Efficient Hardware Architectures for Solving the Discrete Logarithm Problem on Elliptic Curves

Tim Erhan Güneysu

2006-01-31

Diplomarbeit
Ruhr-Universität Bochum

Chair for Communication Security
Prof. Dr.-Ing. Christof Paar

Erklärung

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Ort, Datum _____

Abstract

The utilization of Elliptic Curves (EC) in cryptography is very promising due to their resistance against powerful index-calculus attacks. Since their invention in the mid 1980s, Elliptic Curve Cryptosystems (ECC) have become an alternative to common Public Key (PK) cryptosystems such as RSA. With a significantly smaller bit size, ECC provides similar security than other PK systems (e.g. RSA).

The effort of breaking a cryptosystem mainly defines its security. Hence, a "secure" cryptosystem will most likely not be broken within the next decades even if we take technological progress into account. As a consequence, conventional attacks based on software implementations of cryptanalytical algorithms will most probably never succeed in breaking actual ciphers. It is widely accepted, that the only feasible way to attack such cryptosystems is the application of dedicated hardware.

In times of improved hardware manufacturing and increasing computational power, the issue arises how secure the small key lengths of ECC are, facing a massively parallel attack based on special-purpose hardware.

This is the first work presenting an architecture and an FPGA implementation of an attack on ECC. We present an FPGA based multi-processing hardware architecture for the Pollard-Rho method for EC over GF(p) which is, to our current knowledge, believed to be the most efficient attack against ECC. The implementation is running on a conventional low-cost FPGA as it can be found, e.g., in the parallel code breaker machine COPACOBANA. The latter provides a parallel cluster of FPGAs, providing a large quantity of computational power [KPP+06]. Thus, fairly accurate estimates about the cost of an FPGA-based attack can be given.

Furthermore, we will project the results on actual ECC key lengths (e.g. $k = 160$ bit) and estimate the expected runtimes for a successful attack. Since FPGA-based attacks are out of reach for such key lengths, we present estimates for an ASIC design. As a result, ECC over GF(p) and bit sizes of $k > 160$ can be considered to be infeasible to break with current algorithms as well as computational and financial resources.

Contents

1. Introduction

1.1. Motivation

In ancient times, the task of ensuring communication security was not an issue: With no noticeable communication except direct talk, there was only a negligible probability of being eavesdropped without being noticed. In the last centuries more and more communication paths have been created to satisfy the people's desire for interaction. With the years, those channels become faster, wider and more accessible for everyone. Nowadays, an enormous amount of data is flooding the communication lines like the wires of the Internet each day, revealing more information about individuals than appreciated.
This new situation demands a greater use of information and communication security. Protecting data in this context means to encrypt sensible information in a way that eavesdroppers should not be capable to recover its plaintext. It is important to remark that the security of all known and practical cryptosystems is based on computational assumptions. Thus, it is not impossible to reveal the secret of encrypted information for an intruder without the secret key. The ciphertext is rather encrypted in a way that is infeasible to break it with current cyptanalytical algorithms and computational resources. Although this strategy is common practice, it is not bullet-proof leaving an exiguous option for an attacker to derive the secret with yet unconsidered methods. Due to these technical improvements and the ongoing increase of computational power, it becomes more and more important to secure our sensible data with effective cryptosystems from being tapped by unauthorized audience.
In the mid-seventies, public-key (PK) cryptography was proposed by Diffie and Hellman [DH76] for public use, introducing a new era of cryptography. Unfortunately, with better and faster cryptanalytical attacks, growing security parameters are required to preserve the desired data protection standard [LV01]. The field of PK cryptography was expanded by Miller and Koblitz in 1985 independently proposing the Elliptic Curve Cryptography (ECC). Elliptic Curves augment established PK algorithms and provide the advantage that powerful index calculus attacks cannot be applied. This results in significantly smaller security parameter, providing an equivalent protection compared to factoring-based and classical discrete logarithm techniques for PK cryptography.
In this thesis, we want to explore the most efficient attacks on ECC which are currently known with respect to special-purpose hardware. Further, to limit the

scope of this work, we will only regard the class of Elliptic Curves over GF(p) which have been less examined in context of hardware aspects so far.
To current knowledge, the best method for attacking Elliptic Curves (EC) is a multi-processing derivative of the Pollard-Rho method [Pol78]. This technique was published by Wiener and Oorschot in 1999 and includes a rough and quite optimistic estimation for a hardware machine capable to break a curve over GF(2^{155}). Inspired by this statement, this work covers the following aspects:

1. We propose and implement a hardware design for EC over GF(p).
2. We build a hardware architecture capable to mount an efficient attack against EC over GF(p). This involves the creation of external software components required for administration and management.
3. Upon first results, we estimate the expected runtime to break an EC over GF(p) complying to relevant a recent security parameters.
4. We give estimations for breaking ECC Challenges [Cer06] using our special-purpose hardware.

To our knowledge, items 2-4 have not been discussed earlier in the open literature.

1.2. Thesis Outline

We will commence this thesis with a brief review about related publications and projects in Chapter 2. This includes work concerning ECC operations in hardware as well as the topic of efficiently performed attacks. We will restrict ourselves to publications with relevance to ECs over GF(p) at this point.
Chapter 3 covers relevant details about the mathematical background for the work with ECC. This part also intends to provide a very brief survey of known, and published attacks. Because we will focus on the security of general ECC, we will not provide a detailed description of attacks on "weaker" curves with special characteristics. Furthermore, we will mention and explain the currently most efficient way to break ECC which forms the basis for hardware implementations. The next chapter provides a general description of mounting an attack against ECC over GF(p). This includes the discussion and choice of relevant parameters as well as the construction of a reference model for a software implementation. The findings of this chapter will be used for conceptual purposes as well as for a performance comparison with respect to a hardware based design in Chapter 6. In Chapter 5 we describe the actual hardware realization. We will present an FPGA based platform capable to perform massively parallel computations. In other words, we will translate the general model from Chapter 4 into transistor based logic.

We will analyze our results in Chapter 6 and compare the time and resource consumption of our software and hardware implementations. With those results, we are able to give an estimate of expected runtimes for attacks against recent EC over GF(p) and ECC challenges. Based on this discussion, we will provide a security analysis in respect of financial considerations. In Chapter 7, we finally provide a security assessment of ECC with respect to hardware based attacks and recommend some options for future research.
In the Appendix, we will present the implementation of a serial communication channel for data exchange between an FPGA and a host computer, mostly used for testing purposes. Furthermore, the operation of the arithmetic controller is highlighted in detail. In addition, the reader will find a summary about used notation and symbols as well as abbreviations and signal naming conventions. Furthermore, some practical insights are given by the status report of an ECC reference attack.

2. Previous Work

In the following sections, we will briefly summarize published work with relevance to this thesis. On the one hand, this includes hardware implementations of EC over prime fields, which are the basic target of analysis. On the other hard, we will highlight publications and projects with respect to attacks on ECC in hardware. This treatment is limited to prime field ECC.
Koblitz [Kob87] and Miller [Mil85] have independently proposed to use the group of points on an elliptic curve defined over a finite field as a new paradigm in PK cryptography. There have been several books studying ECC in further detail. To mention only three, the interested reader is recommended to take notice of [Men93] by Menezes (which is perhaps somewhat dated by now), [BSS99] by Blake, Seroussi and Smart as well as [HMV04] by Hankerson, Menezes and Vanstone.

2.1. Hardware for ECC over Prime Fields $\mathbb{F}_p$

With respect to hardware operation, the work of Orlando and Paar suggests a first design for a scalable GF(p) elliptic curve processor in programmable hardware [OP01]. They implemented an arithmetic unit on an FPGA capable to perform field additions and semi-systolic multiplications required for projective EC computations. The authors reported to run their system at 40 MHz, requiring 11,416 LUTs, 5,735 Flip-Flops, and 35 Block-RAMS for computations on the EC over GF($2^{192} - 2^{64} - 1$). Table 2.1 shows their assumed latency times for a point operation (for a given bit size $k = \lceil \log_2(p) \rceil$), excluding any overhead and processing costs for additions. This first approach has been picked up by [OBPV03]: Their

Operation	**Required Cycles**	**Required Time**
EC point doubling	$15.5k$	62 μs
EC point addition	$15.5k$	62 μs

Table 2.1.: GF(p) arithmetic unit performance on a Virtex-E XCV1000E FPGA at 40 MHz [OP01]

architecture is based on a five-layered design using a Montgomery representation for a fast modulus reduction. Furthermore, the architecture prefers computations in projective coordinates but offers an interface for feeding in affine points. Table 2.2 provides the performance results of their work for bit size $k = 160$: The

Operation	**Required Cycles**	**Required Time**
EC point doubling	$40k + 38$	70 μs
EC point addition	$42k + 56$	74 μs

Table 2.2.: GF(p) arithmetic unit performance on a Virtex-E XCV1000E FPGA at 91.308 MHz [OBPV03]

authors reported their architecture running for $k = 160$ on a Virtex-E XCV1000 FPGA at a maximum clock speed of 91.308MHz and a similar area consumption as in [OP01].
In contrast to the presented architectures, Daly, Marnane, Kerins, and Popovici [DMKP04] proposed an architecture relying only on affine computations. This demands the necessity of an additional inverter circuit but reduces the overall latency for a single point computation. The authors stated for a bit length of $k = 160$ on a Xilinx Virtex-2 XC2V2000 FPGA an area occupation of 1854 slices and a maximum clock speed of 40.28MHz. The poor performance with respect to a reduced maximum frequency is compensated by the low runtime, depicted in Table 2.3.

Operation	**Required Cycles**	**Required Time**
EC point doubling	$6k + 12$	24.30 μs
EC point addition	$5k + 9$	20.23 μs

Table 2.3.: GF(p) arithmetic unit performance on a Virtex-2 XC2V2000 FPGA at 40.28 MHz [DMKP04]

Besides the presented architectures, it might be useful to review basic hardware realization of arithmetic functions. For an extensive coverage of modular addition and multiplication, the interested reader is referred to [Koc95].

2.2. Hardware Attacks on ECC

To our knowledge, hardware based attacks on elliptic curve cryptosystems have not been implemented up to now. Proposals for hardware based attacks are very rare and those which exist do not apply to the scope of this work. In contrast to curves in GF(2^m), curves over GF(p) have not been examined. The most important work in this field is provided by Wiener and van Oorschot [vOW99]. It provides a rough estimate for building a hardware platform to break a curve over GF(2^{155}). However, some important information on specific details of the algorithm have been omitted.
To our knowledge, no further attempts have been taken to address the issue of breaking a GF(p) based ECC with special purpose hardware.

3. Elementary Concepts and Mathematical Background

In 1985, Neal Koblitz [Kob87] and Victor Miller [Mil85] proposed elliptic curves (EC) as a new design for public key cryptosystems. Although being mathematically elegant, ECs present more complexity with respect to implementation than conventional PK cryptography such as RSA [RSA77]. But due to the fact that more sub-exponential attacks do not apply to elliptic curves, they can be used with noticeably smaller bit lengths with respect to conventional PK methods.

3.1. Introduction to Elliptic Curve Cryptography

ECs themselves are an artificial definition based on a mathematical ellipsoid equation of third degree. A tuple of elements taken from a finite field $\mathbb{F}_p = GF(p)$ satisfying the equation is called "point" on that curve. To build a group from this curve for enabling real computations on this structure, a closed group operation must be defined. This group operation is called point addition.
Before proceeding with any mathematical formulas, there should be some short notice concerning our notation. During this thesis, we should agree upon the following symbols for simplicity and consistency (see as well the overview in Appendix A.4):

- An **elliptic curve** $E(\mathbb{F}_p)$ is defined over prime fields in all concerns of this thesis, hence p is prime and $p > 3$. A parameter $k = \lceil log_2(p) \rceil$ specifies the size in bits of p.
- The **order of the curve** m is given by $m = \mathrm{ord}(E(\mathbb{F}_p))$.
- A **base point** $P \in E(\mathbb{F}_p)$ with order n is defined by $n = \mathrm{ord}(P)$.
- A **further point** $Q \in \langle P \rangle$ provided by an unknown dependency $\ell \in \{2, \ldots, n-1\}$ related to P by $Q = \ell P$.

Commonly, an EC over $\mathbb{F}_p$ with characteristic $p \neq 2, 3$ is defined using a simplified Weierstrass equation in affine coordinates[HMV04]:

$$E : y^2 = x^3 + ax + b \ (a, b \in \mathbb{F}_p, 4a^3 + 27b^2 \neq 0) \tag{3.1}$$

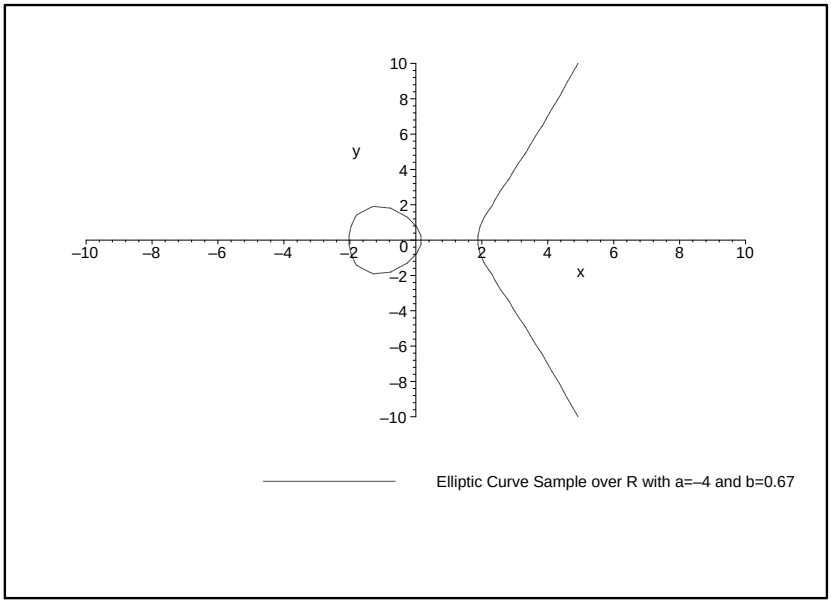

Figure 3.1.: Elliptic curve $y^2 = x^3 - 4x + 0.67$ over $\mathbb{R}$

The property of security of an EC relies directly on its underlying finite field. For ECs over $\mathbb{R}$ we have a good visual understanding (cf. Figure 3.1) and this intuitive way of comprehension makes them unusable for a cryptographic context. Consequently, two main classes of finite fields have been identified which are more appropriate for our purposes. These are the prime GF(p) fields with a large, prime value p and the binary extension fields GF(2^m). During this work, we will focus on the analysis of prime fields only.

Using ECs for cryptographic applications, a mathematical primitive needs to be identified. This primitive is a characteristic providing a one-way-feature, i.e. easy to compute when a required secret is known but hard to invert without that knowledge. Such a problem for EC can be built upon the Elliptic Curve Discrete Logarithm Problem.

3.2. The Elliptic Curve Discrete Logarithm Problem

The Elliptic Curve Discrete Logarithm Problem (ECDLP) is basically an extension to the basic Discrete Logarithm Problem (DLP) in modular groups. The latter problem is well known, e.g., in protocols based on Diffie-Hellmann key exchange [DH76] and ElGamal encryption [ElG85] schemes. The only change from DLP to ECDLP is harbored in the choice of the underlying group and corresponding elements. Both definitions will be presented enabling a direct comparison.

- **DLP for Prime Fields**
  Let p be a prime with $p > 3$ and $\mathbb{F}_p = GF(p)$ the Galois Field over p.

Given two elements $g, h \in \mathbb{F}_p$, where g is generator of a sufficiently large subgroup $\langle g \rangle$, find the integer ℓ such that $\ell \cdot g = h$ holds. The parameter ℓ is often denoted as discrete logarithm $\ell = \log_g(h)$.

- **ECDLP for Prime Fields**
  Let p a prime with $p > 3$ and $\mathbb{F}_p = GF(p)$ the Galois Field over p. Given the elliptic curve E over the finite field $\mathbb{F}_p$ and two points P and Q with $P \in E(\mathbb{F}_p)$ and $\langle P \rangle$ a sufficiently large subgroup, find the integer ℓ such that $\ell \cdot P = Q$ wher $Q \in \langle P \rangle$ holds. Here, the parameter ℓ is sometimes denoted as elliptic curve discrete logarithm $\ell = \log_p(Q)$

Obviously, aside from different groups and associated operations, both definitions are basically the same. The differences are indeed that DLP uses elements directly from the finite field whereas ECDLP defines a intermediate mathematical layer based on ECs. Hereby, the definition of DLP respective finite fields encloses the application of two group operations - addition and multiplication. In the domain of Elliptic Curve Cryptography (ECC) only one operation is supported - the point addition. This is definitely an (great) advantage and a (small) disadvantage at same time. On the one hand, the operation for multiplying points, which is a basic function in cryptography, must be simulated by intelligent concatenation of several point additions. On the other, the lack of multiplication capabilities and the associated additional group structure makes some subexponential attacks infeasible. The index-calculus methods are based on multiplicative characteristics of finite fields which are not available in the context of elliptic curves. Therefore, currently only fully exponential algorithms remain as candidates for attacking the security of ECC.

3.3. Known Algorithms to Attack Elliptic Curves

As briefly mentioned, all recently known attacks on ECC have fully exponential complexity [HMV04]. This statement is true for general ECs and excludes attacks on special subclasses like supersingular and anomalous curves. This thesis intends to analyze the practical security of ECC in general and, thus, will not take curves with weak cryptographic properties into account. The interested reader can find additional information on particular attacks of certain curves in Section 3.4.
The cryptographic primitive ECDLP, respectively the resolution of parameter ℓ, can be solved using the following techniques:

3.3.1. Naïve Exhaustive Search

This method sequentially adds the point $P \in E(\mathbb{F}_p)$ to itself. The addition chain $P, 2P, 3P, 4P, \ldots$ will eventually reach Q and then ℓ with $\ell \cdot P = Q$ is obtained. In

worst case, this computation can take up to n steps where $n = ord(P)$, making this attack infeasible for practice when n is large.

3.3.2. Baby Step Giant Step

The Baby Step Giant Step algorithm as introduced 1971 by D. Shanks [Sha71] is an improvement to the naïve approach as introduced above. It takes advantage of ideal class numbers of quadratic fields in two different steps which need to be combined afterwards. For $n = \mathrm{ord}(P)$, temporary memory for about $\sqrt{n}$ points and roughly additional $\sqrt{n}$ computational steps are required. Due to the fact that the Baby Step Giant Step is known not to be the most efficient algorithm when attacking ECC, it will not play a greater role in further discussion.

3.3.3. Single-Processor Pollard-Rho (SPPR)

The Pollard-Rho attack as proposed by J. Pollard in 1978 [Pol78] is a collision based algorithm based on a random walk in the point domain of an EC. Although it has similar computational complexity compared to the Baby Step Giant Step algorithm of about $\sqrt{\pi n/2}$, it is superior due to its negligible memory requirements. In combination with the employment of parallel processing, the Pollard-Rho is the fastest known attack [HMV04] against ECC. Thus, the parallel version of this algorithm will be the main focus of this thesis and will be discussed in Section 3.3.4.

To explain the Pollard-Rho in more detail, it should be explained why point collisions can help to reveal the ECDLP. Let $R_1 = c_1P + d_1Q$ and $R_2 = c_2P + d_2Q$ be two points with $R_1, R_2 \in E(\mathbb{F}_p)$ and $R_1 = R_2$ but $c_1 \neq c_2$ and $d_1 \neq d_2$. Then the following statements hold [HMV04]:

$$\begin{aligned} R_1 &= R_2 \\ c_1P + d_1Q &= c_2P + d_2Q \\ (c_1 - c_2)P &= (d_2 - d_1)Q \\ (c_1 - c_2)P &= (d_2 - d_1)\ell P \\ (c_1 - c_2) &= (d_2 - d_1)\ell \bmod n \\ \ell &= (c_1 - c_2)(d_2 - d_1)^{-1} \bmod n \end{aligned} \tag{3.2}$$

Hence, it is obvious that in case of a point collision in the subgroup of P the ECDLP can be solved efficiently. Next is the issue of how to find such a collision. The simplest approach would be to take a starting point $S = c_sP + d_sQ$ with $c_s, d_s \in_R \{2, \ldots, n\}$ chosen randomly. A second point $T_1 = c_{t1}P + d_{t1}Q$ with other randomly chosen coefficients is used to compute $R_1 = S + T_1$. Then a third random point T_2 determined the same way will lead to a further point $R_2 = S + T_2$

which is compared against previous results. This procedure can be continued until a correspondence of points is located with the drawback that all results (about $\sqrt{n}$ due to the birthday paradox) need to be stored. The enormous space requirements would make this attack similarly costly as the Baby Step Giant Step algorithm.

The better solution is to have of a random walk [Pol78] within the group. This is a pseudo-random function determining a collision candidate using an addition chain with a finite set of pre-initialized random points. In other words, we have a function f taking a current point X_j of the EC as input and computes its successor X_{j+1} by simply adding another point. A repetition of this procedure produces a pseudo-random trail of points in the domain of $\langle P \rangle$. The other point, which is added each time, is determined from a set $\mathcal{R}$ of previously randomly chosen points and is selected in each iteration by a partitioning function. Let $R_i \in \mathcal{R}$ be the i-th out of s total random points with $i = \{0, \ldots, s-1\}$. Then, we can define a partitioning function g which determines the next random point R_i to add:

$$g : E(\mathbb{F}_p) \rightarrow \{0, \ldots, s-1\} \ : \ X \mapsto i.$$

When we integrate g into the function f, we obtain a next point X_{j+1} by:

$$f : E(\mathbb{F}_p) \rightarrow E(\mathbb{F}_p) \mid X_{j+1} := X_j + R_{g(X_j)}.$$

Due to the finiteness of $\mathcal{R}$, the trail generated by a repetitive utilization of f will always run into a cycle and therefore eventually collide in some point. The shape depicted by the random walk is similar to the greek letter ρ and hereby eponym for this algorithm. The collision itself can easily detected using Floyd's cycle finding algorithm which requires only a second computation advancing twice as fast as the first one [MvOV96]. Hence, except for two computations, no additional storage is required. Figure 3.2 depicts the construction of the trail respectively a random walk in the Pollard-Rho algorithm. Assume X_0 to be the starting point of the trail. A repeated application of f with $X_{j+1} = f(X_j)$ will lead to each next point in the walk. Finally, we will encounter a collision caused by a duplicate visit at point X_3 and X_9, respectively.

The complexity for this Single-Processor Pollard-Rho (SPPR) algorithm is derived directly from the collision probability given by the birthday paradox [Pol78]. The birthday paradox deals with the random and repetitive selection of elements from a distinct set until a duplicate is chosen. Assuming Z to be a random variable for the number of chosen elements, the probability that j out of n elements can be selected without duplication is

$$Pr(Z > j) = (1 - \frac{1}{n})(1 - \frac{2}{n}) \ldots (1 - \frac{j-1}{n}) \approx e^{\frac{-j^2}{2n}}.$$

This finally leads to an expected number of distinct elements of roughly $\sqrt{\pi n/2}$ before a collision occurs and the algorithm terminates. A proof of this statement can be found in [vOW99, FO90].

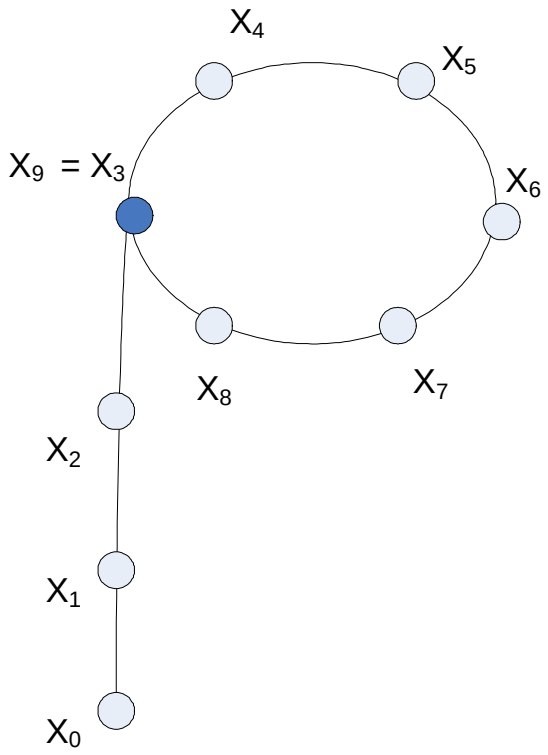

Figure 3.2.: Single Processor Pollard-Rho (SPPR)

3.3.4. Multi-Processor Pollard-Rho (MPPR)

The Multi-Processor Pollard-Rho is assumed to be the best known attack against ECC [HMV04]. Proposed by van Oorschot and Wiener [vOW99], it is basically a variation of the previously presented SPPR with some modification for better support of the parallel idea and to achieve a linear speedup with the number of available processors. Because of the fact that a set of several processors W can not easily contribute to the work on a single trail due to limitations in data distribution, another approach is favored for the Multi-Processor Pollard-Rho (MPPR) method.
Here, each processor $w_i \in \mathcal{W}$ starts an individual trail but does not primarily focus on ending in a cycle like the SPPR. In fact, a selection criterion for points on the trail is defined which mark a small partition of all computed points as "distinguished". For example, we simply can assign this property to points with an x coordinate showing a specific number of consecutive zero bits. Each processor transmits these distinguished points to a central unit or server which keeps track of all those points and checks them for duplicates. If such a duplicate, respectively a collision is finally found among all points on the central unit, the algorithm terminates successfully.

Example: Let $E(\mathbb{F}_p)$ be an EC with $k = \lceil log_2(p) \rceil = \lceil log_2(n) \rceil = 64$ bits. Then we can define a point to be distinguished if and only if the δ Most Significant Bits (MSB) of its x-coordinate are cleared. For instance, for $\delta = 16$, we will limit our efforts to find a collision among a smaller subset $D \subset \langle P \rangle$ with an upper bound $|D| = 2^{64-16} = 2^{40}$ points. This increases our chance to detect a collision among points in this subset and further we can efficiently use a multitude of parallel processors $\mathcal{W}$ in order to collect these points.

The difference between SPPR and MPPR becomes visible from Figure 3.3. Please notice the separate trails per processor where spots colored dark blue represent points designated as distinguished. Additionally, the eventual collision of two processors in a distinguished point is highlighted in red.

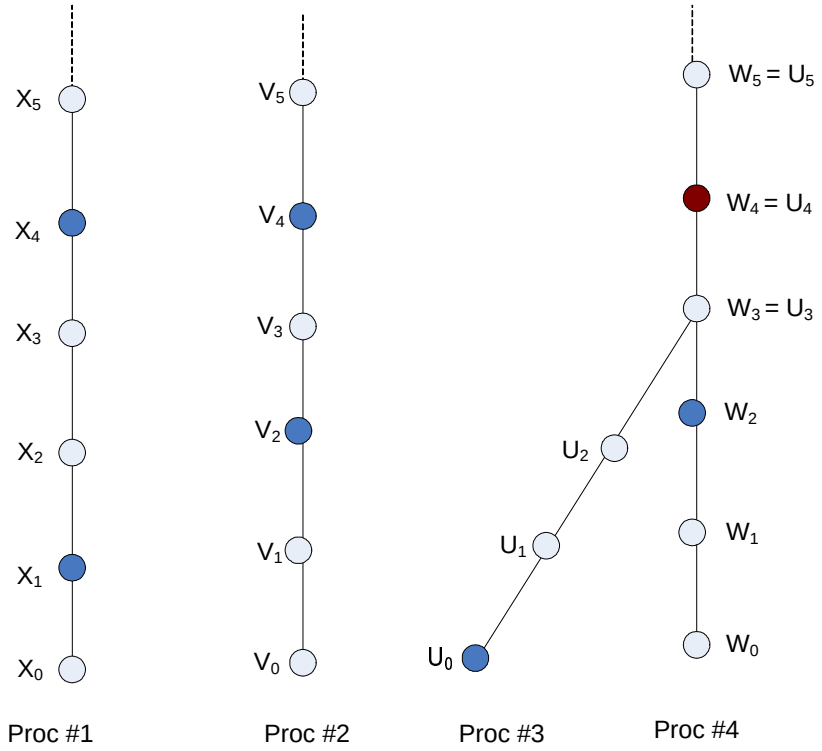

Figure 3.3.: Multi-Processor Pollard-Rho (MPPR)

After the rough outline of MPPR basic operation, it should become clear how the linear speed advantage with respect to SPPR is obtained. As seen with SPPR, we can expect $T = \sqrt{\frac{\pi n}{2}} + c$ points for the MPPR to compute until a collision occurs, i.e., one trail hits the trail of another processor. Due to the fact that this is a multiprocessor algorithm, a correcting factor c is appended to accommodate the additional overhead of collecting points in parallel. Assuming all available

processors to directly contribute to achieve threshold T, the workload of a single processor $w \in \mathcal{W}$ is reduced linearly to $\sqrt{\frac{\pi n}{2}} / |\mathcal{W}| + c$ [vOW99].

Further details of the MPPR method can be obtained from following pseudo code [HMV04]:

Algorithm 1 Multi-Processor Pollard-Rho

Input: $P \in E(\mathbb{F}_p); n = ord(P), Q \in \langle P \rangle$
Output: The discrete logarithm $\ell = log_P(Q)$

1: Select the size s of finite set with random points
2: Select a partitioning function $g : \langle P \rangle \rightarrow \{0, 2, \ldots, s-1\}$
3: Select a set D out of $\langle P \rangle$ satisfying the distinguished point property
4: **for** $i = 0$ to $s - 1$ **do**
5: Select random coefficients $a_i, b_i \in_R [1, \ldots, n-1]$
6: Compute i-th random point $R_i \leftarrow a_i P + b_i Q$
7: **end for**
8: **for** each parallel processor **do**
9: Select starting coefficients randomly $c, d \in_R [1, \ldots, n-1]$
10: Compute a starting point $X \leftarrow cP + dQ$
11: **repeat**
12: **if** $X \in D$ is a distinguished point **then**
13: Send (c, d, X) to the central unit/server
14: **end if**
15: Compute partition of current point $i = g(X)$.
16: Compute next point $X \leftarrow X + R_i;\ c \leftarrow c + a_i \ mod\ n;\ d \leftarrow d + b_i \ mod\ n$
17: **until** a collision in two points was detected on the server
18: **end for**
19: Let the two colliding triples in point Y be (c_1, d_1, Y) and (c_2, d_2, Y)
20: **if** $c_1 = c_2$ **then**
21: **return** failure
22: **else**
23: Compute $\ell \leftarrow (c_1 - c_2)(d_2 - d_1)^{-1} \ mod\ n$
24: **return** ℓ
25: **end if**

3.4. Further Attacks on Elliptic Curves

The previous section mentioned only a short evolutionary path from a naïve to the best known attack on general ECC. Of course, there are further algorithms which either do not exceed the performance of the presented algorithms or make special assumptions with respect to the underlying ECs.

