Integer Factorization based on the Elliptic Curve Method using Reconfigurable Hardware

Christian Röpke

March 26, 2008

Diploma Thesis

Ruhr-Universität Bochum
Department of Electrical Engineering
and Information Sciences

Chair for Communication Security
Prof. Dr.-Ing. Christof Paar
Advisor: Dipl.-Ing. Tim Güneysu

Erklärung/Statement

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

I hereby declare that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by references to other authors.

Datum/Date	Christian Röpke

Abstract

In this work we present a novel and efficient implementation of the elliptic curve method for factoring small to mid-sized integers (up to a few hundred bits). Among others, the ECM is an important tool to accelerate co-factorizations required in context of the general number field sieve.

Like for other ECC-based algorithms, the efficiency of the underlying modular arithmetic for the ECM is of crucial importance. Thus, we discuss new strategies for implementing a high-performance multiplier and adder/subtracter unit in reconfigurable hardware. The optimization of these fundamental operations basically relies on the intensive use of DSP elements integrated in modern field programmable gate arrays. Based on this improved arithmetic, we achieve a significant gain in performance of the ECM (Phase 1).

Finally, we discuss the integration of the ECM on a massively parallel FPGA-cluster, namely COPACOBANA (Cost Optimized PArallel COde Breaker), enabling a multitude of simultaneous factorizations on a single hardware platform.

Contents

1. Introduction

In 1977, Ronald L. Rivest, Adi Shamir, and Leonard Adleman invented the RSA algorithm which is one of the most frequently used methods in the field of public-key cryptography. Its security relies on the apparent intractability of the integer factorization problem. Informally speaking, it is easy to multiply large prime numbers but it is hard to factor large composite numbers to get all (or at least one) of its prime factors. Even before that proposal, many mathematicians and cryptologists are working hard on finding sufficient solutions for this problem.

The main ideas behind modern factoring algorithms are based on suggestions proposed by Fermat (1607-1665) [9] and Kraïtchik (1882-1957) [22]. Fermat expresses composite integers as a difference of two squares of the form

$$n = p \cdot q = x^2 - y^2$$

where all given numbers are positive integers. This equation is satisfied for all odd composites n. Then, if x and y are determined in some way, the factors p and q of number n can easily be computed by $p = (x + y)$ and $q = (x - y)$. Kraïtchik extends Fermat's idea by two new suggestions. First, Kraïtchik was also looking for differences of squares which are multiples of n such that

$$k \cdot n = k \cdot p \cdot q = x^2 - y^2$$

where k is a positive integer. For suitable x and y, the factors p and q can again easily be determined by computing $p = \gcd(x + y, n)$ and $q = \gcd(x - y, n)$. Kraïtchik secondly suggests a method for finding such suitable x and y. These ideas are described in more detail in Appendix A.1.

Based on these suggestions of Fermat and Kraïtchik, Algorithm 1 informally may describe the main loop of modern factoring algorithms.

The number field sieve (NFS) is a modern factoring method. It was proposed by J.P. Buhler, H.W. Lenstra Jr. and C. Pomerance in 1992 but was originally based on a work by J.M. Pollard in the late 80's. The general number field sieve (GNFS) is a derivative of the NFS and is the fastest known algorithm for factoring general large numbers. Similar to Algorithm 1, it is principally structured in three steps where the most time is spend on the first step since this is the most complex

Algorithm 1 Informally description of modern factoring methods

Input: $n = p \cdot q$, n odd and $a, b, n \in \mathbb{N}$

Output: p, q

1: Determine x, y for which $x^2 \equiv y^2 \pmod{n}$ holds
2: Compute $x = \sqrt{x^2}$ and $y = \sqrt{y^2}$
3: Compute $p = \gcd(x + y, n)$ and $q = \gcd(x - y, n)$
4: **return** p, q

one. Step one includes a polynomial selection, a relation collection (including a sieving step) and some linear algebra computations. In step two, the square root is calculated and finally the factors are determinated. In its sieving step, mid-sized composites of up to a few hundred bits, which are a product of only a few numbers, must be factorized. This task can be efficiently realized by the elliptic curve method (ECM), which was invented by H.W. Lenstra Jr. in 1985. Since roughly ten billion of such mid-sized composites need to be processed, speeding up the factorization by the ECM is of crucial importance to accelerate the GNFS.

In this work we present a novel and efficient implementation of the elliptic curve method for factoring small to mid-sized integers (up to a few hundred bits). Like for other ECC-based algorithms, the efficiency of the underlying modular arithmetic for the ECM is of crucial importance. Thus, we discuss new strategies for implementing a high-performance multiplier and adder/subtracter unit in reconfigurable hardware. The optimization of these fundamental operations basically relies on the intensive use of DSP elements integrated in modern field programmable gate arrays. Based on this improved arithmetic, we achieve a significant gain in performance of the ECM (Phase 1). Finally, we discuss the integration of the ECM on a massively parallel FPGA-cluster, namely COPACOBANA (Cost Optimized PArallel COde Breaker), enabling a multitude of simultaneous factorizations on a single hardware platform.

For this purpose, this thesis is structured in eight further chapters. Chapter 2 gives a complete overview of available FPGA implementations in the field of the elliptic curve method. In Chapters 3 and 4, basic definitions and facts about required mathematics and the underlaying hardware are presented. Then, in Chapter 5 the elliptic curve method is introduced and important improvements are discussed. Based on this description, in Chapter 6 the ECM hardware architecture is developed and appropriate algorithms are proposed. Chapter 7 describes the realization of our architecture on an FPGA device and Chapter 8 shows the achieved results which are obtained during sufficient hardware simulations. Finally, Chapter 9 concludes the implementation results and gives a brief outlook to future work.

2. Related Work

The first publication of a realized hardware implementation of the elliptic curve method and the first description of GNFS acceleration through hardware-based ECM are presented in [34, 12, 40].

The authors use elliptic curves in Montgomery's form and realize the Montgomery ladder for performing the scalar multiplication on elliptic curves. Deducted from software experiments, the following ECM parameters are chosen for finding 40-bit factors in 200-bit numbers: $B_1 = 960$, $B_2 = 57000$, and $D = \{6, 30, 60, 210\}$. Other parameters are $k_N = 1375$ (the bit length of k) and the bit length of the to be factored number $n = 198$. A hardware-software co-design is implemented using an ARM7TDMI microprocessor and a Xilinx Virtex 2000E-6 FPGA. Figure 2.1 gives a brief overview of the implemented hardware. The central control unit is

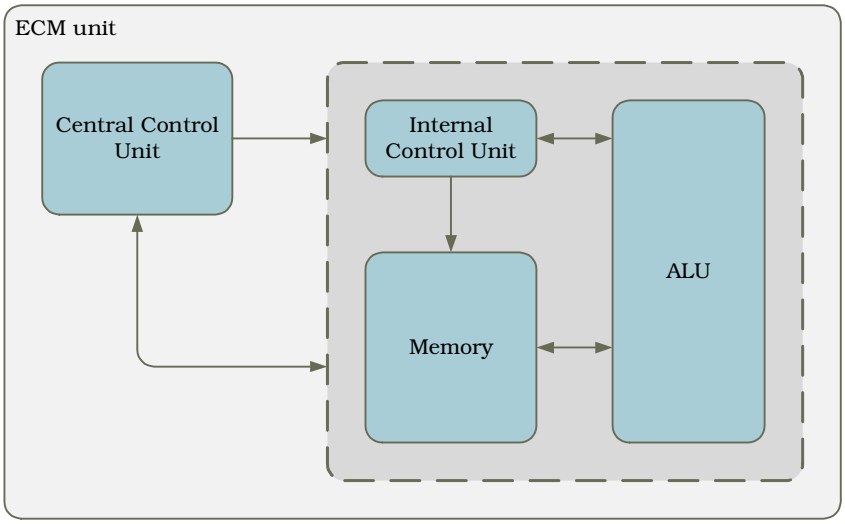

Figure 2.1.: ECM hardware design [34]

implemented by the microprocessor, it coordinates the data exchange before and after computations, and it generates instructions which control the entire ECM unit. The ECM unit is implemented by the FPGA and consists of an internal control unit, memory, and an arithmetic logic unit (ALU). The internal control

unit controls data transfer between the memory and the ALU, the memory contains the data set needed for performing the ECM, and finally, the ALU performs computations modulo $2N$ where N is the number to be factored.

The hardware-software co-design achieves the results listed in Table 2.1. Phase 1

Table 2.1.: Implementation results [34]

Operation	Time
Phase 1	912 ms
Phase 2 (estimate)	1879 ms

is implemented completely and its execution time is 912 ms whereas the execution time of Phase 2 is estimated at least 1879 ms.

The authors already discuss optimizations such as integrating multiple ALUs in one ECM unit, but in fact, a hardware system including three ECM units per FPGA is realized, and each of such units consist of a single ALU.

In [13, 14], – as far as we know – the fastest realized ECM hardware implementation including support for Phase 1 and Phase 2 is presented.

Elliptic curves are represented in Montgomery's form and the Montgomery ladder is used for performing the scalar multiplication on elliptic curves. Most ECM parameters are adopted from previous works to get 40-bit factors from approximately 200-bit composite numbers: $B_1 = 960$, $B_2 = 57000$, and $D = \{30, 210\}$.

Furthermore, a deeply optimized method for realizing the Montgomery ladder with two multipliers and one addition/subtraction unit is presented. Also, Phase 2 of the elliptic curve method is realized in reconfigurable hardware for the first time.

The realized architecture of the ECM hardware is pictured in Figure 2.2. In this architecture, an ECM system is represented by one FPGA which communicates with a connected host system. The host system performs administrative and controlling operations whereas the FPGA executes the cost-intensive operations such as the scalar multiplication. One ECM system consists of several ECM units, a global memory, and an instruction set. The global memory and instruction set are shared by implemented ECM units. Each unit is able to perform both Phase 1 and Phase 2 of the elliptic curve method. The number of ECM units per ECM system depends on the resources available in the chosen hardware platform. One ECM unit incorporates two multipliers, one addition/subtraction unit, and some local memory.

The architecture is implemented for the parameters $n = 198$ and $k_N = 1375$ on

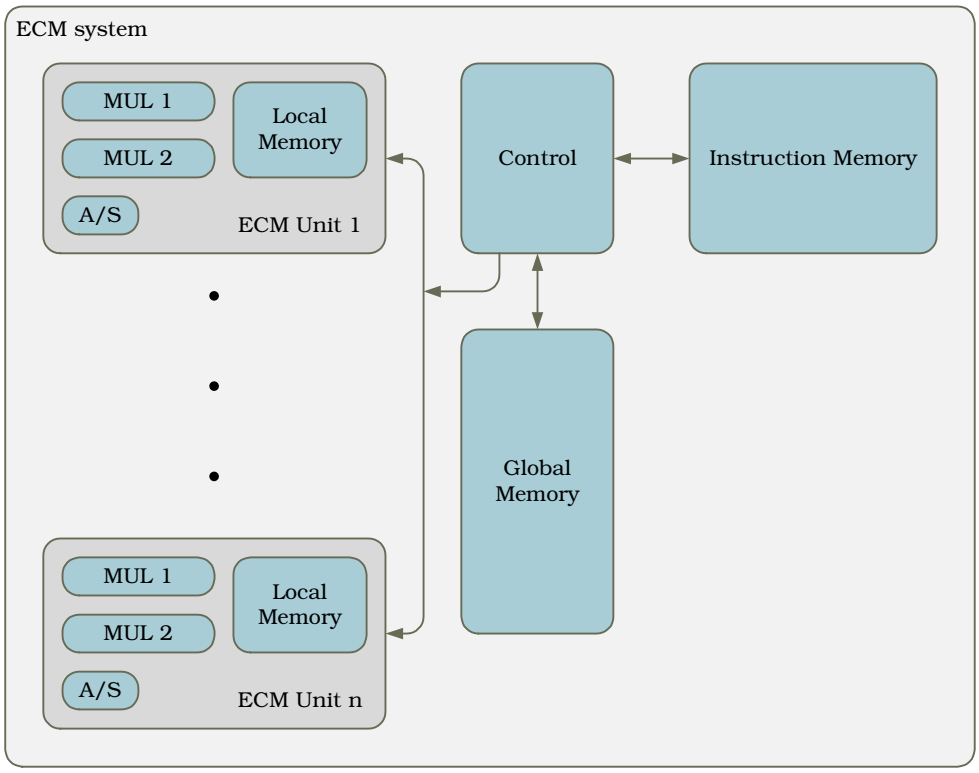

Figure 2.2.: ECM hardware design [13, 14]

several FPGA devices. Table 2.2 gives a brief overview of the implementation results.

The authors improve the proof-of-concept work of [34, 12, 40] by a factor of 6.8 (Phase 1) and 11.8 (Phase 2) regarding to the number of needed clock cycles, and by a factor of 9.3 (Phase 1) and 16.1 (Phase 2) regarding to the running time needed for the corresponding phase. In this work, Phase 2 of the elliptic curve method is realized completely on an FPGA device for the first time.

In [10], a fully parallel and pipelined modular multiplier circuit exhibited a significantly improvement over previous works is presented.

A 135-bit hardware multiplier is implemented which intensively use digital signal processing (DSP) elements which are provided by a Virtex-4 (XC4VSX25-10) FPGA. In contrast to previous works, a massively parallelized architecture is realized whereas previous multiplier implementations are based on serial architectures.

Table 2.2.: Implementation results [13, 14]

Results	Virtex (XCV2000E-6)	Spartan 3 (XC3S5000-5)	Spartan 3E (XC3S1600E-5)	Virtex 4 (XC4VLX200-11)
Max. number of ECM units per FPGA	7	10	4	27
Max. clock frequency for one ECM unit	54 MHz	100 MHz	93 MHz	135 MHz
Time for Phase 1 & 2	62.8 ms	33.9 ms	36.5 ms	25.2 ms
Cost of one FPGA	\$1230	\$130	\$35	\$3000
Number of ECM ops. per sec. [1]	111	295	109	1073
Number of ECM ops. per sec. per \$100 [1]	9	227	311	36

The implementation obtains the results given in Table 2.3. Note that 129 DSP

Table 2.3.: Implementation results [10]

135-bit Designs	Freq. [MHz]	Area [Slices]	# of DSPs
Mult.	220	3405 (33%)	128 (100%)
Add/Sub	245	446 (4%)	0

elements are used for the implemented multiplier but only 128 DSP elements are provided by the chosen hardware platform. Hence, the operation of one DSP element is implemented by logic blocks.

