures allow very high flexibility by supporting a large number of fields with arbitrary parameters. Some even include support for other crypto-systems like RSA. Clearly, for small devices which are not able to perform cryptographic computations in reasonable time the approach to support a wide range of cryptographic algorithms makes sense. However, many designs target server systems and usually no reasons are given while this broad support is advantageous. After all a modern server system possesses one or more fast processors and so is able to compute all crypto-system operations easily. Therefore a server side cryptography acceleration should not only perform well but it should outperform the normal software implementations. Both high performance and wide support of various crypto-systems and fields are hardly achievable at the same time, at least under the constraint that the system costs should stay in reasonable limits.

This work uses a different approach. It proposes a crypto-processor optimized for a specific application, namely the Austrian e-Government project Bürgerkarte. This system—and many others in this field—use a specific crypto-system and a specific finite field, in fact an elliptic curve over a 192-bit field with a NIST reduction prime. Especially the optimization for the special reduction prime gains much performance because this nearly halves the area of the algorithmic unit in comparison to other architectures commonly used for reduction with arbitrary primes. In the targeted server system most of the cryptographic tasks can be expected to be of the supported type and the hardware acceleration can take place. The few requests for other types can be computed by the software without a real impact on the total performance because of their small share. In the future larger fields will become more widely used to hold the level of security. Then the FPGA can be easily reconfigured with a version of the processor that is parameterized for this new common field.

The design objective to build are real useable system is complied with the integration of the hardware support into the Java JCE library. This allows an easy integration in existing applications and also simplifies the use in newly created systems. In fact only a single line of code has to be added to a Java program using the JCE library and a JAR file created in this work has to be included in the class path when executing the program. This enables the use of the hardware acceleration and also allows transparent switching to the software implementations for not supported fields.

To satisfy the constraint to keep costs within limits the processor is targeted on a PCI FPGA board, although producing an ASIC in standard cell technology would be possible and test place and route runs were performed in this work. Implemented on a PCI board, which is equipped with a Xilinx Spartan 3-1500—a midrange FPGA—2100 elliptic curve point multiplications per second are achieved.

This is in the range of software implementations on high-end systems when pre-computation can be used. This is not the case for signature verification which is probably the most important task in a typical server system. Then the pure software solution only achieves one tenth of the ECCP performance. In addition the processor supports simultaneous point multiplication of two points which improves verification performance even more. When more performance is required a high end FPGA can be used, on a Virtex-4 LX200 about 14000 point multiplications are achievable. Then the fastest ASIC solutions known to the author are outperformed, which do have significant higher unit costs for the expectable quantity to be produced. The mid-range implementation still is able to beat the other FPGA implementations known to the author by far in terms of performance.

7.2 Future Work

The most obvious feature which can be added is the support for more fields than the 192-bit field. The most important part is already done as the Java RTL level model supports all recommended NIST primes up to 521 bit. This part is not trivial because the 192-bit prime reduction is a special case where only additions are used. For most of the larger fields this is not true, here subtraction is used too. The reduction was designed to always return positive numbers because the use of negative number would complicate the complete architecture significantly. To add support for this larger fields only the reduction already available on RTL-level in Java must be implemented in VHDL, which should be possible in less than a day. Support for more fields directly leads to the fact that runtime switching between two fields could be useful. This can be either be done on driver level by putting cores parameterized for different fields into the chip or on the core level itself by supporting more than one field on each core by multiplexing multiple reduction units. The latter is more efficient in terms of scheduling but uses more hardware, probably reduces maximum clock rate and is more complicated.

Not implemented in the hardware is the hybrid-redundant-binary adder reduction. On the one hand because of the lack of time on the other hand the optimum parameterization on the target hardware uses a quite small radix of 8 which reduces the influence on performance of this measure. For other hardware targets, or for applications where response time is of more importance than in this work, where throughput is the only performance criteria, the hybrid adder concept probably is advantageous. The basic idea of the concept is that FPGAs perform ripple carry addition quite efficiently. While long ripple carry adders are slow, computing chunks of for example four bits could allow faster redundant-to-binary conversion which is for higher radices the dominant part of the multiplication. Wöckinger [35] showed that using this concept for all adders in the architecture is not useful because the impact on the clock frequency is too high. However, a modification of this concept by using the 4-bit ripple adders only in the feedback loop, is very promising because in particular for high radices this path is shorter than the radix multiplier path and so no slowdown is expected, while the advantage of reducing the number of cycles for the redundant-to-binary conversion still is striking. Thus a better scalability for higher radices should be achieved.