Supersingular Curves When considering ECs with charactistic equal to 2 or 3, the curve might be supersingular which is a special feature concerning the curve's discriminant. This enables a special attack as proposed by Frey and Rück in 1994 [FR94].

Anomalous Curves For anomalous curves, i.e., a curve $E(\mathbb{F}_q)$ with exactly q points, Semaev [Sem98] and Smart [Sma97] offer an easier way to compute the discrete logarithm. This kind of attack is very specific and can simply be avoided by ensuring that the number of points of an EC does not match the number of elements in the field.

Subfield Curves Assuming small finite fields $\mathbb{F}_{2^B}$ and associated curves $E(\mathbb{F}_{2^{Bd}})$ the Pollard-Rho can be accelerated by a factor $\sqrt{d}$. This will lead to a expected runtime of only $\sqrt{(\pi n/d)}/2$ [GLV00].

Composite Order of Base Point Having a base point P with order $n = ord(P)$ and n not prime, a Pohlig-Hellman decomposition can be used to take advantage of the factorization of $n = p_1^{\nu_1} p_2^{\nu_2} \dots p_k^{\nu_k}$. This will reduce the complexity of a subsequent attack to the complexity of a similar attack on the largest identified prime p_i with $i \in \{1, \dots, k\}$[PH78].

ECDLP in Subintervals Finally, the Pollard-Lambda method, also known as method of catching kangaroos, is only a bit slower than the presented Pollard-Rho method with an expected runtime of $3.28\sqrt{b}$ for a b-sized interval and negligible space requirements [HMV04]. In case the search interval can be reduced a priori ($b < n$), the Pollard-Lambda method can be even faster with values $b < 0.39n$. Besides, it gives the option of parallelization with linear speedup as well. But with respect to general groups the Pollard-Rho method is still the best choice.

3.5. Elliptic Curve Representation

The context of this work requires a brief introduction to the representation of elliptic curves. In the Section 3.1, elliptic curves have been discussed only very roughly, without any details concerning advanced computational aspects such as, e.g., coordinate systems. However for a hardware based design, it is evident to choose the best fitting computational model and coordinate representation, respectively.
In the introduction of Section 3.1 we have agreed on ECs over prime fields $\mathbb{F}_p$ with p prime and $p > 3$ to exclude any problems from supersingularities. Furthermore, the parameter n should be implicitly considered as prime to avoid Pohlig-Hellman decompositions which spoil statements about the security of a specific bit length because of their possibility of further reduction.

3.5.1. Affine Coordinates

Keeping these commitments in mind, an important issue is the choice of a coordinate system. Using the Weierstrass equation from (3.1), the following affine representation for points R_1, R_2, and a resulting point R_3 for a group operation applies:

$$R_i \in E(\mathbb{F}_p);\ R_i = (x_i, y_i);\ i = 1 \ldots 3$$

The group operation to add two points $R_1 + R_2 = R_3$ is defined in affine coordinates as follows:

- **Curve addition formulae** ($R_3 = R_1 + R_2$ with $R_1 \neq \pm R_2$)

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \\ \lambda &= (y_2 - y_1)(x_2 - x_1)^{-1} \end{aligned}$$

- **Curve doubling formulae** ($R_3 = 2R_1$)

$$\begin{aligned} x_3 &= \lambda^2 - 2x_1 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \\ \lambda &= (3{x_1}^2 + a)(2y_1)^{-1} \end{aligned}$$

When classifying the number and type of suboperations, it will reveal the curve addition to require 6 ADDitions/SUBtractions, 3 MULtiplications/SQuarings and 1 INVersion. A curve doubling even consumes 8 ADD/SUB, 4 MUL/SQ and 1 INV. The reason that additions and subtraction as well as multiplication/squarings are considered equivalent can be referred to their inherent similarity of complexity. Both additions and subtractions as well as multiplications and squarings show uniform behavior in terms of performance and can even implemented the same way. Of course, there are options to make squarings significantly faster than standard multiplications [MvOV96] but this will require some extra resources when considering hardware implementations. Thus, for an area efficient design, both operations are considered equivalent so far.
Furthermore, it should be explained why ADD/SUB are listed explicitly which is not common practice. Usually, due to their little impact and relative significance in respect to the runtime of multiplications, they are neglected in most publications. But for a precise analysis of required cycles in hardware applications, it can be of importance to have detailed figures.

3.5.2. Elliptic Curves using Projective Coordinates

Due to the fact that field inversion is most costly with respect to computational expense, quite a lot of efforts were taken to avoid this type of expensive operation.

This improvement can be achieved by a change to another coordinate system. The projective coordinates encapsulate the expensive inversion using a separate Z coordinate to the cost of several additional multiplications. To translate the equation from (3.1) into projective coordinates, one transforms any point R_i using $x_i = X_i/Z_i$ and $y_i = Y_i/Z_i$, yielding following projective curve equation [CMO98]:

$$E : Y^2Z = X^3 + aXZ^2 + bZ^3 \ (a, b \in \mathbb{F}_p, 4a^3 + 27b^2 \neq 0) \tag{3.3}$$

Points R_i in projective coordinates are represented by:

$$R_i \in E(\mathbb{F}_p);\ R_i = (X_i, Y_i, Z_i);\ i = 1 \ldots 3$$

The transformation of group operations according to Equation (3.3) results in:

- **Curve addition formulae** ($R_3 = R_1 + R_2$ with $R_1 \neq \pm R_2$)

$$\begin{aligned} X_3 &= vA \\ Y_3 &= u(v^2X_1Z_2 - A) - v^3Y_1Z_2 \\ Z_3 &= u^3Z_1Z_2 \\ u &= Y_2Z_1 - Y_1Z_2 \\ v &= X_2Z_1 - X_1Z_2 \\ A &= u^2Z_1Z_2 - v^3 - 2v^2X_1Z_2 \end{aligned}$$

- **Curve doubling formulae** ($R_3 = 2R_1$)

$$\begin{aligned} X_3 &= hs \\ Y_3 &= w(4B - h) - 8{Y_1}^2s^2 \\ Z_3 &= 8s^3 \\ w &= a{Z_1}^2 + 3{X_1}^2 \\ s &= Y_1Z_1 \\ B &= X_1Y_1s \\ h &= w^2 - 8B \end{aligned}$$

Counting operations for this coordinate system, one will find a number of 6 ADD/SUB and 14 MUL/SQ for curve addition and 15 ADD/SUB and 12 MUL/SQ for curve doubling. The displacement concerning the ratio between additions and doubling in affine and projective coordinate systems is striking at this point. Where affine coordinates have an advantage when adding two distinct points regarding point duplication, in the projective domain doubling is the faster operation. Relevant for this statement are the number of MUL/SQ operations which are much more costly than ADD/SUB which can be performed in constant time.

3.5.3. Elliptic Curves using Other Coordinate Systems

Of course, there has been extensive research in this field [CMO98] by finding new and mixed coordinate systems like the Jacobian [CMO97] and Chudnovsky Jacobian [CC86] coordinate systems to achieve a minimal number of basic field operations for both curve operations. To avoid exceeding the limit of this thesis with facts which will not be of relevance for this work, it should be mentioned that most of those advanced coordinate systems exploit mathematical advantages based on projective coordinates to achieve better results.
As a conclusion for this chapter, it should be clear that the choice whether to use affine, projective or other coordinates refers to the issue how fast and resource efficient a field inversion might be implemented. In case that no other restrictions apply, one will favor affine coordinates if a field inversion can be computed faster than 10 MULs. However, in case of a hardware implementation, the additional area consumption has to be taken into account.

4. General Model for MPPR

Having discussed the MPPR basics in the previous chapter, this chapter will consider how to translate this idea into a general model for developing a software reference implementation. An early step of this development process is to identify required components and characteristics whose detailed features can be refined afterwards. Of course, this first approach does not include precise specifications for a final implementation. It should rather give an impression about the relationship between the interacting components and can be seen as an abstract component model. Hence, it will depict and encompass all data flow and required entity interaction up to a specified level of precision where it would become too specific to a dedicated environment. In other words, the general model will cover only the *macro design* and neglects any *micro design* like a detailed implementation of finite field operations. Using a finite field operation as a given and atomic operation enables the modeling of a software and hardware independent design for MPPR.
Another aspect of the creation of a general model is the possibility to study and analyze the required parameters for the MPPR method. This includes the size of the distinguished point subset, the number of random points used and centralized storage requirements. Hence, at the end of this chapter, we will provide a list with recommended settings which are optimized for the use in any (hardware) implementation.
Obviously, the MPPR algorithm requires a central server unit for collecting points from the computational processors $\mathcal{W}$. Analyzing Algorithm 1, the central unit needs to be equipped with following capabilities:

- A **communication controller** for data exchange with the $|\mathcal{W}|$ point processor
- A **database** for storing the tuples (c, d, Y) for a point $Y = cP + dQ$ in a sorted table according to Y for efficient point recovery.
- A unit for **validating distinguished points** received from a processor. This step is mandatory in terms of defective processors which might spoil the entire computational result by transmitting incorrect points.
- The arithmetic resources for **computing the discrete logarithm** from a colliding point.

- An **EC generator** for testing purposes. This unit is optional when external curve parameters are specified.

Putting all these building blocks together, a data flow model as depicted in Figure 4.1 appears.

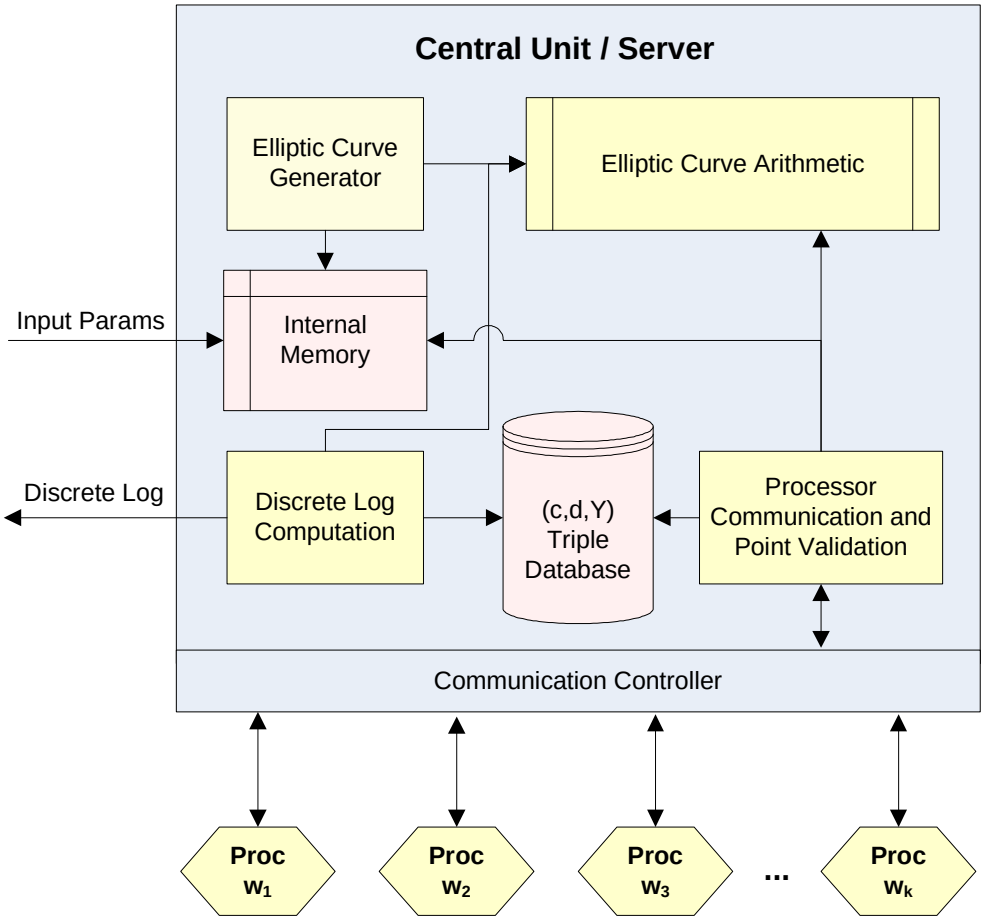

Figure 4.1.: A general component model for MPPR

With a rough understanding about the components in the central unit, we can continue to analyze the point processors which are put into charge to perform the actual computations. Each processor w_i of the set of all processors $\mathcal{W}$ will require at least the following functions:

- A **computational unit providing field arithmetic** in $\mathbb{F}_p$ for points, and for $\mathbb{F}_n$ to compute related coefficients in $\langle P \rangle$ (remember that n is prime).

- A **computational unit providing elliptic curve arithmetic** respectively a point addition in $E(\mathbb{F}_p)$.

- A **local memory** storing the current point X, its associated coefficients c, d with $X = cP + dQ$ and an array with s random triples (a_i, b_i, R_i) with $R_i = a_iP + b_iQ;\ i = 0 \ldots s-1$ and where s denotes the upper limit of available partitions $s = max(g(\langle P \rangle))$. Optionally, there might be additional storage for buffering distinguished points in case that the data path to the central server unit has not been cleared for transmission.
- A **Pollard-Rho state machine** which incorporates the required steps for determining a distinguished point, selects the next partition i and random tuple (a_i, b_i, R_i) and eventually, computes the next point $X_{succ} = X_{cur} + R_i$ and updates the corresponding coefficients $c_{succ} = c_{cur} + a_i \ mod\ n$ respectively $d_{succ} = d_{cur} + b_i \ mod\ n$.

This item list will result in the processor model shown in Figure 4.2.

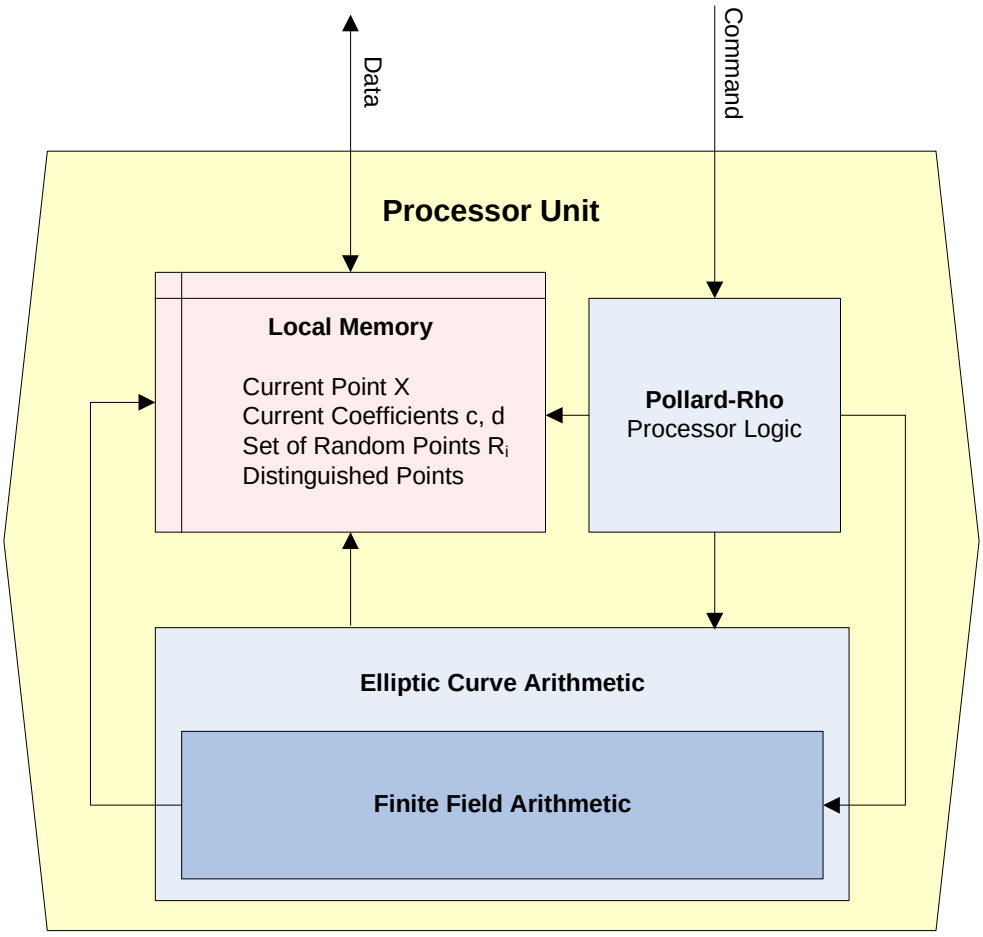

Figure 4.2.: A general processor model for MPPR

With all identified entities and components, the next step is to determine the best choice for design parameters and their associated environment.

4.1. Optimal System Environment and Parameters

Although this section will discuss how to find the best settings for the general model, it might be wise to already restrict the discussion to parameters which are also suitable for use in hardware. Otherwise, this might lead to an excessive view incorporating too many irrelevant software dependent options.
Basically, an optimal model requires a domain in which it is said to be optimal. Such a domain for the general model is given on the one hand by the required time for computing a new point including the update of coordinates (*Time Domain*). On the other hand, there is still the aspect of storage efficiency (*Memory Domain*). In spite of the negligible space requirements in the point processors, the central distinguished point database might easily consume an enormous amount of hard disk space dependent on the size of the set D of potentially distinguished points. For example, consider p to require $k = 160$ bits per value. The reception of a triple (c, d, Y) from a point processor actually translates to the task of storing a tuple with four values $(c, d, Y = (x_1, y_1))$, occupying a total size of $s_{bits} = 4k = 640$ bits, respectively $s_{bytes} = 80$ bytes. Furthermore, one might assume n to be of same size with $k = \lceil \log_2(n) \rceil$ and a common distinguished point D to have $\delta = 16$ leading zero bits in its x value. This will finally lead to a set of distinguished points of about $d_{count} = |D| = 2^{k-\delta} = 2^{160-16} = 2^{144} \approx 2.23 \cdot 10^{43}$ tuples in size. With no further data compression, the lossless storage of all distinguished points would consume $d_{bytes} = d_{count} \cdot s_{bytes} \approx 1.784 \cdot 10^{45}$ bytes, respectively $1.623 \cdot 10^{33}$ terrabytes (TB). This is currently infeasible to handle, the usage of lossy point storing is the only chance to cope with this flood of point data.
Although the general model should not be specific to a dedicated environment, there is another criterion for the hardware context which should be kept in mind. The optimality in area consumption (*Area Domain*), i.e. the number of gates or slices of an FPGA should already be considered to ensure that all elements of the *macro design* will result in an optimal design of an hardware application as well.

4.1.1. Time Domain

The discussion of the time domain will fork again in two ways with respect to achieve best performance when running an MPPR attack. First, it is mandatory to select the best design available to do a single point addition as fast as possible (*micro-time optimization*). Secondly, the collection of distinguished points should be optimized for producing a collision as fast as possible (*macro-time optimization*).

Coordinate Systems Concerning micro-time optimization, the selection of an optimal coordinate system for the MPPR is absolutely mandatory. From Section

3.5 it has become clear that if no further restrictions apply, the choice between using affine, projective or even another coordinate system is based on the speed of the field inversion. Actually, this lies not in the responsibility of this section because we defined field operations to be atomic for this model. Nevertheless, we will encounter other limitations which will put this section in charge of this discussion. When using a coordinate system with a three (or even larger) tuple like the projective, Jacobian or Chudnovsky coordinate system, these representations avoid the inversion by encoding it into an additional coordinate, e.g., related to X/Z and Y/Z, respectively. But this means that we lose uniqueness of the single coordinates as each coordinate $x \in \mathbb{F}_p$ in the affine domain can be expressed by $p-1$ different terms $x = X/Z;\ X, Z \in \mathbb{F}_p$. Although this is no problem for basic EC arithmetic, it will become an issue when considering the MPPR and its requirement to determine a distinguished point. In contrast to the statement in [vOW99] which estimated the costs of a MPPR attack in hardware by using projective coordinates, there actually is no general option to determine a distinguished point using a non-affine coordinate system without subsequent field inversion. In general, the field inversion itself seems to be the only operation to recover the unique characteristic of a distinguished point. Thus, for general curves the usage of a projective coordinate system is inappropriate because it would still require a subsequent inversion $X/Z = X \cdot Z^{-1}$ to be able to detect a collision. Hence, affine coordinates seem to be the most efficient way for coordinate representation in the context of the MPPR method. Consequently, we will spend a lot of efforts to perform the inescapable field inversion as efficient as possible.

Optimizing Point Computations A further step in micro-time optimization is the analysis of the data flow in a single computation of a new point. As we have already seen, the computation itself consists of a curve operation and two appended field additions for coefficient updating. In almost all situations, the point operation itself is basically a point addition rather than a point doubling. The latter one can easily be banned from the design by switching to another partition which will be discussed in detail later. Thus, only referring to a point addition $(x_3, y_3) = X_3 = X_1 + X_2 = (x_1, y_1) + (x_2, y_2)$ with associated coefficients $c_3 = c_1 + c_2$ and $d_3 = d_1 + d_2$, the data flow of an affine point computation is shown in Figure 4.3. Please notice the critical path marked in red. The data flow reveals that only lightweight operations like additions and subtractions are outside the critical path. This indicates only a small benefit from using massive pipelining per processor. The only method which might achieve a significantly better performance might rely on critical path condensing techniques, either by grouping several field operations together or by attempting to decrease the execution time of a single field multiplication or inversion. Both options will require

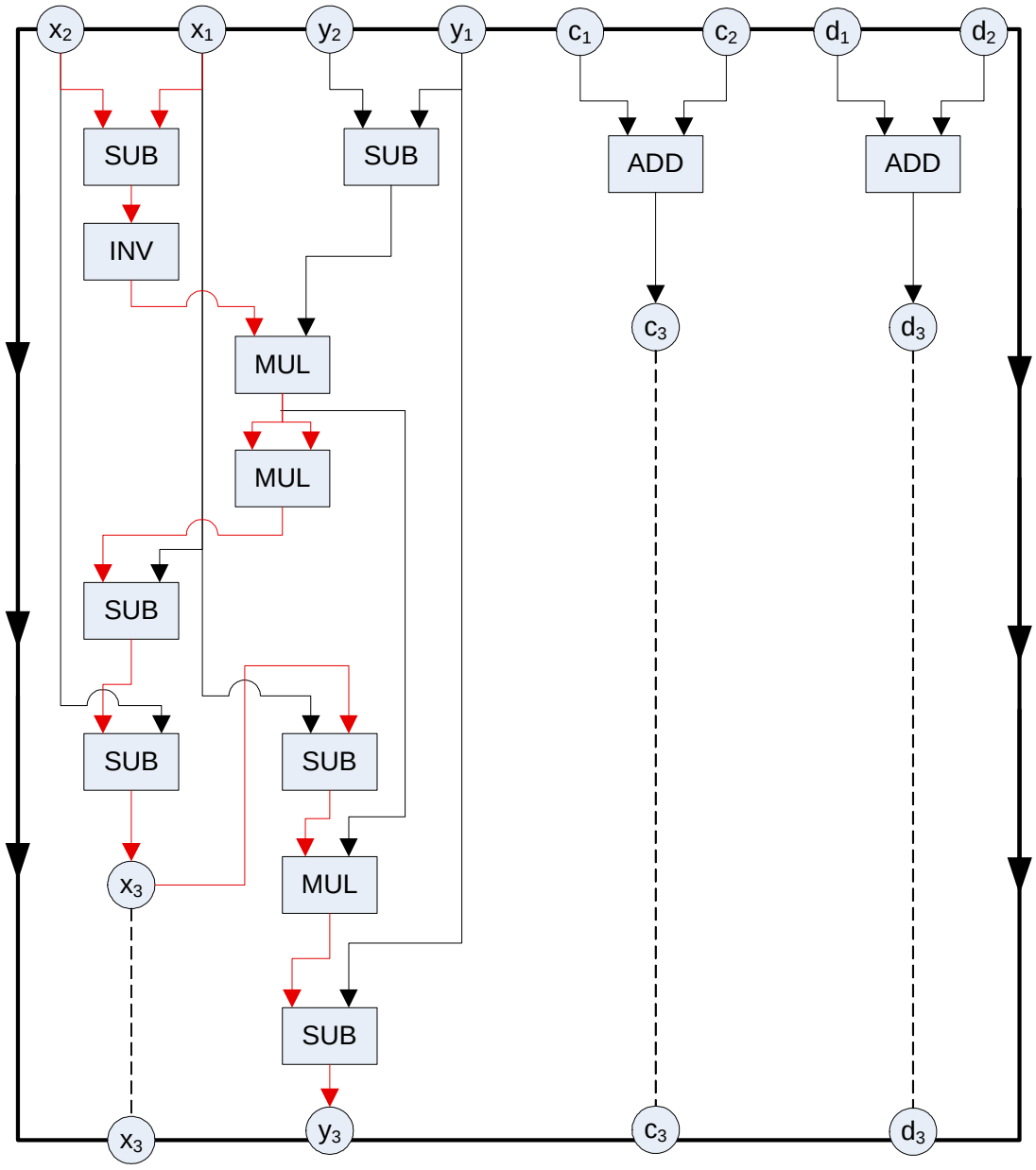

Figure 4.3.: Computation of a next point in MPPR (data flow)

specific implementation details which will exceed the scope of the general MPPR model at this place but will be discussed in Section 5.5.

Algorithmic Improvements After the discussion of the micro-optimization, we should some spend words concerning the macro-optimization domain, i.e. how to achieve an optimum in time to reveal the discrete logarithm in respect to a smallest number of computed distinguished points. Several papers [ESST99, WZ98, Tes01] have been published discussing the options for speeding up the Pollard-Rho algorithm. Here, a rather simple way is the *inverse-point strategy*. This strategy extends the term of a point collision by its inverse, i.e. not only identical points will lead to a collision, rather their corresponding inverses as well. Thus, defining the distinguished point property to specify a point X will also include its inverse X^{-1}. Hence, it will be possible to compute the discrete logarithm ℓ via an additional formula based on Equation (3.2):

$$\begin{aligned} R_1 &= -R_2 \\ c_1P + d_1Q &= -(c_2P + d_2Q) \\ (c_1 + c_2)P &= -(d_1 + d_1)Q \\ (c_1 + c_2)P &= -(d_1 + d_1)\ell P \\ (c_1 + c_2) &= -(d_1 + d_1)\ell \ mod\ n \\ \ell &= -(c_1 + c_2)(d_1 + d_2)^{-1} \ mod\ n \end{aligned}$$

This improvement will increase the overall performance of MPPR by a factor of $\sqrt{2}$ because not only the identity suffices to provoke a collision but also the inverse. This will finally result in an improved runtime per processor of

$$T_{w_i} = \sqrt{\pi n}/(2 \cdot |M|) + c. \tag{4.1}$$

A further option to discuss is the number of partitions which are used to determine a next random point in the Pollard-Rho's walk. It is not obvious how many partitions will provide a benefit for the speed of the MPPR. Pollard proposed in this work [Pol78] the use of three partitions and therefore three distinct random points. However, there have been studies showing that this choice is not optimal as it is not as random as it is supposed to be [Tes98]. Instead, for example, a 20-adding walk is proposed in [Tes01], providing better performance. But the disadvantage of using 20 partitions is the simple fact that the partitioning is not that easy. Using powers of two for partitioning yields the option to determine a random point by just taking the lower bits of a current value (e.g. x coordinate). This idea was picked up in [GLV00] where the use of partitions with $s = \{16, 32\}$ is favored. For such partition sizes, the algorithm is also easy to implement in hardware, thus, it is probably the best choice in this context.

4.1.2. Memory Domain

As already observed, the MPPR algorithm requires only a small internal memory per processor. But indeed, a central unit challenged by the task to store every distinguished point can easily dispatch terrabytes of hard disk space. Furthermore, the central unit not only needs to store the points, it rather must be capable of detecting a collision very efficiently. This points to the utilization of a data dictionary with emphasis on a fast insertion and multiple entry detection.

Data Organization Probably the most efficient method is the application of a degenerated hash table. A common hash table is created with additional storage space to ensure only a minimal amount of rehashing due to collisions of the hash function. This "spare" storage regularly consumes up to a third in addition the required space in total. In context of MPPR and its enormous demand for memory, this is not feasible. But the constraint that only a single collision needs to be found in some point of time with no (or at most little) demand for keeping other points, makes it possible to deal with a reverse approach. Instead of using a hash table which is larger than the total amount of potential entries, one might use one which is intentionally too small. Of course, this will cause internal collisions due to the fact that different points are affected with the same hash key. Unfortunately, overwriting the same hash position with different points will make collisions less probable but it is indeed the only method to cope with the mass of data. It should be noted that the probability of finding a colliding point with overwriting in hash positions can be estimated using following formula:

- Assuming no limitation in the memory $z = \infty$ will reduce the collision detection among n values to the birthday paradox, where a random variable Z requires at least j selections before a duplication occurs:

$$P(Z > j) = e^{-j^2/(2n)}$$

- Defining the memory to be limited to z entries, this means that a collision can only be detected among those z entries. Furthermore, it must be reflected that previous values are overwritten with the proportion of z/n in each step. Eventually, one obtains [vOW99]:

$$P(Z > j) = (1 - z/n)^{j-z} e^{-z^2/(2n)};\ j < z \tag{4.2}$$

Point Compression We can derive from Equation (4.2) that the more space for storing distinguished points is available, the more probable a collision finally is. Let $k = 160$ be the bit size of p for an EC $E(\mathbb{F}_p)$. Storing a single distinguished point will occupy $s_{bits} = 4k = 640$ bits for a tuple $(c, d, Y = (x, y))$ (see Section 4.1). For a specific value $x_1 \in \mathbb{F}_p$ of a point $X_1 = (x_1, y_1)$ only its additive

inverse $X_2 = \overline{X_1}$ with $X_2 = (x_1, y_2)$, where $y_2 = p - y_1$ exists in the EC group, and enables for the option to compress the y-coordinate. The two possibilities of $y_{i,0}, y_{i,1}$ for any particular x_i of a point $X_i = (x_i, y_{i,j})$ $(j \in \{0, 1\})$ can be reduced to store the index j, i.e. to store just a single bit for y. In practice, the assignment of the index j to a y coordinate can be performed by determining the logical relationship for y_j with $y_0 < \frac{p}{2}$ and $y_1 > \frac{p}{2}$, respectively. In other words, if the most significant bit of y_j is one, this will result in $j = 1$ to be stored as substitute for y_j, otherwise $j = 0$. This will decrease the memory usage from $s_{bits} = 4k$ to $s_{bits} = 3k + 1$.
Beyond the previous discussion, it is even possible to neglect the y coordinate at all. Due to the coefficients c and d, it can be easily recovered in case a collision in x has been detected. Thus, we can abandon a point's y coordinate, leaving us with $s_{bits} = 3k$.