In Table 2.4, the authors compare the extrapolated implementation results with

[1] This was calculated with the maximal number of ECM units.

the results obtained by [13, 14]. Note that the authors use the Xilinx's EasyPath

Table 2.4.: Extrapolated implementation results [10]

Phase 1, 135-bit	[13, 14]	[10]
FPGA	XC3S5000-5	XC4VX25-10
Slices per ECM unit	2300 (7%)	6006 (58%) (+ 128 DSP48)
#bRAMs per ECM unit	2 (2%)	31 (24%)
Max. frequency	100 MHz	220 MHz
$\#T_{clk}$ for one modular multiplication	151	1
$\#T_{clk}$ per Phase 1	$1.22\ 10^6$	13750
#Phase 1 ops. per sec. per ECM unit	82	16000
#ECM units per FPGA	14	1
#Phase 1 ops. per sec. per FPGA	1148	16000
FPGA price (quantity)	\$130 (10^4)	\$116 (2500)
#Phase 1 ops. per sec. per \$100	883	13793
Improvement factor		15.6

solution for calculating the gained improvement factor. By using this solution, FPGA devices can be shipped supporting only previously defined operations. Unfortunately, it is not possible to reprogram such devices after shipping. Also remark that at most Phase 1 is implemented by one multiplier circuit (including an addition/subtraction unit) requiring all DSP elements which are dependent to each other. But the authors do not describe efficient ways about filling the long pipeline which is required by the multiplier, and there is no description about the realization of the scalar multiplication which is required by Phase 1. This may indicate that the implemented circuit is not able to perform the entire Phase 1 independently without a big I/O communication overhead while transferring/receiving data to/from the host system. Furthermore, support for Phase 2 is not described. However, the authors estimated an improvement factor of 15.6 regarding to previous works for Phase 1.

Apart from previous works which are proposed in the field of hardware implementations of the elliptic curve method, in [38] an exponentiation hardware is

presented supporting 512/1024/1536/2048-bit moduli. With this work, the author introduces the first publication of an DSP-based hardware multiplier (even before the proposal of [10]) using a Virtex-4 F12-10 FPGA device. Since [10] indicates that using DSP elements can improve modular computations, this work presents a serial architecture for modular multipliers.

A Montgomery multiplier is realized by using the *Modular Multiplication with Quotient Pipelining* algorithm. The implemented multiplier splits the to be multiplied operands into several blocks and multiplies them block-wise by using the DSP elements. Subsequent required addition tasks are performed by logic blocks to complete calculating the modular multiplication. Figure 2.3 gives a brief overview of the implemented Montgomery multiplier.

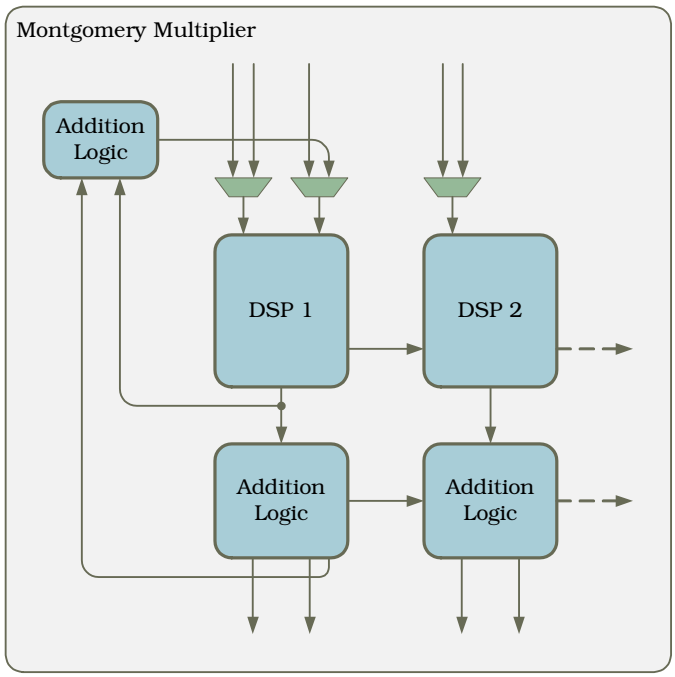

Figure 2.3.: Multiplier hardware design [38]

Since the implementation focus on RSA only, this work does not include separate results for the implemented multiplier.

3. Mathematical Background

In this chapter we discuss selected issues in the field of mathematics which are required for this work.

3.1. Complexity Theory

Complexity theory mainly aims to provide classifying computational problems and allows comparisons of different algorithms. Since the complexity of factoring algorithms, in particular of the Elliptic Curve Method, is discussed in Chapter 5, this section introduces some definitions and basic facts about complexity theory in general (cf. [28]).

Definition 3.1.1. *An algorithm is a well-defined computational procedure that takes a variable input and returns an output.*

In other words, an algorithm can be seen as a computer program which executes some operations sequentially.

Example: The addition of two integers is described in Algorithm 2. The input

Algorithm 2 Addition of positive multi-precision integers

Input: Two n-word integers $u = (u_{n-1,\cdots,u_0})_b$ and $v = (v_{n-1,\cdots,v_0})_b$.
Output: The $(n+1)$-word integer $w = (w_n, \cdots, w_0)$ such that $w = u + v$, w_n being 0 or 1.

1: $carry \leftarrow 1$
2: **for** $i = 0$ to $n-1$ **do**
3: $\quad w_i \leftarrow u_i + v_i + carry \pmod b$
4: $\quad carry \leftarrow \lfloor (u_i + v_i + carry)/b \rfloor$
5: **end for**
6: $w_n \leftarrow carry$
7: **return** $(w_n, \cdots, ..w_0)_b$

consists of two positive integers u and v which are expressed in base b and are

of size n. After performing several steps, the algorithm halts with the output $w = u + v$.

Definition 3.1.2. *The running time of an algorithm on a particular input is the number of primitive operations or steps executed. The worst-case running time of an algorithm is an upper bound on the running time for any input, expressed as a function of the input size. The average-case running time of an algorithm is the average running time over all inputs of a fixed size, expressed as a function of the input size.*

Example: In case of Algorithm 2, the primitive operation is the word-wise addition, the division by base b, and the modulo operation with respect to base b. The two latter operations are negligible, because division by the base can easily be realized by a right shift and calculating the least residue modulo the base can easily be realized by taking the right-most digit of the given integer. Furthermore, any assignments are negligible, too. Hence, the number of primitive operations, i.e. the number of word-wise additions, is n times one/two additions, depending on the *carry*. In worst case, there is always a *carry*, i.e. $u_i + v_i \geq b$ for all i. Then, the worst-case running time is $n \cdot 4$. But assuming in average $u_i + v_i \geq b$ holds half the cases for all i, the average running time is $n \cdot (1,5 + 1,5) = n \cdot 3$.

Definition 3.1.3 (big-O and small-o)**.** *Let f and g be two real functions. Then, f(n) = O(g(n)) if there exists a positive constant c and a positive integer n_0 such that $0 \leq |f(n)| \leq c \cdot |g(n)|$ for all $n \geq n_0$. In contrast, f(n) = o(g(n)) if for any positive constant $c > 0$ there exists a constant $n_0 > 0$ such that $0 \leq |f(n)| < c \cdot |g(n)|$ for all $n \geq n_0$.*

Example: The running time of algorithms cannot always be determined easily such as in case of Algorithm 2. To be able to compare two algorithms, one determines a formula like $f(x) = 2 \cdot n^3 + 3 \cdot n^2 \cdot log(n) + 75 \cdot n^2 + 7 \cdot n + 2000$ where $f(x)$ approximates the running time of the algorithm. With respect to the order notation, $f(x) = O(n^3)$ and obviously, $f(x) = o(n^4)$ where the latter approximation is not a very good one.

Definition 3.1.4. *A polynomial-time algorithm is an algorithm whose worst-case running time function is of the form $O(n^k)$, where n is the input size and k is a constant. Any algorithm whose running time cannot be so bounded is called an exponential-time algorithm. Furthermore, a subexponential-time algorithm is an algorithm whose worst-case running time function is of the form $e^{o(n)}$, where n is the input size.*

Example: Obviously, $f(x)$ is a polynomial-time algorithm. Problems which can be solved with such algorithms can be characterized as easy. All other problems such as the factorization problem are said to be hard.

The complexity of algorithms can also be classified by using the formula

$$L_N(\alpha, c) := e^{(c+o(1))(\ln N)^{\alpha}(\ln \ln N)^{1-\alpha}}$$

with $0 \leq \alpha \leq 1$ and $c > 0$. Depending on α, the algorithm is of polynomial complexity for $\alpha = 0$ and exponential complexity for $\alpha = 1$. For $0 < \alpha < 1$, the algorithm is of subexponential complexity.

3.2. Number Theory

Number theory is one of the oldest and most studied field in mathematics. It studies properties of numbers in general, and integers in particular. Since the elliptic curve method is used for factoring composite numbers, i.e. to find divisors and primes of a composite number, some definitions and facts about primes and composites are introduced in the following (cf. [28] and [36]).

Definition 3.2.1. *Let a, b be integers. Then a divides b (equivalently: a is a divisor of b, or a is a factor of b) if there exists an integer c such that $b = ac$. If a divides b, then this is denoted by $a|b$.*

Example: $a = 3$, $b_1 = 10$, and $b_2 = 12$. Then $a \nmid b_1$, because $b_1 = 2 \cdot 5$ and none of its divisors is 3. But $a \mid b_2$, because $b_2 = 2^2 \cdot 3$.

Proposition 3.2.2. *For all a, b, $c \in \mathbb{Z}$, the following are true:*

- $a|a$
- *If $a|b$ and $b|c$, then $a|c$.*
- *If $a|b$ and $a|c$, then $a|(bx + cy)$ for all x, $y \in \mathbb{Z}$.*
- *If $a|b$ and $b|a$, then $a = \pm b$.*
- *An integer c is a common divisor of a and b if $c|a$ and $c|b$.*

Definition 3.2.3. *A non-negative integer d is the greatest common divisor of integers a and b, denoted $d = \gcd(a, b)$, if*

- *d is a common divisor of a and b; and*
- *whenever $c|a$ and $c|b$, then $c|d$.*

Equivalently, $\gcd(a, b)$ is the largest positive integer that divides both a and b, with the exception that $\gcd(0, 0) = 0$.

Definition 3.2.4. *A non-negative integer d is the least common multiple of integers a and b, denoted $d = \mathrm{lcm}(a, b)$, if*

- *$a|d$ and $b|d$; and*
- *whenever $a|c$ and $b|c$, then $d|c$.*

Equivalently, the least common multiple of integers a and b $\operatorname{lcm}(a,b)$ is the smallest non-negative integer divisible by both a and b.

Examples:

- $\gcd(12,18) = 6$, divisors of 12 are $\{\pm1, \pm2, \pm3, \pm4, \pm6, \pm12\}$; divisors of 18 are $\{\pm1, \pm2, \pm3, \pm6, \pm18\}$; positive common divisors of 12 and 18 are $\{1, 2, 3, 6\}$.
- $\operatorname{lcm}(12,20) = 120$, divisors of 12 are $\{\pm1, \pm2, \pm3, \pm4, \pm6, \pm12\}$; divisors of 20 are $\{\pm1, \pm2, \pm4, \pm5, \pm10, \pm20\}$; positive integers divisible by 12 and 20 are $\{120, 240, 240 \cdot x\}, x \in \mathbb{N}$ and $x \geq 3$.

Remark: The *least common multiple* can also be computed by

$$\operatorname{lcm}(a,b) = (a \cdot b)/(\gcd(a,b)).$$

Definition 3.2.5. *An integer $p \geq 2$ is said to be prime if its only positive divisors are 1 and p. Otherwise, p is called composite. Two integers a and b are said to be relatively prime or coprime if* $\gcd(a,b) = 1$.

Remark: If p is prime and $p|ab$, then either $p|a$ or $p|b$ (or both).

Theorem 3.2.6 (Prime Number Theorem)**.** *Let $\pi(x)$ denote the number of prime numbers $\leq x$. Then*

$$\lim_{x\to\infty} \frac{\pi(x)}{x/ln\ x} = 1.$$

There is an infinite number of primes.

A more explicit estimate for $\pi(x)$ is given by

$$\pi(x) > \frac{x}{\ln x}$$

where π is the number of primes $\leq x$ and $x \geq 17$.

Example: The number of primes $\leq x = 10^{10}$ is bounded by $\pi(x) > 434,294,481$.

Theorem 3.2.7 (Fundamental Theorem of Arithmetic)**.** *Every integer $n \geq 2$ has a factorization as a product of prime powers:*

$$n = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3} \cdots p_k^{e_k},$$

where p_i are distinct primes, and e_i are positive integers. Furthermore, the factorization is unique up to rearrangement of factors.

The fundamental theorem of arithmetic is essentially for factoring integers. It says that every number is either a prime or a composition of primes. Without this assumption, number theory must be reinvented.

Definition 3.2.8. *g(n) is the largest gap that occurs below any prime $p \leq n$.*

Examples: The following examples indicate that the gap between two consecutive primes is small.

- $g(5) = 1$
- $g(11) = 3$
- $g(29) = 5$
- $g(97) = 7$
- $g(4652507) = 153$
- $g(17051887) = 179$
- $g(20831533) = 209$
- $g(47326913) = 219$

This observation is later on used to improve the elliptic curve method (cf. Section 5).

Definition 3.2.9. *Let $N \geq 1$ and let $|(\mathbb{Z}/N\mathbb{Z})^*|$ be denoted by $\varphi(N)$. The function φ is called the Euler totient function and one has $\varphi(N) = |\{x \mid 1 \leq x \leq N, \gcd(x, N) = 1\}|$.*

Theorem 3.2.10 (Euler)**.** *Let N and x be integers such that x is coprime to N, then*

$$x^{\varphi(N)} \equiv 1 \bmod N.$$

Theorem 3.2.11 (Fermat's little theorem)**.** *Let x and p be integers where p is prime, then*

$$x^{p-1} \equiv 1 \bmod p.$$

The Euler theorem is a generalization of Fermat's little theorem. The function φ is also called the *Euler phi function*, and it is often used to compute the number of elements of multiplicative groups (cf. Section 3.4).

3.3. Modular Arithmetic

Many cryptographic systems, also the elliptic curve method, are essentially based on modular arithmetic. Since most corresponding computations are performed by

computers, integers are expressed in radix b representation which can be processed by such hardware components. Beside the representation of integers, this section gives a brief introduction in modular addition, subtraction, and multiplication. In this regard, an efficient algorithm for modular multiplication suggested by Montgomery in [30] and one of its improvements proposed by Orup [33] are described in more detail. See also [28].

Definition 3.3.1. *The representation of a positive integer a as a sum of multiples of powers of b is such that*

$$a = \sum_{i=0}^{n} a_i b^i = a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0$$

where a_i is an integer with $0 \leq a_i < b$ for $0 \leq i \leq n$ and $a_n \neq 0$.

Remarks:

(i) The base b representation of a positive integer $a = a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0$ is usually written as $a = (a_n a_{n-1} \ldots a_1 a_0)_b$. The integers a_i, $0 \leq i \leq n$, are called *digits*. a_n is called the most significant digit or high-order digit; a_0 the least significant digit or low-order digit. If $b = 10$, the standard notation is $a = a_n a_{n-1} \ldots a_1 a_0$.

(ii) It is sometimes convenient to pad high-order digits of a base b representation with 0's; such a padded number will also be referred to as the base b representation.

(iii) If $(a_n a_{n-1} \ldots a_1 a_0)_b$ is the base b representation of a and $a_n \neq 0$, then the precision or length of a is $n + 1$. If $n = 0$, then a is called a single-precision integer; otherwise, a is a multiple-precision integer. $a = 0$ is also a single-precision integer.

Example: The base 2 representation of $a = 123$ is $(1111011)_2$. The base 4 representation of a is $(1323)_4$.

Definition 3.3.2. *If a and b are integers, then a is said to be congruent to b modulo m, written $a \equiv b \pmod{m}$, if n divides (a-b). The integer m is called the modulus of the congruence.*

Example:

- $3 \equiv 15 \pmod{12}$
- $-11 \equiv 17 \pmod{7}$

Remarks:

(i) $a \equiv b \pmod{m}$ if a and b leave the same remainder when divided by n.