An optimization can be done in the simultaneous point multiplication. There is a pre-computation required to calculate $P + Q$ which then is used in the algorithm. The functions to perform a point addition are already offered by the hardware, respectively by the micro code. However, due to time constraints in implementing the hardware, the control does not support this calculation. For JCE applications this is not a significant disadvantage because functions are supplied to compute this point addition easily, and the verification benchmarks suggest that the performance loss is not very pronounced. However, the integration of the addition in the hardware, which should be possible in a day, would relieve the software from all elliptic curve group operations. While this is—apart of the small estimated performance gain—just a matter of elegance for JCE application it is quite relevant for other possible implementations, because then they could do without any algorithms for elliptic

curve arithmetic.

Also on the JCE level optimizations could be performed. Namely the bus transfer is due to restrictions in the JCE library and the BigInteger class quite inefficient. Although at least parts of the implementation represent numbers in integer chunks they do only allow access on a per byte basis. This leads to unnecessary copying activity and to inefficient bus transfers because in addition the byte order is reversed in comparison to the PCI bus byte order and only single bytes can be transfered at each bus cycle. It is questionable whether a significant acceleration can be achieved, because the share of time needed for bus transfers is quite low. Nevertheless an optimization of the bus transfers is desirable.

Probably a useful feature would be the integration $GF(2^m)$-field support into the processor. Many works, for example [13], propose dual field multipliers which work over both prime fields and binary polynomial fields without using much more resources. The approach to use a reduction for a specific polynomial is common in many $GF(2^m)$ hardware solutions—for instance the server side processor proposed in [33]—and so fit well with the $GF(p)$ processor described in this work which uses a reduction for a specific prime. However, it should be verified whether and what applications exists for such a dual-field server side acceleration.

Another interesting work would be the verification of the processor in a real e-Government project. While the integration should be easy when the JCE library is used, the principal task is probably the design of realistic use scenarios. This allows significant performance results with could give estimates whether the share of cryptographic operations is high enough that the system can benefit significantly from the hardware acceleration or whether other operations like XML-parsing are more prevalent which would cut the advantage of the hardware solution.

Appendix A

Abbreviations

AMD Advanced Micro Devices

BRAM Block RAM

CLB Configurable Logic Blocks

CMOS Complementary Metal Oxide Semiconductor

CPU Central Processing Unit

CSA Carry Save Adder

DCM Digital Clock Manager

DRC Design Rule Checking

ECC Elliptic Curve Cryptography

ECCP Elliptic Curve Cryptography Processor

ECDSA Elliptic Curve Digital Signature Algorithm

FPGA Field Programmable Gate Array

GF Galois Field

GCHQ Government Communications Headquarters

HDL Hardware Description Language

ICR Interrupt Control Register

ISR Interrupt Status Register

JCE Java Cryptography Extension

LSB Least Significant Bit

LUT Lookup Table

LVS Layout Versus Schematic

MSB Most Significant Bit

PCI Peripheral Component Interconnect

RAM Random Access Memory

RSA Rivest, Shamir, and Adleman (public key encryption technology)

VHDL Very High-Speed Hardware Description Language

VLSI Very Large Scale Integration

WB Wishbone Bus

XST Xilinx Synthesis Tool

Bibliography

[1] (2005). *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*. IEEE Computer Society. 77

[2] A-SIT Zentrum für sichere Informationstechnologie - Austria (-). *Bürgerkarte*. `http://www.buergerkarte.at/`. 1

[3] Andrews, K. (2004). *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*, Graz University of Technology, Austria. `ftp://ftp2.iicm.edu/pub/keith/thesis/thesis.zip`. ii

[4] ANSI (1993). *X9.30-1993, Part 2. Public key cryptography using irreversible algorithms for the financial services industry: The Secure Hash Algorithm (SHA-1) (Revised)*. 25

[5] ANSI (1995). *X9.30-1995, Part 1. Public key cryptography using irreversible algorithms for the financial services industry: The Digital Signature Algorithm (Revised)*. 22

[6] ANSI (1998). *X9.62-1998. Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm*. 6, 23

[7] AVNET (-). *Spartan-3 1500 Evaluation Kit*. `http://www.em.avnet.com/`. 32, 43, 48

[8] Crowe, F., Daly, A. and Marnane, W. P. (2005). *A Scalable Dual Mode Arithmetic Unit for Public Key Cryptosystems*. In *ITCC (1)* DBL [1], pages 568–573. 3

[9] Desmedt, Y., editor (1994). *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*. Springer. 78

[10] Diffie, W. and Hellman, M. E. (1976). *New Directions in Cryptography*. IEEE Transactions on Information Theory, IT-22(6):644–654. 6

[11] Eberle, H., Gura, N., Shantz, S. C., Gupta, V., Rarick, L. and Sundaram, S. (2004). *A Public-Key Cryptographic Processor for RSA and ECC*. In ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference on (ASAP'04), pages 98–110, Washington, DC, USA (2004). IEEE Computer Society. 3, 71