There might be further ways to reduce the number of bits to store, especially with respect to the coordinates c and d which are not of primary interest for detecting a collision. Thus, for the sake of simplicity, more complex techniques for data management are not regarded in this thesis in order to avoid dynamic data structures which make it much more difficult to efficiently create and relocate data.

Storage Dimensions Finally, the issue of how many points are to be stored in a degenerated hash table should be considered relying on the capabilities of currently available computing environments. Table 4.1 shows the memory requirements in gigabytes for storing 2^z distinguished points with $z = 20 \ldots 32$.

No. of points		**Required Space for DP (bit size k) in GB**					
z	$s_{points} = 2^z$	k=40	k=80	k=128	k=160	k=192	k=256
20	1048576	0.117	0.234	0.375	0.469	0.563	0.75
21	2097152	0.234	0.469	0.75	0.938	1.125	1.5
22	4194304	0.469	0.938	1.5	1.875	2.25	3
23	8388608	0.938	1.875	3	3.75	4.5	6
24	16777216	1.875	3.75	6	7.5	9	12
25	33554432	3.75	7.5	12	15	18	24
26	67108864	7.5	15	24	30	36	48
27	134217728	15	30	48	60	72	96
28	268435456	30	60	96	120	144	192
29	536870912	60	120	192	240	288	384
30	1073741824	120	240	384	480	576	768
31	2147483648	240	480	768	960	1152	1536
32	4294967296	480	960	1536	1920	2304	3072

Table 4.1.: Server memory requirements

We observe that it already can be challenging to store points of a 40 bit EC for $z = 32$ bits when using off-the-shelf hardware. Besides the physical number of points, another issue is how many points in total need to be stored on the central unit during the MPPR's execution. This figure directly depends on a parameter δ, denoting the number of most or least significant bits which are defined to be zero designating a distinguished point. A change in δ will cause the proportion $\Theta = 2^{-\delta}$ of all points to grow or reduce with respect to the size of the set D of distinguished points. Of course, choosing a smaller set D will provide collisions with greater probability. But at the same time, the event of locating a distinguished point becomes less probable and increases the average trail lengths. Assuming, for simplicity, an unlimited size of memory $z = \infty$ [vOW99], the expected runtime for one out of $|W|$ processors for MPPR is $\sqrt{\frac{\pi n}{2}}/\,|\mathcal{W}| + c$ with $c = 1/\Theta$ as a consequence of the birthday paradox combined with multiprocessing. Using this statement, the amount of required storage space on the central unit can directly be derived. By downsizing the partition of all expected points to compute to the dimension of the set of distinguished points one achieves $s_{totalpoints} = \Theta\sqrt{\pi n/2}$ [Tes01]. The value $s_{totalpoints}$ denotes the expected number of points to be stored on the central unit. Unfortunately, we still need to compensate the drawback of limited memory in practice ($z < \infty$). According to Equation (4.2) and referring to [vOW99], limiting the central server memory to a value v will modify the expected runtime of MPPR to:

$$E(Z) = Exp(n, v) = \sum_{j=0}^{v-1} e^{-j^2/(2n)} + (n/v)e^{-v^2/(2n)} \tag{4.3}$$

Following Equation (4.3), we can finally deduce the expected number of distinguished points to be stored on a central server with memory for $v = 2^z$ points and a proportion Θ of the set D:

$$E(Z) = Exp(\Theta, n, v) = \Theta(\sum_{j=0}^{v-1} e^{-j^2/(2n)} + (n/v)e^{-v^2/(2n)}) \tag{4.4}$$

Incorporating the inverse-point strategy which accelerates the search by a factor of $\sqrt{2}$ and resolving $\Theta = 2^{-\delta}$ results in:

$$E(Z) = Exp(\delta, n, v) = 2^{-\delta}(\sum_{j=0}^{v-1} e^{-j^2/(4n)} + (n/v)e^{-v^2/(4n)}) \tag{4.5}$$

With Equation 4.5 it is possible to estimate the expected number of points which are to be handled by the central server based on following parameters:

- δ denoting the number of leading zero bits of the distinguished point property

- n denoting the order of base point P
- v denoting the number of storage elements with $v = 2^z$

Non-Profitable Trails Lastly, there is one remaining fact to be considered which does not directly influence the performance or memory requirements but can make a processor useless when being undetected. It might be possible that a processor takes an unfavorable walk, i.e. it might become stuck in a cycle without finding any further distinguished points due to a repetitive partitioning sequence. Without being resolved, such a processor would not further contribute to the search for distinguished points, thus, there should be a control mechanism which interrupts a processor after a time when no distinguished point was found.
The threshold T for such an interrupt could be based on heuristics. Wiener and Oorschot [vOW99] proposed to determine and use the mean trail length between two distinguished points. The mean trail length directly results from the proportion of the set D and is $1/\Theta$, given by the geometrical distribution for all trail lengths. In case the threshold T exceeds, e.g., the twentieth of an average trail length $(20/\Theta)$, the current trail is eventually abandoned and the processor is reset.

4.1.3. Area Domain

Although information about area consumption of hardware resources is not discussed at this point, there are already some aspects which take influence on the model in this stage. We try to achieve an optimal relationship between the *Time* for execution and the *Area* demand of components and the resulting *AT product* will be the primary measure in this work in terms of optimality.
As briefly mentioned, the substitution of computational units against "simpler" solutions in the design will directly lead to a decrease in area. This indeed applies to the removal of the point doubling facility of ECC. Basically, a point operation between two points $R_1, R_2 \in E(\mathbb{F}_p)$ can consist of a point addition (if $R_1 \neq \pm R_2$) or point doubling (if $R_1 = R_2$). In MPPR, the latter operation will occur very unlikely, hence, just in case the current point is equal to the randomly selected point. An easy way to circumvent the implementation of point doubling is to modify the partitioning function g to never return a random point equal to the current one or its inverse. This can simply be achieved by comparing the x-values of the current and randomly chosen point and, in case of equality, by just switching to another partition. This works if all random points have been generated distinctively, thus for all $R = \{R_i \in E(\mathbb{F}_p);\ i = 0, \ldots, s-1\} = \biguplus R_i$, i.e. all random points R_i in the set R are different. Translated to the hardware context, the entire state machine for point doubling can be replaced by one k-bit comparator.
Regarding Figure 4.3, it already has been discussed that operation pipelining is

rather unfavorable. The critical path nearly encompasses all costly operations, leaving only two additions and a subtraction untouched. Therefore, it is questionable to spend additional hardware resources for a separate field adder/subtracter for those three operations or to do all the computation consecutively. Furthermore it seems that the sequential approach is more efficient in terms of the AT-product. This point will be discussed in detail in Section 5.2.

4.1.4. Refining Parameters

Finally, the results of the previous discussion will be concluded and a set of proposed parameters for MPPR is suggested. The following settings, respectively parameters, have been identified:

1. Bit size of EC $k = \lceil log_2(p) \rceil$: To limit the scope of the analysis, the bit size of a curve should be restricted to $32 < k \leq 160$. Furthermore, for management and storage reasons, all regarded k are divisible by 8.

2. Number of partitions s controlling the number of different random points: Here, $s \in \{4, 8, 16, 32\}$ should be tested using the reference implementation. For hardware purposes, the best performing value from this set will be taken.

3. Distinguished point criterion defining a set D: Due to strong influence of the size of D and its associated proportion Θ, this variable should be fixed in a way to achieve comparable results. Thus, a distinguished point is defined to have the $\delta = \{16, 32\}$ leading bits cleared in its x coordinate.

4. Number of available server storage elements $v = 2^z$: Table 4.1 shows the fast growing nature of the central storage. To remain practically, $z = 24$ should be assumed allowing $v = 2^{24} = 16777216$ elements to be stored on the server, e.g., demanding hard disk space of about 7.5 GB for an EC with $k = 160$.

4.2. Designing a Reference Implementation

After the general discussion in previous sections, it might be useful to have a reference implementation at hand as well for debugging as for parametrization purposes. However, it should be noted, that this reference is primarily designed as a guiding implementation with no claim on being optimized for speed. Regardless, to gain a maximum performance without elaborative optimization, the implementation will be based on the programming language C and the optimized and prebuilt GMP library [Pro05] for field operations. Furthermore, the MPPR reference will run only a single point processor which makes more complex device

communication obsolete. With this level of abstraction and associated prerequisites, the general model discussed so far can be easily translated into a working system on a single-processor environment.
A yet undiscussed issue is the generation of testing data, e.g., the question how to generate an EC for a specific bit size according to the agreements from section 3.3. Of course, there is a lot of literature about EC generation as well as some samples, but when considering to do testing for arbitrary bit sizes, a more flexible EC generating facility becomes essential and will be described in the following.

4.2.1. Generating Elliptic Curves

To run attacks against EC, we must be able to create suitable curves. A naïve idea could be to select all curve relevant parameters randomly for a specific bit length k until the constraint of the discriminant $4a^3 + 27b^2 \neq 0$ is satisfied (cf. Section 3.1). But for a successful attack we will encounter the issue of determining the order of the curve and of a corresponding base point. Those parameters cannot be randomly selected and, in fact, need to be determined from the underlying curve. There are methods to count the points on a curve but these are computationally demanding. The best known algorithm in this domain is the *SEA algorithm* (and derivatives) proposed by Schoof [Sch85] Elkies and Atkin [Elk98].
With the previously mentioned approach, we need to face another problem. Since we had no influence on the random parameter generation, the selected curve might provide only weak security and could be vulnerable to efficient attacks discussed in Section 3.4. Thus, we should consider to satisfy at least the following security constraints when generating curves:

- For EC with group order m and $m = nq$, it should be ensured that q is a prime $> 2^{160}$ for curves to use in today's communication. In this thesis, the restriction $n = 1 \Rightarrow m$ *is prime* applies but q, however, might be below 2^{160} for testing purposes. The use of a prime parameter m will disable the application of the Pohlig-Hellman decomposition [PH78].
- Anomalous attacks can be avoided by choosing $m \neq p$ (cf. Section 3.4).
- Finally, $p^k \neq 1 \ (mod\ m)$ should be ensured for all $1 < k < 20$ to avoid the MOV attack [MOV93].

Consequently, we should abandon our naïve idea for curve generation and already ensure in advance to choose our parameters according to the mentioned constraints. Thus, a better method for curve generation for testing purposes is the use of the *Constructive Weil Descent*. But here, the problem is the limited choice among available EC when using this method [CMO00]. A most promising technique in this domain is the method of *Complex Multiplication*. Although this method is somewhat inconvenient to implement, it is more efficient and flexible

than the other presented algorithms. Due to the limited scope of this thesis, the generation of EC using Complex Multiplication is only roughly outlined, excluding any further discussion of complex structures like Hilbert or Weber polynomials. Please refer to [KSZ02] for further information. The following pseudo code will sketch how to generate an EC of prime order:

Algorithm 2 Complex Multiplication for Elliptic Curve Generation

Input: Discriminant D; a Hilbert or Weber polynomial Y; desired EC bit size k
Output: EC with (a, b, m, n) where m, n prime and a base point $B = (x, y)$

1: **repeat**
2: Generate a random prime p with k bits.
3: Attempt to find (u, v) satisfying $4p = u^2 + Dv^2$ (Cornacchias algorithm).
4: **until** tuple (u, v) exists
5: Determine possible curve orders: $m_1 = p + 1 - u$ and $m_2 = p + 1 + u$.
6: **if** neither m_1 nor m_2 is suitable according security constraints **then**
7: Return to step 1
8: **else**
9: $m \leftarrow m_i$ where $i \in \{1, 2\}$ and m_i matches security constraints
10: **end if**
11: Compute the roots of the polynomial Y modulo p.
12: **if** Y is a Weber polynomial **then**
13: Transform roots of Y to roots of the corresponding Hilbert polynomial
14: **end if**
15: Each root represents a j-invariant, leading to two elliptic curves.
16: Choose the curve $E(\mathbb{F}_p)$ which has order m (probabilistic check).
17: Finally choose a random point as base point on the curve $B \in_R E(\mathbb{F}_p)$
18: Since m is prime and every point in E is a generator, assign $n \leftarrow m$
19: **return** curve and base point

4.2.2. Components and Dependencies

As briefly discussed before, a reference implementation intents to give a rough impression about the behavior and performance of MPPR in software. For a quick realization, the MPPR designed for multiprocessing is degenerated to run on a single CPU. This naïve approach will avoid any problems with processor communication and memory locking.
The issue of the expected software performance rating should be discussed briefly, although it is not of primary interest but might be useful for comparison. On the one hand, it might sound realistic that slower results can be expected on a CPU which handles the central management *and* the point processing with respect to a dedicated processor which only computes distinguished points. On the other hand, the actual additional work for the management reduces to write

data to a hash table and is comparable to the effort to prepare and encapsulate the data for network transmission when considering the dedicated processors. Thus, regardless of their greater responsibility, a combined MPPR management and point processor will probably produce results similar to a single dedicated point processor with only a negligible difference.
Putting all observations together, one can deduce a reference model for a single off-the-shelf processor, as shown in Figure 4.4.

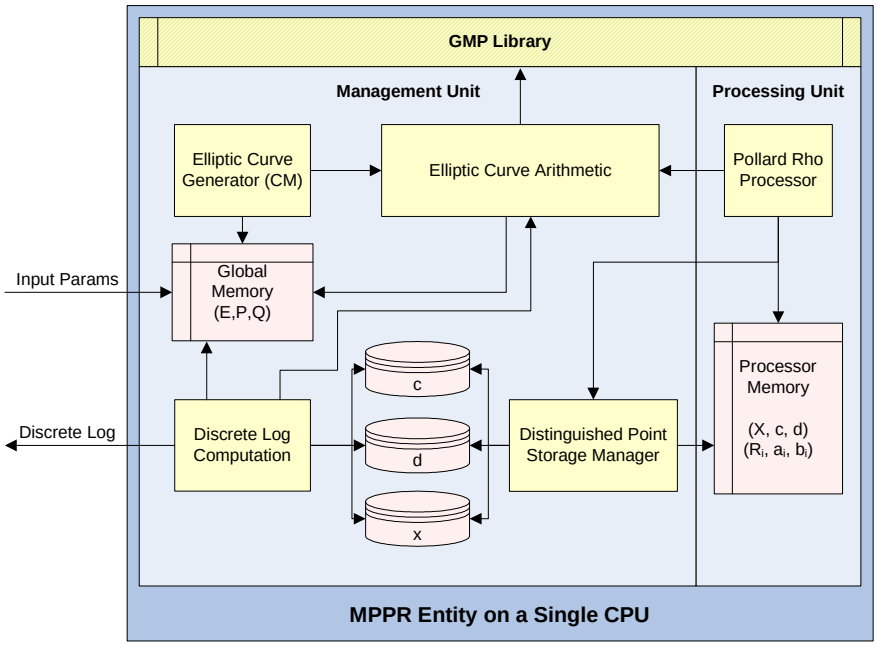

Figure 4.4.: MPPR reference implementation for a single processor

4.2.3. Managing Distinguished Points

It probably strikes that the single distinguished point database from Figure 4.1 has been split into three separate data containers. Obviously, this directly reflects the reduced requirements to store only the triple (c, d, x). Most suitable is the use of three flat files instead of a database system which would bring in too much overhead through, e.g, irrelevant multi-user access and transaction logging. Using flat files and defining bit sizes to comply with 8 bit boundaries, it is rather easy to create and maintain a hash table file for each of the values with common file system functions.
Next, some remarks concerning an appropriate hash function should be made. As we have already seen, it will not be possible to provide enough server space to store every distinguished point. Hence, a mapping between the domain D of distinguished points into the domain of storage elements S is defined. Let $(x_\lambda, y_\lambda) = \lambda \in D$ with $\lambda = c_\lambda P + d_\lambda Q$ be an arbitrary distinguished point. Then,

its associated storage element $(c_i, d_i, x_i) = \sigma_i \in S$ with $i \in \{0, \ldots, |S| - 1\}$ can be obtained by the hash mapping H:

$$H : D \rightarrow S : \lambda \mapsto \sigma_i \mid (\lambda, c_\lambda, d_\lambda) \mapsto (x_i, c_i, d_i) \text{ with } i \equiv x_\lambda \; mod \; |S|$$

In other words, the storage element is simply identified by reducing the x coordinate of point λ to a value i within the boundaries of the storage domain. Having allocated a storage file with 2^z elements for each c, d, x, this operation is the same as taking the $z - 1$ least significant bits of x as hash key i.
Before storing an element x in the server memory, we check if the previously inserted value is the same as the one to be written. This includes a real collision as well an inverse-point collision (see Section 4.1.1).

4.2.4. Implementing the Point Processor

Further attention should be paid to the implementation of the actual point computation for MPPR. Figure 4.3 already introduced the field operations and their interactions. By employment of the GMP library [Pro05] in the reference implementation, it becomes rather straightforward to realize these operations. Provided with the ability to handle integers of arbitrary bit sizes, the GMP library allows the following straightforward code snippets to implement the relevant field operations for addition, subtraction, multiplication, and inversion in $\mathbb{F}_p$.

Field Addition The addition of two elements $u, v \in \mathbb{F}_p$ can be performed by two basic computations described by the following algorithm $\text{ADD}_{\mathbb{F}_p}$. Please note that all statements beginning with "mpz_" are functions of the GMP library and can be referred to in the corresponding documentation [Pro05], if necessary.

Algorithm 3 Field Addition using GMP library ($\text{ADD}_{\mathbb{F}_p}$)

Input: Two elements $u, v \in \mathbb{F}_p$ and p prime
Output: Sum $u + v = r \in \mathbb{F}_p$

```
1: mpz_add(r,u,v); {Add r = u + v in integer domain}
2: if mpz_compare(r,p) ≥ 0 then
3:    mpz_sub(r,r,p); {If r>p, correct result r}
4: end if
5: return r
```

Field Subtraction The subtraction $\text{SUB}_{\mathbb{F}_p}$ of two elements $u, v \in \mathbb{F}_p$ works rather similar to the field addition. Basically, it just flips the addition and correcting subtraction around. Please note that the result can temporarily become negative, thus a signed integer data type or at least a hint for the current sign is required for correct results.

Algorithm 4 Field Subtraction using GMP library ($\text{SUB}_{\mathbb{F}_p}$)

Input: Two elements $u, v \in \mathbb{F}_p$ and p prime
Output: Difference $u - v = r \in \mathbb{F}_p$
1: `mpz_sub(r,u,v);` {Subtract $r = u - v$ in integer domain}
2: **if** `mpz_compare_ui(r,0)` < 0 **then**
3: `mpz_add(r,r,p);` {If u<v, correct result r}
4: **end if**
5: **return** r

Field Multiplication The presented algorithm $\text{MUL}_{\mathbb{F}_p}$ to multiply two elements $u, v \in \mathbb{F}_p$ is very rudimentary. This involves a full multiplication with subsequent reduction. Besides the fact that for computation a temporary memory of twice the input length is required, the reduction might be costly compared to better field multiplication algorithms like the Montgomery or interleaved multiplication [MvOV96]. For now, this basic implementation is sufficient for our purposes. More complex algorithms will be discussed for MPPR in hardware.

Algorithm 5 Field Multiplication using GMP library ($\text{MUL}_{\mathbb{F}_p}$)

Input: Two elements $u, v \in \mathbb{F}_p$ and p prime
Output: Product $uv = r \in \mathbb{F}_p$
1: `mpz_mul(r,u,v);` {Compute $u \cdot v$ in integer domain}
2: `mpz_mod(r,r,p);` {Reduce $r = (u \cdot v)\ mod\ p$}
3: **return** r

Field Inversion Finally, the inversion $\text{INV}_{\mathbb{F}_p}$ of an element $u \in \mathbb{F}_p$ is very simple using the GMP library. Fortunately, there is a function `mpz_invert` performing an optimized extended Euclidean algorithm (GCD) to return the inverse of a group element. This can directly be adopted for our purposes:

Algorithm 6 Field Inversion using GMP library ($\text{INV}_{\mathbb{F}_p}$)

Input: Two elements $u, v \in \mathbb{F}_p$ and p prime
Output: Inverse $u^{-1} = r \in \mathbb{F}_p$
1: `mpz_invert(r,u,p);` {Compute the inverse of u in $\mathbb{F}_p$}
2: **return** r

Having defined the field operations, the next step is their integration in the logic of the MPPR point processor. Table 4.2 will show the command sequence of GMP functions producing the expected result of a point computation $X = X + R = (x_1, y_1) + (x_2, y_2)$ with associated coefficient updating $c = c + a\ mod\ n;\ d = d + b\ mod\ n$. Besides, the presented command sequence can also be taken for itself without referring to the basic field arithmetic above. Thus, it can be seen as

a detailed description of the actual Pollard-Rho idea without relying on a specific implementation of a field operation. Please note the syntax of numbering. A leading 'E' (like in "E01") indicates a part of the elliptic curve addition. A 'C' denotes an operation on the coefficients c or d.

Input:		Current point $X = (x_1, y_1)$ with coefficients $c, d \in \langle N \rangle$, Random point $R = (x_2, y_2)$, with coefficients $a, b \in \langle N \rangle$ Modulus m, order of base point $n = ord(P)$				
Output:		Updated current point X and coefficients c, d				
Register:		$R1, R2, \lambda$				
No.	Description	Field OP	Target	OP1 (u)	OP2 (v)	MOD
E01	$R1 \leftarrow (y_2 - y_1)$	$\text{SUB}_{\mathbb{F}_p}$	$R1$	y_2	y_1	m
E02	$R2 \leftarrow (x_2 - x_1)$	$\text{SUB}_{\mathbb{F}_p}$	$R2$	x_2	x_1	m
E03	$R2 \leftarrow R2^{-1}$	$\text{INV}_{\mathbb{F}_p}$	$R2$	$R2$	-	m
E04	$\lambda \leftarrow (R1R2)$	$\text{MUL}_{\mathbb{F}_p}$	λ	$R1$	$R2$	m
E05	$R1 \leftarrow \lambda^2$	$\text{MUL}_{\mathbb{F}_p}$	$R1$	λ	λ	m
E06	$R1 \leftarrow (R1 - x_1)$	$\text{SUB}_{\mathbb{F}_p}$	$R1$	$R1$	x_1	m
E07	$R1 \leftarrow (R1 - x_2)$	$\text{SUB}_{\mathbb{F}_p}$	$R1$	$R1$	x_2	m
E08	$R2 \leftarrow (x_1 - R1)$	$\text{SUB}_{\mathbb{F}_p}$	$R2$	x_1	$R1$	m
E09	$R2 \leftarrow (x_1 - R1)$	$\text{SUB}_{\mathbb{F}_p}$	$R2$	x_1	$R1$	m
E10	$R2 \leftarrow \lambda R2$	$\text{MUL}_{\mathbb{F}_p}$	$R2$	λ	$R2$	m
E11	$x_1 \leftarrow R1$	-	x_1	$R1$	-	-
E12	$y_1 \leftarrow (R2 - y_1)$	$\text{SUB}_{\mathbb{F}_p}$	$R2$	$R2$	y_1	m
C13	$c \leftarrow (c + a)$	$\text{ADD}_{\mathbb{F}_p}$	c	c	a	n
C14	$d \leftarrow (d + b)$	$\text{ADD}_{\mathbb{F}_p}$	d	d	b	n

Table 4.2.: Command sequence for a single point computation

Obviously, Table 4.2 describes the computation of a single point operation in MPPR. But this does not yet involve the determination of a partition, the selection of a random point as well as the detection of a distinguished point. Thus, the procedural steps E01-C15 are set into context of some additional steps which are described in the following, separate algorithm for the sake of clearness.

Algorithm 7 Software Point Processor

Input: $(X = (x, y), c, d)$; (R_i, a_i, b_i) for $i = 0, \ldots, s-1$; p; n
Output: A distinguished point tuple (X, c, d) with $X \in D$

1: **while** $X \notin D$ **do**
2: Determine new partition i from $i = x \ mod \ s$
3: Compute next point $X = X + R_i$ and coefficients c, d {see E01-C14}
4: **end while**
5: **return** (X, c, d)

4.2.5. Embedding the Point Processor

Finally, the point processing unit requires to be embedded into a surrounding entity which cares for point storage, collision detection and finally the computation of the discrete logarithm. In the reference model this will also take place on the same processor, hence, the point processing entity is implemented as a function call for the central unit. Although the point processor is a piece of software in this stage, it should be remarked that the server components can easily be reused to operate with several hardware processors. Then, instead of a function call to the underlying software layer, a communication controller will be employed for receiving and transmitting data to the attached external processors. The following algorithm is designed for one embedded software point processor but can easily extended for use with further external processors.

Algorithm 8 MPPR Central Unit

Input: $P \in E(\mathbb{F}_p)$; p prime; partition size s; $n = ord(P)$; $Q \in \langle P \rangle$
Output: The discrete logarithm $\ell = log_P(Q)$

1: **for** $i \leftarrow 0$ to $s-1$ **do**
2: Select random coefficients $a_i, b_i \in_R [1, \ldots, n-1]$
3: Compute i-th random point $R_i \leftarrow a_i P + b_i Q$
4: **end for**
5: Select starting coefficients randomly $c, d \in_R [1, \ldots, n-1]$
6: Compute starting point $X \leftarrow cP + dQ$
7: **repeat**
8: Call point processor to return next distinguished point $X = ((x, y), c, d)$
9: Store the returned distinguished point tuple (x, c, d) in separate hash files
10: **until** a collision in the hash file for value x was detected
11: Let the two colliding triples in point X be (c_1, d_1, X) and (c_2, d_2, x)
12: **if** $c_1 = c_2$ **then**
13: **return** failure {We found a useless collision}
14: **else**
15: Compute $\ell \leftarrow (c_1 - c_2)(d_2 - d_1)^{-1} \; mod \; n$
16: **return** ℓ
17: **end if**

5. Hardware Model for MPPR

With the general model at hand, we now can focus on the actual target of this work: Translate the general model to an optimum hardware design. Obviously, most efforts during this process will be taken by the efficient implementation of field operations in terms of an optimal AT-product. But before starting over with designing, it is mandatory to discuss a suitable underlying system environment.

5.1. Model Environment

Considering hardware issues, it is a good idea to recall the prerequisites for MPPR to select a most suitable environment. The following aspects will capture our special attention:

- A multitude of parallel processors.
- Low cost processors.
- Flexibility in parameterization (e.g. bit sizes).

Having this in mind, a decision can be taken concerning the employed hardware device type. Currently, available choices are Application Specific Integrated Circuits (ASIC) or Field Programmable Gate Arrays (FPGA). Whereas ASICs are relatively cheap at high volumes with respect to FPGAs, ASICs have the great disadvantage of being inflexible in terms of logical modifications. Due to the great importance of reconfigurability for this work, we will favor the usage of S-RAM based FPGAs. However, it should be stressed that for actual attacks for which very large numbers of point processors are required, the conversion of an "optimum" FPGA design into an ASIC architecture can be a very good solution. In this thesis, we will employ two different types of FPGAs. First we will use a Xilinx Spartan-3 XC3S200 for testing and second a larger Spartan-3 XC3S1000 FPGA for the final system. Both chips are shipped either as stand-alone processors or soldered on development boards providing external connections via several expansion slots, VGA connectors and RS232 interfaces. Table 5.1 will summarize the features of both FPGAs.

At this point we should briefly explain the FPGA chip itself. The functionality of being reprogammable is realized by smart internal subsections or slices. Each FPGA slice contains two 4-input lookup tables (LUTs), two configurable D-flip

Feature	**XC3S200**	**XC3S1000**
System Gates	200K	1000K
Slices	1,920	7,680
Logic Cells	4,320	17,280
Multipliers (18x18)	12	24
Block RAM Bits	216K	432K
Distributed RAM Bits	30K	120K
Max Single Ended I/O	173	391
RS232[1]	Yes	Yes
USB 2.0[1]	No	Yes
VGA D-SUB[1]	Yes	Yes

Table 5.1.: Device features of SPARTAN-3 XC3S200 and XC3S1000

flops, multiplexers, dedicated carry logic, and gates used for creating slice based multipliers. Each LUT can implement an arbitrary 4-input boolean function. Coupled with dedicated logic for implementing fast carry circuits, the LUTs can also be used to build fast adder/subtracters and multipliers of essentially any word size [Xil06d]. For the FPGAs mentioned above, four slices are grouped into one Configure Logical Block (CLB).
The chip version shipped on development boards can be used for single unit testing, i.e. to check the function of the MPPR. For testing and debugging purposes, the on-board communication interfaces can be utilized. For achieving a real benefit of a multiprocessing application like the MPPR, however, the built-in interfaces do not provide enough flexibility to connect several boards efficiently. Thus another approach is preferred, i.e. taking an architecture capable to carry and manage up to 120 FPGAs with an integrated interconnection [KPP+06].

5.1.1. The COPACOBANA Design

The idea briefly mentioned in the previous section has been realized in [KPP+06]. The authors have designed a flexible architecture targeting the massively parallel execution of suitable cryptanalytic algorithms. In total, 120 FPGAs of type Xilinx XC3S1000 can simultaneously contribute to a distributed task. The data transfer between FPGAs and a remote host is realized by a common 64-bit interconnect which is capable to address each computational unit separately. A schematic design of the COPACOBANA machine is depicted by Figure 5.1.