(ii) $a \equiv a \pmod{m}$.

(iii) If $a \equiv b \pmod{m}$ then $b \equiv a \pmod{m}$.

(iv) If $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv c \pmod{m}$.

(v) If $a \equiv a_1 \pmod{m}$ and $b \equiv b_1 \pmod{m}$, then $a + b \equiv a_1 + b_1 \pmod{m}$ and $ab \equiv a_1 b_1 \pmod{m}$.

Two integers x and y can be added, subtracted, or multiplied modulo m using the following algorithms.

Algorithm 3 Modular addition

Input: Two positive integers $x, y < m$ and a modulus m
Output: $z \equiv x + y \pmod{m}$

1: Compute $x + y$
2: **if** $x + y \geq m$ **then**
3: Subtract the modulus m from z
4: **end if**
5: **return** z

Algorithm 4 Modular subtraction

Input: Two positive integers $x, y < m$ and a modulus m
Output: $z \equiv x - y \pmod{m}$

1: Compute $x - y$
2: **if** $x - y < 0$ **then**
3: Add the modulus m to z
4: **end if**
5: **return** z

Algorithm 5 is significantly more complex than Algorithms 3 and 4. The big disadvantage of this algorithm is the expensive division in step 2.

Montgomery proposed an algorithm in which this division is prevented. Therefore, integers are converted to the Montgomery representation. Let M be the positive modulus, an let R be an integer such that $R = b^k > m$ and $\gcd(m, R) = 1$. Then,

Algorithm 5 Classical modular multiplication

Input: Two positive integers x, y and a modulus m
Output: $x \cdot y \pmod{m}$
1: Compute $x \cdot y$
2: Compute the remainder r when $x \cdot y$ is divided by m
3: **return** r

the two integers of the Montgomery multiplication a and b are converted to the Montgomery representation such that $\tilde{x} \equiv x \cdot R \pmod{m}$ and $\tilde{y} \equiv y \cdot R \pmod{m}$. Addition of two such integers can be performed such that

$$\tilde{a} + \tilde{b} \equiv a \cdot R + b \cdot R \equiv (a + b) \cdot R \equiv \widetilde{(a + b)} \pmod{M}.$$

In contrast to the addition, the modular multiplication is adapted in a way that the Montgomery multiplication ($MMul$) of x and y is defined as: $MMul(x, y) := x \cdot y \cdot R^{-1} \pmod{m}$ which is described in Algorithm 6. The Montgomery multiplication

Algorithm 6 Montgomery multiplication

Input: Integers $M = (m_{n-1}m_{n-2} \dots m_1 m_0)_b$, $x = (x_{n-1}x_{n-2} \dots x_1 x_0)_b$, and $y = (y_{n-1}y_{n-2} \dots y_1 y_0)_b$ with $0 \leq x, y < M$, $R = b^n$ with $\gcd(M, b) = 1$, and $M' = -M^{-1} \bmod b$
Output: $x \cdot y \cdot R^{-1} \bmod M$
1: $A \leftarrow 0$ // $(A = (a_n a_{n-1} a_{n-2} \dots a_1 a_0)_b)$
2: **for** i from 0 to $(n-1)$ **do**
3: $u_i \leftarrow (a_0 + x_i \cdot y_0) \cdot M' \bmod b$
4: $A \leftarrow (A + x_i \cdot y + u_i \cdot M)/b$
5: **end for**
6: **if** $A \geq M$ **then**
7: $A \leftarrow A - M$
8: **end if**
9: **return** A

of two integers is performed such that

$$MMul(\tilde{x}, \tilde{y}) \equiv \tilde{x} \cdot \tilde{y} \cdot R^{-1} \equiv x \cdot R \cdot y \cdot R \cdot R^{-1} \equiv (x \cdot y) \cdot R \pmod{M}.$$

Integers $\tilde{x}$ in Montgomery representation can be converted back to the normal representation by performing

$$MMul(\tilde{x}, 1) \equiv x \cdot R \cdot R^{-1} \equiv x \pmod{M}.$$

Algorithm 6 benefits from preventing a multi-precision division. In step 4, a division by the base b is needed but since $(A + x_i \cdot y + u_i \cdot M)$ is a multiple of the base b, this division can be preformed efficiently by a right shift.

Orup invented some improvements of the Montgomery multiplication in [33]. Regarding to our hardware implementation, the high radix algorithm *Modular Multiplication with Quotient Pipelining* is described in more detail.

Let M be a modulus such that $M > 2$ and $\gcd(M, 2) = 1$. Let k, n be positive integers such that $4\tilde{M} < 2^{k \cdot n}$ where $\tilde{M} = (M' \bmod 2^{k(d+1)}) \cdot M$ and $d \geq 0$ is a delay parameter. Let R^{-1} be an integer such that $2^{k \cdot n} \cdot R^{-1} \bmod M = 1$ and let M' be given by $(-M \cdot M') \bmod 2^{k(d+1)} = 1$. Let A be the multiplicand such that $0 \leq A \leq 2\tilde{M}$. Let B be the multiplier in radix 2^k representation such that $0 \leq B \leq 2\tilde{M}$ and $B = \sum_{i=0}^{n+d} (2^k)^i \cdot b_i$ where $b_i \in \{0, 1, \ldots, 2^k - 1\}$ for $0 \leq i < n$ and $b_i = 0$ for $i \geq n$. The result of Algorithm 7 is $S_{n+d+2} \equiv A \cdot B \cdot R^{-1} \pmod{M}$ where $0 \leq S_{n+d+2} < 2\tilde{M}$.

Algorithm 7 Modular multiplication with quotient pipelining [33]

Input: $0 \leq A, B \leq 2\tilde{M}$, $M'' = \frac{\tilde{M}+1}{2^{k(d+1)}}$, delay parameter d and word width k
Output: $S_{n+d+2} \equiv A \cdot B \cdot R^{-1} \pmod{M}$ and $0 \leq S_{n+d+2} < 2\tilde{M}$
1: $S_0 = 0$, $q_{-d} = 0$, $q_{-d+1} = 0$, $\ldots$, $q_{-1} = 0$
2: **for** $i = 0$ to $n + d$ **do**
3: L1 : $q_i \equiv S_i \pmod{2^k}$
4: L2 : $S_{i+1} = \frac{S_i}{2^k} + q_{i-d} M'' + b_i A$
5: **end for**
6: $S_{n+d+2} = 2^{kd} \cdot S_{n+d+1} + \sum_{j=0}^{d-1} q_{n+j+1} \cdot 2^{kj}$
7: **return** S_{n+d+2}

In contrast to the classical Montgomery multiplication, this algorithm mainly benefits from the fact that no operation is needed for determining the quotient q_i. It can be determined only by the selection of the right-most digit.

Remarks:

- The maximal number of digits n of $B = \sum_{i=0}^{n-1} (2^k)^i \cdot b_i$ can be computed by

$$n = \left\lceil \frac{k(d+1) + h + 2}{k} \right\rceil$$

where $2 < M < 2^h$ and $\gcd(M, 2) = 1$. This is true, because of the following: Let $\tilde{M}_{max} = (2^{k(d+1)} - 1) \cdot (2^h - 1)$ be the maximal possible value of $\tilde{M}$ where $\tilde{M} = (M' \bmod 2^{k(d+1)}) \cdot M$. Since $4\tilde{M} < 4\tilde{M}_{max} + 1$ is equivalent to $4\tilde{M} < 2^{k(d+1)+h+2}$, $k(d+1) + h + 2 \stackrel{!}{=} k \cdot n$ such that $4\tilde{M} < 2^{k \cdot n}$ is satisfied.

- The system parameter M'' is bounded such that $1 \leq M'' < 2^h$. Since $(2^{k(d+1)}) \mid (\tilde{M}+1)$ [33], $M''_{min} = 1$ is true because $M'_{min} = 0$. $M''_{max} < 2^h$ is true since $(\tilde{M}_{max}+1)(2^{k(d+1)}) = 2^h$.
- Since $0 \leq A, B < M$ for the first input values A and B, Algorithm 7 can be used iteratively such that the result S_{n+d+2} can be reused as new input value for A or B.

Finally, it might be interesting that for determining M' an inverse must be computed for each modulus M which is usually cost-intensive. In case of Algorithm 7, the used modulus for computing the inverse is a power of two and the exponent is small. Hence, computing the inverse should not be expensive in case of this performing algorithm.

3.4. Algebraic Structures

Many modern crypto systems and the elliptic curve method are based on algebraic structures. Some information and basic facts about these structures are introduced in this section (cf. [6] and [28]).

Definition 3.4.1. *A group $\langle G, \circ \rangle$ consists of a set G with a binary operation $\circ$ on G satisfying the following axioms*

1. Closure. *$\circ$ is closured, that is for all $a, b \in G$ we have $a \circ b \in G$.*
2. Associativity. *$\circ$ is associative, that is for all $x, y, z \in G$ we have $(x \circ y) \circ z = x \circ (y \circ z)$.*
3. Identity element. *G has an identity element e, that is for all $x \in G$ we have $x \circ e = e \circ x = x$.*
4. Inverse. *For every $x \in G$ there exists y, an inverse of x such that $x \circ y = y \circ x = e$.*

Remarks:

(i) The group G is said to be commutative or abelian if the group operation is commutative, i.e. for all $a, b \in G$ we have $a \circ b = b \circ a$.

(ii) The identity element of a group is necessarily unique as well as the inverse of an element.

(iv) A set G with the binary operation $+$ is said to be an additive group denoted by $\langle G, + \rangle$ and the identity element is 0.

(v) A set G with the binary operation $\cdot$ is said to be a multiplicative group denoted by $\langle G, \cdot \rangle$ and the identity element is 1.

Definition 3.4.2. *The cardinality* $|\langle G, \circ \rangle|$ *or* $|G|$ *of group* G*, i.e. the number of its elements, is also called its order. A group is finite if its order is finite.*

Definition 3.4.3. *A non-empty subset* H *of a group* G *is a subgroup of* G *if* H *contains the identity element* e *of* G *and* H *is itself a group with respect to the operation of* G*. If* H *is a subgroup of* G *and* $H \neq G$*, then* H *is called a proper subgroup of* G*.*

Definition 3.4.4. *A group* G *is cyclic if there is an element* $\alpha \in G$ *such that for each* $b \in G$ *there is an integer* i *with*

$$b = \underbrace{\alpha \circ \alpha \circ \cdots \circ \alpha}_{i-times}.$$

Such an element α *is called a generator of* G*.*

Remarks:

(i) If G is a group and $a \in G$, then the set of all "powers" of a forms a cyclic subgroup of G, called the subgroup generated by a, and denoted by $\langle a \rangle$.

(ii) Every subgroup of a cyclic group G is also cyclic. More precisely, if the order of G is n, then for each divisor d of n, G contains exactly one cyclic subgroup of order d.

Definition 3.4.5. *Let* G *be a group and* $a \in G$*. The order of an element* a *is defined to be the least positive integer* t *such that*

$$\underbrace{a \circ a \circ \cdots \circ a}_{t-times} = e,$$

provided that such an integer exists. If such a t *does not exist, then the order of* a *is defined to be* ∞*.*

Remark: Let G be a group, and let $a \in G$ be an element of finite order t. Then $|\langle a \rangle|$, the size of the subgroup generated by a, is equal to t.

Theorem 3.4.6 (Lagrange)**.** *Let* G *be a finite group and* H *be a subgroup of* G*. Then the order of* H *divides the order of* G*. As a consequence, the order of every element also divides the order of* G*.*

Example: The multiplicative group $\langle \mathbb{Z}_{11}, \cdot_{11} \rangle$ is of order $\varphi(11) = 11 - 1 = 10$. Then, the order of possible subgroups can only be $1, 2, 5$, or 10.

Definition 3.4.7. *A ring $\langle R, \circ, \Diamond \rangle$ consists of a set R together with two binary operations $\circ$ and $\Diamond$ on R, satisfying the following axioms.*

1. *$\langle R, \circ \rangle$ is a commutative group with an identity element denoted by $e_\circ$.*
2. *$\Diamond$ is associative and has an identity element $e_\Diamond$ with $e_\Diamond \neq e_\circ$.*
3. *$\Diamond$ is distributive over binary operation $\circ$, that is for all $x, y, z \in R$, $x \Diamond (y \circ z) = x \Diamond y \circ x \Diamond z$ and $(y \circ z) \Diamond x = y \Diamond x \circ z \Diamond x$.*

Remark: The ring is said to be commutative, if the binary operation $\Diamond$ is commutative.

Example: $\langle \mathbb{Z}_M, +_M, \cdot_M \rangle$ is a commutative ring with $\mathbb{Z}_M = \{0, 1, 2, 3, \ldots, M-1\}$ and M is composite. For some elements a, i.e. for which $\gcd(a, M) = 1$ holds, the multiplicative inverse exists. But for all other non-zero elements, the inverse does not exist.

Definition 3.4.8. *A field is a commutative ring such that every nonzero element is invertible.*

Example: For prime numbers p, $\langle \mathbb{Z}_p, +_p, \cdot_p \rangle$ is a field since for all elements a, it is satisfied that $\gcd(a, p) = 1$.

Definition 3.4.9. *A finite field is a field $\mathbb{F}$ which contains a finite number of elements. The order of $\mathbb{F}$ is the number of elements in $\mathbb{F}$.*

Theorem 3.4.10. *For any primes p and any positive integers d there exists a finite field with $q = p^d$ elements. This field is unique up to isomorphism and is denoted by $\mathbb{F}_q$ or GF(q).*

Proposition 3.4.11. *The characteristic of a field $\mathbb{F}$ denoted by $char(\mathbb{F})$ is either 0 or a prime number p.*

Example: In case of $\mathbb{F}_{p^m}$ where p is prime and $m > 0$, the characteristic of this finite field is p.

Definition 3.4.12. *A finite field that does not contain any proper subfield is called a prime field.*

Example: The prime field of $\mathbb{F}_{p^m}$ is $\mathbb{F}_p$ where p is prime.

3.5. Elliptic Curves

Mathematicians have studied elliptic curves for over a hundred years. In 1985, N. Koblitz [18] and V. Miller [29] independently proposed using elliptic curves in cryptography. Since the elliptic curve method is based on elliptic curve arithmetics, this section introduces some definitions and facts about elliptic curves (cf. [6, 19, 37, 15, 27, 20]).

Definition 3.5.1. *An elliptic curve $\mathcal{E}$ over a field K denoted by $\mathcal{E}/K$ is defined by the simplified affine Weierstraß equation*

$$\mathcal{E} : y^2 = x^3 + ax + b$$

where $a, b \in K$, $char(K) \neq 2, 3$, and $\Delta \neq 0$ where Δ is the discriminant of $\mathcal{E}$. In this case, Δ is defined as $\Delta = -16(4a^3 + 27b^2)$.

Remarks:

(i) Note that if $\mathcal{E}$ is defined over K, then $\mathcal{E}$ is also defined over any extension field of K.

(ii) The condition $\Delta \neq 0$ ensures that the elliptic curve is smooth, that is, there are no points at which the curve has two or more distinct tangent lines.

Example: Figure 3.1 graphs an elliptic curve $\mathcal{E}(\mathbb{R})\backslash\{\infty\}$ over the field $\langle \mathbb{R}, +, \cdot \rangle$ where ∞ is the point at infinity.