[12] Elgamal, T. (1985). *A public key cryptosystem and a signature scheme based on discrete logarithms*. IEEE Transactions on Information Theory, 31(4):469–472. 6, 22

[13] Großschädl, J. and Savas, E. (2004). *Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m)*. In Joye and Quisquater [20], pages 133–147. 74

[14] Hankerson, D., Menezes, A. and Vanstone, S. (2004). *Guide To Elliptic Curve Cryptography*. Springer Professional Computing. Springer. vi, vii, 1, 7, 16, 17, 20, 23, 25, 26

[15] IAIK (-). *IAIK Elliptic Curve Crytography Library*. `http://jce.iaik.tugraz.at/products/17_ecc/index.php`. 64, 69

[16] IEEE (2000). *Technical Report IEEE Std 1363-2000. IEEE Standard Specifications for Public-Key Cryptography.* vii, 12

[17] IEEE (2001). *Standard Verilog Hardware Description Language*. Revision of IEEE Standard 1364-1995,2001. 34

[18] IEEE (2002). *IEEE Standard VHDL Language Reference Manual*. 34, 37

[19] Joan Daemen, V. R. (2001). *The Design of Rijndael. AES - The Advanced Encryption Standard*. Springer. ISBN: 3540425802. 6

[20] Joye, M. and Quisquater, J.-J., editors (2004). *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*. Springer. 77

[21] Koblitz, N. (1987). *Elliptic curve cryptosystems*. In Mathematics of Computation, 48, pages 203–209. 8

[22] Lim, C. H. and Lee, P. J. (1994). *More Flexible Exponentiation with Precomputation.* In Desmedt [9], pages 95–107. 13

[23] Miller, V. S. (1986). *Use of elliptic curves in cryptography*. In Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85, pages 417–426, New York, NY, USA (1986). Springer-Verlag New York, Inc. 8

[24] Montgomery, P. L. (1985). *Modular Multiplication Without Trial Division.* Mathematics of Computation, 44(170):519–521. 16

[25] Opencores.org (2002). *Specification: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. Revision: B3*. `http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf`. v, 61

[26] Opencores.org (2003). *Opencores PCI Bridge*. `http://www.opencores.org/projects.cgi/web/pci/home`. 42

[27] Orlando, G. and Paar, C. (2001). *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. 3, 27

[28] Pollard, J. M. (1978). *Monte Carlo methods for index computation mod p*. Mathematics of Computation, 32:918–924. 8, 25

[29] Pollard, J. M. (1988). *Factoring with cubic integers*. unpublished. 25

[30] Rivest, R. L., Shamir, A. and Adelman, L. M. (1977). A METHOD FOR OBTAINING DIGITAL SIGNATURES AND PUBLIC-KEY CRYPTOSYSTEMS. Technical Report MIT/LCS/TM-82. 6, 7

[31] Satoh, A. and Takano, K. (2003). *A Scalable Dual-Field Elliptic Curve Cryptographic Processor*. IEEE Trans. Comput., 52(4):449–460. 3, 71

[32] Virtual Silicon (2003). *0.13um High Density Standard Cell Library*. For 0.13um UMC process. 45

[33] Wolkerstorfer, J. (2004). *Hardware Aspects of Elliptic Curve Cryptography*. PhD dissertation, Graz University of Technology, Austria, Institute for Applied Information Processing and Communications. vi, 10, 18, 30, 35, 57, 74

[34] Wolkerstorfer, J. and Bauer, W. (2002). *A PCI-Card for Accelerating Elliptic Curve Cryptography*. In Proceedings of Austrochip 2002. `http://www.iaik.tu-graz.ac.at/research/publications/2002/ACHIP2002-ECC.htm`. 3

[35] Wöckinger, T. (2005). High-Speed RSA Implementation for FPGA Platforms. Master's project, Graz University of Technology, Austria, Institute for Applied Information Processing and Communications. 1, 51, 73

[36] Xilinx (2003). *Development System Reference Guide*. v, 44

[37] Xilinx (2003). *Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs*. `http://www.xilinx.com/bvdocs/appnotes/xapp462.pdf`. 43

[38] Xilinx (2003). *XST User Guide*. `http://toolbox.xilinx.com/docsan/xilinx6/books/docs/xst/xst.pdf`. 43

[39] Xilinx (2005). *Spartan-3 FPGA Family: Complete Data Sheet*. `http://direct.xilinx.com/bvdocs/publications/ds099.pdf`. v, 41, 42

[40] Xilinx (2005). *Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs*. `http://www.xilinx.com/bvdocs/appnotes/xapp464.pdf`. 42

[41] Örs, S. B., Batina, L., Prenel, B. and Vandewalle, J. (2005). *Hardware Implementation of an Elliptic Curve Processor over GF(p) with Montgomery Modular Multiplier*. International Journal of Embedded Systems (IJES), page 15. 4, 71