To consider modular design guidelines, the FPGAs are not soldered directly on a single backplane. Rather, six of them are grouped an a single modules in standard DIMM format. This makes it easy to run the COPACOBANA in different stages of expansion, e.g. with 30, 60, or a maximum of 120 FPGAs. This

[1] Communication interfaces are only available for FPGAs on development boards

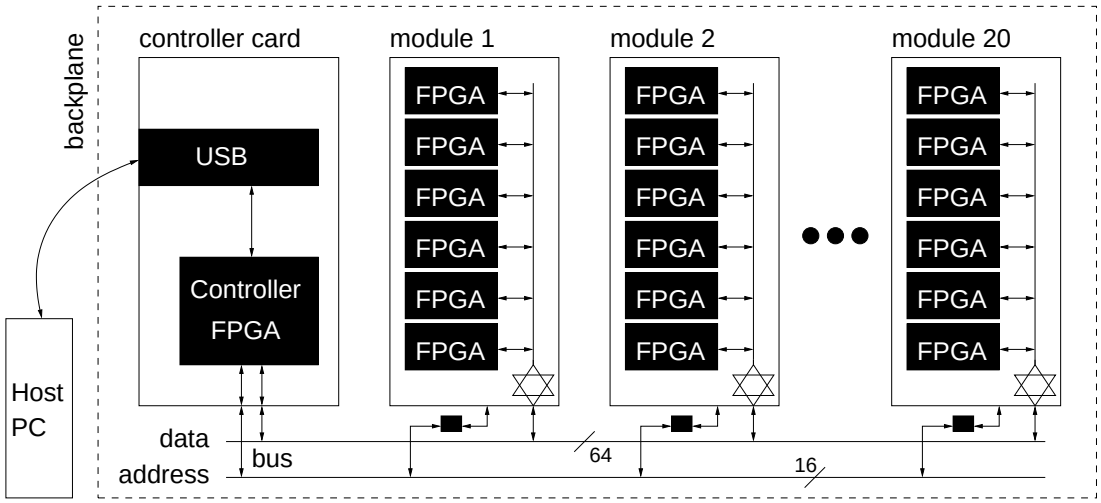

Figure 5.1.: COBACOBANA architecture from [KPP+06]

figure implicitly contains a limit of 20 DIMM modules fitting in the backplane. A single DIMM module is shown as a schematic in Figure 5.2

With the design introduced above, a highly scalable and flexible architecture is available to perform a multitude of single FPGA computations in parallel. This is an optimal prerequisite for running an MPPR attack on ECC. For further information and applications concerning the COPACOBANA design, the interested reader is referred to [KPP+06].

5.1.2. Programming FPGAs

Creating a hardware implementation can be performed by assembling a schematic circuit design. But for large applications, this approach is impractical due to its fast growing complexity. Thus, it is much more appropriate to use a higher level of abstraction like hardware descriptions providing syntaxes known from common computer languages. Public representatives are Verilog [Pal03] and VHDL [Ash01] which are capable to model complete systems from scratch using a language comparable to high-level programming languages such as C++ to some extent. Instead of being compiled like in case of executable code, the hardware description is synthesized, translated and mapped onto the available hardware environment or FPGA's slices, respectively. Furthermore, both hardware description languages are standardized and vendor-independent, therefore the design can rather easily be ported to another hardware system or FPGA. This also includes the translation into an ASIC design. Obviously, this implies that no FPGA specific optimizations have been made during the design stage.
Finally, the issues which of both language to choose and which tools to use for development remain. The vendor Xilinx ships its boards with a starter kit containing a development environment called "Xilinx ISE" [Xil06b]. This application

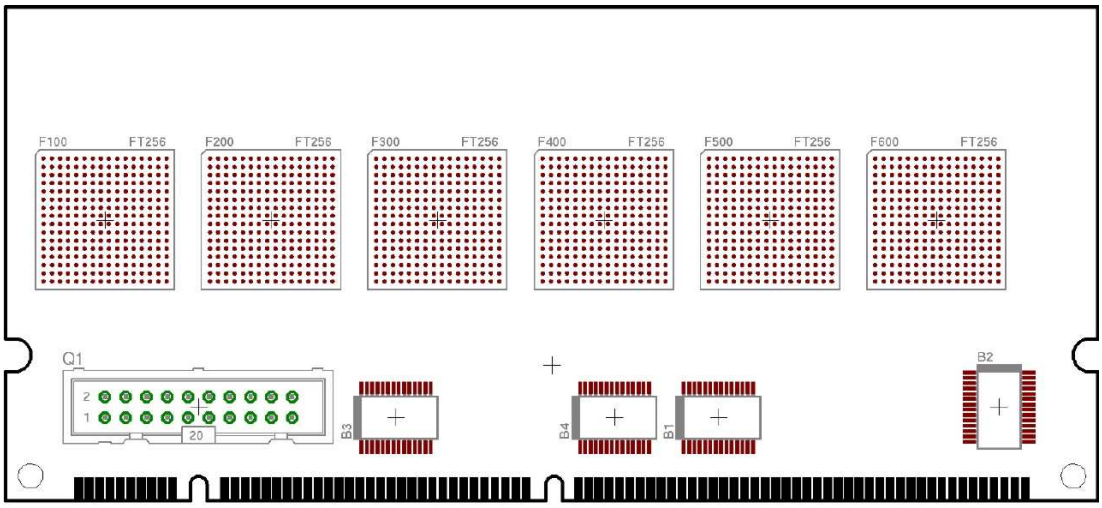

Figure 5.2.: COBACOBANA DIMM module [KPP+06]

suffices for our design needs. Due to the fact that the created code is not compiled, there is a necessity to simulate the behavior in advance to verify a successful operation. This simulation is taken place using the Xilinx's software 'ModelSim' [Xil06c] which is an absolute must-have in terms of design testing and debugging. Finally, we should address the issue which of the two available Hardware Definition Languages (HDL) to choose. Although Verilog is simpler in respect to its semantics, we will favor VHDL because of its greater flexibility. But in common practice, this is more a matter of personal taste.

5.2. Basic Architecture

5.2.1. Software vs. Hardware

Switching over to the hardware domain from our software reference will bring in some additional aspects to be considered. First, an FPGA is a piece of logic which should be programmed in a way to make use of strong parallelism. In contrast, a processor in an off-the-shelf computer might provide some internal pipelining and even dual core computing (referring to currently available Athlon64 X2, AMD Opteron [AMD06] and Intel Itanium processors [Int06]). But it is not capable changing its logic according to the needs of an executed application. This also manifests in their instructions usually being processed sequentially. Consequently, a standard CPU has a fixed count of internal gates for specific purposes. Each running application needs to cope with the amount of hardware support provided by the processor. An FPGA however, is capable extending logical parts with more importance to an application to benefit a faster execution.
The weaknesses of an FPGA in direct comparison to off-the-shelf CPUs are

located in their clocking and design optimization. Where common CPUs are clocked with several gigahertz due to straight optimizations, FPGAs are internally restricted to operate only in a spectrum of hundreds of megahertz. This is a direct drawback for the internal structure which provides the great flexibility of being reconfigured. Dependent on the optimality factor of the programmed design of an application, the actual clocking speed will even further be degraded. Consequently, an FPGA can only acquire a speed advantage towards classical CPUs by a customization providing a highly optimized and parallel architecture.

5.2.2. Model Conversion to Hardware

In Chapter 4 we discussed a general model to implement the MPPR attack. Now, we focus on some hardware specific adaptions to exploit the additional benefits of parallelism in hardware.
The point addition is determined to be the most time consuming operation in MPPR. Unfortunately, because of the strong serial nature (see 4.1.1), it is rather doubtful if parallelism can help at this point. The critical path in Figure 4.3 only excludes three additions and subtractions which might be computed by separate units. Due to the little relevance of additions and subtractions in respect to the overall performance, this method will lead just to a worse AT product with great certainty.
Another idea would be to condense the critical path by combining operations. Such an option yields the trick of the simultaneous inversion [HMV04] due to the fact that the field inversion is definitely the most expensive operation in affine point operations. It is performed by inverting a product $\alpha = a_1 a_2 \dots a_{k-1} a_k$ of several elements $a_1, \dots, a_k \in \mathbb{F}_p$ and subsequently recover for an element a_i by multiplication with irrelevant elements a_j with $i \neq j$ the inverse, i.e.,

$$a_i^{-1} = (a_1 \dots a_{i-1} a_{i+1} \dots a_l)\alpha^{-1} = (a_1 \dots a_{i-1} a_{i+1} \dots a_l)(a_1 \dots a_l)^{-1}.$$

Of course, this trick only pays off in case that inversions are extraordinarily expensive, compensating the additional costs of the control overhead as well as the additional combining and recovering multiplications. Hence, due to its significant additional costs which actually makes it only interesting for small $l \leq 4$, this idea should not directly be merged into our design. We can revise our decision when we have determined the actual speed of an inversion.
Another opportunity for optimization could lie within the special treatment of squaring. In the previous software model, these operations have been neglected and simply mixed up with multiplications. Here, the question is if a dedicated unit for squaring can significantly increase the overall performance. Of course, the achieved benefit must justify the additional area on the FPGA. If the AT product does not decrease, it makes no sense to employ it. Again, due to the execution of only a single squaring per point addition, it is rather doubtable if this

is an improvement. This issue will be picked up when specific field multiplication algorithms for hardware are chosen.
Although it has been shown that in-line pipelining can not be expected to accelerate point additions, there still is the option of duplicating the entire point processing unit itself on the FPGA. This option will definely provide a linear speedup when leaving the little additional controlling overhead out of consideration. The drawback of using several point processing cores is the more sophisticated way of communication between the server and the FPGA. Talking about a single core will enable us to transmit any distinguished point directly to the central unit. Indeed, this will not be possible when having several cores. Here, the input and output demands multiplexing. In case that two cores attempt to send at the same time, locking and buffering mechanisms are further required. Of course, the introduction of a point buffer (PB) into the design will provide a benefit as well: it becomes possible for a core to write a distinguished point into a centralized buffer on the FPGA and to continue its search for a next one immediately. Consequently, the central unit can independently access the PB and retrieve the stored points without interrupting any of the cores. Ensuring a flawless operation, this technique requires an additional locking mechanism pointing out which component has the permission to access and modify the contents of the PB.
Considering the last aspect, it might be wise to rewrite our architecture from Figure 4.1 to a new hardware layer model. Figure 5.3 depicts an abstract structure which presents an outline for a more detailed hardware modeling later on. At this

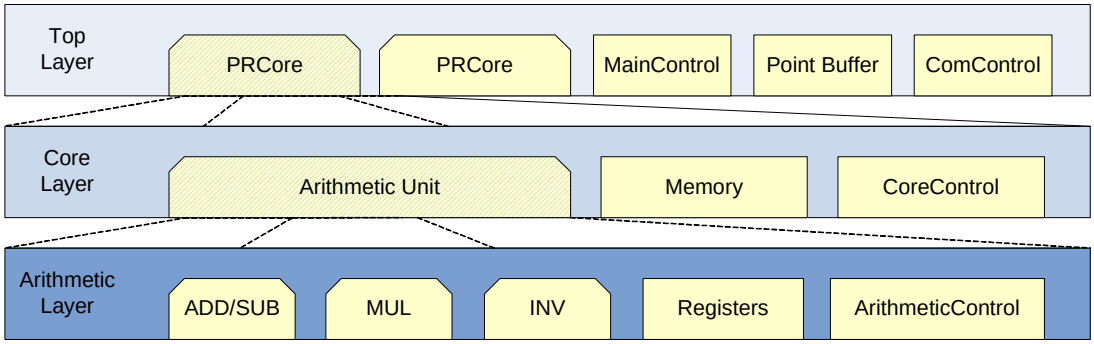

Figure 5.3.: Abstract hardware layers of an MPPR processor

point, we should agree upon the notion of an FPGA based point processor to act as a passive device. Hence, it is dependent on external data and commands sent by an active central unit. In the previous model which runs on a single device, it was easily possible to start and control the point processing unit using a function call. With an external FPGA, this is significantly more difficult.
Besides a communication controller which is primarily designed to exchange the raw data on a physical level, there must be some entity extracting the incom-

ing raw bits and interpreting them as commands to know how to instruct its dependent point processing cores. A new component called *Processor Controller* (PC) accommodates to this: Its basic function is to take the bit stream from the communication controller, reorganize it to data and commands and pass them to the corresponding cores. Furthermore, it is capable to control the buffer for data retrieval because it is mandatory to have an interface where the central server unit can access the FPGA's PB.

With this insight, we now can start to discuss the top layer which primarily encapsulates exactly this functionality.

5.3. Top Layer Design

In the previous section, the employment of multiple point processing cores on one FPGA has been proposed to increase the number of simultaneous computed points by parallel processing. Unfortunately, the management of several cores yields some difficulties with respect to data addressing and resource sharing. Whereas the input of new data can rather easily be distributed by using a chip select (CS) signal to a dedicated core, the output of found distinguished points need first to be resolved by a k-bit multiplexer and then buffered in some memory (PB) to allow the cores to continue their work immediately. Besides, the buffer needs to be managed using a Lock Controller (LC) to avoid access confusion among the contributing parties or nodes. The LC is basically a one-bit-per-node register denoting the current lock status, several inputs for *Lock Requests* (LKR_x) and the same amount of outputs for telling the connected units to which one the *Lock* is *Granted* (LKG_x). The locking mechanism encompasses as well signals for each core as well as an additional signal ($LKRC$ and $LKGC$) for the PC which is required when the server sends a request for data retrieval. Furthermore, the LC can control the output multiplexer as well as the PB's Write Enable (WE). Due to the fact, that the locking itself is rather trivial consisting of simple logical combinations, we will put this aside and refer the reader to Appendix A.3
As already mentioned, the function of the PC is to translate the transmitted bits from the server into commands and delegate them to the cores by setting the appropriate CS_x. The PC always operates in a request-response mode, i.e. for each command received from the server, a response or status message is generated and transmitted back to the caller. This ensures the remote station that a command has been successfully processed. For a better overview about the functionality of the PC, we should define its command set at this point:

- **RESET command**. All cores can be initialized separately using another CS_x signal. This is important, because a core may become stuck in a loop without finding further distinguished points. In this case, the server has to resolve this idling situation and reset the affected computational unit.

- **LOAD command**. Before starting the computation, each core must be provided with initial information about its starting position and an individual table containing random point data. A specific CS_x and $ADDR$ signal will denote the core and the associated memory position to be loaded. Due to the fact, that the top layer does not include any memory modules itself (except the point buffer), the LOAD command is passed directly to each core which will do the actual processing.

- **RUN command**. When the cores have been initialized and reset correctly, they can be switched to running mode by issuing this command.

- **NEXT_PT command**. The server can send this command to the FPGA to check if its central distinguished point buffer contains some points. In case that the buffer was already locked while this command is called, the pointer to the topmost element in the PB is decreased by one. Furthermore, this command appends an array of status flags to the response. This can be evaluated by the server to determine an halted processor core due to an infinite cycle.

- **LOCK command**. In case that the server has detected available distinguished points on the FPGA, it needs to lock the buffer to gain exclusive access to it. Only in locked mode it is possible to retrieve the buffer data and delete a topmost element. During the locking period, no core can write further points to the PB.

- **UNLOCK command**. After locking and retrieving point data from the buffer, the server must call UNLOCK to allow the cores writing further points to the buffer.

- **RETRIEVE command**. When the server has requested exclusive access permission to the PB, it can use the RETRIEVE command to read the topmost element from the buffer.

We have excluded the communication controller (ComCTL) from discussion so far. The reason for this is that only the actual data interface of this component is of interest, i.e. that it receives and transmits bits and indicates the data status using interrupts. The actual work of generating command requests and responses is in the responsibility of the PC. Hence, the implementation of the ComCTL and an associated communication path might vary due to its environment, e.g. being part of the COPACOBANA this will be a bus controller module. For a standalone FPGA on a development board the data exchange could rely on the RS232 or USB interface.
Figure 5.4 shows a schematic plan of a setting with two Pollard-Rho Cores (*PRCore*) representing the top layer of our MPPR design in hardware. Please refer to the appendix A.5 for further information concerning signal and port naming

and specifications. Obviously, the top layer is easily scalable. The only change which is required to add further cores to the design is to bring in additional Chip Selects (CS_x) on the PC as well as extending the LC with further request (LKR_x) and granted (LKG_x) ports. Hence, dependent on the EC bit lengths and available slices on the FPGA, the design can be adapted to a maximum number of fitting cores. This directly allows the number of points being computed in parallel to be optimized.

5.4. Core Layer Design

The previous section has discussed a top or management layer which is primarily responsible for data exchange with the external server unit. Consequently, we now can proceed with the computational components (PRCore) represented by the core layer. Here, we need to accomplish the remaining issues from section 5.2.2, e.g., if it makes sense to add additional hardware for squaring and parallel computation of non-critical operations. The resolution of these issues directly depends on information about details of field operations in hardware for a cost-benefit analysis. For this, we need a pivot point as reference to discuss if a solution is better or not. Our metric for deciding whether a solution is good or not will be the AT product: The lower the AT-product, the more efficient and, hence, better the design is. Due to an expected low performance gain of our outstanding aspects, we will start with an implementation saving most hardware area and which provides a high potential to achieve good timings. Obviously, this excludes all of the open issues at this point which would require a significant amount of hardware area with only a little effect on the system performance.

Central Arithmetic Unit Further, all required operations should make use of extensive resource sharing, i.e. by combining all field operations in a central Arithmetic Unit (AU) for GF(p) operations. This GF(p)-AU provides the function of all required field operation in a single unit enabling single path routing of the costly k-bit input and output data. A detailed discussion and implementation of the Arithmetic Layer can be found in Section 5.5.

Memory Management A further component with relevance to this layer is the Core Controller (CC) which is primarily responsible for managing the operations of the AU and delegating its output to memory locations. The latter aspect directly leads us to the third type of component - a memory module. For further details, it is convenient to recall our data requirements for field operations from Table 4.2 which simply can be converted to memory constraints. It is clear that the AU will demand three concurrent inputs to be available. Two providing the operands $OP1$ and $OP2$ and a third for the modulus MOD. The modulus itself is

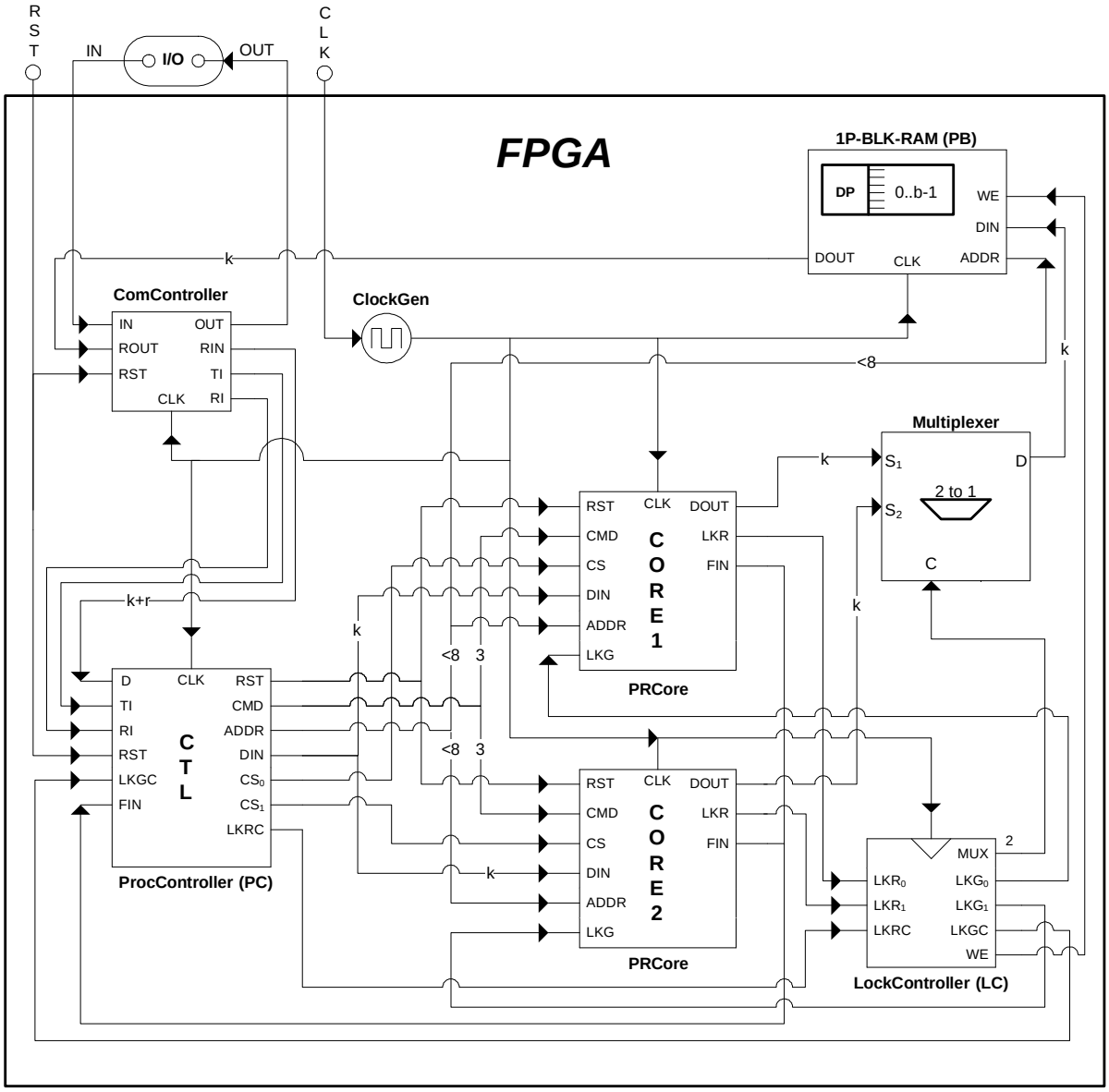

Figure 5.4.: Top layer design (schematic overview)

relatively static, it only changes between n for coefficient operations and m for all other purposes. The operands, however, are changing dynamically. Additionally, some values have to be available as first *and* second operand - either at the same time. Consequently, a common data pool is a further constraint allowing a concurrent dual access. Exactly this prerequisite can be satisfied using a dual-port block memory module as it is widely available on most FPGAs. Such a module allows concurrent reading of the same memory segment using two independent access paths. Simultaneous writing is supported as well but only in different memory elements. Otherwise, a result will become undefined.
This dual port main memory contains all k-bit wide register values for the current point computation, including starting point X, associated coefficients c, d as well as temporary values like $\lambda, R1, R2$. Besides, it can hold the random point data $(R_i = (x_i, y_i), a_i, b_i)$ for $i = 0..s-1$ in upper memory regions as well. Hence, register and random data is available via a common access path requiring no further multiplexing.
The latter aspect makes it desirable to similarly spend a separate single-port memory module for storing the modulus MOD either. Of course, this approach might appear wasteful due to the fact that only two values need to be stored. Considering the alternative, in other words by placing two individual registers for m and n with different access and control paths will require output multiplexing and routing. Concluding, the two-value memory approach is obviously the less expensive one.

Core Controlling Next, we should spent some words about the CC and how it manages the AU. For each field operation, the CC starts driving the memory address lines ($ADDR_A$, $ADDR_B$ and $ADDR_M$) to retrieve operands for input to the AU. Next, the AU is notified about the type of operation which is one out of *ADD(0), SUB(1), MUL(2), INV(3)*, and a reset signal (RST). The AU indicates a terminated computation by raising the operation finished ($OPFIN$) signal and the CC can store the result to memory (by enabling WE_B) at a new location pointed by $ADDR_B$.
From the top layer design, we have already encountered some commands used by the server to control the FPGA's operation. Some of them are directly delegated to the cores to be processed. This includes:

- **CORE RESET**. When a core receives a reset, it reinitializes its components and state machine. Then, it will start over at the beginning of a point computation. Furthermore, if the core was in running state before, it is stopped.

- **CORE LOAD**. When residing in stopped mode, a core can load its memory with a k-bit value from an external data path. Therefore, it drives the first address line $ADDR_A$ to the specified memory location and enables

writing to memory using the data from the input port DIN_A. The LOAD command can basically write into all memory elements, including the realm of register and random point data.

- **CORE RUN**. This command simply puts the core from the halted into the running mode. It should be invoked, when all data has been successfully loaded. The current state of a core is passed to the top layer. This is useful to tell the central station whether a core has aborted its operation due to a too long trail length.

A schematic overview of the discussed core layer design is given by Figure 5.5.

We can now approach the final and most challenging arithmetic layer.

5.5. Arithmetic Layer Design

The details of the arithmetic layer has been neglected throughout the thesis so far. The general model has assumed field operations as atomic entities provided by the GMP library, leaving their actual implementation yet undiscussed. Now, all prerequisites and proposals will be compiled building an Arithmetic Unit which offers a functionality for doing field additions, subtractions, multiplications and inversions. Obviously, we should start with introducing each of those operation separately.

5.5.1. Field Addition

Together with the subtraction, the field addition can be seen as the simplest operation. It can described entirely by Algorithm 9.

Algorithm 9 General Field Addition

Input: Two elements $u, v \in \mathbb{F}_p$ and p
Output: Sum $u + v = r \in \mathbb{F}_p$

1: $r \leftarrow u + v$
2: **if** $r >= p$ **then**
3: $\quad r \leftarrow r - p$
4: **end if**
5: **return** r

This notation looks rather similar to the Algorithm 3 that we have created using GMP functions. But this time, we definitely have to care how to perform such an underlying k-bit addition and subtraction operation. Obviously, it is easy to use a simple k-bit carry-propagation adder/subtracter (CPA) which is barely an array of k interconnected full adders. But this type of unit brings in a significant latency due to the required time for carry propagation. The more

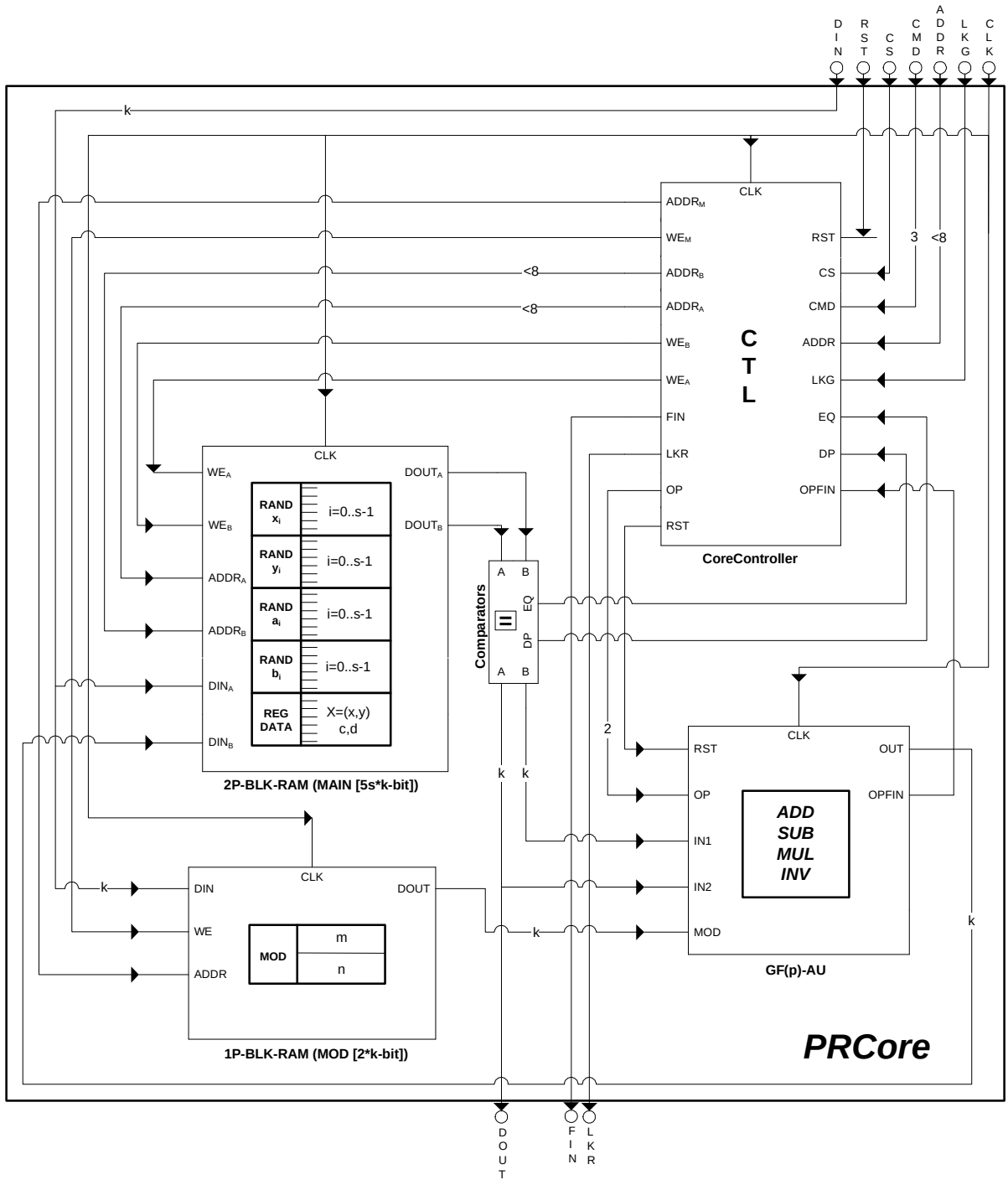

Figure 5.5.: Core layer design (schematic overview)

bits respectively full-adders are added at a time, the longer the propagation path will become. This directly reduces the maximum clock speed of the final system. There are several ways to cope with this issue. One option is to choose another basic adder type. Suitable models are the carry-look-ahead (CLA) and the carry-save adders (CSA). But both types bring in additional constraints. For example, the CLA is rather resource consuming in respect to FPGA area. The CSA, being one of the fastest type of adders, is based on a three-to-two I/O model, thus inappropriate for direct use in our field adding operation. Employing a CSA will finally require a subsequent addition which combines the two outputs of a CSA to a single result. Of course, there are further adder types, having a similar ratio between benefits and disadvantages. But these should not play a role at this point. The interested reader might be referred to [Koc95] for further information.
Another option to achieve a better latency is the usage of *chunking*, i.e., the entire value to be added is chopped into small chunks or limbs which are processed separately per clock cycle. This reduces the propagation time enormously, e.g., having k-bit values and using r chunks, we only need to consider the latency time of a k/r-bit adder. The drawback of this solution is the additional overhead and the increased number of clock cycles. First, it takes much more effort to take care of the adding and storing of correct chunks which will require additional r-k-shift registers. Further, each chunk computation requires a separate clock cycle. In terms of complexity, the computational effort increase from $O(1)$ to $O(k)$, what is not preferable.
Another option to overcome the problem of the propagation delay could be the use of a precompiled and highly optimized core module for the specific FPGA. The so called Intellectual Property (IP)-cores provided by the FPGA vendor can simply be used like the integration of a library in a software application. Besides, IP-cores often make use of device specific characteristics, such as a high-speed propagation path, etc. [Xil06a]. This seems currently to be the most straightforward and reasonable option because it encapsulates rather complex bit manipulations with a good ratio between resource consumption and latency times.

5.5.2. Field Subtraction

As already discussed, the subtraction within a field operates the same way as the addition. Again, this can easily obtained from the following mathematical description: The only difference between addition and subtraction actually is located in its control logic. In conclusion, this indicates that both operations are merged onto the same hardware already at this stage.