Remark: An elliptic curve candidate whose discriminant Δ does not satisfy $\Delta \neq 0$ is obviously not an elliptic curve. Such a curve is denoted in the following as cubic curve because of the degree of the right side polynomial, i.e. $x^3 + ax + b$.

Definition 3.5.2. *The group law for elliptic curves over a field K with $char(K) \neq 2, 3$ is defined as follows:*

1. Identity. *$P \oplus \infty = \infty \oplus P = P$ for all $P \in \mathcal{E}/K$.*
2. Negatives. *If $P = (x, y) \in \mathcal{E}/K$, then $(x, y) + (x, -y) = \infty$. The point $(x, -y)$ is denoted by $-P$ and is called the negative of P; note that $-P$ is indeed a point in $\mathcal{E}/K$. Also $-\infty = \infty$.*
3. Point addition. *Let $P = (x_1, y_1) \in \mathcal{E}/K$ and $Q = (x_2, y_2) \in \mathcal{E}/K$, where $P \neq \pm Q$. Then $P \oplus Q = (x_3, y_3)$, where*

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \text{ and } y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1.$$

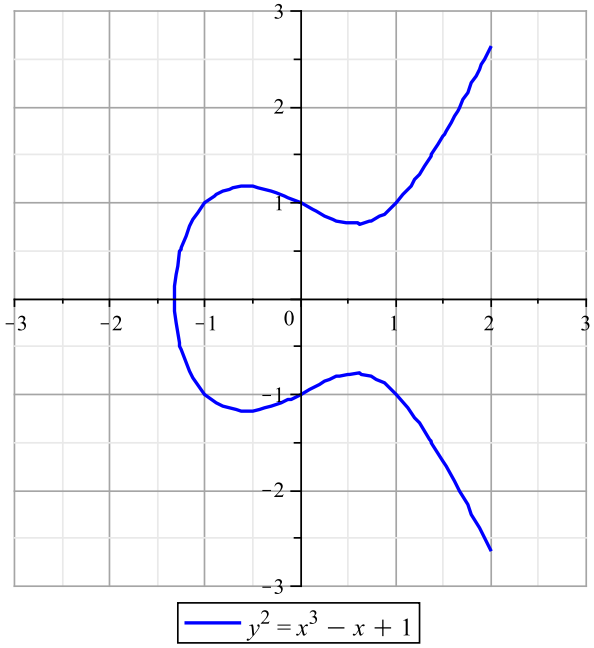

Figure 3.1.: Elliptic curve $\mathcal{E}$ over the finite filed $\mathbb{R}$

4. Point doubling. *Let* $P = (x_1, y_1) \in \mathcal{E}/K$, *where* $P \neq -P$. *Then* $2 * P = (x_3, y_3)$, *where*

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \text{ and } y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1.$$

Example: Figures 3.2 and 3.3 illustrate the group law for elliptic curves over the field $\langle \mathbb{R}, +, \cdot \rangle$ geometrically.

Theorem 3.5.3. *Let* $\mathcal{E}$ *be an elliptic curve given by the Weierstraß equation and let* $\oplus$ *be the rules defined in Definition 3.5.2. Then,* $\langle E, \oplus \rangle$ *is an abelian group with identity element* ∞. *If* $\mathcal{E}$ *is defined over a field* K, *then* $\mathcal{E}/K$ *is a subgroup of* $\mathcal{E}$.

Definition 3.5.4. *The point multiplication or scalar multiplication on an elliptic curve* $\mathcal{E}$ *of a scalar* $k \in \mathbb{N}$ *and a point* $P \in \mathcal{E}$ *is defined such that*

$$k * P = \underbrace{P \oplus P \oplus \cdots \oplus P}_{k-times}.$$

Since $\langle \mathcal{E}, \oplus \rangle$ is an abelian group, some properties are recapitulated in the following.

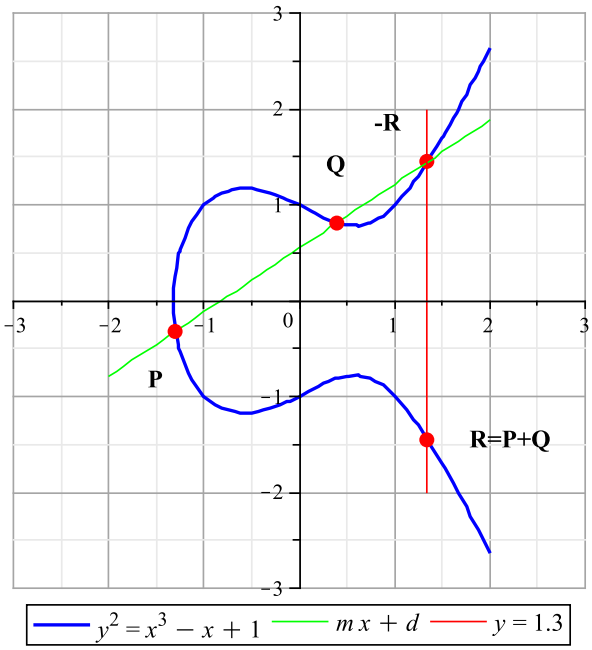

Figure 3.2.: Point addition on elliptic curves $\mathcal{E}_i$ over the finite field $\mathbb{R}$

(i) The number of elements of a group is called its cardinality or order. The order is denoted by $|\langle \mathcal{E}, \oplus \rangle|$ or $\#\mathcal{E}(K)$.

(ii) The order n of an element $P \in \mathcal{E}$ is the least positive integer such that $n * P = \infty$.

(iii) $\langle E, \oplus \rangle$ is called a cyclic group $\mathcal{E}$ if there exists an element $P \in \mathcal{E}$ which generates all elements of $\mathcal{E}$ and is denoted by $\langle P \rangle$. Element P is then called a generator of G. The order of the generator is of the same size as the group order.

(iv) For all points $P_i \in \mathcal{E}$ is true that $k * P_i = \infty$ where k is the group order of $\mathcal{E}$.

Theorem 3.5.5. *Let K be a finite field with $char(K) \neq 2, 3$ and let $\mathcal{E}$ be an elliptic curve over K. The simplified projective Weierstraß equation is defined as*

$$\mathcal{E} : Y^2Z = X^3 + aXZ^2 + bZ^3$$

where $a, b \in K$. The projective point $(X : Y : Z), Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z)$. The point at infinity ∞ corresponds to $(0 : 1 : 0)$, while the negative of $(X : Y : Z)$ is $(X : -Y : Z)$.

Theorem 3.5.6. *Let $\mathcal{E}_M$ be an elliptic curve expressed in Montgomery's form,*

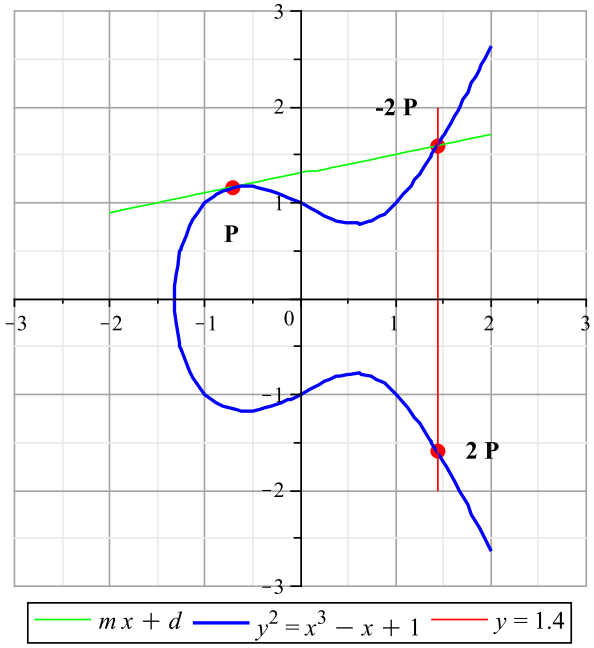

Figure 3.3.: Point doubling on elliptic curves $\mathcal{E}_i$ over the finite field $\mathbb{R}$

that is

$$\mathcal{E}_M : By^2 = x^3 + Ax^2 + x.$$

Theorem 3.5.7. *Let $P = (X_P : Y_P : Z_P)$ and $Q = (X_Q : Y_Q : Z_Q)$ be two points on $\mathcal{E}_M$. Addition and doubling on $\mathcal{E}_M$ is given by the following formulas where $Y_{P,Q}$ never appears.*

Addition *$(P \neq Q)$:*

$$X_{P+Q} = Z_{P-Q}((X_P - Z_P)(X_Q + Z_Q) + (X_P + Z_P)(X_Q - Z_Q))^2$$
$$Z_{P+Q} = X_{P-Q}((X_P - Z_P)(X_Q + Z_Q) - (X_P + Z_P)(X_Q - Z_Q))^2$$

Doubling *$(P = Q)$:*

$$4X_P Z_P = (X_P + Z_P)^2 - (X_P - Z_P)^2$$
$$X_{2P} = (X_P + Z_P)^2 (X_P - Z_P)^2$$
$$Z_{2P} = 4X_P Z_P((X_P - Z_P)^2 + ((a+2)/4)(4X_P Z_P))$$

The y-coordinate can be retrieved, if needed, by the following formula

$$y_n = \frac{(x_1 x_n + 1)(x_1 x_n + 2a) - 2a - (x_1 - x_n)^2 x_{n+1}}{2by_1}$$

where $P = (x_1, y_1)$. x_R and x_{R+1} are the affine x-coordinates of the points $n * P$

and $(n+1) * P$*, respectively.*

Theorem 3.5.8. *A curve in simplified affine Weierstraß can be converted to Montgomery' form if and only if*

- *the polynomial* $x^3 + ax + b$ *has at least one root* α *in* $\mathbb{F}_p$*,*
- *the number* $3\alpha^2 + a$ *is a quadratic residue in* $\mathbb{F}_p$*.*

Remark: It is alway possible to convert a curve in Montgomery's form into simplified Weierstraß equation, putting $a = 1/B^2 - A^2/3B^2$ and $b = -A^3/27B^3 - aA/3B$. But the converse is false. Not all elliptic curves can be written in Montgomery's form.

A scalar multiplication of a scalar k and a point P which is an element of an elliptic curve can be performed by using the Montgomery ladder (Algorithm 8). Note that the Montgomery ladder is able to perform scalar multiplications for any

Algorithm 8 Montgomery ladder

Input: $P_0 = (x_0 : z_0)$, $k = (k_{l-1}, k_{l-2}, ..., k_2, k_1, k_0)_2$, $\mathcal{E}_M$
Output: $k * P_0$

```
Q ⇐ P_0
P ⇐ 2 * P_0
for i = l − 2 downto 0 do
  if k_i = 1 then
    Q ⇐ P ⊕ Q
    P ⇐ 2 * P
  else
    P ⇐ P ⊕ Q
    Q ⇐ 2 * Q
  end if
end for
return Q
```

kind of elliptic curves including curves in Montgomery 's form.

4. Hardware Background

In this chapter we give basic information about the hardware we used for our implementation.

4.1. Overview of Integrated Circuits

Integrated circuits (IC) are miniaturized electronic circuits with a high component density. Figure 4.1 gives a brief overview of subclasses of such circuits. Usually,

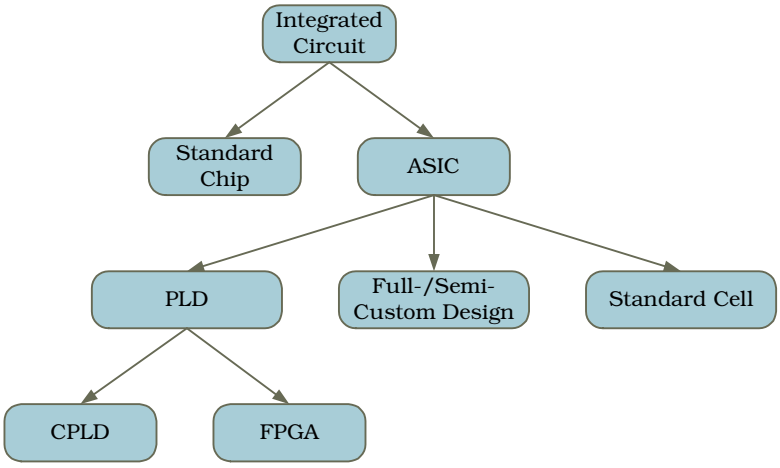

Figure 4.1.: Technology overview

ICs are classified in two subclasses. On the one hand, standard chips like microprocessors are developed for a large part of the market. Such integrated circuits are mainly used for multi-functional tasks. On the other hand, application specific integrated circuits (ASIC) are used for implementing customer-specific hardware which may not be realized by standard chips. Beside the class of standard cells and full-/semi-custom design, programmable logic devices (PLD) are part of this class. Standard cells have fixed interfaces and can be wired in a way such that the combination results in the desired functionality. In contrast, full-/semi-custom design allows to define any kind of interface where the hardware is realized by

defining masks which are used to etch the functionality into the hardware. Typically, programmable logic devices consists of already etched hardware which can be programmed in different ways. For example, field programmable gate arrays (FPGA) are programmed by configuring the hardware (e.g., by wiring lines inside a switch matrix). Another main difference between full-/semi-custom design and field programmable gate arrays is the fact, that FPGA devices can be programmed completely by the customer whereas full-/semi-custom designed devices are mainly programmed by some technology vendor.

4.2. Virtex-4

The field programmable gate array is introduced by means of the Xilinx Virtex-4 SX35 FPGA. Its structure is illustrated in Figure 4.2. In general, FPGA devices

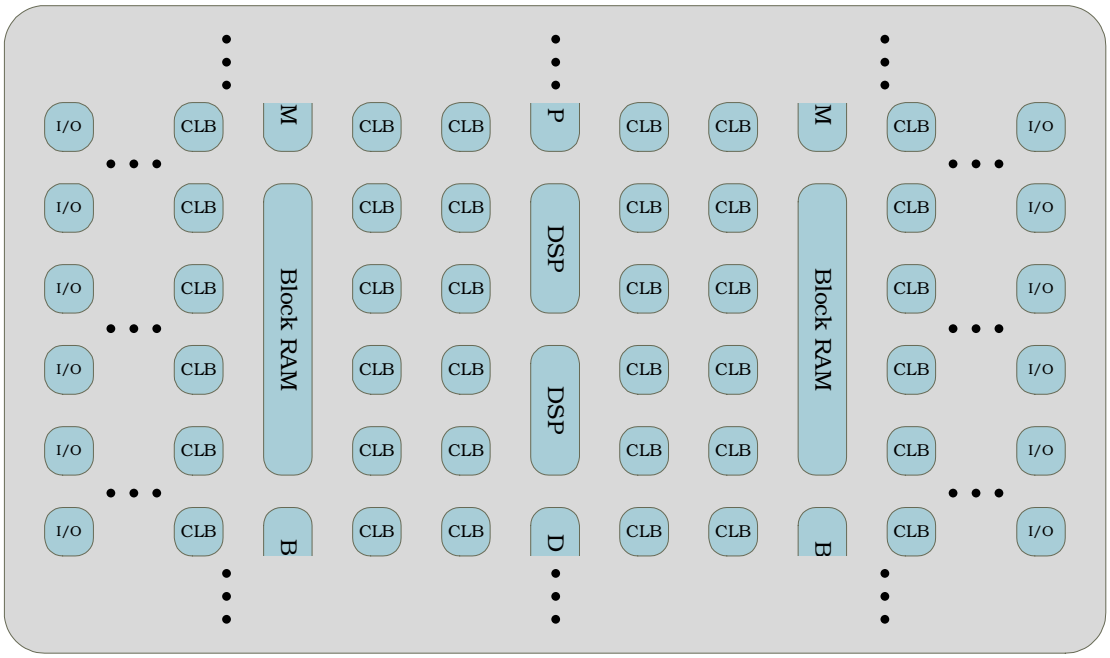

Figure 4.2.: Virtex-4 schematic structure

consist of a lot of identical and free programmable control logic blocks (CLB) and input/output (I/O) blocks which are arranged in a matrix-like structure where each row is connected to each column. Those blocks can be connected to each other by a matrix switch which is available for each of such blocks. The functionality of a circuit is programmed inside the CLBs whereas the communication to other components outside the FPGA is provided by the I/O blocks.

In contrast to a typical FPGA, Virtex-4 SX35 FPGA devices additionally integrate hardware macros like digital signal processing (DSP) and block RAM (BRAM) elements. Since these components are intensively used in our implementation, CLBs, DSPs, and block RAMs are described in more detail next.

4.2.1. Control Logic Block

In Figure 4.3, a control logic block of a Virtex-4 SX 35 FPGA is depicted. Each

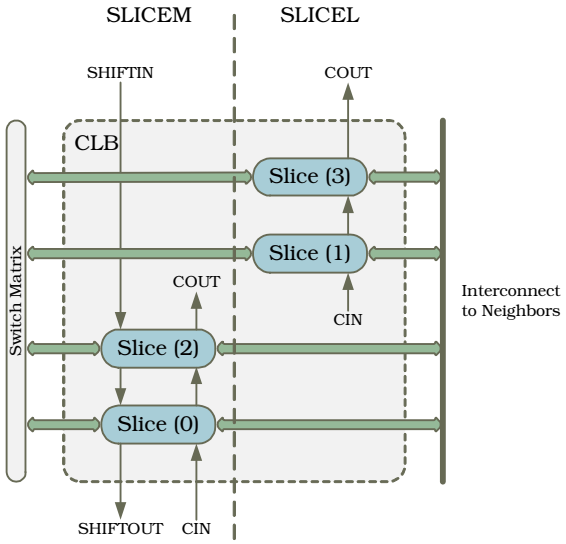

Figure 4.3.: Virtex-4 control logic block

CLB is connected to one matrix switch which allows the connection to other components of the FPGA. The control logic block consists of four slices which are grouped pair-wise. The left hand group is called SLICEM whereas the name of the right one is SLICEL. Each slice group has its own carry logic which can be connected to other slice groups placed below or above a considered CLB. Both slice groups consists of two function generators, two storage elements, its own carry logic, and some multiplexer and arithmetic gates. In addition to SLICEL, slices of SLICEM can be configured to act as additional storage (distributed RAM) or 16-bit shift register.

The two function generators can be configured as lookup table (LUT) with four inputs. The LUT can implement any boolean function of four inputs where the carry propagation delay through the LUT is independent from the realized function. Signals of the implemented function can leave the slice directly without passing any storage element.

The lookup table available in slices of SLICEM can also be configured as distributed RAM, ROM, and shifter. In case of distributed RAM, one LUT can be implemented as 16×1-bit RAM where it can be read and write synchronously to the clock signal which is the same as for the storage element. Lookup tables realized in SLICEM can also be configured as a 16×1-bit ROM which is loaded at the device configuration step. Finally, each SLICEM lookup table can be implemented as 16-bit shift register. Such shifters can be cascaded to a larger shift register by `SHIFTIN` and `SHIFTOUT` signals which can be wired to below and above placed slices.

Multiplexers can be implemented by using lookup tables and multiplexer integrated in each slice. Figure 4.4 pictures an 4x1 multiplexer. This multiplexer is

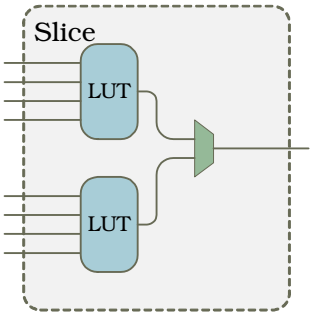

Figure 4.4.: Virtex-4 4×1 multiplexer

realized by two lookup tables of the same slice. The output of the tables are connected to the integrated multiplexer which routes the correct signal to the output. Wide multiplexers are realized by several lookup tables of different slices which are used in a level of logic.

4.2.2. Digital Signal Processing

The digital signal processing (DSP) element is a fast hardware macro for integer arithmetic at least available on FPGA devices of the Virtex-4 family. Among others, it supports arithmetic functions such as multiplication, multiplication followed by accumulation, multiplication followed by addition, three-input addition, and 17-bit shifting. The Virtex-4 SX35 FPGA integrates 192 DSP elements arranged in four vertical columns with 48 DSP elements per column. The structure of the Virtex-4 also indicates that DSP elements can be cascaded by fast connections to the next above or below placed DSP. Figure 4.5 illustrates a possible configuration of a DSP element. Usually, signals `A` and `B` are used as input for the

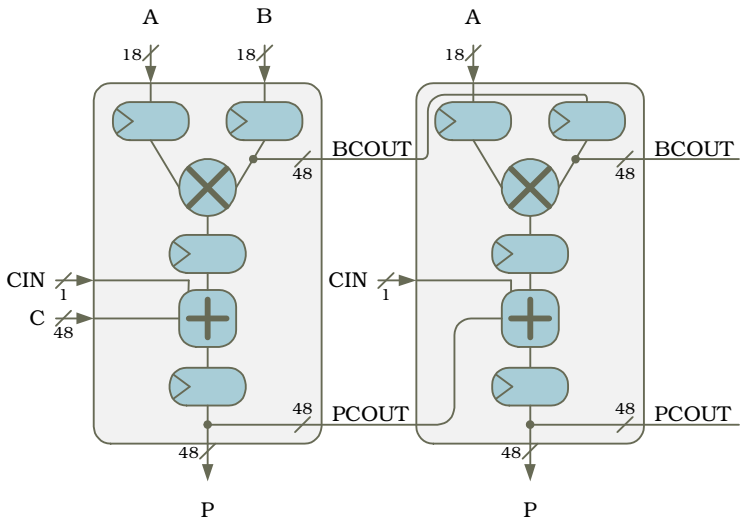

Figure 4.5.: Two cascaded DSP elements

signed 18×18-bit two's complement multiplier. It is followed by a signed 48-bit signed adder/subtracter/accumulator element which also uses signals `C` and `CIN` for further inputs. The result is returned as signal `P` at the output port of the DSP element. The same output is also routed to `PCOUT` which can be forwarded as well as the `BCOUT` signal to a cascaded DSP element. Since an internal 48-bit bus is available, data is extended suitable to this size. The DSP functionality is configured by the `OPMODE` signal which can be altered at each clock cycle. The general function of the DSP element in Figure 4.5 is $P = C \pm (A \cdot B + CIN)$ when operation mode `0x35` is used. The figure also indicates that various pipeline register can be configured.

Furthermore, Figures 4.6 and 4.7 show DSP elements configured to perform the functions $P = PCOUT = P \pm (A \cdot B + CIN)$ (operation mode `0x25`) and $P =$

$PCOUT = Shift(PCIN) \pm (C + P \pm CIN)$ (operation mode `0x5E`) which are intensively used by our implementation.

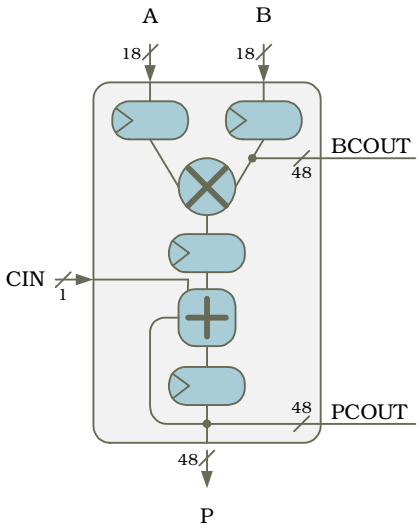

Figure 4.6.: DSP performing $P = PCOUT = P \pm (A \cdot B + CIN)$

Figure 4.7 also indicates the 17-bit shift register for multi-precision arithmetic.

4.2.3. Block RAM

Block RAM elements (BRAM) are multi-functional fast storage elements. The Virtex-4 SX35 FPGA consists of 192 of such elements which are arranged in eight columns on the left and right hand side of the DSP columns. In each of the eight columns, 24 BRAM elements are placed. Figure 4.8 shows a BRAM which can store 18Kbit of data. One BRAM has two input and output ports which are symmetrical and totally independent sharing only the stored data. Each port can be configured separately such that the input size can be 1-bit, 2-bit, or 36-bit where the corresponding data width is 16Kbit, 8Kbit, or 512-bit (16Kx1, 8Kx2, or 512x36). During write and read operations which are performed synchronously, the output can reflect the new data being written, the previous data being overwritten, or the output can remain unchanged. In the latter case, the previous read data is returned. To increase the performance of the block RAM elements, pipeline register can be configured. Furthermore, two or more BRAM elements can be combined to build a deeper storage.

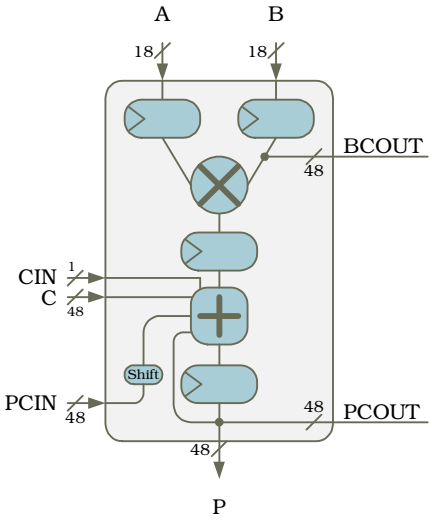

Figure 4.7.: DSP performing $P = PCOUT = Shift(PCIN) \pm (C + P \pm CIN)$

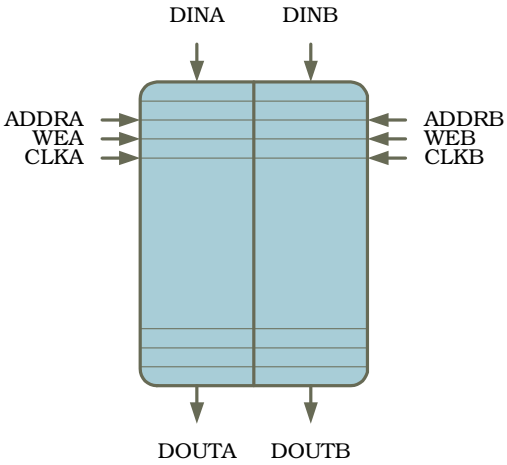

Figure 4.8.: Virtex-4 block RAM

4.3. VHDL Development Flow

VHDL is a hardware description language and it is used for developing hardware on a more abstract layer. VHDL stands for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. Using VHDL a hardware is modeled in four main phases: design, specification, development, and implement phase. Figure 4.9 gives an overview of the hardware modeling process. Modeling a hardware

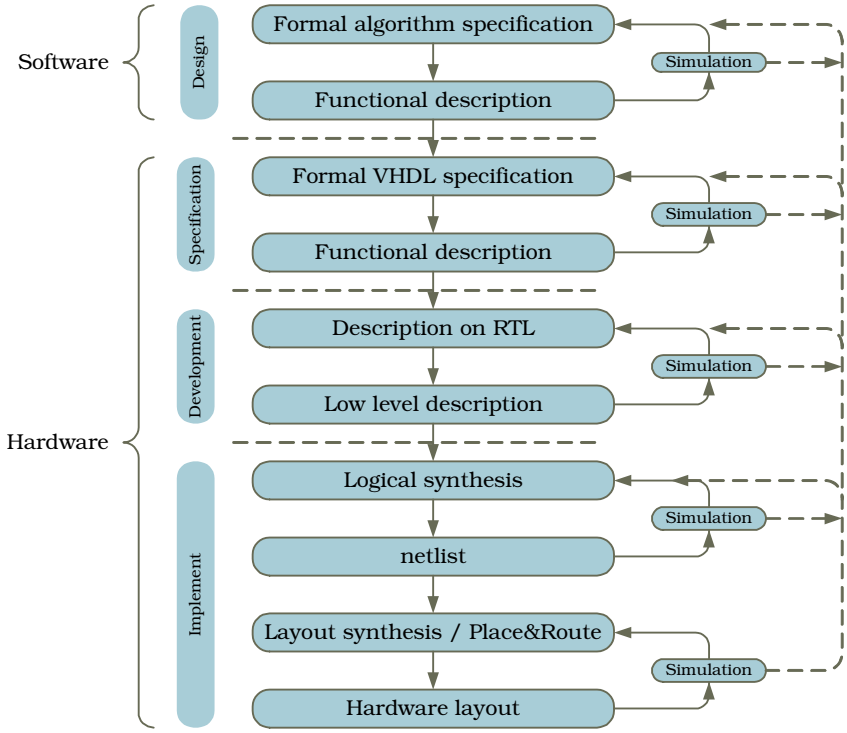

Figure 4.9.: VHDL development flow

system starts with a design in which the desired algorithm is implemented on a very high level. One can take advantage of software modeling tools to ensure that principle functionalities to be implemented system are correct and realizable in hardware. Simulations supports the design modeling (e.g., by finding errors and optimizing the algorithm, respectively).

Next, a formal specification is described by use of several VHDL models defining individual components of the desired system. Here, the behavior of a system can be described algorithmically. The system in this development phase can be simulated concerning its functionality by means of a VHDL simulator.

The development phase substantiates the previously defined VHDL models. This

results in a description which can be synthesized in terms of finite state machines.

The implement phase consists of two main parts. In the logical synthesis, the VHDL source code is transformed to an optimized circuit of logical gates. The netlist includes all used gates and their connections to each other. In the last step, the mapping of the netlist and physical function blocks is created. This layout synthesis depends on the manufacturer's library which defines the implementation of the different gates. On this layer, the simulated time behavior is equal to the expected time behavior of the target hardware.

In each phase, the functionality can be simulated such that misbehavior can be eliminated on each level of development.

5. Elliptic Curve Method

The elliptic curve method (ECM) [25] firstly invented in 1985 by H.W. Lenstra Jr. is a good choice for factoring mid-sized composite numbers of up to 200-bit numbers [34]. Its idea is based on J. Pollard's $p-1$ method [28] but it is more efficient. In contrast to the $p-1$ method, the ECM is more flexible in finding factors.

The idea behind the $p-1$ method is to compute a product Q of all powers of primes where all primes q_i of Q are less than a bound B_1 and all powers of primes q_i^l are $q_i^l \leq n$, where n is the number to be factored. Q is said to be B_1-smooth, i.e. all prime factors of Q are $q_i \leq B_1$. If the group order $p-1$ of the group $\mathbb{Z}_p^*$ is also B_1-smooth, then Q is a multiple of $p-1$ where p is a factor of n. As a consequence of Fermat's little theorem (Theorem 3.2.11), the greatest common divisor of $a^Q - 1$ and n is a factor of n. If the $p-1$ method fails, i.e. $p-1$ is not B_1-smooth, then a higher bound B_2 can be chosen. Again, Q is calculated with respect to the new bound B_2, expecting that $p-1$ is B_2-smooth now.