Algorithm 10 General Field Subtraction

Input: Two elements $u, v \in \mathbb{F}_p$ and p
Output: Difference $u - v = r \in \mathbb{F}_p$

1: $r \leftarrow u - v$
2: **if** $r < 0$ **then**
3: $\quad r \leftarrow r + p$
4: **end if**
5: **return** r

5.5.3. Combined Field Addition and Subtraction

Assuming a k-bit IP-Core adder/subtracter, we can create a schematic design which incorporates both operations from above in a single schematic design depicted in Figure 5.6. Basically, it is based on a single k-bit adder/subtracter unit whose input are multiplexed according to the situation (cf. lines 1 and 3 of Algorithms 9 and 10). An intermediate result is temporarily stored in a k-bit register and guarantees a stable output after the input values have been withdrawn. Beside the computational components, an interesting point is the condition han-

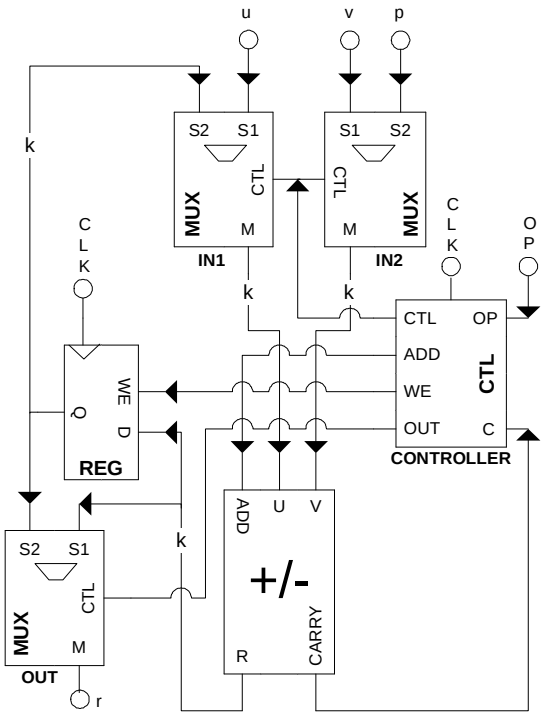

Figure 5.6.: Field addition and subtraction (schematic overview)

dling. Table 5.2 gives an impression about the internal control logic for both operations. Please be aware that a set Write Enable (WE) will make the reg-

OP	CLK	Description	Control	Q	R
ADD	1	$r \leftarrow u + v$	WE $\leftarrow 1$, ADD $\leftarrow 1$, CTL $\leftarrow 0$	*undef*	$u + v$
ADD	2.1	$r \leftarrow r - p$	WE $\leftarrow 0$, ADD $\leftarrow 0$, CTL $\leftarrow 1$	$u + v$	$r - p$
ADD	2.2	set output mux	OUT $\leftarrow$ (CARRY $= 1$)	$u + v$	$r - p$
SUB	1	$r \leftarrow u - v$	WE $\leftarrow 1$, ADD $\leftarrow 0$, CTL $\leftarrow 0$	*undef*	$u - v$
SUB	2.1	set output mux	OUT $\leftarrow$ (CARRY $= 0$)	$u - v$	$u - v$
SUB	2.2	$r \leftarrow r + p$	WE $\leftarrow 0$, ADD $\leftarrow 1$, CTL $\leftarrow 1$	$u - v$	$r + p$

Table 5.2.: Addition/subtraction control table

ister store a value from its data input D in the current cycle. The state of the ADD flag indicates either an addition or subtraction in the computational unit. The $CARRY$, or in this case $BORROW$ value, is required to determine the correct result. In case that the $CARRY$ after a subtraction becomes $CARRY$=1, this will indicate an integer underflow, thus the result for input u, v is negative: $u - v < 0 \Rightarrow u < v$. This information will be used to simulate the if-conditions and route the correct output. Last, the CTL and OUT signals will define the current state of multiplexers (CTL= $0 \Rightarrow S_1$; CTL= $1 \Rightarrow S_2$).

5.5.4. Field Multiplication

Now, we should have a closer look onto the field multiplication which is definitely more challenging. This already manifests in its extended complexity. Whereas the addition/subtraction has constant or at most linear complexity (in respect to full adders), the multiplication will take us quadratic efforts for calculation [MvOV96]. Again, we should attempt to find a design which fits our requirements for an optimal AT-product best. As already mentioned, an uninspired school-book-multiplication followed by a full reduction (see Section 4.2.4) is definitely no option in hardware. Smarter ideas for multiplication are available by the Montgomery [Mon85] and Interleaved Multiplication [Bla83]. These methods yield their performance benefit of about k clock cycles respectively the classical multiplication by alternating the stages of multiplication and reduction. This will avoid to grow the result to twice the size of the original input data and concurrently, make an immediate reduction step rather simple. Hereby, the Montgomery Multiplication (MM) is the even cheaper one because only two instead of three inner computations are necessary [Ama05]. Regardless, a small disadvantage for MM might be the need for a change of the data representation. But we will see that for the field inversion an algorithm exists with same constraints making the additional efforts more attractive. Excluding further alternatives using systolic arrays [Kim01] due their specificness in terms of resource sharing efforts, we will propose the use of a MM unit.

Excursus to Montgomery Representation Before we continue with the multiplication itself, we should discuss the required value transformation into the Montgomery domain. The MM achieves its performance by substitution of integer division required for reduction by divisions by two, respectively right-shifts. This effect is obtained by converting input values $a \in \mathbb{F}_p$ to a new radix representation with $\tilde{a} = aR \; mod \; p$ with a radix $R = 2^h$ and $R > p$. With respect to this thesis, the number of radix bits h is defined as $h = k + 2$.
The element transformation is simple a one-to-one mapping of each element a onto another element $\tilde{a}$ in the same field $\tilde{a} \in \mathbb{F}_p$. Of course, the inversion, respectively back transformation can achieved easily. Let $\tilde{a}$ be a value in Montgomery domain with $\tilde{a} = aR \; mod \; p$. The reverse transformation $\tilde{a}R^{-1} = a \; mod \; p$ can be obtained by

$$\begin{aligned} \hat{p} &= -p^{-1} \; mod \; R \\ U &= \tilde{a}\hat{p} \; mod \; R \\ a &= [(\tilde{a} + Up)/R] \; mod \; p \end{aligned}$$

Although the Montgomery arithmetic provides us with the great advantage of fast reductions, it becomes clear that this benefit suffers from the necessity to transform and reconvert any values from and to Montgomery domain. Consequently, the Montgomery arithmetic only amortizes when considering multiple subsequent operations. Fortunately, a conversion does not affect any characteristics of the number domain, hence, required criteria and uniqueness for determining distinguished points in MPPR remain still valid. This can be referred to the homomorphism feature between standard and Montgomery domain.
In other words, we are able to convert all EC parameters to Montgomery domain and remain throughout the entire computation in this domain. After all computations have been finished, we simply convert back to standard domain. This and the arithmetic in Montgomery domain retains us from performing expensive divisions to maintain a modulus reduction. Further information concerning the Montgomery transformation can be found in [MvOV96, Koc95].
After having introduced the Montgomery domain, we can come back to the discussion of the Montgomery multiplication. Below, the MM is shown in Algorithm 11.

The transformation of the pseudo code to the data flow model, depicted in Figure 5.7, will directly reveal an architectural hardware bottleneck. It can be obtained that two registers are necessary to prevent the signal propagation path to become too long. Passing through two adder units in a single cycle would unacceptably reduce the maximum clock speed. But with incorporation of a second register, each iteration of line 3 to 5 will require two clock cycles.

Algorithm 11 Montgomery Multiplication

Input: $\tilde{u}, \tilde{v} \in \mathbb{F}_p$ with $\tilde{u} = uR \; mod \; p; \tilde{v} = vR \; mod \; p; \; R = 2^h, \; h = k + 2$
Output: Product $\tilde{u}\tilde{v}R^{-1} = \tilde{r} = rR \in \mathbb{F}_p$

1: $m \leftarrow 0$
2: **for** $i = 0$ to $h - 1$ **do**
3: $\quad m \leftarrow m + \tilde{v} \cdot \tilde{u}_i$ {add $\tilde{v}$ if bit i in $\tilde{u}$ is one}
4: $\quad m \leftarrow m + p \cdot m_0$ {add p if m is odd}
5: $\quad m \leftarrow m >> 1$ {divide m by 2 [r-shift by 1 bit]}
6: **end for**
7: **if** $m \geq p$ **then**
8: $\quad m \leftarrow m - p$
9: **end if**
10: **return** $\tilde{r} \leftarrow m$

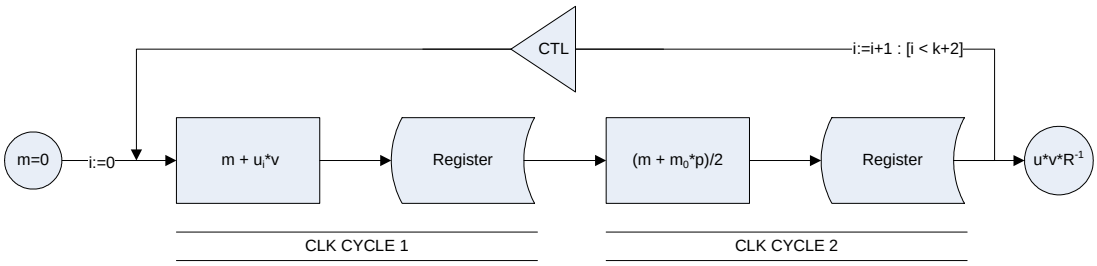

Figure 5.7.: Montgomery multiplication (data flow)

A very fast MM implementation making use of CSAs is presented in [Ama05]. Instead of the issues of standard adders, it has the drawback of two area-consuming CSA units and a split multiplication result in a *CARRY* and *SUM* part. As already mentioned, the final combination of those values will require a further standard adder, reducing the performance advantage of this alternative MM architecture.
Considering a good resource sharing factor for the AU design, it might be wise to check for a greatest intersection of resources required for addition/subtraction as well for multiplication. Taking this as a priority and reusing the adder/subtracter IP-core will take us back again to the double-registered performance problem.
To defuse this issue, a more intelligent control logic can help to suppress irrelevant computation cycles, i.e. an operation with no change to the result is simply removed by the controller saving a clock cycle. There a two options for cycle optimizations:

- **Remove empty $\tilde{v}$ additions** (line 3 in Algorithm 11). In each iteration of MM, it is tested if the current bit is $\tilde{u}_i = 0$. In this case, an empty addition $m \leftarrow m + 0 \cdot v$ would be performed which is skipped.

- **Avoid empty p reductions** (line 4 in Algorithm 11). Checking for the Least Significant Bit (LSB) of m will enable us to determine in advance if m is odd or even. In case it is even, we can again save the reduction step cycle.

Let us take a look at the savings in terms of complexity. We have seen that the expected runtime of the MM algorithm based on a double registered design will take $T = 2(k + 2)$ steps as a direct result from the length of the main loop and number of iterations. Assuming a uniformly distributed value u where $k = \lceil log_2(p) \rceil = \lceil log_2(\tilde{u}) \rceil$, we can consider roughly $k/2$ bits set to one. Thus, skipping all additions for zero bits in $\tilde{u}$ will already reduce the average runtime of the proposed MM to $T = 1.5(k + 2)$.
Considering the reduction steps, we should first focus on the responsible execution condition. In each iteration, a reduction occurs when the LSB of m is $m_0 = 1$. Hence, we have to follow the data flow to get an understanding how this can happen. A direct influence has the division by two (line 5 of Algorithm 11) where the bit m_1 will become the new LSB m_0. Obviously, due to the uniform distribution of $\tilde{u}$, this is with probability of $Pr_{DIV2}(m_0 = 1) = 1/2$. A further change in the LSB might happen during an $\tilde{v}$-addition and is dependent if $\tilde{v}$ is odd or even. Fortunately, a detailed case differentiation in $\tilde{v}$ can be omitted, because both cases are symmetric, i.e. unconcernedly if m_0 has become zero or one from division by two, an optional v addition will again map the result into domains of equal size. Hence, we can estimate the final probability $Pr(m_0 = 1) = 1/2$ that a LSB will become odd through manipulation by division (line 5 of Algorithm 11) and addition steps (line 3 of Algorithm 11). Therefore, we can downgrade the complexity estimation for the reduction step indeed by another $1/2(k+2)$, leaving a total expected runtime of $T = k + 2$ in average case. This is finally a similar result with respect to a more costly CSA implementation. We only need to accept the slower maximum clock speed which results from using IP-cores instead of the utilization of CSAs. This is of course not desirable, but tolerable concerning the next operation - the slow and expensive inversion. With this performance result, we can evaluate the option for an additional squaring unit. Due to the fact that common squaring techniques in GF(p) do not undermine the lower complexity limit of h, we can forbear from the idea to integrate a separate square unit.
Figure 5.8 depicts the schematic design of a Montgomery Multiplicator.

Some word should be spent concerning the applied control logic. Clearly, the controlling state machine will have two main states, one for adding $\tilde{v}$ to m ($ADD\text{-}V$) and another when reducing m to an even value ($ADD\text{-}P$). Skipping operations means nothing else than remaining in the current state. The state machine diagram in Figure 5.9 shows an abstract description of controller function. A detailed listing specifying each control flag is omitted at this point for the sake of clarity.

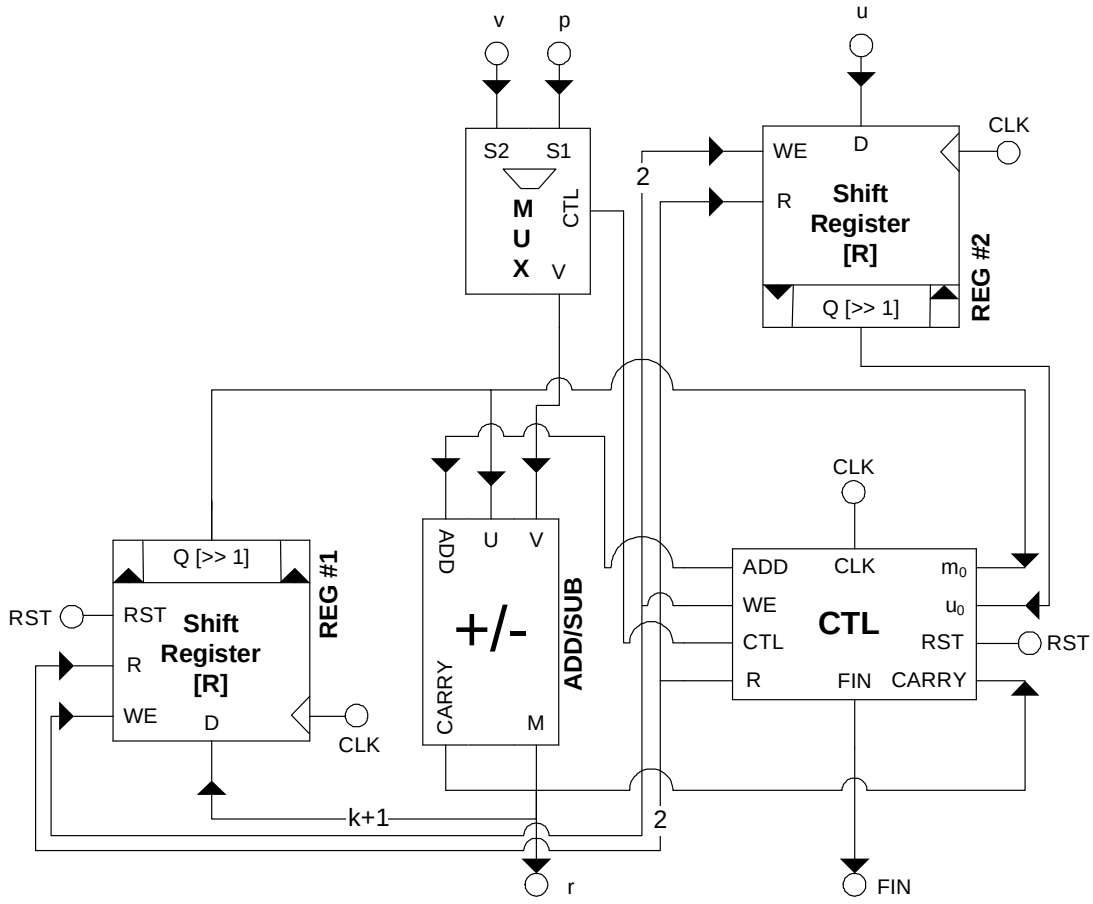

Figure 5.8.: Montgomery multiplicator (schematic design)

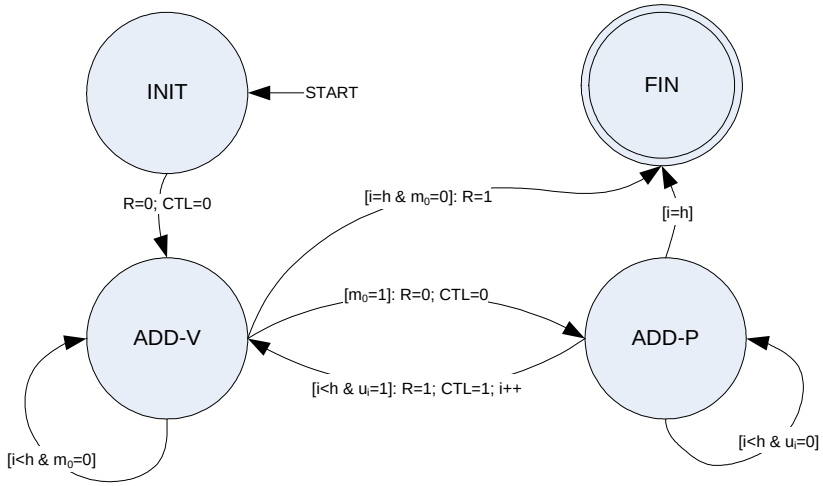

Figure 5.9.: Montgomery multiplicator (state machine)

5.5.5. Field Inversion

Commonly known, the field inversion counts for the most expensive operation in elliptic curve operation. This always calls for methods to avoid this type of operation (see Section 3.5.2). Unfortunately, those common possibilities do not apply to the MPPR, as we have to detect collisions which require unique coordinate representation. Thus, we will keep our focus on achieving the best trade-off between speed and area consumption for the inversion module.
Usually, for inverting elements the extended GCD algorithm is the most common algorithm [MvOV96]. For hardware purposes, derivatives have been developed for use with binary operations. The best known representatives are the Binary extended GCD for standard affine coordinates [MvOV96] and the Kaliski algorithm [Kal95] for values residing within the Montgomery domain. There is also another method for computing an inverse to an element in a finite field. Let $a \in \mathbb{F}_p$ be an element for which we should compute an inverse with $a \cdot a^{-1} = 1 \; mod \; p$. Then we can compute a^{p-2} because $a^{p-2} \cdot a = a^{p-1} = 1 \bmod p$ according to Fermat's Little theorem [Kob94]. But this operation will require an exponentiation engine which is even slower than a field inversion[1]. Thus, GCD-based techniques are preferred for our purpose.
Because we already have entered the realm of Montgomery coordinates by the MM, we will further favor the method by Kaliski for inversion. Kaliski has defined a sequence of two algorithms for inverting elements. The first phase called "AlmostMontgomeryInverse" computes for an element a its biased inverse $a^{-1}2^z$. The second phase is used to correct the bias and restore the Montgomery radix representation. The following code snippets will explain the details of the Kaliski inversion [DMP04].

The intermediate values r and z are then used in the second algorithm, where the desired radix representation is corrected.

Obviously, the two phases are originally designed to take an input element from standard domain. Feeding an $\tilde{a} = aR = a2^h \; mod \; p$ will lead to $r = a^{-1} \cdot 2^{z-h} \; mod \; p$ returning an incorrect result. Some authors [HMV04, DMP04] propose to compute the real inverse followed by a MM at this point to recover the Montgomery representation. But this additional multiplication will cost many unnecessary clock cycles. Thus, it is more appropriate to adapt the phase II algorithm to convert r directly to Montgomery domain.

Obviously, we retrieved a result $r = a^{-1} \cdot 2^{(z-h)} \; mod \; p$ from phase I. With our adapted phase II, we correct our intermediate value by further $2h - z$ multiplications by two. Consequently, we obtain $r \cdot 2^{2h-z} = a^{-1} \cdot 2^{(z-h)+(2h-z)} = a^{-1}2^h$ and converted the inverse back to Montgomery domain. In [DMP04] an inversion from Montgomery to Montgomery domain is assumed to take $4m+4$ cycles including the MM operation. Let us review our adapted design in terms of com-

[1] Using the binary method [MvOV96], we need to consider about $1.5k$ modular multiplications per exponentiation.

Algorithm 12 Almost Montgomery Inverse (Phase I)

Input: $a \in \mathbb{F}_p$ and p
Output: Intermediate values r and z where $r = a^{-1} \cdot 2^z \; mod \; p$ and $h \leq z \leq 2h$

1: $u \leftarrow p$, $v \leftarrow a$, $r \leftarrow 0$, $s \leftarrow 1$
2: $k \leftarrow 0$
3: **while** $v > 0$ **do**
4: **if** u is even **then**
5: $u \leftarrow u/2$, $s \leftarrow 2s$
6: **else if** v is even **then**
7: $v \leftarrow v/2$, $r \leftarrow 2r$
8: **else if** $u > v$ **then**
9: $u \leftarrow (u - v)/2$, $r \leftarrow r + s$, $s \leftarrow 2s$
10: **else**
11: $v \leftarrow (v - u)/2$, $s \leftarrow r + s$, $r \leftarrow 2r$
12: **end if**
13: $k \leftarrow k + 1$
14: **end while**
15: **if** $r \geq p$ **then**
16: $r \leftarrow r - p$ {make sure that r is within its boundaries}
17: **end if**
18: **return** $r \leftarrow p - r$

Algorithm 13 Montgomery Inverse (Phase II)

Input: $r = a^{-1} \cdot 2^z \; mod \; p$, z from Almost Montgomery Inverse and p
Output: $a^{-1} \cdot 2^h \; mod \; p$

1: **for** $i = 0$ to $(z - h)$ **do**
2: **if** r is even **then**
3: $r \leftarrow r/2$
4: **else**
5: $r \leftarrow (r + p)/2$
6: **end if**
7: **end for**
8: **return** r

Algorithm 14 Modified Montgomery Inverse (Phase II)

Input: $r = a^{-1} \cdot 2^{z-h} \; mod \; p$, z from Almost Montgomery Inverse and p
Output: $a^{-1} \cdot 2^h \; mod \; p$

```
1: for i = z to 2h do
2:     r ← 2r
3:     if r ≥ p then
4:         r ← r − p
5:     end if
6: end for
7: return r
```

plexity. From phase I, we obtain a counter z varying from h to $2h$. The counter is indeed a direct representative for the number of cycles consumed. In phase II, the current value of z is advanced from $h \leq z \leq 2h$ to a fixed $2h$, i.e. phase II takes only $2h-z$ cycle which are necessary to get to the total value of $z+(2h-z) = 2h$. Hence, the inversion has a fixed complexity of $2h$ cycles which is only the double in comparison to a MM. This very good result implicitly negates the utilization of the simultaneous-inverse technique (cf. Section 4.1.1).
To translate the inverting algorithm into hardware, we will start with an abstract model for a better overview about contributing components. By analyzing the methodology carefully, we will recognize a necessity for three concurrent arithmetic operations per clock cycle. Two are required to compute the differences of $u - v$ and $v - u$ enabling us to distinguish which case to take. At the same time, the sum of $r + s$ must be computed. Besides the arithmetics, we need to include a shifting layer, i.e. a component which is capable to perform multiplications and divisions by two (i.e. LR-shifts by one). At last, a row of multiplexers will feed the correct input into the three arithmetic units. Please note that there might be some more efficient routing options when considering not just a single layer rather than a second row of multiplexers. However, this view of efficiency only considers the saving of routing paths but covertly increases the signal propagation time when passing through a second row of components. Due to the fact that the inverter design is indeed already rather slow in terms of a maximum clocking speed, all attempts should aim onto the improvement of a short signal propagation. This finally leads us to the use of a single layer of multiplexers.
The compiled abstract design and the corresponding data flow is represented by Figure 5.10. It provides a valuable impression how the schematic design will look alike. Most parts can be adopted one-by-one but there are some additional aspects demanding a separate handling. For example, the shifting layer in Figure 5.10 will always modify values which have passed the registers and arithmetic unit. From the modified Kaliski algorithm we know that there are cases in which it is preferable to shift directly and internally (e.g. u or v is even). Thus, the shifting layer should be incorporated into the registers which are augmented with

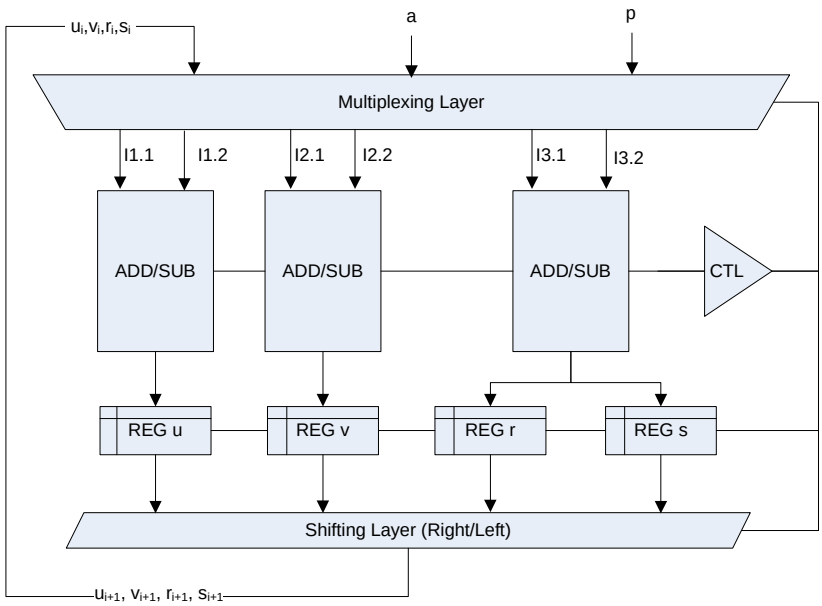

Figure 5.10.: Montgomery inversion (data flow)

a direct feedback path. This will enable an internal shift within a register without taking a route through the arithmetic units.
Further, the case differentiation in the inversion algorithm will reveal another difficulty. First, we have to decide which fork in the *if*-condition to take and secondly we need to perform the nested statements. When desiring to do this most efficiently, we should try to implement a loop iteration in a single cycle. Breaking up a single iteration into j cycles can improve the overall clocking speed of the system, but to compensate the additionally taken cycles we need to achieve a performance factor at least by j. Due to other system constraints (e.g. latency times in adder/subtracter units), this is hardly feasible. Consequently, we should consider to merge all operations per iteration in a single cycle.
Returning to the issue of chaining the case differentiation (I) and statement execution (II) in a single cycle, we will see that for (I) we need to perform the elaborate arithmetic operation $(u - v)$ and $(v - u)$, respectively. Both results are compared to determine the greater one. This test finally leads to the branching in (II), hence the controller must capable to handle the fork within a cycle. Of course, a register based controller logic is not suitable at this point due to its latency time of at least one clock cycle. Thus, additional components need to be introduced which care for the register's write enablers (WE). These components called WriteThrough controller (WT) basically consist of a multiplexer driven by some surrounding control logic. This method of in-cycle-controlling make it possible to directly act on the different outputs of the arithmetic units. Figure 5.11 shows the conversion of the abstract data model into a schematic design

encompassing the extended features described above.

Finally, we should take a closer look onto the underlying control logic. For the sake of simplicity, we will discuss only the states required for an inversion. The precise controlling for right and left shifts at the right time is rather trivial and can be obtained directly from the algorithm above. Figure 5.12 shows the state diagram for the inversion procedure displaying the main transitions.

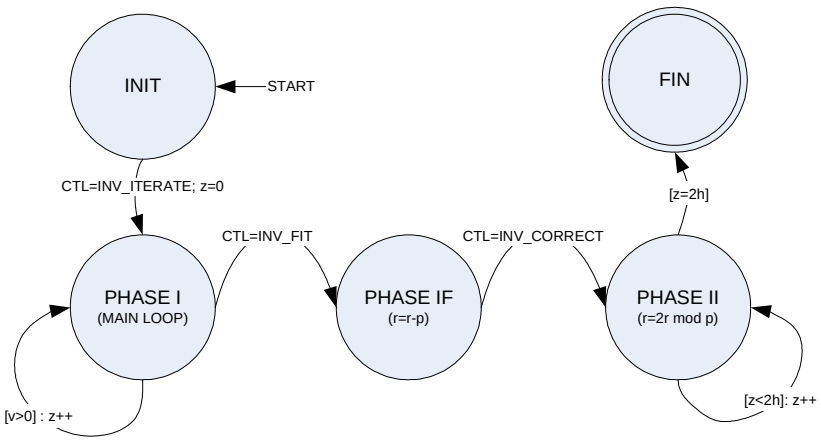

Figure 5.12.: Montgomery inverter (state machine)

5.5.6. Implementing Field Operations

With the input from previous chapters, we can now start in assembling the collected information to build a computational unit on the FPGA. We have chosen our favorite algorithms in respect to optimality in clock cycles and a maximum overlapping in FPGA area. This is a good preparation to build a single input arithmetic unit capable to perform all required operations within the same set of resources.

Having a look at the literature about similar GF(p)-AUs will recover only very little information. There are two relevant publications in this field: on the one hand the work of Örs, Batina, Preneel and Vandewalle [OBPV03] proposing a five-layered architecture with basic arithmetics using projective coordinates. Unfortunately, the internal conversion from affine to projective coordinates is rather costly, thus taking per point addition $42k + 56$ clock cycles. A better and more recent approach was made by Daly, Marnane, Kerins and Popovici [DMKP04] which also assume affine coordinates in Montgomery domain. Their results will of $5k + 9$ per point addition will be similar to the performance reached by our design. With the simple idea to overlap the number of required k-bit components per operation we can take reference to the information in Table 5.3. Be aware that the values in square brackets denote the bit width of the associated component.