In contrast, if ECM fails, i.e. the group order $|\mathcal{E}(\mathbb{Z}_q)|$ of a chosen curve $\mathcal{E}/\mathbb{Z}_M$ is not smooth with respect to a bound B, curve parameters can be changed which results in a different group order. Then, there is another chance that $|\mathcal{E}(\mathbb{Z}_q)|$ is B-smooth without increasing bound B. This takes advantage in the runtime of the ECM algorithm compared to the $p-1$ method. Pollard's $p-1$ method requires $O(B \log_p(B))$ modular multiplications whereas the complexity of the elliptic curve method is reduced to $O(\exp((\sqrt{2}+o(1))\sqrt{\log q}\log\log q)M(N))$ where $M(N)$ is an upper bound for the time needed to perform a single point addition on elliptic curves modulo N.

Based on Lenstra's observation, Brent [3] and Montgomery [31] deeply improved Lenstra's ECM algorithm by adding a continuation (also called Phase 2) which makes ECM even more efficient. In the following, the standard continuation and improved standard continuation are described in more detail. Other improvements such as Brent's birthday paradox continuation [3] or Montgomery's FFT continuation [32] are not discussed here, because their implementation seems to be too complex in hardware.

5.1. ECM Algorithm

Lenstra's ECM algorithm (Phase 1) uses elliptic curve arithmetic with projective coordinates to find factors in composite numbers. Let $M \in \mathbb{N}$ be a composite number such that $q \in \mathbb{N}$ is an unknown factor of M. Let also $B_1, B_2 \in \mathbb{N}$ be two upper smoothness bounds, let p be all primes that are $\leq B_1$, and let p^{e_p} be all powers of primes $\leq B_2$. Then, $\mathcal{E}/\mathbb{Z}_M$ is actually not an elliptic curve but a cubic one, because $(\mathbb{Z}_M, +_M, \cdot_M)$ is not a finite field since M is composite. Nevertheless, elliptic curve arithmetic can also be used for adding and doubling points on a cubic curve. Therefore, curve parameters and an initial point $P \in \mathcal{E}/\mathbb{Z}_M$ which is not the point at infinity ($P \neq \mathcal{O}$) are chosen randomly. Let also be k the least common multiple of all powers of primes $q_i \leq B_1$ that are $q_i^l \leq B_2$. For such a k, the scalar multiplication $Q = k * P$ is calculated by using elliptic curve arithmetic. Finally, the greatest common divisor of M and the z-coordinate of the resulting point Q is calculated. If it results in an integer between 1 and M, a non-trivial factor is found. Algorithm 9 outlines these considerations and a more detailed view is described afterward.

Algorithm 9 ECM: Phase 1

Input: $M \in \mathbb{N}$, smoothness bounds B_1 and B_2, list of prime numbers $p \leq B_2$
Output: $q \in \mathbb{N}$, so that $q \mid M$; or fail

1: Choose a random curve $\mathcal{E}/\mathbb{Z}$ and a point $P \in \mathcal{E}(\mathbb{Q}), P \neq \mathcal{O}$.
2: Calculate

$$k = \prod_{p<B_1} p^{e_p} \qquad e_p = \max\{n \in \mathbb{N} : p^n \leq B_2\}$$

3: Calculate $Q = kP = (x_Q : y_Q : z_Q)$
4: **if** $1 < q = \gcd(z_Q, M) < M$ **then**
5: **return** q
6: **else**
7: **return** fail // *or continue with Phase 2*
8: **end if**

Assuming q to be a prime factor of a composite number M, then $\mathcal{E}$ over the finite field $(\mathbb{Z}_q, +_q, \cdot_q)$ would be an elliptic curve $(\mathcal{E}/\mathbb{Z}_q)$ with the group order $|\mathcal{E}(\mathbb{Z}_q)|$ and for any point $P \in \mathcal{E}/\mathbb{Z}_q$, it holds that $|\mathcal{E}(\mathbb{Z}_q)| * P = \mathcal{O}$. This also holds true for all multiples $[l \cdot |\mathcal{E}(\mathbb{Z}_q)|] * P = \mathcal{O}$ with $l \in \mathbb{N}$. Let k be an integer such that $k = l \cdot |\mathcal{E}(\mathbb{Z}_q)|$, then $k * P = \mathcal{O} = (0 : 1 : 0)$.

In case of ECM, we do not choose an elliptic curve but $\mathcal{E}/\mathbb{Z}_M$ which is a cubic curve, where M is the composite number to be factored. Now, all calculations are done in $\mathbb{Z}_M$ which means that $k * P$ results in a point $Q = (x_Q : y_Q : z_Q)$. Since q is

an unknown prime factor of M and $\mathcal{O}$ is $(0:1:0)$, we know that $z_Q \equiv 0 \pmod{q}$. In other words, z_Q is a multiple of q, if $k = l \cdot |\mathcal{E}(\mathbb{Z}_q)|$. Hence, q divides $\gcd(z_Q, M)$ and $\gcd(z_Q, M)$ computes a factor of M. Thus, ECM allows us to find an unknown factor q of a composite number M.

Note that there is a low probability that $k * P$ cannot be computed in $\mathbb{Z}_M$. In such a case, also a non-trivial factor of M is found. Assuming affine coordinates, we compute $\frac{y_2-y_1}{x_2-x_1}$ in case of point adding and $\frac{3x_1^2+a}{2y_1}$ in case of point doubling. Such calculations can only be performed modulo M, if and only if $\gcd(x_2 - x_1, M) = 1$ and $\gcd(2y_1, M) = 1$ is satisfied. In other cases, the greatest common divisor is a factor of M.

The elliptic curve method can be used to find non-trivial divisors of a composite number n. Regarding to complexity, Algorithm 9 (Phase 1) can be repeated with several random chosen curves and suitable parameters. In this case, let $n \in \mathbb{N}$ be the number to be factored which is not a prime power and not divisible by two or three. Let also g be a positive integer.

A non-trivial divisor of n can be found with probability of at least $1 - e^{-g}$ within time

$$g \cdot K(p) \cdot M(n)$$

where p denotes the least prime divisor of n and $K(x)$ is a function such that $K(x) = \exp(\sqrt{(2 + o(1)) \log x \log\log x})$. $M(n)$ is an upper bound for the time needed to perform a single point addition on an elliptic curve mod n where $M(n) \approx O((\log n)^2)$.

All divisors of n can be found by repeating Algorithm 9 with several random chosen curves and suitable parameters. Unknown non-trivial divisors of n can be found within time

$$g \cdot K(p') \cdot M(n)$$

where p' is the corresponding least divisor of n, with the exception of the largest prime factor. The time which is needed to perform the total factorization of n is expected to be at most

$$L(n)^{1+o(1)}$$

for $n \to \infty$ and where $L(x)$ is a function such that $L(x) = e^{\sqrt{\log x \log\log x}}$.

Thus, the elliptic curve method is applicable for factoring numbers which are a composition of mostly small prime numbers. Hence, the worst case occurs if the second largest prime factor of n is not smaller than $\sqrt{n}$. Then, n is the product of some small primes and two large prime numbers that are of the same order of magnitude.

In contrast to the elliptic curve method, the running time of other factoring algorithms of the same complexity is basically independent of the size of the prime factors of n. ECM is substantially faster than these algorithms if the second largest prime factor of n is much smaller than $\sqrt{n}$.

5.2. Standard Continuation

If $1 < q = \gcd(z_Q, M) < M$ is not satisfied, which means that Algorithm 9 could not find a factor of M, it might be possible that k just miss a large factor x, such that $k \cdot x = l \cdot |\mathcal{E}(\mathbb{Z}_q)|$. In such a case, $k * P = \mathcal{O}$ holds. To increment k, a prime factor p can be used to calculate $p * Q_0 = p * (k * P) = (p \cdot k) * P$. If $p * Q_0 = \mathcal{O}$ is satisfied, again a non-trivial factor of M is found. The probability that k contains all prime factors of the group order $|\mathcal{E}(\mathbb{Z}_q)|$ can be increased, if $p_i * Q_0$ is computed iteratively where p_i is between B_1 and B_2. The resulting z-coordinates are multiplied by each other to $d = \prod_i z_{pQ_i}$. The greatest common divisor of d and M calculates a factor of M if the group order is B_2-smooth. This idea is commonly denoted as standard continuation which is outlined in Algorithm 10.

Algorithm 10 ECM: Phase 2 (standard continuation)

Input: $M \in \mathbb{N}$, smoothness bounds B_1 and B_2, list of prime numbers $p \leq B_2$
Output: $q \in \mathbb{N}$, so that $q \mid M$; or fail

1: $d := 1$
2: **for** $p : B_1 < p \leq B_2$ prime **do**
3: Calculate $pQ = (x_{pQ} : y_{pQ} : z_{pQ})$
4: $d \Leftarrow d \cdot z_{pQ} \pmod M$
5: **end for**
6: **if** $1 < q = \gcd(d, M) < M$ **then**
7: **return** q
8: **end if**
9: **return** fail // *or restart with other curve parameters*

There is no advantage in using Phase 2 if $p_i * Q$ is computed such as in Phase 1. It is more efficient to exploit the fact that the maximal gap of two successive primes D is small (e.g., $D = 153$ for all primes less than $4.5 \cdot 10^6$). Thus, we can precompute a small set of points $i * Q$, where $i = 2, 4, 6, 8, \ldots, D$, and determine consecutive primes p_i whose difference is not greater than D. Then, $p_i * Q$ can be computed as described in Table 5.1.

Table 5.1.: Determination of product d w/o performing scalar multiplications

$p_1 * Q =$	$p_1 * Q$			$d = z_{p_1 Q}$
$p_2 * Q =$	$p_1 * Q$	$+$	$(p_2 - p_1) * Q$	$d = d \cdot z_{p_2 Q}$
$p_3 * Q =$	$p_2 * Q$	$+$	$(p_3 - p_2) * Q$	$d = d \cdot z_{p_3 Q}$
		$\ldots$		
$p_{j+1} * Q =$	$\underbrace{p_j * Q}$	$+$	$\underbrace{(p_{j+1} - p_j) * Q}$	$d = d \cdot z_{p_{j+1} Q}$
	computed iteratively		precomputed	

Essentially, such a way of computing $p_{j+n} * Q$ replaces the scalar multiplication by a point addition in each iteration. Finally, computing $\gcd(d, M)$ might result in a factor of M.

5.3. Improved Standard Continuation

Additionally to the method described in Table 5.1, the standard continuation can be improved by a suggestion of Montgomery [31]. Montgomery's key idea is, to test if $\gcd(z_{p_i * Q_0}, M)$ results in a non-trivial factor without computing $p_i * Q_0$. Therefore, all numbers between B_1 and B_2 are represented by $m \cdot D \pm j$ with $M_{MIN} = \left\lfloor (B_1 + \frac{D}{2})/D \right\rfloor \leq m \leq M_{MAX} = \left\lceil (B_2 - \frac{D}{2})/D \right\rceil$ and $1 \leq j \leq \left\lfloor \frac{D}{2} \right\rfloor$. Since, only primes of this representation are of interest, the set J_S includes all primes between B_1 and B_2 and a few composites such that $J_S = \left\{ j : 1 \leq j \leq \left\lfloor \frac{D}{2} \right\rfloor, \gcd(j, D) = 1 \right\}$. Since $p_i = m \cdot D \pm j$ is satisfied, we know that if $p_i * Q_0 = \mathcal{O} \Leftrightarrow (m \cdot D \pm j) * Q_0 = \mathcal{O} \Leftrightarrow m \cdot D * Q_0 = \mp j * Q_0$. This is satisfied if and only if $x_{mD*Q_0} \cdot z_{j*Q_0} - x_{j*Q_0} \cdot z_{mD*Q_0} \equiv 0 \pmod{q}$. Thus, $m \cdot D * Q_0$ and $j * Q_0$ can be computed instead of $p_i * Q_0$ which reduces the complexity of the standard continuation.

An algorithm for implementing Phase 2 efficiently is described in [13, 14] which, among others, takes advantage of Montgomery's suggestion. In the precomputation step, a prime_table is computed for all pairs (m, j) where $j \in J_S$ and $m \in M_T$ with $M_T = \{m : M_{MIN} \leq m \leq M_{MAX}\}$. Entries for prime_table(j, m) are 1 when $m \cdot D \pm j$ is prime, and 0 otherwise. Furthermore, a GCD_table is computed, where GCD_table$(j) = 1$ when $j \in J_S$, i.e. $\gcd(j, D) = 1$, and 0 otherwise. Then, set $S = \{j * Q_0 : j \in J_S\}$ is precomputed whereas set $T = \{m \cdot D * Q_0 : m \in M_T\}$ is computed iteratively such as described in Algorithm 11.

Algorithm 11 Improved standard continuation

Input: $M \in \mathbb{N}$, smoothness parameter B_1 and B_2, list of prime numbers $p \leq B_2$
Output: $q \in \mathbb{N}$, so that $q \mid M$; or fail

Precomputations:
$M_{MIN} = \left\lfloor (B_1 + \frac{D}{2})/D \right\rfloor$, $M_{MAX} = \left\lceil (B_2 - \frac{D}{2})/D \right\rceil$
for each $j = 1$ to $\frac{D}{2}$ **do**
  if $\gcd(j, D) = 1$ **then**
    GCD_table(j) = 1 and add j to J_S
  end if
end for
for each $m = M_{MIN}$ to M_{MAX} **do**
  for each $j = 1$ to $\frac{D}{2}$ **do**
    if $m \cdot D \pm j$ is prime **then**
      prime_table(m, j) = 1
    end if
  end for
end for
$Q \leftarrow Q_0$
for $j = 1$ to $\frac{D}{2}$ step 2 **do**
  if GCD_table(j) = 1 **then**
    store $Q = j * Q_0$ in S
  end if
  $Q \leftarrow Q \oplus 2 * Q_0$
end for

Main computations:
$d \leftarrow 1$, $Q \leftarrow D * Q_0$, $R \leftarrow M_{MIN} * Q$
for each $m = M_{MIN}$ to M_{MAX} **do**
  for each $j \in J_S$ **do**
    if prime_table(m, j) = 1 **then**
      retrieve $j * Q_0$ from table S
      $d \leftarrow d \cdot (x_R \cdot z_{jQ_0} - x_{jQ_0} \cdot z_R)$
    end if
  end for
  $R \leftarrow R \oplus Q$
end for
if $1 < q = \gcd(d, M) < M$ **then**
  return q
end if
return fail *// or restart with other curve parameters*

5.4. ECM Operations

A careful analysis of Phase 1 and the improved standard continuation (in the following denoted as Phase 2) shows that in case of Phase 1 the scalar multiplication $k * P$, and in case of Phase 2 the scalar multiplications $j * Q_0$, $D * Q_0$, $M_{MIN} * Q$, the point addition $R \oplus Q$, and the modular multiplication $d \cdot (x_R \cdot z_{jQ_0} - x_{jQ_0} \cdot z_R)$ need to be computed efficiently. Obviously, the scalar multiplication is the most computationally intensive operation. Hence, we should spend particular efforts to optimize this operation most.