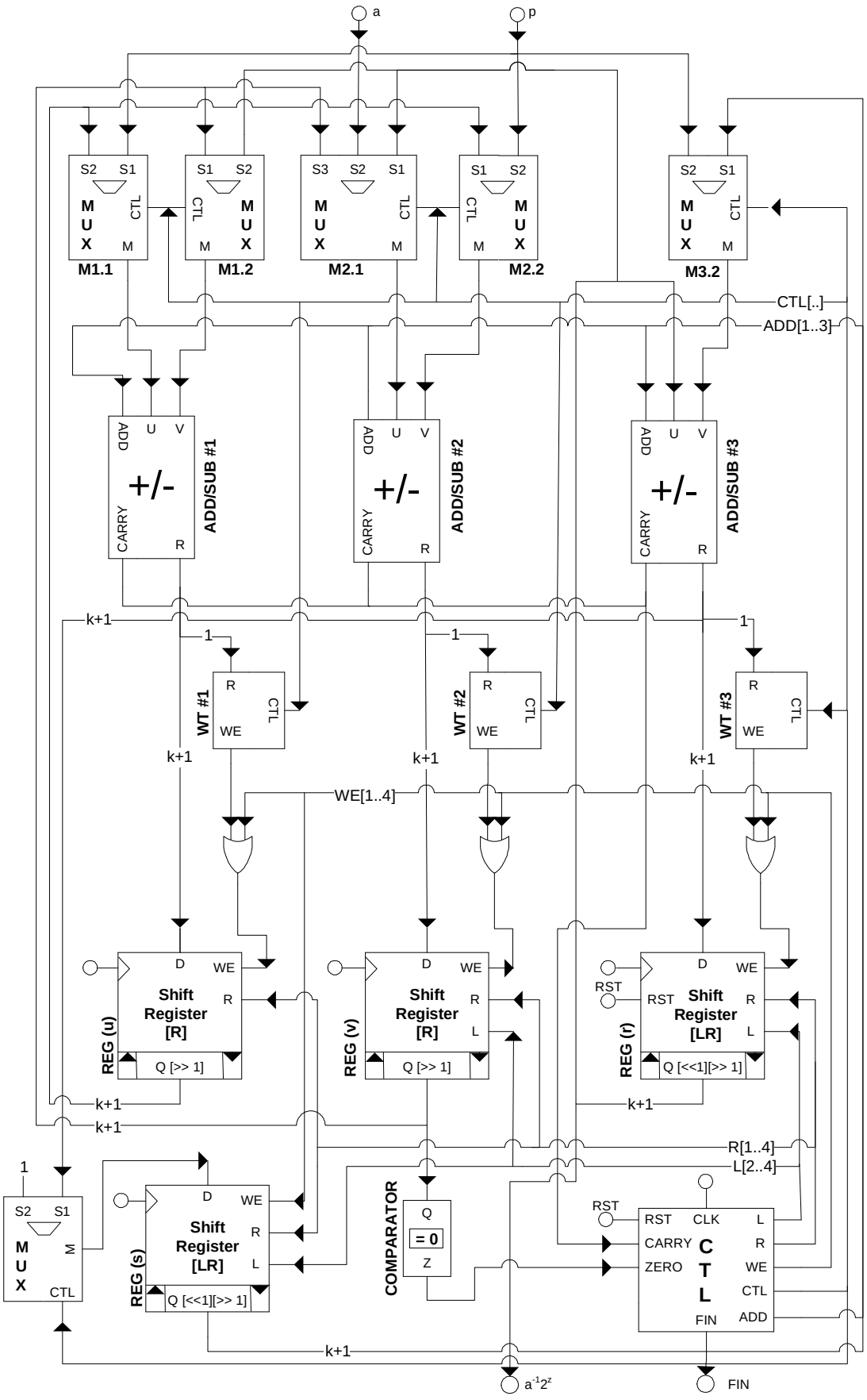

Figure 5.11.: Montgomery inversion (schematic overview)

Operation	MUX Count	ADD/SUB Count	REG Count
ADD	3 $[k]$	1 $[k]$	1 $[k]$
SUB	3 $[k]$	1 $[k]$	1 $[k]$
MUL	1 $[k+1]$	1 $[k+1]$	2 $[k+1]$
INV	5 $[k+1]$	3 $[k+1]$	4 $[k+1]$

Table 5.3.: Resource consumption of hardware operations

MUX	MUX Port	Input From	Applicable to Operation
M1.1	S1	IN1	ADD, SUB, $\text{INV}_{[INIT]}$
M1.1	S2	REG1	$\text{INV}_{[PHASE\ I]}$
M1.2	S1	IN2	ADD, SUB
M1.2	S2	REG2	$\text{INV}_{[PHASE\ I]}$
M2.1	S1	MOD	$\text{INV}_{[INIT;\ PHASE\ II]}$
M2.1	S2	REG2	$\text{INV}_{[PHASE\ I;\ FIN]}$
M2.1	S3	REG1	ADD, SUB
M2.2	S1	REG1	$\text{INV}_{[PHASE\ I]}$
M2.2	S2	MOD	ADD, SUB, $\text{INV}_{[FIN]}$
M2.2	S3	REG3	$\text{INV}_{[PHASE\ II]}$
M3.1	-	REG3	ALL
M3.2	S1	REG4	$\text{INV}_{[PHASE\ I]}$
M3.2	S2	IN2	$\text{MUL}_{[ADD-V]}$
M3.2	S3	MOD	$\text{MUL}_{[ADD-M]}$, $\text{INV}_{[PHASE\ II]}$
OUTPUT	S1	REG1	$\text{ADD}_{[0]}$, $\text{SUB}_{[1]}$
OUTPUT	S2	REG2	$\text{ADD}_{[1]}$, $\text{SUB}_{[0]}$, $\text{INV}_{[FIN]}$
OUTPUT	S3	REG3	$\text{MUL}_{[FIN]}$

Table 5.4.: Multiplexer input control table

Hence, the smallest intersection of components will determine the minimum number of components utilized for an AU. It strikes that the inversion provides by far the most elements, therefore the resulting design will probably look most alike to the presented inverter design. Besides the operational components, a subsequent output multiplexer is additionally employed to route the corresponding result from the registers to the upper layer. The final design for the Arithmetic Unit is presented by Figure 5.13. Of course, to coerce a multiplication and addition/subtraction into a substantially different design of an inverter demands some change in routing. Consequently, it should be considered to employ a minimal set and dimension of multiplexers. For a better overview, Table 5.4 will summarize the optimized signal paths through all input multiplexers. Besides the rerouted data input to multiplexers, we have to consider some inconsistencies between internal and external bit lengths. The two operands *IN1* and *IN2* as well as the modulus *MOD* which are provided by upper layers, have a bit size of

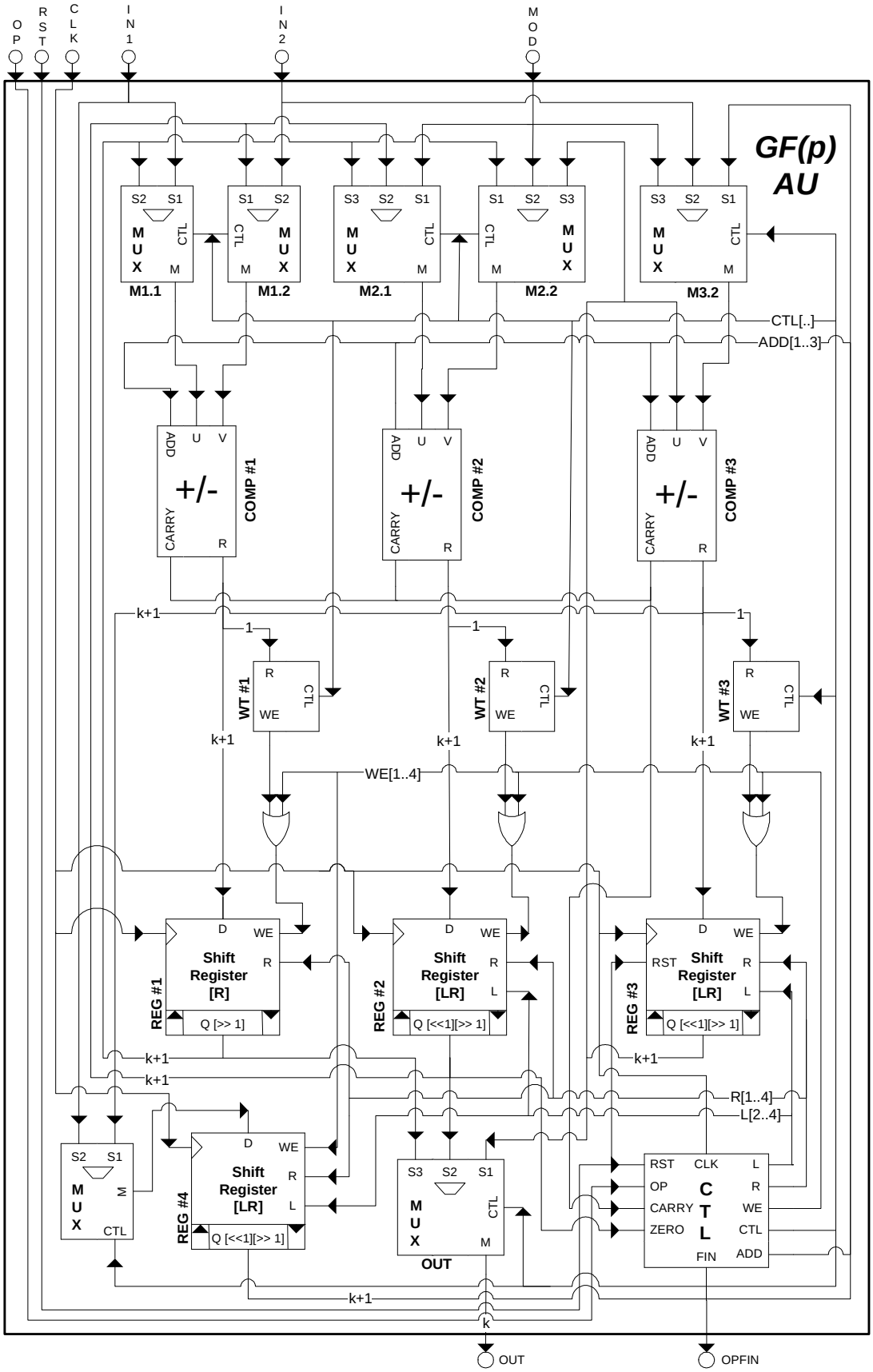

Figure 5.13.: Arithmetic layer design (schematic overview)

k. Regardless, in the arithmetic unit we use a bit size of $k + 1$ to reflecting the growing operands during multiplication and inversion. Hence, we internally have to add a leading zero to k-bit values to restore the compatibility of data ports.
Furthermore, we should take a look at the Arithmetic Controller (AC). Due to several parallel processes, it is rather difficult to merge all control statements into a single listing without losing the claim for clearness. Thus, we will omit a detailed listing of controller instructions in this chapter and refer the reader to Appendix A.1 where this is presented in full detail.
With the description of the arithmetic layer, we finally have completed the description of the essential components on one FGPA. For information about the communication between the host system and an FPGA, the interested reader will find a serial controller implementation in Appendix A.2. This controller is used to provide the physical data layer for exchange of information between the contributing parties. Especially, it is useful for debugging purposes on a development board, since the COPACOBANA design has an integrated bus-driven communication path.

6. Comparing Architectures

So far, we have discussed how to create a model for the MPPR method for general (software) implementations as well as for special-purpose hardware. Next, we will have a look at the architectures' overall performance. In Section 4.1.4, we have fully specified most of the relevant parameters with a few exceptions. Thus, we will consider our software reference implementation first to resolve those outstanding issues and use some of the provided information to define a complete testing suite. Further, we will compare the collected results of the general model to those of our hardware model.

6.1. Software Performance

Our software implementation is tested on an Intel Pentium M 735 processor running at a clock speed of 1,7 GHz. Including the costly logic for generating ECs using the Complex Multiplication method, the compiled implementation occupies roughly 187 Kb of hard disk space.

Determining the Partition Size We will now determine empirical values for the outstanding issues, i.e., we have not specified a fixed number of partitions which is supposed to provide best results. From Section 4.1.4, we should consider a suitable value for $s = 2^\sigma$ with $s = \{4, 8, 16, 32\}$ for a series of bit sizes between $k = 32, 40, 48, 56$. Although the selected bit sizes k are rather small in respect of practical bit lengths, there is no reason to assume that a best k from the chosen set will not be the best for all k. The small bit lengths have been selected, so that it is possible to break the ECDLP in a reasonably amount of time for curves of that dimensions. Please note, that a distinguished point property of 15 most significant zero bits is assumed. This finally leads to an average trail length of $1/\Theta = 1/2^{-15} = 32768$. Table 6.1 summarizes the following average results for $i = 3$ repetitions.

Obviously, the partition $s = 16$ achieves best performance for all selected bit sizes k. Hence, we will fix the value of partitions s for further use to $s = 16$ which seems to provide the best choice for a random walk.

Parameters		Average Performance		
Bit size k	**Partitions s**	**Trail Length**	**Dist. Points**	**Total Time**
32	4	25,635	6	4.15 sec
32	8	35,094	5	4.05 sec
32	16	20,356	4	**3.75 sec**
32	32	27,280	3	3.93 sec
40	4	24,042	70	13.0 sec
40	8	27,253	51	12.0 sec
40	16	20,927	27	**7.11 sec**
40	32	16,918	69	10.4 sec
48	4	21,226	1,024	137 sec
48	8	20,604	940	120 sec
48	16	22,247	520	**74.3 sec**
48	32	25,848	803	129 sec
56	4	22,993	16,115	2450 sec
56	8	32,897	8,410	1810 sec
56	16	28,983	8,797	**1610 sec**
56	32	23,483	11,955	1680 sec

Table 6.1.: MPPR software performance for different bit sizes $k = \{32, 40, 48, 56\}$ and partition sizes $s = \{4, 8, 16, 32\}$ on Pentium M@1.7GHz

Verifying Runtime Expectations Another interesting aspect is the comparison of our empirical results for the given bit sizes to the expected runtimes. Due to the very limited number of computed distinguished points ($< 2^{24}$) for the selected bit sizes, we can use (4.1) to verify our results.
Assume n to be a value with k bits, thus n will be in the range $2^{k-1} < n < 2^k$. Taking the upper and lower bound as input to Equation (4.1) and $c = 1/\Theta = 2^{15}$ and computing the total number of computed points for $s = 16$ from Table 6.1, we will encounter the results presented by Table 6.2. All in all, the empirical values seem to match the projected ones. There are some minor deviations which can be directly referred to the small number of $i_{max} = 3$ executed repetitions, e.g, the computational point count for $k = 40$ is out of the expected range in T_L and T_H. We have chosen this rather small number of samples because the time complexity for this test is growing rapidly with the number of repetitions.

k	$T_L = \sqrt{\pi 2^{k-1}}/2 + c$	$T_H = \sqrt{\pi 2^k}/2 + c$	$Mean(T_L, T_H)$	**Comp. Pts**
32	$1.07 \cdot 10^5$	$1.24 \cdot 10^5$	$1.15 \cdot 10^5$	$7.46 \cdot 10^4$
40	$7.23 \cdot 10^5$	$9.95 \cdot 10^5$	$8.59 \cdot 10^5$	$5.72 \cdot 10^5$
48	$1.06 \cdot 10^7$	$1.49 \cdot 10^7$	$1.28 \cdot 10^7$	$1.16 \cdot 10^7$
56	$1.68 \cdot 10^8$	$2.38 \cdot 10^8$	$2.03 \cdot 10^8$	$2.55 \cdot 10^8$

Table 6.2.: Deviations in empirical and estimated runtimes

Point Throughput Next, it might be useful to determine the number of computed points per second. For this purpose, all produced points are counted during the MPPR loop for a period of time $t = 20sec$. Due to the lack of efforts for intensive software optimization, we will use these figures only for getting a clue for the expressiveness of hardware performance values. Table 6.3 shows the software performance based on computed points per second for a specific bit size k.

k	**Elapsed Time (sec)**	**# Total Points**	**Points per Second**
40	20,27	3,667,465	181,000
64	20,13	2,828,323	141,000
80	20,00	2,183,735	109,000
96	21,06	2,065,726	98,100
128	21,05	1,533,239	72,800
160	20,00	1,228,050	61,400
192	20,59	1,011,931	49,100
240	23,13	934,985	40,400
256	20,86	799,804	38,300

Table 6.3.: Point additon performance on Pentium M@1.7GHz

As expected, the computational performance degrades with increasing bit sizes. Obviously, a doubling in the bit size leads to 40-50% less computed points per second. This might be surprising as the multiplication is known to have a quadratic complexity. An explanation could be the integer representation of the GMP libary which uses several "limbs" to fit portions of a multi-precision values into arithmetic registers. In case of our Pentium M we can expect up to 4 limbs (for $k = 256$) each with 64 bits to be handled by the Pentium's MMX unit. Due to this small number of limbs and resulting inner multiplications, the additional effort to perform a multiplication for a larger bit size does not have such a significant impact on the overall performance as one could expect. The loss in performance is rather driven by the inversion where the extended GCD algorithm with logarithmic complexity needs to be taken into account [MvOV96]. Both, the main influence which can be traced back to the inversion and the minor impact for multiplications result in a nearly linear change of the overall performance for increasing bit sizes. This is visually presented in Figure 6.1.

6.2. Hardware Performance

For estimating the hardware performance, we first need some additional information. In contrast to the Pentium M 735 running at a fixed clock speed of 1.7GHz, the clocking of FPGAs is subject to propagation delays and varies with the bit size k. Furthermore, it is much more accurate to determine an exact number of required clock cycles which will directly result in a number of computed points

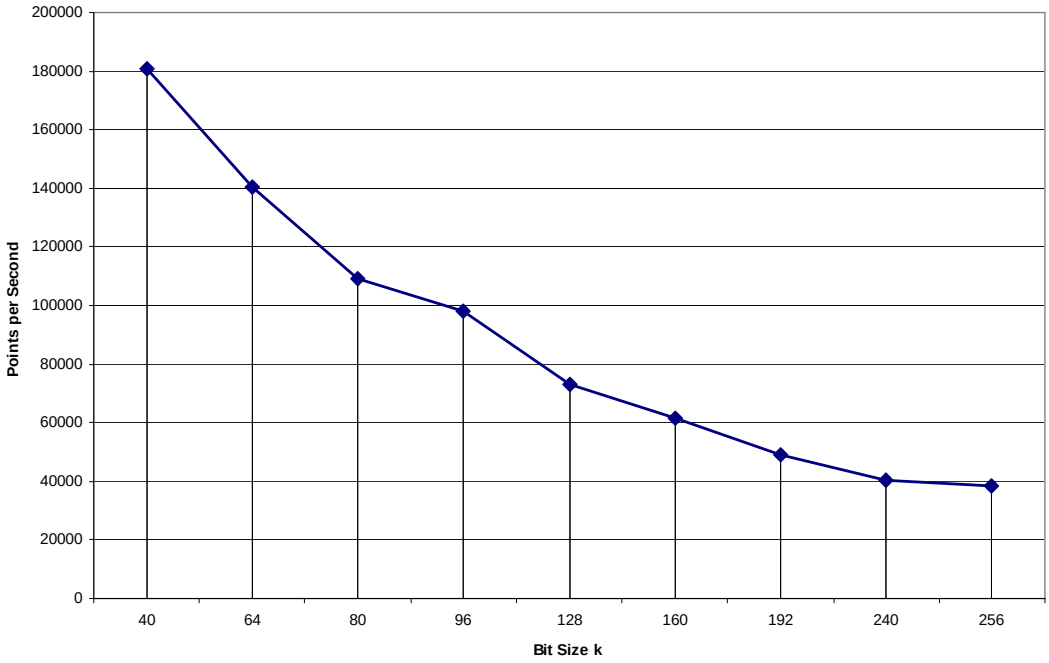

Figure 6.1.: Point throughput of Pentium M@1.7GHz

per second. But this obviously requires knowledge about precise execution times and number of field operations. To address this issue, the following listing will recall our achievements with respect to the required cycle count per operation (see Appendix A.1).

- **Field Addition**. Due to the bit-parallel design of our modular addition operation, we need a single cycle $T_{ADD} = 1$ for computation. In addition to this, two cycles for resetting ($T_{RST} = 2$) and another for signaling its termination ($T_{FIN} = 1$) need to be spent.

- **Field Subtraction**. Because it is logically similar to the addition, we will charge $T_{SUB} = 1$ as well for the subtraction. Of course, additional $T_{RST} = 2$ and $T_{FIN} = 1$ cycles need to be considered.

- **Field Multiplication**. According to the complexity analysis in section 5.5.4, we can assume $T_{MUL} = k + 2 = h$ as an expected average runtime. This figure yet excludes $T_{RST} = 2$ and $T_{FIN} = 1$ cycles for preparation and termination.

- **Field Inversion**. The complexity of a modified Kaliski inversion has been determined to the fixed complexity of $T_{INV} = 2(k+2) = 2h$. Again, further $T_{RST} = 2$ and $T_{FIN} = 1$ cycles are required for administration.

Setting the complexities into context to the number of operations for a single iteration in MPPR, we can compute an exact number of cycles. With reference to Section 3.5.1, an elliptic curve point addition in affine coordinates can be performed using 6 subtractions, 3 multiplications and a single inversion. With the updating of coefficients, demanding another two additions and some administrative cycles, we can finally compile a list for the total cycle consumption of a single MPPR iteration shown in Table 6.4.

Context	**Operation Type**	**Count**	T_{CMP}	T_{RST}	T_{FIN}
Administration	Partitioning	1	1/2	-	-
Administration	DP Check	1	1/2	-	-
Point Addition	Subtractions	6	6	12	6
Point Addition	Multiplications	3	$3h$	6	3
Point Addition	Inversion	1	$2h$	2	1
Coefficient Update	Addition	2	2	4	2
Total	-	14	$5h + 9$	24	12

Table 6.4.: Required cycles for one MPPR iteration with $h = k + 2$ and bit size k

Regardless of a small constant, we achieve a similar performance compared to the architecture in [DMKP04]. Combining the required clock cycles for resetting, finishing, and the actual computation time T_{CMP} directly provides the effective number of cycles for a single iteration of an MPPR core. Hence, we can assume for further analysis the following straightforward complexity function:

$$T(k) := T_{CMP}(k) + T_{RST} + T_{FIN} = 5h + 45 = 5k + 55 \text{ with } h = k + 2. \quad (6.1)$$

Synthesis Results With the knowledge of the performance of a single core, we can now synthesize the design to estimate the required area for various bit sizes k. Those figures immediately determine the maximum number of cores fitting on one FPGA. The Table 6.5 shows the area and clocking constraints for a Xilinx SPARTAN-3 XC3S1000, providing 17,280 logic cells in 7,680 slices with a variable number of cores. Please notice that the figure presenting the number of used slices as well includes all top-layer components for management and communication.

It strikes that architectures with a device usage close to 100 % have significantly longer signal propagation periods than others with some unused FPGA area (e.g. see $k = 128$ or $k = 96$ with 4 cores). This effect can be directly traced back to the more condensed packaging which leaves only little flexibility for extensive route optimizations after synthesis. Hence, longer signal paths need to be taken into account, leading to a slower maximum clocking speed.

Further, we have as well tested and synthesized the MPPR on the Xilinx SPARTAN-3 XC3S200. Here, the maximum number of 1,920 available slices is a limiting factor. This allows only the utilization of a single core with a maximum bit size of $k = 80$, resulting in a device usage of 99% what implies rather

k	**Cores**	**Min. Period**	**Max. Clock**	**Slices**	**Usage**	**Slices/Core**
160	2	25.000 ns	40.000 MHz	6,450	83 %	3,230
128	3	24.900 ns	40.100 MHz	7,561	98 %	2,520
96	3	20.800 ns	48.100 MHz	6,091	79 %	2,030
96	4	22.600 ns	44.300 MHz	7,564	98 %	1,890
80	4	19.600 ns	50.900 MHz	6,820	88 %	1,710
64	4	18.200 ns	54.800 MHz	5,789	75 %	1,450
64	5	19.200 ns	52.000 MHz	6,817	88 %	1,360

Table 6.5.: Area and timing constraints on XC3S1000 FPGA

bad routing delays. Hence, we will only focus on more expressive figures for the larger XC3S1000 FPGA for our analysis.

Point Throughput With our complexity function in (6.1), it is rather straightforward to derive the performance details for the selected architectures. This includes a precise statement about the number of required cycles per iteration as well as the number of points per second.

k	**Cores**	T_k	**Time per Pt**	**Pts per Core/sec**	**Total Pts/sec**
160	2	855	21.400 μs	46,800	93,600
128	3	695	17.300 μs	57,800	173,000
96	3	535	11.100 μs	90,000	270,000
96	4	535	12.100 μs	82,700	331,000
80	4	455	8.940 μs	111,900	447,000
64	4	375	6.840 μs	146,200	585,000
64	5	375	7.210 μs	138,600	693,000

Table 6.6.: Performance of MPPR on XC3S1000 FPGA

It must be mentioned that all figures above exclude any additional cycles for writing to the distinguished point buffer, respectively the time for waiting for a granted buffer lock. Computing the discrete logarithm of a smaller EC, the buffer locking might become a bottleneck, especially when deploying larger numbers of cores on an FPGA. This can easily be avoided by reducing the size of the distinguished point set D. In direct consequence, this will produce longer search trails for distinguished points and collisions of two or more cores requesting a buffer lock at the same time will become a rather rare event.

AT-Considerations An outstanding issue is the rating of a reached optimization level in terms of the AT-product. Unfortunately, all possible variations from our currently presented architecture which have been in disposition throughout

this work, have been identified providing only a negative influence. This includes the application of operation pipelining, simultaneous inversions, word-wise adders and subtracters as well as the additional unit for squaring. Thus, we now have an architecture whose AT-product can be considered as optimal for our purpose, although we do not have a direct relation. Defining the AT-product for this work as the product of required time in μs multiplied with the number of used slices per core will lead us to the compilation of AT-products as shown in Table 6.7. We can state that a doubling in a bit size k will roughly increase the AT-product by a factor $f = 4$. An error of this estimation can directly referred to the changing weight of the management logic which has been averaged over the number of available cores. The smaller the total number of cores, the greater the per-core-share of management logic, which directly influences the AT-product in a negative manner.
Unfortunately, we can not give further statements with respect to a "best" AT-product at this point because increasing bit sizes will obviously result in increased execution times *and* a larger demand for area on the FPGA. At this point, a comparison will only make sense referring to the AT-products of different architectures using the same parameters. Due to the absence of further MPPR implementations in hardware, we are obliged to skip this consideration.

k	**Time per Pt**	**Used Slices/Core**	**AT-Product**
160	21.400 μs	3,230	68,900
128	17.300 μs	2,520	43,600
96	12.100 μs	1,890	22,900
80	9.000 μs	1,710	15,200
64	7.210 μs	1,360	9,830

Table 6.7.: AT-products of the MPPR processor on an XC3S1000 FPGA

Now we will investigate the performance on the COPACOBANA architecture. The previous tables present figures stating the performance on a single XC3S1000 FPGA only. Let us assume the computational power of 120 simultaneous computing XC3S1000 FPGAs, as in case with COPACOBANA. In the following, we will estimate the expected performance since the COPACOBANA is not yet fully functional.
We need to ensure that each contributing FPGA does not interfere too often with others when transmitting distinguished points to the server. As previously described, this can easily be guaranteed by decreasing the size of the set D, e.g., by demanding $\delta > 16$ bits to be zero to satisfy this constraint. An appropriate choice of δ will cause a core to compute several minutes until distinguished point is found and need to be sent to the server. For example, a value of $\delta = 24$ producing an average trail length of about $\Theta = 1/2^{-24} = 16777216$ will require a core for a bit length of $k = 160$ to compute for about 6 minutes until a suitable

point is found. Having 240 cores on the COPACOBANA, this will result in one point to be transmitted on average every 1.5 seconds. Of course, this situation can further be relaxed by simply increasing δ. With the previous discussion, the following performance values (Table 6.8) can be expected from a run on the COPACOBANA platform.

k	**Cores per FPGA**	**Total Cores**	**Computed Pts/sec**
160	2	240	$11.2 \cdot 10^6$
128	3	360	$20.8 \cdot 10^6$
96	3	360	$32.4 \cdot 10^6$
96	4	480	$39.7 \cdot 10^6$
80	4	480	$53.7 \cdot 10^6$
64	4	480	$70.2 \cdot 10^6$
64	5	600	$83.2 \cdot 10^6$

Table 6.8.: Estimated performance for MPPR on a COPACOBANA in full expansion stage (120 XC3S1000 FPGAs)

6.3. Comparing Software and Hardware Performance

This chapter is dedicated to compare the software and hardware performance of the architectures presented in this thesis. A direct line-up of the simple reference software implementation against the optimized hardware architectures should be regarded with reasonable skepticism. An optimized software implementation for a better comparison might be provided by the work of Brown, Hankerson, Lopez, and Menezes [BHLM01], using a Pentium II 400 MHz workstation. Unfortunately, this work considers only NIST fields and relies on ancient hardware, thus it might be more adequate for assessment to use our inelegant software reference. Using a correction factor of 25% which might be achieved by using assembler code and an inherently better design will place our software solution in more competitive position against its relative in hardware. Table 6.9 faces the corrected performance values of the software implementation with the best hardware figures, i.e. using a maximum number of cores fitting on one Xilinx XC3S1000. At this point, it should be remarked that the software and hardware comparison will only give reasonable figures when comparing single chips. Hence, it relates underlying hardware where an Intel Pentium M with 140 million transistors against a single XC3S1000 FPGA (1 million gates) is taken into account. Of course, a direct lineup of hardware logic is not absolutely precise since different portions of transistors are used for cache and memory on both chips and are not available for boolean computations. However, for our comparison we will set

the number of transistors in direct relationsship which will emphasize the benefit of using special-purpose hardware. This brings in another bias in favor of our FPGA architecture.

k	**XC3S1000 FPGA**	**Pentium M@1.7GHz (+25%)**	**HW/SW-Ratio**
160	93,600 pts/sec	76,800 pts/sec	1.22
128	173,000 pts/sec	91,000 pts/sec	1.90
96	331,000 pts/sec	123,000 pts/sec	2.70
80	447,000 pts/sec	136,000 pts/sec	3.28
64	693,000 pts/sec	176,000 pts/sec	3.95

Table 6.9.: Software vs. hardware performance rating

Cost Analysis We have related the software and hardware performance with respect to their throughputs in point computations. Now, we should take the monetary aspects into account. This monetary aspect is of central importance if we want to evaluate the actual threat for real-world ECC implementations. In January 2006, a Pentium M 735 processor costs about US$ 220 whereas the Xilinx SPARTAN-3 XC3S1000 can be purchased for US$ 50 per chip[1]. Additionally, we need a housing and peripheral interconnection for each chip.
For the sake of simplicity, we will take the existing COPACABANA architecture for embedding the FPGAs which provides an integrated platform to manage up to 120 FPGAs of type XC3S1000. For such a machine, the authors report the costs for construction to be less than US$ 10,000 [KPP+06].
Taking this amount of US$ 10,000 as reference, we can estimate a corresponding number of workstations with Intel Pentium M processors. Assuming additional US$ 180 for a housing, a mainboard, main memory and a physical storage facility (such as a hard disk drive or a flash memory device for the operation system), we can assess the costs for a single station with about US$ 400. This corresponds to 25 workstations which can be related to a single COPACOBANA machine in terms of asset costs. Of course, there are further aspects, e.g. the power consumption which is left out of consideration at this point. Table 6.10 and Figure 6.2 emphasize the relationship with respect to the computational power for a US $ 10,000 MPPR-attack in terms of software and hardware implementations.

Clearly, the hardware architectures are much more dependent on the varying bit size than a software implementation. This effect is even enhanced with the employment of several cores, becoming available with smaller bit sizes. Although an additional core on the FPGA might slightly decrease the maximum clock speed of the entire system, this negative factor is instantly overwhelmed by the additional computational power provided by this extra point processor.

[1]Reseller prices.

k	**25 Pentium M@1.7GHz**	**COPACOBANA**(120 XC3S1000)
160	$1.92 \cdot 10^6$ pts/sec	$11.2 \cdot 10^6$ pts/sec
128	$2.28 \cdot 10^6$ pts/sec	$20.8 \cdot 10^6$ pts/sec
96	$3.07 \cdot 10^6$ pts/sec	$39.7 \cdot 10^6$ pts/sec
80	$3.41 \cdot 10^6$ pts/sec	$53.7 \cdot 10^6$ pts/sec
64	$4.39 \cdot 10^6$ pts/sec	$83.2 \cdot 10^6$ pts/sec

Table 6.10.: MPPR hardware vs. software performance for a US$ 10,000 attack

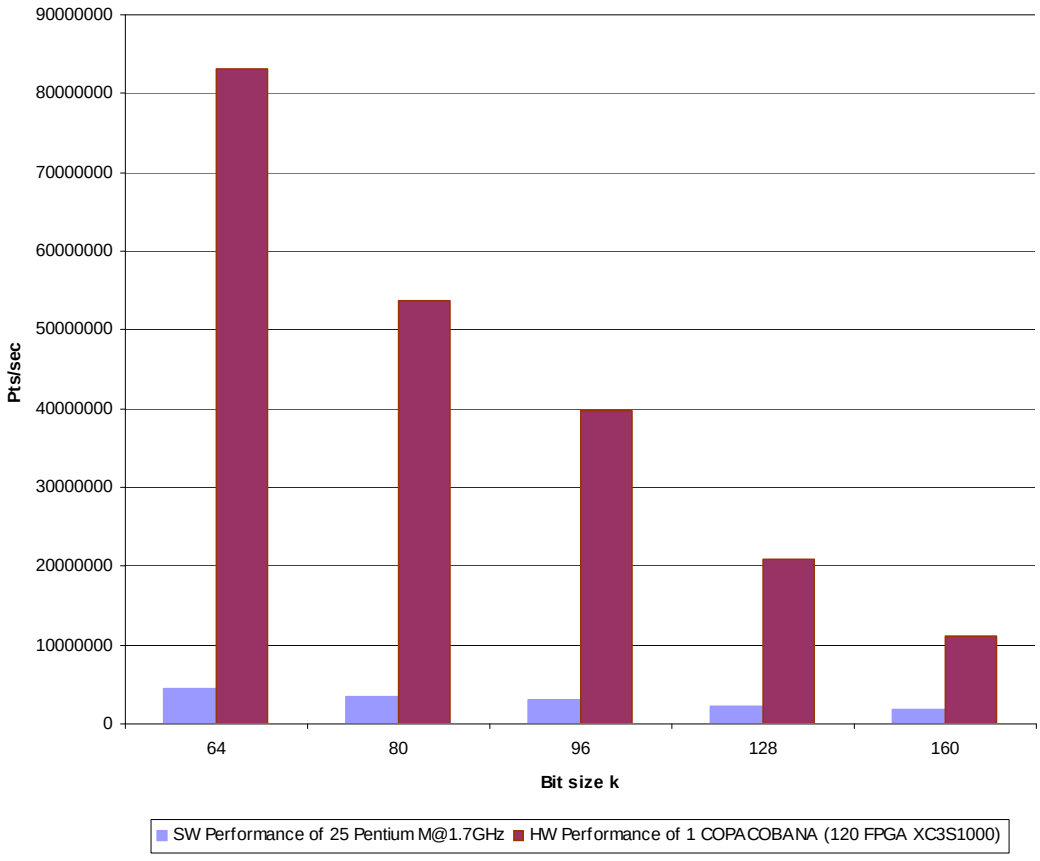

Figure 6.2.: MPPR hardware vs. software performance for a US$ 10,000 attack

6.4. Projected Runtimes for MPPR

To give some clues about the expected runtime to break ECC over GF(p) with a specified bit length k, we will regard the following three architectures. First, we have our reference software representing a single off-the-shelf Pentium M 735. Secondly, we will estimate the performance for a single XC3S1000 FPGA architecture on a development board. Last, we will give statements for expected runtimes on the COPACOBANA system. Projections are performed for bit lengths $k = \{64, 80, 96, 128, 160\}$ using a distinguished point proportion of $\Theta = 2^{-24}$.
Using Equation (4.1), we compute the expected number of points for a specified bit length to determine the ECDLP of an associated curve. We disregard the negative effect of point overwriting at this point and, therefore, use the simplified estimation formula. This is useful to examine at which bit lengths the negative influence becomes significant. Table 6.11 displays an upper (T_H) and lower (T_L) computational boundary and their associated expected number of points to be stored on the server (cf. S_H and S_L).

k	$T_L = \sqrt{\pi 2^{k-1}}/2 + c$	$T_H = \sqrt{\pi 2^{k}}/2 + c$	$S_L = \Theta \cdot T_L$	$S_H = \Theta \cdot T_H$
64	$2.71 \cdot 10^9$	$3.82 \cdot 10^9$	$1.61 \cdot 10^2$	$2.28 \cdot 10^2$
80	$6.89 \cdot 10^{11}$	$9.74 \cdot 10^{11}$	$4.11 \cdot 10^4$	$5.81 \cdot 10^4$
96	$1.76 \cdot 10^{14}$	$2.50 \cdot 10^{14}$	$1.05 \cdot 10^7$	$1.49 \cdot 10^7$
128	$1.16 \cdot 10^{19}$	$1.64 \cdot 10^{19}$	$6.89 \cdot 10^{11}$	$9.74 \cdot 10^{11}$
160	$7.58 \cdot 10^{23}$	$1.07 \cdot 10^{24}$	$4.52 \cdot 10^{16}$	$6.39 \cdot 10^{16}$

Table 6.11.: Estimated MPPR runtimes and produced distinguished points

Assuming a geometrical distribution of distinguished points we can show that only negligible effects from point overwriting are to be expected for bit sizes ≤ 96 bits. This statement is true for a provided server storage size of $2^{24} \approx 1.678 \cdot 10^7$ points.
The alignment of our architectures to the expected numbers of points to compute will reveal their performance in respect to the assigned issue and bit length, respectively. The Table 6.12 presents the results for a expected computation time for a single machine/chip. The figures are determined by computing the mean from the expected total number of required points for solving the ECDLP.

Obviously, a successful computation for $k > 96$ seems to be unrealistic with any of the presented architecture when considering only a single machine. This is emphasized by the visual impression in Figure 6.3. Here, the data from above is set into context of logarithmic complexity.

Single Month Attack The number of days required for computation can only be cut by massive employment of further machines. We define a specific time constraint to be met and count the necessary machines. Following the attempt

k	$Mean(T_H, T_L)$	**Pentium M@1.7GHz**	**XC3S1000**	**COPACOBANA**
64	$3.25 \cdot 10^9$	5.14 h	78.1 min	39.1 sec
80	$8.32 \cdot 10^{11}$	70.5 d	21.5 d	4.30 h
96	$2.13 \cdot 10^{14}$	55.1 y	20.4 y	62.1 d
128	$1.40 \cdot 10^{19}$	$4.86 \cdot 10^6$ y	$2.55 \cdot 10^6$ y	$2.14 \cdot 10^4$ y
160	$9.15 \cdot 10^{23}$	$3.78 \cdot 10^{11}$ y	$3.10 \cdot 10^{11}$ y	$2.58 \cdot 10^9$ y

Table 6.12.: Expected runtimes for the presented architectures

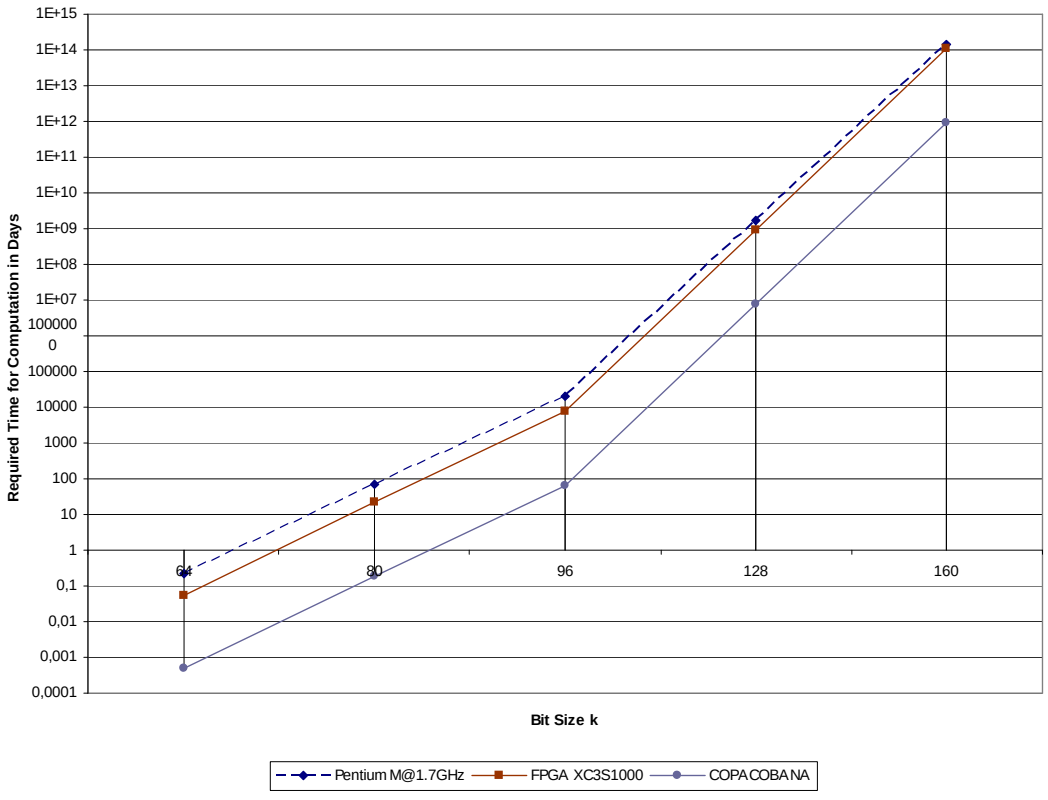

Figure 6.3.: Projected computation times for presented architectures

to break ECC with a bit length of k in a maximum of one month (30 days) will lead us to the numbers represented by Table 6.13.

k	# Pentium M@1.7GHz	# XC3S1000	# COPACOBANA
80	3	1	1
96	670	249	3
128	$59.1 \cdot 10^6$	$31.0 \cdot 10^6$	$259 \cdot 10^3$
160	$4.60 \cdot 10^12$	$3.77 \cdot 10^12$	$31.4 \cdot 10^9$

Table 6.13.: Number of required machines for a single month attack

All in all, with our optimum hardware design it will take at least about 31.42 billion machines of type COPACOBANA for performing an attack on a $k = 160$ bit EC within a month. Please be aware that our estimation has yet excluded any negative effects due to memory constraints. In respect of monetary aspects, we can estimate a single month attack on ECC with $k = 160$ bit to dispatch about US\$ $3.14 \cdot 10^{14}$. This is really an impressive fact for the resistance of Elliptic Curves against known attacks on special-purpose hardware.

Security of RSA vs. ECC For a comparison with RSA, we can cite Shamir and Tromer which optimistically estimated a factorization of RSA1024 to cost about US\$ 10 million and last about one year [ST03]. Another more realistic RSA security consideration has been published by Franke, Kleinjung, Paar, Pelzl, Priplata, and Stahlke [FKP+05] who assumed an one-year attack to cost about US\$ 200 million. According to the wide-spread opinion of [LV01], an RSA cryptosystem with $k = 1024$ bits is considered to provide a similar security like an ECC based system with only $k = 160$ bits.
Adapting our figures to a similar setting as in [ST03, FKP+05], we obtain a demand for at least 2.58 billion COPACOBANA machines to perform an attack on ECC with $k = 160$ bits in one year. This leads to an expense of US\$ $2.58 \cdot 10^{13}$ which is still far beyond any of the two estimations for the security of RSA1024. As a consequence, we can assume ECC with $k = 160$ bits to provide a greater security than RSA with $k = 1024$ bits when drawing a balance with respect to monetary aspects.

ECC Challenges Since 1997, the Mississauga company Certicom has issued a challenge: A list of elliptic curves and associated ECDLP parameters which should be broken with computational power to prove the security of ECC [Cer06]. For EC over GF(p), exercises for following bit sizes k have been defined with $k = \{79, 89, 97, 109, 131, 163, 191, 239\}$. Certicom has given some estimation of machine days required for solving each challenge. Consequently, the issue arises how fast our architectures are with respect to the challenges and figures provided

by Certicom. Certicom uses for their runtime estimations an Intel Pentium 100 as reference machine. Table 6.14 will summarize our findings:

k	**Est.** [Cer06]	**Pentium M@1.7GHz**	**XC3S1000**	**COPACOBANA**
79	146 d	49.0 d	15.3 d	3.06 h
89	12.0 y	4.64 y	1.62 y	4.93 d
97	197 y	74.7 y	30.7 y	93.4 d
109	$2.47 \cdot 10^4$ y	$5.57 \cdot 10^3$ y	$2.91 \cdot 10^3$ y	24.3 y
131	$6.30 \cdot 10^7$ y	$1.40 \cdot 10^7$ y	$7.40 \cdot 10^6$ y	$6.17 \cdot 10^4$ y
163	$6.30 \cdot 10^{12}$ y	$1.09 \cdot 10^{12}$ y	$9.15 \cdot 10^{11}$ y	$7.62 \cdot 10^9$ y
191	$1.32 \cdot 10^{17}$ y	$2.17 \cdot 10^{16}$ y	$1.89 \cdot 10^{16}$ y	$1.57 \cdot 10^{14}$ y
239	$3.84 \cdot 10^{24}$ y	$4.44 \cdot 10^{23}$ y	$8.62 \cdot 10^{23}$ y	$7.18 \cdot 10^{21}$ y

Table 6.14.: Expected runtimes for Certicom challenges [Cer06]

Considering the current Certicom Challenge of $k = 131$ bits for ECs over GF(p), we can estimate a required computational power of roughly at least 62,000 COPACOBANA machines for solving the ECDLP within a year. Unlike $k = 160$, this is also an enormous but not an absolutely infeasible amount of computational power.
Analyzing our performance for the last solved challenge with $k = 109$ bits, we can state that about 300 COPACOBANA machines are already sufficient to break this ECDLP in only 30 days. This single-month attack on ECC with $k = 109$ bits can be realized with about US$ 3 million.

6.5. Estimating an ASIC Design for MPPR

Before closing this chapter, we would like to estimate the performance ASIC chip design based on our FPGA architecture. Let us assume that we are able to create an ASIC chip encompassing 10 million transistors. This is a tenfold increase of available hardware area with respect to the XC3S1000 FPGA. Due to the different internal layout, we can expect to run such a chip at clock speeds far beyond those of FPGA's. A not unrealistic estimation is a maximum clock frequency of 500 MHz. These design parameters denote a computational power which might even exceed the COPACOBANA. Table 6.15 will give a rough figure about projected runtime for such a virtual chip design to break ECC over GF(p) with k-bits.

With respect to our projections for the COPACOBANA from Section 6.4, we will encounter an slight increase of computational power for this fictive processor. Assuming this high-performance chip to cost only US$ 50 per unit including overhead in high-volume production, we can estimate the runtimes for MPPR attacks with respect to different financial considerations. Table 6.16 shows the estimated cost-performance relationship for our ASIC design. Please note that

k	# Cores	ASIC Performance (pts/sec)	Expected Runtime
64	50	$66.6 \cdot 10^6$	49.0 sec
80	40	$44.0 \cdot 10^6$	5.26 h
96	40	$37.4 \cdot 10^6$	65.9 d
128	30	$21.6 \cdot 10^6$	$2.05 \cdot 10^4$ y
160	20	$11.7 \cdot 10^6$	$2.48 \cdot 10^9$ y

Table 6.15.: MPPR performance estimates for an ASIC design with $10 \cdot 10^6$ transistors running at 500 MHz

these figures only represent the asset costs and exclude any expenses for operation. We can state that we still need to accept an expected computational time of more

	Expected runtimes in years for attacks with			
k	US\$ $100 \cdot 10^3$	US\$ 10^6	US\$ $10 \cdot 10^6$	US\$ $100 \cdot 10^6$
128	$1.03 \cdot 10^1$	$1.03 \cdot 10^0$	$1.03 \cdot 10^{-1}$	$1.03 \cdot 10^{-2}$
160	$1.24 \cdot 10^6$	$1.24 \cdot 10^5$	$1.24 \cdot 10^4$	$1.24 \cdot 10^3$
192	$9.64 \cdot 10^{10}$	$9.64 \cdot 10^9$	$9.64 \cdot 10^8$	$9.64 \cdot 10^7$
256	$1.09 \cdot 10^{21}$	$1.09 \cdot 10^{20}$	$1.09 \cdot 10^{19}$	$1.09 \cdot 10^{18}$

Table 6.16.: Cost-performance consideration for MPPR attacks with $10 \cdot 10^6$ transistors per ASIC running at 500 MHz

than 1200 years with an expense of US\$ 100 million for an MPPR attack. If we relate this to the context of RSA1024, we need to afford US\$ $1.24 \cdot 10^{11}$ in order to expect a successful termination of an MPPR attack within a single year. This is still far beyond the estimations for the corresponding RSA1024 attack (with a factor of about $620 : 1$).

7. Discussion

Finally, we like to summarize our findings of this thesis. Furthermore, open issues and unconsidered aspects will be stated in Section 7.2 to point out the scope for future research.

7.1. Conclusions

This thesis was dedicated to demonstrate the security of ECC over GF(p) against efficient attacks implemented in hardware. For this purpose, we have proposed a scalable architecture, realizing a point processing core for the MPPR method what is supposed to be the most efficient attack on ECC to current knowledge. Compared to [OBPV03], we have incorporated an underlying arithmetic unit for curve operations which is faster and uses less hardware resources. Furthermore, the runtime performance of our AU is superior considering the implementation of [DMKP04] who present their results based on a larger and better Virtex-2 FPGA.
We need to remark that our AU design was primarily focused on the optimal implementation of an entire MPPR system what consequently implies a reduced comparability of subcomponents. Although we have logically separated the arithmetic unit in our design, we still encounter interferences and additionally required logic for interaction with upper layers. This distorts a direct comparison with other units which are optimized for acting as a stand-alone implementation. Furthermore, area considerations are affected by the employment of different FPGA types. For example, the utilization of a low-cost SPARTAN-3 XC3S1000 FPGA which is inferior to the Virtex-2 XC2V2000 chip used in [DMKP04], will lead to worse expected routing results, e.g., caused by the shorter maximum column paths. This and further aspects can cause the same implementation to a require an increased numbers of slices on less efficient FPGAs.
Our findings based on the performance of arithmetic units and our Pollard-Rho core are presented in Table 7.1.

Regarding the attack performance, we have already achieved very good results with a single FPGA in contrast to a software implementation. Using a scaling according to asset costs of US$ 10,000, we have come up with a balanced performance ratio for our architectures between roughly $6:1$ for $k = 160$ and $19:1$ for $k = 64$ bits in favor of our hardware design. Applying this benefit to a COPACOBANA machine finally leads us to our final conclusion. A single

Implementation	**Time per EC Addition**	**Used Slices**
Arithmetic Unit by [OP01]	62.000 μs	n/a for $k = 160$
Arithmetic Unit by [OBPV03]	74.000 μs	3,480 (extrapolated)
Arithmetic Unit by [DMKP04]	20.230 μs	1,854
Our Arithmetic Unit	19.995 μs	2,664
Our PRCore	21.400 μs	3,230[1]

Table 7.1.: Design parameters for arithmetic units with $k = 160$ bits

COPACOBANA machine can be expected to take less than 4h 20min to solve the ECDLP for an EC with $k = 80$ bits. However, it takes expected 2.58 billion years to break an EC with $k = 160$ bits. Other way around, for an attack based on a multitude of machines which should last less than a single month, we will need to deploy about 31.42 billion machines or about $3.77 \cdot 10^{12}$ single FPGAs of type Xilinx XC3S1000. Based on current knowledge and the COPACOBANA architecture, an MPPR attack which is expected to be successful within one year would dispatch asset costs of about US$ $2.58 \cdot 10^{13}$, excluding additional costs for storage and electric power. This can currently be considered as unfeasible with available financial resources. Furthermore, it exceeds by far estimations for breaking RSA cryptosystems. Compared to [ST03] and [FKP$^+$05] assuming a successful RSA1024 attack in one year to cost US$ 10 million and US$ 200 million, respectively, we can attest ECC over GF(p) with $k = 160$ bits to be much more secure than its assumed RSA relative with $k = 1024$ bits.
Further refinement of our architecture might lead to an improvement in performance. But with great certainty, such changes will not draw general ECs over GF(p) with bit lengths of $k \geq 160$ into the zone of weak cryptosystems.

7.2. Future Work

Due to limitations in time, the scope of our work was limited. This encompasses several ideas which might be promising for further research projects. The following list will give a small number of yet unconsidered further options.

- **Further Improvements of the Arithmetic Unit.** It has become clear that the performance of the MPPR directly depends on the area and timing constraints of its underlying arithmetics. Thus, further improvements in this field can have a significant reduction of the computational time. This might involve especially a closer look to the inverter design which can be considered as most influential. Here, some ideas like the integration of the work of [dDBQ04] and [Gut02] might be useful.

[1] Besides the AU, this includes additional overhead for the MPPR logic, main memory, command processing and communication components

- **Using Projective Coordinates.** Our attack was based on affine coordinates due to the ambiguity of projective coordinates in respect to the distinguished point determination. In case that a new criterion for projective elements can be identified, this will eliminate the utilization of the costly inverter and allows a new design running at higher clock speeds.

- **Further Improvements to the Hardware.** A very self-evident idea is the evolution of underlying hardware machines. Using a new COPACOBANA v2.0 capable to provide a housing for a number of PRCores far beyond of the current design. For example, with new manufacturing techniques and better FPGAs, we can employ more cores on a single chip. This will further decrease expected runtimes for MPPR.

- **Finding Trail Shortcuts.** The main idea of the MPPR is that the trail of computed points of one processors hits another. When identifying a methodology to couple the starting and random points of all contributing processors in a way that the probability of a trail collision is significantly increased, this can lead to a performance boost with immense impact. This includes similar approaches like the notion extension of a collision or the use of isomorphism [Tes01]. To some extent, this already has been successfully implemented by the inverse-point strategy (see Section 4.1.1).

- **Developing New Attacks.** New parallel attacks can revolutionize the view on the security of ECC. This might lead to a change of an optimal attack scenario with special purpose hardware.

A. Appendix

A.1. Arithmetic Unit Controller

The following controller instruction tables attempt to provide an understandable impression about the internal logic of our arithmetic unit. Each arithmetic operation is presented in a separate listing for the sake of clarity.
With the attempt to represent hardware processes including sequentially *and* concurrently executed instructions, we will encounter the problem to illustrate such a description in an understandable way in a single sheet. Thus, parallel processes are designated by a subcategory in the clock cycle column, e.g. $1(x)$ where x denotes the identity number of a parallel task in cycle 1. Sequential instructions are simply designated by an ascending clock cycle number. Further, the internal change of control flags like *WE, R* and *L* are implicitly contained in assignments ($REG1 \leftarrow \ldots$) and shifts ($REG1 >> 1$). Please note the classification of clock cycles into reset (RST), computational (CMP) and finishing (FIN) cycles.

Class	**Cycle**	**Control Description**	**Active Components**
RST	1	Awaiting input from RAM	IN1, IN2, MOD
RST	2	REG1 ← IN1 + IN2	COMP1, M1.1, M1.2
CMP	1	REG2 ← REG1 - MOD	COMP2, M2.1, M2.2
FIN	1(1)	Determine OUT ← {REG1, REG2}	OUT, CARRY[2]
FIN	1(2)	Indicate end of operation	-

Table A.1.: Arithmetic unit control table for addition

In the listing A.1 for the field addition, we define a RESET and FINISH period lasting over two and one cycle, respectively. Hence, the actual computation demands only another single clock cycle. The reason why we extend the notion of the RESET to span over two cycles, is caused by the claim for conformity in respect of multiplication and inversion. Here, a two cycle reset will become evident and for simplicity this RESET length will be defined for the addition and subtraction, too. The following Table A.2 shows the listing for the AU's field subtraction. The Montgomery Multiplication is a bit more complex due to its volatile nature caused by cycle skipping. Thus, we will shorten the listing style to create some space for more relevant information. For the ability to save cycles, conditional branching is required. This is realized by new columns for displaying conditions and states in the corresponding control table. Furthermore, we will

Class	**Cycle**	**Control Description**	**Active Components**
RST	1	Awaiting input from RAM	IN1, IN2, MOD
RST	2	REG1 ← IN1 - IN2	COMP1, M1.1, M1.2
CMP	1	Determine OUT ← {REG1, REG2}	OUT, CARRY[1]
FIN	1(1)	REG2 ← REG1 + MOD	COMP2, M2.1, M2.2
FIN	1(2)	Indicate end of operation	-

Table A.2.: Arithmetic unit control table for subtraction

encounter the necessity to handle sequential and parallel statements *in a single cycle*, for which the cycle notation needs a change. For example, the expression $i(b|3)]$ denotes a control command in its i-th iteration to be executed in parallel with at least two further statements (second entry in parentheses determines the number of parallel processes) after the signal propagation has passed the previous statements (first letter in $(b|3)$ shows the sequential order). Hence, all control statements with $(a|x)$ have already been passed respectively executed before any statement with $(b|x)$ draws our attention. With the extended notation, the control for the AU's Montgomery Multiplication is listed as shown in Table A.3. The listing reveals itself as a combination of the MM's state machine and instructions from the algorithm.

Class	**Cycle**	**State**	**Condition**	**Control Description**
RST	1	$undefined$	*none*	Awaiting input from RAM
RST	$2(a\|1)$	$undefined$	*none*	REG3 ← 0
RST	$2(a\|2)$	$undefined$	*none*	REG4 ← IN1
RST	$2(a\|3)$	$undefined$	*none*	$i \leftarrow 0$
RST	$2(a\|4)$	$undefined$	*none*	Switch state to ADD-V
CMP	$i(a\|1)$	ADD-V	$\text{REG4}_{LSB} = 0$	REG3 ← REG3 + IN2
CMP	$i(b\|1)$	ADD-V	$\text{REG3}_{LSB} = 1$	Switch state to ADD-M
CMP	$i(b\|2)$	ADD-V	$\text{REG3}_{LSB} = 0$	REG3 ← REG3 >> 1
CMP	$i(b\|3)$	ADD-V	$\text{REG3}_{LSB} = 0$	REG4 ← REG4 >> 1
CMP	$i(b\|4)$	ADD-V	$\text{REG3}_{LSB} = 0$	$i \leftarrow i + 1$
CMP	$i(a\|1)$	ADD-M	$\text{REG3}_{LSB} = 1$	REG3 ← REG3 + MOD
CMP	$i(a\|2)$	ADD-M	$\text{REG4}_{LSB} = 0$	REG3 ← REG3 >> 1
CMP	$i(a\|3)$	ADD-M	$\text{REG4}_{LSB} = 0$	REG4 ← REG4 >> 1
CMP	$i(a\|4)$	ADD-M	$\text{REG4}_{LSB} = 0$	$i \leftarrow i + 1$
CMP	$i(b\|1)$	ADD-M	$\text{REG4}_{LSB} = 1$	Switch state to ADD-V
CMP	$i(c\|1)$	*all*	$i = k + 1$	Switch to FIN
FIN	1	FIN	*none*	Indicate end of operation

Table A.3.: Arithmetic unit control table for multiplication

Eventually, we will provide the control table for the embedded Montgomery Inversion. Using the extended notation, Table A.4 displays the logical signal

routing for this operation. Again, we will recognize the structure of the state machine as well as the recoded instructions for both phases.

Class	**Cycle**	**State**	**Condition**	**Control Description**
RST	1	$undefined$	*none*	Awaiting input from RAM
RST	$2(a\|1)$	$undefined$	*none*	REG1 ← MOD (+0)
RST	$2(a\|2)$	$undefined$	*none*	REG2 ← IN1 (+0)
RST	$2(a\|3)$	$undefined$	*none*	REG3 ← 0
RST	$2(a\|4)$	$undefined$	*none*	REG4 ← 1
RST	$2(a\|5)$	$undefined$	*none*	$i \leftarrow 0$
RST	$2(a\|6)$	$undefined$	*none*	Switch state to PHASE I
CMP	$i(a\|1)$	PHASE I	$\text{REG1}_{LSB} = 0$	REG1 ← REG1 >> 1
CMP	$i(a\|2)$	PHASE I	$\text{REG1}_{LSB} = 0$	REG4 ← REG4 << 1
CMP	$i(b\|1)$	PHASE I	$\text{REG2}_{LSB} = 0$	REG2 ← REG2 >> 1
CMP	$i(b\|2)$	PHASE I	$\text{REG2}_{LSB} = 0$	REG3 ← REG3 << 1
CMP	$i(c\|1)$	PHASE I	CARRY(1)=0	REG1 ← (REG1 - REG2) >> 1
CMP	$i(c\|2)$	PHASE I	CARRY(1)=0	REG3 ← REG3 + REG4
CMP	$i(c\|3)$	PHASE I	CARRY(1)=0	REG4 ← REG4 << 1
CMP	$i(d\|1)$	PHASE I	CARRY(1)=1	REG2 ← (REG2 - REG1) >> 1
CMP	$i(d\|2)$	PHASE I	CARRY(1)=1	REG4 ← REG3 + REG4
CMP	$i(d\|3)$	PHASE I	CARRY(1)=1	REG3 ← REG3 << 1
CMP	$i(e\|1)$	PHASE I	*none*	$i \leftarrow i + 1$
CMP	$i(e\|2)$	PHASE I	REG2=0	REG3 ← MOD + REG3
CMP	$i(e\|3)$	PHASE I	REG2=0	Switch state to PHASE IF
CMP	$i(a\|1)$	PHASE IF	CARRY(3)=0	REG3 ← REG3 - MOD
CMP	$i(a\|2)$	PHASE IF	*none*	Switch state to PHASE II
CMP	$i(a\|1)$	PHASE II	$i < (2k + 4)$	REG3 ← REG3 << 1
CMP	$i(b\|1)$	PHASE II	CARRY(3)=0	REG3 ← REG3 - MOD
CMP	$i(c\|1)$	PHASE II	$i = (2k + 4)$	Switch state to FIN
FIN	1	FIN	*none*	Indicate end of operation

Table A.4.: Arithmetic unit control table for inversion

A.2. Server ⇔ Processor Communication

In previous chapters, the actual exchange of data between the central server unit and the point processors has been excluded because it has been regarded as being less important. Indeed, the communication facility is not subject to a very strict requirement referring to the transfer capacity. This is because of the easy way to adjust the distinguished point criterion accordingly which directly influences the amount of data being produced by the cores. Thus, a low-speed communication between server and computational units can already be sufficient.