5.4.1. Scalar Multiplication on Elliptic Curves

The Montgomery ladder can be implemented efficiently by using two multipliers and one adder/subtracter [13]. Using this combination, one step of the Montgomery ladder, i.e. $2 * P$ and $P \oplus Q$, can be computed in six steps as described in Table 5.2

Table 5.2.: One step of the Montgomery ladder ($2 * P$ and $P \oplus Q$)

Adder/Subtractor	Multiplier 1	Multiplier 2
$a_1 = x_P + z_P$ $s_1 = x_P - z_P$		
$a_2 = x_Q + z_Q$ $s_2 = x_Q - z_Q$	$m_1 = s_1^2$	$m_2 = a_1^2$
$s_3 = m_2 - m_1$	$m_3 = s_1 \cdot a_2$	$m_4 = s_2 \cdot a_1$
$a_3 = m_3 + m_4$ $s_4 = m_3 - m_4$	$x_{2P} = m_5 = m_1 \cdot m_2$	$m_6 = s_3 \cdot a_{24}$
$a_4 = m_1 + m_6$	$x_{P \oplus Q} = m_7 = a_3^2$	$m_8 = s_4^2$
	$z_{P \oplus Q} = m_8 \cdot x_{P-Q}$	$z_{2P} = m_{10} = s_3 \cdot a_4$

5.4.2. Point Addition on Elliptic Curves

The configuration of two multipliers and one adder/subtracter can also be used to implement the point addition in an efficient way. This is described in Table 5.3, where the second type of ECM operations is computed within six steps.

Table 5.3.: Point addition ($P \oplus Q$)

Adder/Subtractor	Multiplier 1	Multiplier 2
$a_1 = x_P + z_P$ $s_1 = x_P - z_P$		
$a_2 = x_Q + z_Q$ $s_2 = x_Q - z_Q$		
	$m_3 = s_1 \cdot a_2$	$m_4 = s_2 \cdot a_1$
$a_3 = m_3 + m_4$ $s_4 = m_3 - m_4$		
	$m_7 = a_3^2$	$m_8 = s_4^2$
	$z_{P \oplus Q} = m_8 \cdot x_{P-Q}$	$x_{P \oplus Q} = m_7 \cdot z_{P-Q}$

5.4.3. Repeatedly Modular Multiplications

The third type of ECM operations is the modular multiplication of coordinates of the intermediate calculated points. Product d can be computed step-wise as described in Table 5.4.

Table 5.4.: Calculating accumulated product d

Adder/Subtractor	Multiplier 1	Multiplier 2
	$m_1 = x_n \cdot z_0$	$m_2 = x_0 \cdot z_n$
$d_{0n} = m_1 - m_2$	$m_3 = x_n \cdot z_1$	$m_4 = x_1 \cdot z_n$
$d_{1n} = m_3 - m_4$	$d = d \cdot d_{0n}$	$m_1 = x_n \cdot z_2$
	$d = d \cdot d_{1n}$	$m_2 = x_2 \cdot z_n$
	$\cdots$	

6. ECM Hardware Architecture

Our hardware architecture is designed to compute integer factorizations with the elliptic curve method including both Phase 1 and 2. We follow the proposal of [13, 14] which is – as far as we know – the fastest hardware implementation available supporting the entire ECM algorithm. In contrast to that work, we assume that the intensive utilization of DSP elements accelerate the ECM hardware significantly such that we achieve a much better cost-benefit ratio for middle-cost FPGA devices as gained before. For that purpose, we also employ a special purpose hardware called COPACOBANA which consists of multiple independent Virtex-4 FPGA devices to massively parallel computations.

6.1. Architecture Overview

The COPACOBANA is an FPGA-cluster which is originally designed for breaking ciphers [23]. Our architecture benefits from this special purpose hardware as central system performing the cost-intensive operations required by the elliptic curve method. Such operations are the scalar multiplication as well as the point addition on elliptic curves, and the multi-precision modular multiplication which are introduced in Section 5.4. An overview of our architecture is given in Figure 6.1. It pictures the COPACOBANA which consists of 16 FPGA modules each equipped with eight Virtex-4 FPGA devices. Each FPGA can be accessed by a 16-bit address bus and data is transferred via a shared 64-bit data bus. All 128 FPGA devices are completely independent and execute operations in parallel. Note that all FPGA devices placed on the same module are programmed with the same configuration. The COPACOBANA is connected to a host system via an Ethernet connection. The host is responsible for performing only low-cost operations needed to control the COPACOBANA and managing the execution flow of the elliptic curve method.

In our architecture, a single FPGA integrated in the COPACOBANA represents one ECM system which works entirely independently and in parallel to other ECM systems. Each ECM system consists of multiple ECM units and each unit is capable to perform the elliptic curve method on a separate data set. The number

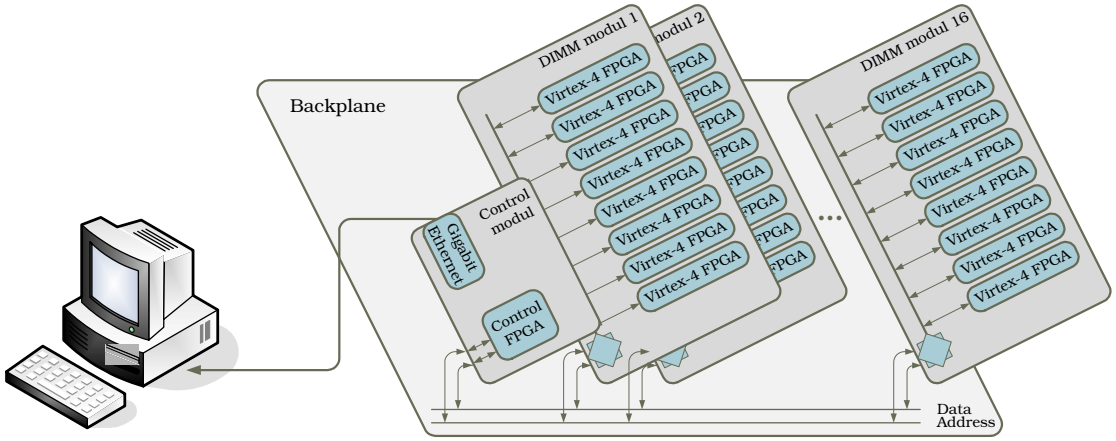

Figure 6.1.: Architecture overview

of ECM units of one ECM system depends on resources available with the chosen hardware platform. The following figure indicates that such a hardware configuration allows a highly configurable network of ECM systems supporting a variety of different integer bit lengths. Figure 6.2 shows such a network of ECM systems in which three different ECM systems support numbers with length of 0 up to 66, 67 up to 151, and 152 up to 202 bit. Another system is configured to support all of those bit lengths. In the following, we concentrate on ECM systems supporting numbers of the same order of magnitude due to easier administration.

The interactions between the COPACOBANA and the host system during the runtime of the elliptic curve method is discussed next.

Phase 1:

1. Depending on the number M, the host system chooses the coordinates of a suitable starting point $P_0 \in \mathcal{E}_M$ and a random curve parameter $a \in \mathbb{Z}_M$, and computes $a_{24} = \frac{a+2}{4} \pmod M$ as well as the Phase 1 parameter k. Note that M, P_0 and a_{24} can be different for each ECM unit supporting the same bit length whereas k is common for all those ECM units. All parameters are transferred via the Ethernet connection and the shared bus of the COPACOBANA to a dedicated ECM unit which supports the corresponding bit length.
2. Each ECM unit calculates one scalar multiplication $Q_0 = k * P_0 = (x_{Q_0} : y_{Q_0} : z_{Q_0})$ which is provided to the host system.
3. The host systems computes $\gcd(z_{Q_0}, M)$ and decides if Phase 2 is continued or if a non-trivial factor of M is found.

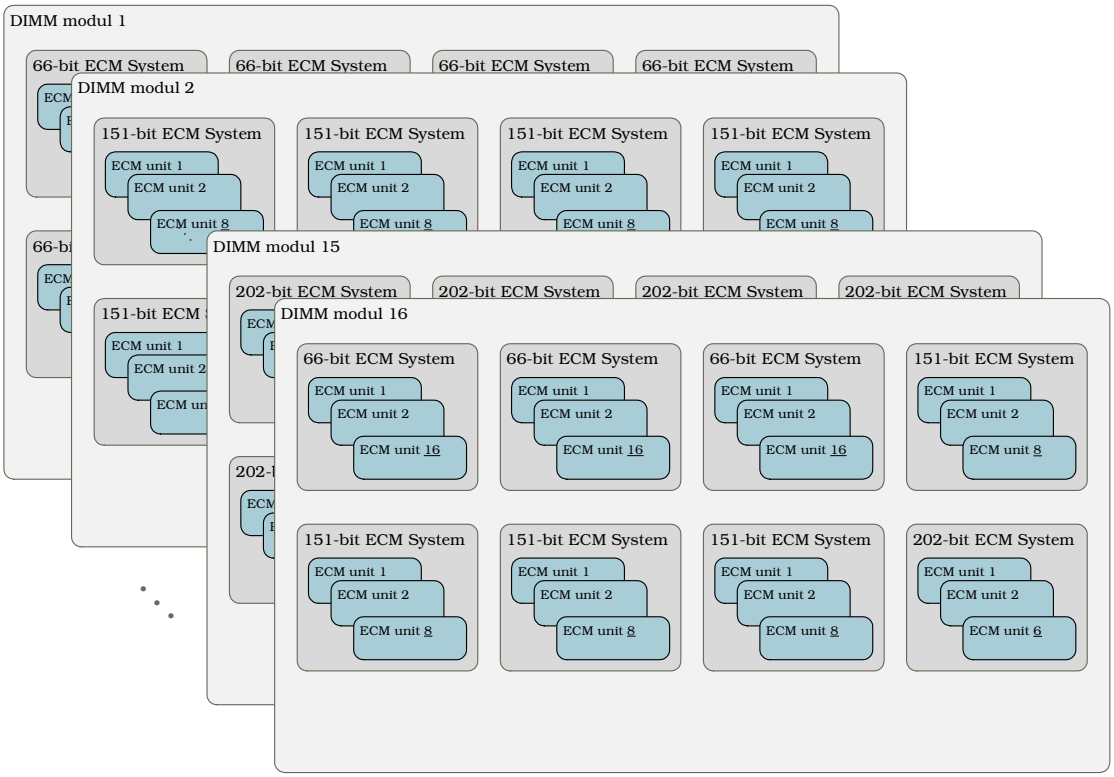

Figure 6.2.: Configurable network of ECM systems

Phase 2:

1. The host system precomputes the prime_table as well as the GCD_table, and the COPACOBANA precomputes the scalar multiplications $Q = D * Q_0$, $M_{MIN} * Q$, and $j * Q_0$ according to Algorithm 11. After the precomputation step, the host system transfers the precomputed tables to the COPACOBANA and to dedicated ECM units, respectively.
2. Each ECM unit calculates $d = d \cdot (x_R \cdot z_{jQ_0} - x_{jQ_0} \cdot z_R)$ and $R = R \oplus P$ iteratively and provides the result of the accumulated product d to the host system.
3. The host system retrieves d and computes $\gcd(d, M)$ and determines, if a non-trivial factor is found or not. If not, Phase 1 can be restarted with different curve parameters.

With high probability ($> 80\%$) a non-trivial factor is found after performing the elliptic curve method with approximately 20 different curves [34].

ECM systems which are specialized for a fixed bit length are illustrated in Fig-

ure 6.3. In such a system, all ECM units support numbers of the same order

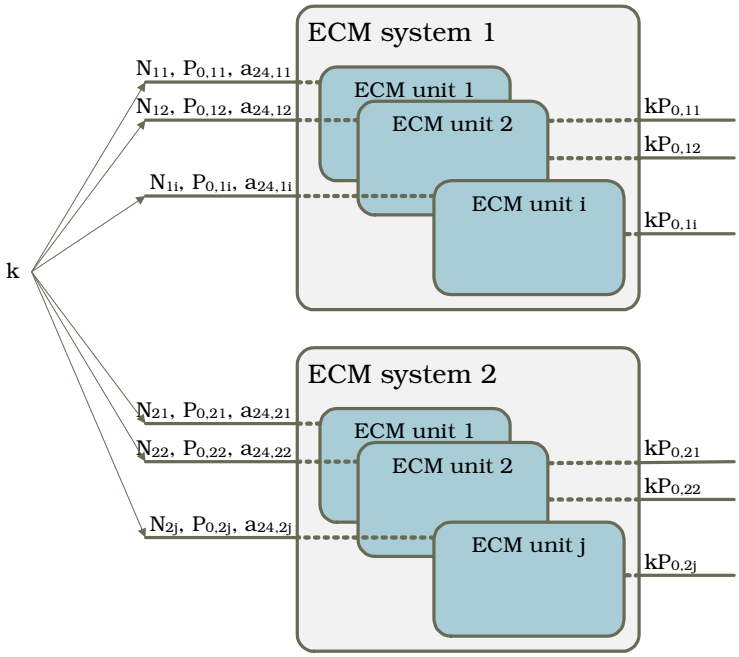

Figure 6.3.: Example of an ECM system supporting Phase 1

of magnitude but with different values. In case of Phase 1, the parameters are $M_{ij}, P_{0_{ij}}$, and $a_{24_{ij}}$ such that $M_{nm} \neq M_{op}, P_{0_{nm}} \neq P_{0_{op}}$, and $a_{24_{nm}} \neq a_{24_{op}}$ with $1 \leq n, m, o, p \leq i, j$. In contrast, parameter k is the same for all of the ECM units.

One ECM unit is realized by two independent multipliers, one addition/subtraction unit, and some local memory which follows the optimizations described in Section 5.4. Figure 6.4 pictures such an ECM unit. Regarding to previous works, we expect to improve the low-level units by using algorithms optimized for being performed by means of fast DSP elements. Since low-level operations (e.g., modular multiplications) are performed multiple times in high-level operations (e.g., scalar multiplication on elliptic curves), improvements in low-level units provide an acceleration of the entire ECM hardware.

6.2. DSP-optimized Algorithms

We use DSP elements provided by the FPGA devices on our hardware platform to implement the low-level units, in particular the modular multiplier and the modular adder/subtracter. For that purpose, we adapt the *Modular Multiplication*

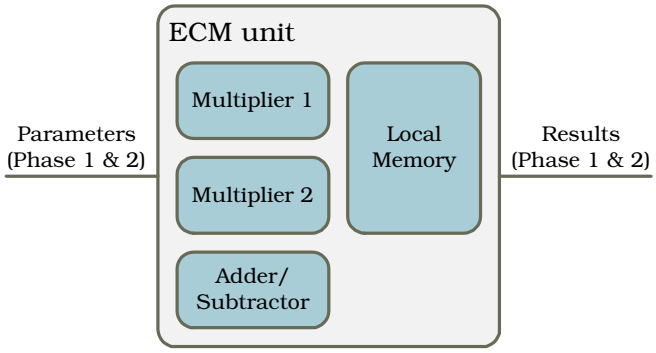

Figure 6.4.: ECM unit

with Quotient Pipelining algorithm [33] to realize a modular multiplier by intensive DSP utilizations. Moreover, we optimize the modular addition/subtraction to be applicable with DSP elements.