Besides the bus controller utilized in the COPACOBANA, another data path to the FPGA can be a serial line controller commonly known as RS232 interface or UART controller. This interface can be extremely useful for debugging purposes of a single FPGA on a development board. Because of the physical presence of an RS232 connector on the actual development boards of the Xilinx FPGAs [Xil06d] which were used for testing and debugging, we only need to add an appropriate controller to our design (see Figure 5.4). On the server station, we further need a piece of software responsible for the remote serial data end point. Usually, modern operating system like Windows [Mic06] and Linux [FB06] already incorporate such a functionality. Hence, the server part requires only little efforts to receive and transmit serial data.

A.2.1. UART on the FPGA

First, we need to agree on a common serial protocol. One of such is the transmission of 8 serial bits including a start and stop bit with no parity at a transmission speed of 9600 baud (see Figure A.1).

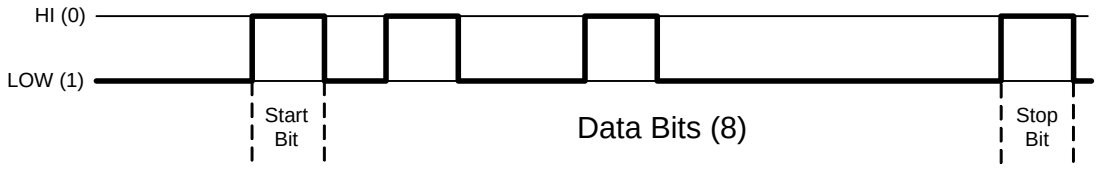

Figure A.1.: Data flow in serial communication

This means that 9600 data bits can be transferred per second (the baud rates do not take control bits into account) or $\lfloor 9600/4k \rfloor = \lfloor 2400/k \rfloor$ points per seconds. For example, for $k = 160$ bit we can transmit 15 points in each direction which is sufficient when considering computations on ECs with bit sizes > 64 and distinguished points start to become rare.
A central issue is the conversion from a parallel signal internally used by the point processor to a serial bit stream for transmission and data reception. Basically, the controller consists of three main components. A transmitter T with associated b-to-1-bit output buffer, a receiver R including an 1-to-b-input buffer, and finally a clock divider responsible to activate the T and R unit at the correct times in a 9600 Baud stepping. The buffers are shift registers with one b-bit and one 1-bit port. The output register can store all b bit in one cycle, from which a single bit can be sent to the transmitter at a time. The transmitter can request the next bit from the register by invoking a bit-shift. The register of the receiver works exactly vice versa.

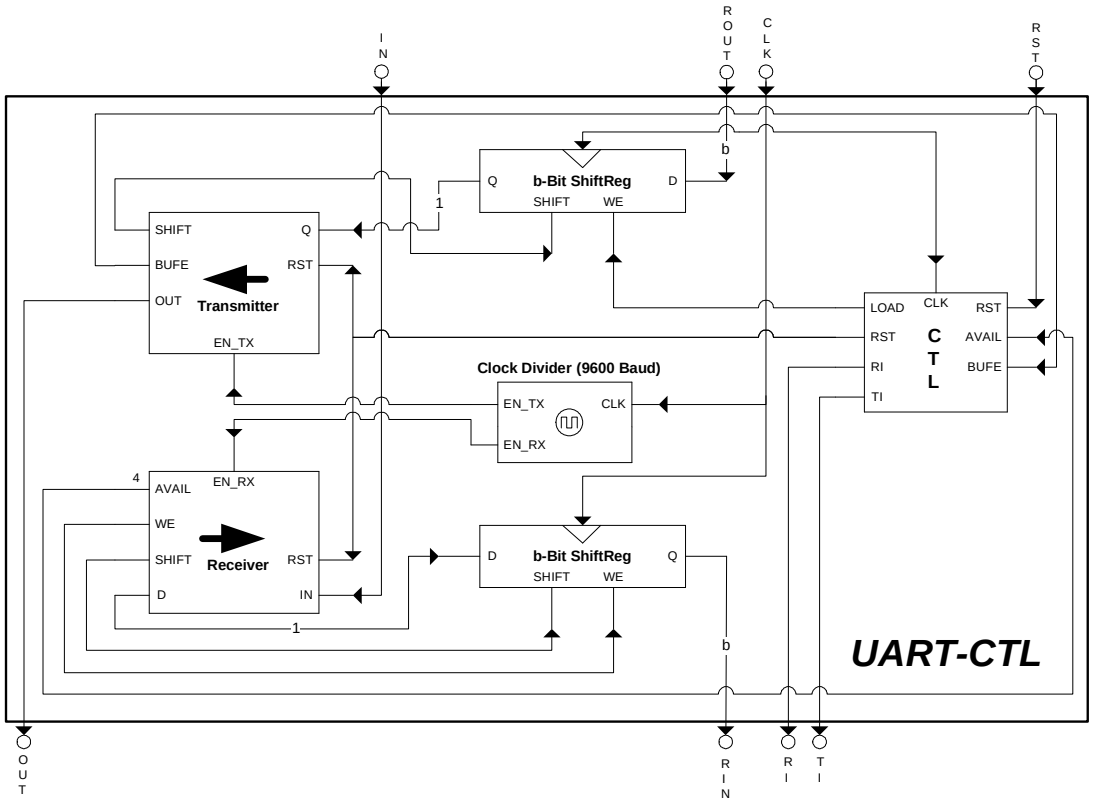

Figure A.2.: Serial line communication controller

Our schematic model displayed in Figure A.2 does not include details supporting robustness of the communication protocol. Thus, it excludes a function to report framing and parity errors (if used), indicating a communication run out-of-synch. These features can easily be added to the design and have just been omitted for a better readability.
Spending some words on the operation of the transmitting and receiving unit should sketch the frame construction and bit sending: On each activation signal from the clock divider, a start bit (HI) is generated by the transmitter followed by sequence of 8 bits from the output register. After the stop bit, a data frame is complete and can be interpreted by the remote station. These steps are repeated until the buffer is empty ($BUFE$) and a transmission interrupt (TI) notifies the upper layer of the necessity to refill it again.
The data reception works analogously. Here, we receive a start bit from the remote station, indicating an incoming transmission. Then, the receiver will pass bitwise the received data byte to the input buffer and advances the buffer position to the next bit. After a received stop bit, a single byte has been successfully read from an external source. This procedure is continued until the receiver recognizes an input buffer entirely filled up with unprocessed data (using an integrated counter). The event of new data being available is indicated by the receiver using the $AVAIL$ flag which will raise an reception interrupt RI in the controller.
We should remark that the size b of the buffers should be byte-aligned. Of course, it is possible to insert some padding data in case that $8 \nmid b$, but for simplicity we assume that the buffer size b is a multiple of 8. This will avoid any confusion with the protocol.

Design Parameters The additional overhead for our serial controller on an FPGA has not yet discussed. This is an important issue since a design for a communication controller should require only a marginal number of FPGA slices leaving sufficient area for the main components providing the processor's intended functionality. The design parameters for our serial controller for various I/O bit size are listed in Table A.5.

b	**Min. Period**	**Max. Performance**	**Slices of XC3S200 FPGA**
16	6.08 ns	164 MHz	105
32	6.08 ns	164 MHz	182
64	6.10 ns	164 MHz	197
80	6.15 ns	163 MHz	229
128	6.44 ns	155 MHz	318

Table A.5.: Design parameters for the serial controller on SPARTAN-3 XC3S200

A.2.2. UART on the Server

So far, we have discussed the serial communication in context to the FPGA. By now, we will have a look how to receive the data from the FPGA on the remote host system. Due to the fact that modern operating system control all the computer's resources, they also provide standard interfaces to access them. While Microsoft Windows allows a communication via the serial controller using APIs [Mic06], Linux will grant the usage of the serial line simply by accessing a device file, e.g. `/dev/ttyS0`. Besides the easy access to the RS232 port via a device file, Linux supports very simple commands for changing the associated device parameters, e.g. the communication speed, etc. The following extract of a C-code listing outlines how to initiate a serial communication in a Linux environment according to a 9600 Baud 8-1 protocol [FB06].

```
int  port_desc;
struct termios   io_params;

// open the com port
port_desc = open( port_name, O_RDWR | O_NOCTTY );

// if result is negative, we got an error
if ( port_desc <0 )
{
...  // return error
}

/* save current settings first */
tcgetattr( port_desc,&saved_io);

/* init new port settings with zeroes */
bzero( &io_params, sizeof(io_params));

/*
  BAUDRATE: Set bps rate.
  CRTSCTS : output hardware flow control
  CS8     : 8n1 (8bit,no parity,1 stop bit)
  CLOCAL  : local connection, no modem control
  CREAD   : enable receiving characters
*/
io_params.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;

/*
  IGNPAR  : ignore bytes with parity errors
  IGNBRK  : ignore line breaks
  IGNCR   : ignore CR's
*/
io_params.c_iflag = IGNPAR | IGNBRK | IGNCR;
io_params.c_oflag = 0;

/* set input mode (non-canonical) */
```

```
io_params.c_lflag = 0;

/* sets the time to wait for next incoming data */
io_params.c_cc[VTIME] = DATA_WAIT_TIME;

/* sets the amount of characters to receive at a time */
io_params.c_cc[VMIN] = DATA_BLOCK_SIZE;

/* now clean the modem line and activate the settings for the port */
tcflush( port_desc, TCIFLUSH );
tcsetattr( port_desc, TCSANOW, &io_params );

/* now we can start with transmission/reception
 ...
```

The transmission and reception of several bytes of data can simply be performed by a single block-wise read and write command. For reading, it should be remarked that the settings from above will block a read for a specific amount of time $t =$ `DATA_WAIT_TIME` until $b =$ `DATA_BLOCK_SIZE` bytes have been received. If less than b bytes have been transmitted within the period t, the read is aborted and the actually received bytes returned. This situation might indicate the presence of a communication or synchronization error. The following command will demonstrate the data exchange:

- Reading b bytes from the serial line `port_desc` into a buffer:
  `bytes_rcvd = read( port_desc, buffer, DATA_BLOCK_SIZE );`

- Writing b bytes to the serial line `port_desc` from a buffer:
  `bytes_trsmt = write( port_desc, buffer, DATA_BLOCK_SIZE );`

Putting this code together in a code library will provide an easy way to access data from the FPGA for the server. Furthermore, with some minor changes to the core layer respectively the data path to the point processors, it is even possible to read the current values of the FPGA's block memory. This is enormously useful considering debugging purposes and is used for our implementation.

A.3. Buffer Locking Logic

To control the point buffer input, it is evident to prioritize the lock requesters. Algorithm 15 shows how and if a lock for the buffer is granted and revoked. If a lock has been granted, this is indicated by the one-bit *LOCK* register. Furthermore, the multiplexer for the data input to the point buffer is changed accordingly.

Algorithm 15 Lock Request Resolution

Input: *LOCK*, LKR_i and *LKRC* signals (i denotes the i-th out of ϕ cores)
Output: *LOCK*, LKG_i and *LKGC* signals (i denotes the i-th out of ϕ cores)

if no current *LOCK* and LKR_1 is set **then**
  Set *LOCK* and LKG_1
  Set *MUX* and *WE* for buffer writing from Core 1
else if no current *LOCK* and LKR_2 is set **then**
  Set *LOCK* and LKG_2
  Set *MUX* and *WE* for buffer writing from Core 2
  ...
else if no current *LOCK* and LKR_ϕ is set **then**
  Set *LOCK* and LKG_ϕ
  Set *MUX* and *WE* for buffer writing from Core ϕ
else if a current *LOCK* is set **then**
  Unset the *LOCK* signal and revoke $LKG_1 \cdots LKG_\phi$
  Unset *WE* for Point Buffer
end if

A.4. List of Notations and Symbols

$a_i \iff i$-th random coefficient $a_i \in \{1, \ldots, n-1\}$ used for $R_i = a_iP + b_iQ$
$b_i \iff i$-th random coefficient $b_i \in \{1, \ldots, n-1\}$ used for $R_i = a_iP + b_iQ$
$c \iff$ current coefficient $c \in \{1, \ldots, n-1\}$ used for $X = cP + dQ$
$d \iff$ current coefficient $d \in \{1, \ldots, n-1\}$ used for $X = cP + dQ$
$D \iff$ set of all distinguished points $D \subset \langle P \rangle$
$\delta \iff$ number of zero bits in x-coordinate designating a distinguished point
$h \iff$ bits of Montgomery radix with $h = k + 2$
$\ell \iff$ discrete logarithm $\ell = log_P(Q)$
$k \iff$ number of bit size to represent values of E, e.g., $k = \lceil log_2(p) \rceil$
$m \iff$ order of elliptic curve E with $m = ord(E)$
$n \iff$ order of base point P $n = ord(P)$
$p \iff$ prime value determining field size $\mathbb{F}_p$ with $p > 3$
$P \iff$ base point (generator) of EC $(x_P, y_P) = P \in E(\mathbb{F}_p)$
$Q \iff$ second point in $(x_Q, y_Q) = Q \in \langle P \rangle$ and $Q = \ell P$
$R_i \iff i$-th random point $(x_{R_i}, y_{R_i}) = R_i \in \langle P \rangle$ obtained by $R_i = a_iP + b_iQ$
$s \iff$ number of available partitions
$T \iff$ threshold defining the maximum number of computed points X without $X \in D$
$\Theta \iff$ proportion of distinguished points with respect to $\langle P \rangle$
$w_i \iff i$-th point processor $w_i \in W$
$\mathcal{W} \iff$ set of all available point processors
$z \iff$ the number of leading/trailing bits of a distinguished point

A.5. List of Signals and Ports

Table A.6 presents signal and port names for the top layer of the MPPR design in hardware.

Component	**Port**	**Type**	**Port Name**	**Bit Size**
PC	RI	IN	Receive Interrupt (ComCTL)	1
PC	RIN	IN	Received Input (ComCTL)	k+r
PC	DOUT	IN	Buffer Data Out	k
PC	LKGC	IN	Lock Granted for Command	1
PC	FIN	IN	Finished	1
PC	TI	OUT	Transmit Interrupt (ComCTL)	1
PC	TOUT	OUT	Transmitted Output (ComCTL)	k+r
PC	CMD	OUT	Core Command	3
PC	ADDR	OUT	Memory Address	<8
PC	DIN	OUT	Core Data Input	k
PC	CS_x	OUT	Chip Select for Core x	1
PC	RST	OUT	Command Reset	1
PC	LKRC	OUT	Lock Requested by Command	1
ComCTL	IN	IN	External Input	BUS
ComCTL	TOUT	IN	Output for Transmission	k+r
ComCTL	RST	IN	Communication Reset	1
ComCTL	OUT	OUT	External Output	BUS
ComCTL	RI	OUT	Receive Interrupt	1
ComCTL	TI	OUT	Transmit Interrupt	1
ComCTL	RIN	OUT	Received Input	k+r
LC	LKR_x	IN	Lock Requested by Core x	1
LC	LKRC	IN	Lock Requested by Command	1
LC	LKG_x	OUT	Lock Granted to Core x	1
LC	LKGC	OUT	Lock Requested by Command	1
LC	WE	OUT	Write Enable for PB	1
LC	MUX	OUT	Multiplexer Control	1
PRCore	RST	IN	Core Reset	1
PRCore	CMD	IN	Core Command	3
PRCore	CS	IN	Chip Select	1
PRCore	DIN	IN	Core Data Input	k
PRCore	ADDR	IN	Address Line	<8
PRCore	LKG	IN	Lock Granted to Core	1
PRCore	DOUT	OUT	Core Data Output	k
PRCore	LKR	OUT	Lock Requested by Core	1
PRCore	FIN	OUT	Core Status (IDLE)	1

Table A.6.: Top layer components (Figure 5.4)

Table A.7 presents signal and port definitions used by a PRCore.

Component	**Port**	**Type**	**Port Name**	**Bit Size**
CC	RST	IN	Command Reset	1
CC	CS	IN	Core Chip Select	1
CC	CMD	IN	Command	3
CC	ADDR	IN	Address Line	<8
CC	LKG	IN	Lock Granted for Buffer Write	1
CC	EQ	IN	Equality between IN1 and IN2	1
CC	DP	IN	IN1 satisfies Distinguished Point Criterion	1
CC	OPFIN	IN	AU Operation Finished	1
CC	$ADDR_M$	OUT	Address Line to Modulus Memory	1
CC	WE_M	OUT	Write Enable to Modulus Memory	1
CC	$ADDR_A$	OUT	Address Line for Data Path A	<8
CC	$ADDR_B$	OUT	Address Line for Data Path B	<8
CC	WE_A	OUT	Write Enable for Data Path A	1
CC	WE_B	OUT	Write Enable for Data Path A	1
CC	FIN	OUT	Core Halted (Finished)	1
CC	LKR	OUT	Lock Requested for Buffer	1
CC	OP	OUT	AU Operation	2
CC	RST	OUT	AU Reset	1
AU	RST	IN	AU Reset	1
AU	OP	IN	AU Operation	2
AU	IN1	IN	AU Operand 1	k
AU	IN2	IN	AU Operand 2	k
AU	MOD	IN	AU Modulus	k
AU	OUT	OUT	AU Result	k
AU	OPFIN	OUT	Operation Finished	1
COMP	A	IN/OUT	Comparator Input A	k
COMP	B	IN/OUT	Comparator Input B	k
COMP	EQ	OUT	Compares A & B for Equality	1
COMP	DP	OUT	A satisfies Distinguished Point Criterion	1

Table A.7.: Core layer definitions (Figure 5.5)

Signals and Ports of the Arithmetic Layer are depicted in Table A.8.

Component	**Port**	**Type**	**Port Name**	**Bit Size**
AC	RST	IN	Reset Signal	1
AC	OP	IN	Operation to be performed	2
AC	CARRY	IN	Carry flags from Adders/Subtracters	3
AC	ZERO	IN	Flag denoting $REG2 = 0$	1
AC	L	OUT	Left Shift Flags	3
AC	R	OUT	Right Shift Flags	4
AC	WE	OUT	Write Enable to Registers	4
AC	CTL	OUT	Control States	STS
AC	ADD	OUT	Add Flags for Adders/Subtracters	3
LR-REG	RST	IN	Register Reset	1
LR-REG	WE	IN	Write Enable Flag	1
LR-REG	R	IN	Right Shift Flag	1
LR-REG	L	IN	Left Shift Flag	1
LR-REG	D	IN	$k+1$-bit Register Input	k+1
LR-REG	Q	OUT	$k+1$-bit Register Output x	k+1
WT	R	IN	LSB of Register Output	1
WT	CTL	IN	State information	STS
WT	WE	OUT	In-Cycle Write Enable	1

Table A.8.: Arithmetic layer definitions (Figure 5.13)

A.6. List of Abbreviations

AC	$\Longleftrightarrow$	Arithmetic Controller
AU	$\Longleftrightarrow$	Arithmetic Unit for GF(p)
AT	$\Longleftrightarrow$	Area-Time Product
CC	$\Longleftrightarrow$	Core Controller
ComCTL	$\Longleftrightarrow$	Communication Controller
EC	$\Longleftrightarrow$	Elliptic Curve
ECC	$\Longleftrightarrow$	Elliptic Curve Cryptography (ECC)
FIN	$\Longleftrightarrow$	Finished
MPPR	$\Longleftrightarrow$	Multi-Processor Pollard-Rho
LC	$\Longleftrightarrow$	Lock Controller
PC	$\Longleftrightarrow$	Processor Controller
RST	$\Longleftrightarrow$	Reset
SPPR	$\Longleftrightarrow$	Single-Processor Pollard-Rho

A.7. Reference Run with $k = 40$ Bits

```
Generating an elliptic curve y^2 = x^3 + a*x + b with 40 bits.
Elliptic curve E: y^2 = x^3 + a*x + b has indeed a bit size of 40 bits

E(a) = 661113824350
E(b) = 793723350633
E(p) = 1058942403199
Order of curve ord(E) = 1058942385139
Base point P = (58972220422,8413406201);
Order of base point ord(P) = 1058942385139
Further point Q = (211024733107,699847587373);
Secret factor Q = l*P with l = 421830503563
Identified prime order ord(P)=1058942385139 with 40 bits

Starting computation of ECDLP:
Allocating memory for computation...Memory acquired.

Enabling GMP Modular Arithmetic...Done

Creating hash files for storing disinguished points...
File creation complete.

Setting up processor(s)...
Configuring processor parameters to use 16 partitions

Starting to collect distinguished points...
...

It seems that
29608128902P + 43247106713Q collides with 113050827328P + 183310176942Q.
Verification for 29608128902P + 43247106713Q = (32497296,365953781440)
Verification for 113050827328P + 183310176942Q = (32497296,365953781440)

Computation of logarithm : l = (113050827328 - 29608128902)*
(43247106713 - 183310176942)^(-1) mod 1058942385139

### Computed logarithm is l = 421830503563 ###

Required distinguished points for computation: 25
Average trail length: 22186

Elapsed total time (sec): 14.850000
```

List of Figures

List of Tables

Bibliography

[Ama05] David Narh Amanor. Efficient hardware architectures for moduluar multiplication. Master's thesis, University of Applied Sciences Offenburg and Ruhr-University Bochum, Germany, 2005.

[AMD06] AMD. Processor Design for Athlon64 X2 and Opteron CPUs, 2006. Website: `http://www.amd.com/`.

[Ash01] Peter J. Ashenden. *The Designers Guide to VHDL*. Morgan Kaufmann, 2001.

[BHLM01] Michael Brown, Darrel Hankerson, Julio López, and Alfred Menezes. Software implementation of the NIST elliptic curves over prime fields. *Lecture Notes in Computer Science*, 2020:pp.250, 2001.

[Bla83] G.R. Blakley. A computer algorithm for the product ab modulo m. *IEEE Trans. on Computers*, 32:497–500, 1983.

[BSS99] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[CC86] D.V. Chudnovsky and G.V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Math.*, 7:385–434, 1986.

[Cer06] Certicom. Certicom ECC challenge, 2006. `http://www.certicom.com`.

[CMO97] Henri Cohen, Atsuko Miyaja, and Takatoshi Ono. Efficient elliptic curve exponentiation. *Advances in Cryptology-Proceedings of ICICS'97*, 1334:282–290, 1997.

[CMO98] Henri Cohen, Atsuko Miyaja, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. *AsiaCrypt*, 1998.

[CMO00] Henri Cohen, Atsuko Miyaja, and Takatoshi Ono. Constructive and destructive facets of weil descent on elliptic curves. Technical Report CSTR-00-016, Department of Computer Science, University of Bristol, October 2000.

[dDBQ04] Guerric Meurice de Dormale, Philippe Bulens, and Jean-Jacques Quisquater. An Improved Montgomery Modular Inversion Targeted for Efficient Implementation on FPGA, 2004. International Conference on Field-Programmable Technology.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22:644–654, 1976.

[DMKP04] Alan Daly, William Marnane, Tim Kerins, and Emanuel Popovici. An FPGA implementation of a GF(p) ALU for encryption processors. Technical report, University College Cork, Cork, Ireland, 2004.

[DMP04] Alan Daly, Liam Marnaney, and Emanuel Popovici. Fast Modular Inversion in the Montgomery Domain on Reconfigurable Logic. Technical report, University College Cork, Cork, Ireland, 2004.

[ElG85] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31:469–472, 1985.

[Elk98] N. D. Elkies. Elliptic and modular curves over nite elds and related computational issues. *Computational perspectives on number theory*, 7:21–76, 1998.

[ESST99] A.E. Escott, J.C. Sager, A.P.L. Selkirk, and D. Tsapakidis. Attacking elliptic curve cryptosystems using the parallel pollard rho method. *CryptoBytes*, 4(2):15–19, 1999.

[FB06] Gary Frerking and Peter Baumann. Serial Programming HOWTO, 2006. Linux HOWTO Websites. `http://www.tldp.org/HOWTO/Serial-Programming-HOWTO/`.

[FKP+05] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, and C. Stahlke. SHARK — A Realizable Special Hardware Sieving Device for Factoring 1024-bit Integers. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *LNCS*, pages 119–130. Springer-Verlag, August 2005.

[FO90] P. Flajolet and A. M. Odlyzko. Random mapping statistics. *Lecture Notes in Computer Science*, 434:329–354, 1990.

[FR94] G. Frey and H. Rück. A remark concerning m-divisibilty and the discrete logarithm in the divisor class group of curves. *Mathematics of Computation*, 62:865–874, 1994.

[GLV00] R. Gallant, R. Lambert, and S. Vanstone. Improving the parallelized pollard lambda search on binary anomalous curves. *Mathematics of Computation*, 69:1699–1705, 2000.

[Gut02] Adnan Abdul-Aziz Gutub. Scalable VLSI Architecture for GF(p) Montgomery Modular Inverse Computation, 2002. IEEE Computer Society Annual Symposium on VLSI.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, New York, 2004.

[Int06] Intel. Processor design for itanium cpus, 2006. Website: `http://www.intel.com/`.

[Kal95] B. S. (Jr.) Kaliski. The Montgomery Inverse and its Applications. *IEEE Transactions on Computers*, 44:10641065, 1995.

[Kim01] Chinuk Kim. VHDL implementation of systolic modular multiplication on RSA cryptosystem. Master's thesis, City University of New York, 2001.

[Kob87] Neil Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[Kob94] Neil Koblitz. *A Course in Number Theory and Cryptography*. Springer Verlag, New York, 1994.

[Koc95] Cetin Kaya Koc. RSA Hardware Implementation. Technical report, RSA Laboratories, 1995.

[KPP+06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking Ciphers with COPACOBANA — A Cost-Optimized Parallel Code Breaker, 2006. To Appear.

[KSZ02] E. Konstantinou, Y.C. Stamatiou, and C. Zaroliagis. On the efficient generation of elliptic curves over prime fields, 2002.

[LV01] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[Men93] A. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[Mic06] Microsoft. Visual Basic .NET Code Sample: Using the COM Port in VB.NET, 2006. Seriall Communication Tutorial for VisualBasic: `http://www.microsoft.com`.

[Mil85] V. S. Miller. Use of elliptic curves in cryptography. *Advances in cryptology proceedings of Crypto '85*, 218:417–426, 1985.

[Mon85] P. L. Montgomery. Modular multiplication wihtout trial division. *Math. Computation*, 44:519–521, 1985.

[MOV93] Alfred J. Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Trans. Inform. Theory*, 39:1639–1646, 1993.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, New York, 1996.

[OBPV03] S.B. Örs, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of elliptic curve processor over GF(p). *Proceedings of the Application-Specific Systems, Architectures, and Processors ASAP*, page 433443, 2003.

[OP01] G. Orlando and C. Paar. A scalable GF(p) elliptic curve processor architecture for programmable hardware. *Proceedings of Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2001)*, 2162:356–371, 2001.

[Pal03] Samir Palnitkar. *Verilog HDL. A Guide to Digital Design and Synthesis*. Prentice Hall PTR, 2003.

[PH78] S.C. Pohlig and M. E. Hellman. An improved algorithm for computing discrete logarithms over GF(p) and its cryptographic significance. *IEEE Trans. Inf. Theory*, 24:106–110, 1978.

[Pol78] J. Pollard. Monte carlo methods for index computation (mod p). *Mathematics of Computation*, pages 918–924, 1978.

[Pro05] GNU Project. GNU multiple precision arithmetic library (GMP), 2005. `http://www.swox.com/gmp` (last checked 01/31/06).

[RSA77] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and Public-Key Cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[Sch85] R. Schoof. Elliptic curves over nite elds and the computation of square roots mod p. *Math. Comp.*, 44:483–494, 1985.

[Sem98] I. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in charactistic p. *Mathematics of Computation*, 67:353–356, 1998.

[Sha71] D. Shanks. Class number, a theory of factorization and genera. *Proc. Symp Pure Math.*, 20:415–440, 1971.

[Sma97] N. Smart. Announcement of an attack on the ECDLP for anomalous elliptic curves, 1997.

[ST03] Adi Shamir and Eran Tromer. Factoring Large Numbers with the TWIRL Device. In *Advances in Cryptology — Crypto 2003*, volume 2729 of *LNCS*, pages 1–26. Springer, 2003.

[Tes98] Edlyn Teske. Speeding up Pollard's Rho method for computing discrete logarithms. *Lecture Notes in Computer Science*, 1423:541–554, 1998.

[Tes01] E. Teske. Square-root algorithms for the discrete logarithm problem, 2001.

[vOW99] P.C. van Oorschot and M.J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12:1–28, 1999.

[WZ98] M. Wiener and R. Zucherato. Faster attacks on elliptic curve cryptosystems. *Proceedings of SAC - Workshop on Selected Areas in Cryptography*, 1556:190–200, 1998.

[Xil06a] Xilinx. Xilinx IP-Cores, 2006. IP-Center website. `http://www.xilinx.com/ipcenter/`.

[Xil06b] Xilinx. Xilinx ISE Webpack, 2006. Integrated System Environment for FPGA programming. `http://www.xilinx.com/ise/logic\_design\_prod/webpack.htm`.

[Xil06c] Xilinx. Xilinx ModelSim III, 2006. Simulation Tool for FPGA programming. `http://www.xilinx.com/ise/optional\_prod/mxe.htm` (last checked 01/31/06).

[Xil06d] Xilinx. Xilinx SPARTAN-3 FPGAs, 2006. FPGA information website. `http://www.xilinx.com/products/silicon_solutions/`.