6.2.1. Montgomery Multiplication

Since the modular multiplication is the most cost-intensive operation in modular arithmetic systems (except for the modular inverse), this unit has a significant impact on the running time of the implementation of the elliptic curve method. Modular arithmetic in Montgomery domain benefits from the fact that the more cost-intensive division which is needed to determine the residue in normal domain is avoided. The *Modular Multiplication with Quotient Pipelining* algorithm is an optimized variant of the Montgomery multiplication which prevents the expensive determination of the quotient in each round and it also provides a high radix representation of the multiplier. In [38], a first DSP adaptation of this algorithm has been proposed. We show that our implementation is faster by design, since the number of rounds could be optimized such that delays are eliminated and no logic blocks are used for relatively complex addition tasks.

We now present our modified high radix multiplication algorithm that can be implemented efficiently by the intensive utilization of DSP elements. Therefore, we firstly recapitulate some basic facts about the *Modular Multiplication with Quotient Pipelining* algorithm (Algorithm 7). Its system parameters satisfies the following constraints.

- Modulus $M > 2$ and M is odd.
- M' is given by $(-M \cdot M') \bmod 2^{k(d+1)} = 1$ with $k, n > 0$.
- $\tilde{M} = (M' \bmod 2^{k(d+1)})M$.

- $4\tilde{M} < 2^{k \cdot n}$.
- $M'' = \frac{\tilde{M}+1}{2^{k(d+1)}}$.
- R^{-1} is given by $2^{k \cdot n} \cdot R^{-1} (\bmod M) = 1$.
- Delay parameter $d \geq 0$.
- Multiplicand A with $0 \leq A \leq 2\tilde{M}$.
- Multiplier B with $0 \leq B \leq 2\tilde{M}$ in radix 2^k representation such that $B = \sum_{i=0}^{n+d} (2^k)^i \cdot b_i$ where $b_i \in \{0, 1, \ldots, 2^k - 1\}$ for $0 \leq i < n$ and $b_i = 0$ for $i \geq n$.
- Result S_{n+d+2} is given by $S_{n+d+2} \equiv A \cdot B \cdot R^{-1} \pmod{M}$ where $0 \leq S_{n+d+2} < 2\tilde{M}$.

The multiplier B is expressed in radix 2^k representation such that n is the number of 2^k blocks of B. Since our hardware platform provides fast unsigned 17×17-bit multiplications, we also need to express the input parameters A and M'', the intermediate results S_i, and the final result S_{n+d+2} in radix 2^k representation. Obviously, $k = 17$ since DSP elements also include 17-bit shifters. The input parameters A and B, and the resulting parameter S_{n+d+2} are of the same order of magnitude such that the result can be used as input parameter and thus, the algorithm can be reused repeatedly. Note that $0 \leq A, B < M$ holds for the initial values of A and B.

Since A and B are less or equal to $2\tilde{M}$, since $M'' < \tilde{M}$, and since S_i and S_{n+d+2} are at least less than $2\tilde{M}$, all these parameters can be expressed in radix 2^k representation with the same number of blocks n. This is because $0 \leq A, B \leq 2\tilde{M}$, $M'' = \frac{\tilde{M}+1}{2^{k(d+1)}}$ as well as $0 \leq S_{n+d+2} < 2\tilde{M}$ is satisfied by constraints described above. Step six of Algorithm 7 shows, that also $S_i < S_{n+d+2}$ is satisfied. Hence, these parameter are expressed as needed as follows: $A = \sum_{i=0}^{n-1} a_i \ (2^k)^i$ with $a_i \in \{0, 1, \ldots, 2^k - 1\}$, $B = \sum_{i=0}^{n+d} b_i \ (2^k)^i$ with $b_i \in \{0, 1, \ldots, 2^k - 1\}$ for $0 \leq i < n$ and with $b_i = 0$ for $i \geq n$, and $S_i = \sum_{j=0}^{n-1} S_{i,j} \ (2^k)^j$ with $m_i \in \{0, 1, \ldots, 2^k - 1\}$. Since M'' is much smaller than $2\tilde{M}$, this parameter needs some further consideration.

Lemma 6.2.1. *Assume M is given such that $2 < M < 2^h$. Then M'' satisfies $0 < M'' < 2^h$ and can be expressed in radix 2^k representation such that $M'' = \sum_{i=0}^{n-1} m_i \ (2^k)^i$ with $m_i \in \{0, 1, \ldots, 2^k - 1\}$ for $0 \leq i < \lceil \frac{h}{k} \rceil$ and $m_i = 0$ for $i \geq \lceil \frac{h}{k} \rceil$.*

Informal Proof: Remark M'' is given by $M'' = \frac{\tilde{M}+1}{2^{k(d+1)}}$ with $2^{k(d+1)} \mid (\tilde{M}+1)$ [33] and $\tilde{M} = (M' \bmod 2^{k(d+1)})M$. Thus, $\tilde{M}_{max} = (2^{k(d+1)}-1)(2^h-1)$. Hence,

$$M''_{max} = \frac{\tilde{M}_{max}+1}{2^{k(d+1)}} \Leftrightarrow M''_{max} < \frac{\tilde{M}_{max}+2}{2^{k(d+1)}}$$

$$\Leftrightarrow M''_{max} < \frac{2^{k(d+1)+h} - 2^{k(d+1)} - 2^h + 3}{2^{k(d+1)}}$$

Since $(-2^{k(d+1)} - 2^h + 3) < 0$,

$$M''_{max} < \frac{2^{k(d+1)+h}}{2^{k(d+1)}} = 2^h.$$

On the other hand: since k, d, and $\tilde{M}$ always remains positive,

$$M'' > 0.$$

□

Our adaptation of the *Modular Multiplication with Quotient Pipelining* algorithm is outlined in Algorithm 12. Note that this algorithm can also be reused repeatedly by using results as inputs.

Regarding to the system parameters adopted from Algorithm 7, we add the number of DSP elements δ as new parameter. For relatively small multiplier units (e.g., with input bit sizes with less than 256 bits), δ can be chosen equal to the number of blocks n. In such case, each 17×17-bit multiplication is performed by a dedicated DSP element. For example, $a_0 \cdot b_i$ is always performed by the leftmost DSP element whereas the rightmost DSP element computes $a_{n-1} \cdot b_i$. For realizing bigger multipliers (e.g., with input bit sizes with more than 1024 bits), a constant number of DSP elements can be used to perform one multi-precision multiplication by repeated use of DSP elements. In such case, $n = \delta \cdot r$ with $r \in \mathbb{N}$ and r is the number of iterations for a single DSP element. For example, [38] implements a multiplier with $\delta = 17$ and $r \in \{2, 4, 6, 8\}$. Such a multiplier is able to handle 512-bit moduli with $r = 2$, 1024-bit moduli with $r = 4$, 1536-bit moduli with $r = 6$, and 2048-bit moduli with $r = 8$. Since the elliptic curve method handles relatively small moduli (e.g., numbers to be factored may be less than 200-bit), we will focus on the case with $\delta = n$ only.

Lemma 6.2.2. *In both cases, the number of blocks n is obtained by*

$$n = \left\lceil \frac{k(d+1)+h+2}{k} \right\rceil.$$

Algorithm 12 Modular multiplication with quotient pipelining optimized for DSP usage

Input: Multiplicand $A = \sum_{i=0}^{n-1} a_i \, (2^k)^i$; multiplier $B = \sum_{i=0}^{n+d} b_i \, (2^k)^i$; $0 \leq A, B \leq 2\tilde{M}$; $M'' = \sum_{i=0}^{n-1} m_i \, (2^k)^i$; $0 < M'' < 2^h$; delay parameter $d \geq 0$; word width $k = 17$; number of blocks n.

Output: $S_{n+d+2} \equiv A \cdot B \cdot R^{-1} \pmod{M}$; $S_i = \sum_{j=0}^{n-1} S_{i,j} \, (2^k)^j$; $0 \leq S_{n+d+2} < 2\tilde{M}$.

$S_{0,j} \leftarrow 0$
$q_{-d} \leftarrow 0, \cdots, q_{-1} \leftarrow 0$
for $i = 0$ to $n + d$ **do**
  $q_i \leftarrow S_{i,0} \pmod{2^k}$
  $S_{i+1,0} \leftarrow \left[S_{i,1} \pmod{2^k} + q_{i-d} \cdot m_0 + b_i \cdot a_0\right] \pmod{2^k}$
  for $j = 1$ to $n + d - 1$ **do**

$$S_{i+1,j} \leftarrow \left[S_{i,j+1} \pmod{2^k} + q_{i-d} \cdot m_j + b_i \cdot a_j + \underbrace{\left\lfloor \frac{S_{i+1,j-1}}{2^k} \right\rfloor}_{carry} \right] \pmod{2^k}$$

  end for
  $S_{i+1,n+d} \leftarrow \left[q_{i-d} \cdot m_{n+d} + b_i \cdot a_{n+d} + \left\lfloor \frac{S_{i+1,n+d-1}}{2^k} \right\rfloor \right] \pmod{2^k}$
end for
$S_{n+d+2} \leftarrow 2^{k \cdot d} \cdot S_{n+d+1} + \sum_{j=0}^{d-1} q_{n+j+1} \cdot 2^{kj}$
return S_{n+d+2}

Informal Proof: From Lemma 6.2.1 recall that $\tilde{M}_{max} = (2^{k(d+1)} - 1)(2^h - 1)$. Hence,

$$\tilde{M} < \tilde{M}_{max} + 1 \Leftrightarrow \tilde{M} < 2^{k(d+1)+h} - 2^{k(d+1)} - 2^h + 2$$

Since $(-2^{k(d+1)} - 2^h + 2) < 0$,

$$\tilde{M} < 2^{k(d+1)+h} \Leftrightarrow 4\tilde{M} < 2^{k(d+1)+h+2}$$

and thus

$$\Rightarrow k(d+1) + h + 2 \stackrel{!}{=} k \cdot n \Leftrightarrow n = \left\lceil \frac{k(d+1) + h + 2}{k} \right\rceil$$

□

Table 6.1 gives an overview of the number of DSP elements δ used for corresponding moduli 2^h.

In contrast to previous works, our modular multiplication unit takes advantage of a reduced number of rounds $R = n + 1$ needed to be performed before the resulting product is available. This is caused in the optimized architecture in which the delay parameter d could be minimized to zero.

Number of DSP elements δ	Max. modulus 2^h
$\delta = 2$	$h \leq 15$
3	32
4	49
5	66
6	83
7	100
8	117
9	134
10	151
...	
i	$15 + (i-2) \cdot 17$

Table 6.1.: Number of DSP elements for moduli 2^h

6.2.2. Modular Addition and Subtraction

The second arithmetic core component inside our ECM unit is the modular addition and subtraction unit. Usually, a modular addition is calculated by performing the addition $S = A + B$ where A and B are the summands, and S is the resulting sum. Then, since $S < 2M$ where M is the corresponding modulus, we need to test if $S \leq M$. In case, if A and B are small enough then S is already reduced. Otherwise, $S = S - M$ needs to be performed such that $S \in \mathbb{Z}_M$. Similarly, the modular subtraction $S \equiv A - B \pmod{M}$ is usually calculated by $S = A - B$ followed by checking if $S < 0$. Depending on S, $S = S + M$ or S remains unchanged. Since both operations are very similar, they can be merged easily.

A conditional branch (i.e., determining if $S \geq M$ or not) is inappropriate in hardware. Thus, another technique is preferred to realize our modular adder/subtracter. This technique includes calculating $S = A + B$ and $S' = A + B - M$ at the same time in case of addition, and $S = A - B$ and $S' = A - B + M$ in case of subtraction. Consequently, the hardware only needs to check a borrow bit to choose if S' or S is the correct result. Algorithm 13 outlines this technique optimized for use with DSP elements.

As expected, the modular addition/subraction in much easier to design than the modular multiplication. Based on the optimizations introduced in Section 5.4, we require the addition/subtraction to be two times faster than a multiplication. We designed a addition/subtraction unit with two DSP elements where the number of block n and the word width k remain the same as for the modular multiplier.

Since our modular multiplication handles input values $A, B < 2\tilde{M}$, we choose $2\tilde{M}$

Algorithm 13 Modular addition and subtraction optimized for DSP usage

Input: $A = \sum_{i=0}^{\frac{n-1}{2}} a_i \ (2^{2\cdot k})^i$; $B = \sum_{i=0}^{\frac{n-1}{2}} b_i \ (2^{2\cdot k})^i$; $2\tilde{M} = \sum_{i=0}^{\frac{n-1}{2}} m_i \ (2^{2\cdot k})^i$; $0 \leq A, B < 2\tilde{M}$; word width $k = 17$; number of blocks n;

Output: $S = \sum_{i=0}^{\frac{n-1}{2}} s_i \ (2^{2\cdot k})^i \equiv A \pm B \pmod{2\tilde{M}}$

```
c_{-1} ← 0;
for j = 0 to (n-1)/2 do
    (c_j, s_j) ← (a_j ± b_j ± c_{j-1})
end for
c̃_{-1} ← 0;
for j = 0 to (n-1)/2 do
    (c̃_j, s'_j) ← (a_j ± b_j ± c_{j-1} ∓ m_j ∓ (c_{j-1} + c̃_{j-1}))
end for
if S' < 0 (in case of addition) then
    return S
else
    return S';
end if
if S < 0 (in case of subtraction) then
    return S
else
    return S';
end if
```

as modulus for the modular addition/subtraction. Only this choice allows the implementation of Algorithm 13 such that only one addition/subtraction of the modulus is needed to compute the correct result, depending on the operation. Note that $\tilde{M}$ is a multiple of M and therefore calculations of combinations of results of both Algorithm 12 and Algorithm 13 result in correct values congruent modulo M.

Example: Assume that A as well as B are expressed in Montgomery representation, M is the modulus, and R is a parameter for the Montgomery multiplication $MMul$. With $A = 26031944, B = 8071945$, $M = 13111977$, and $R^{-1} = 9365426$ the modulus $\tilde{M}$ for the addition/subtraction can be computed: $\tilde{M} = 1122477015039$. Then, computations modulo $2\tilde{M}$ are congruent to computation modulo M which is depicted by the following example.

$$MMul(A, B) = C = 312586125451$$

$$A + B \pmod{2\tilde{M}} = 34103889$$

$$C - D \pmod{2\tilde{M}} = 312552021562 \equiv \mathbf{1825813} \pmod{M}$$

$A \cdot B \cdot R^{-1} \pmod{M} = 9705748$

$A + B \pmod{M} = 7879935$

$C - D \pmod{M} = \mathbf{1825813}$

7. ECM Implementation

We have implemented the elliptic curve method completely using high-level language support by VHDL. Since software experiments indicate that most to be factored integers returned from the GNFS in its sieving step are less than 140-bit long, a system handling up to 151-bit moduli has been targeted. An ECM system has been implemented to perform all operations needed by Phase 1 and Phase 2, most important the scalar multiplication and point addition on elliptic curves. In fact, all operations which can be calculated by means of two modular multipliers and one modular adder/subtracter can be computed by our system if suitable instructions are performed. Phase 1 has been realized completely in hardware whereas the implementation of Phase 2 is still in progress. Figure 7.1 illustrates the current ECM system.

The ECM system consists of eight different ECM units, an instruction set, and some memory in which k is stored. Each ECM unit performs the operations both for Phase 1 and 2, and the instruction memory includes instructions which encode, for example, operation modes and addresses of local memories. After starting the ECM hardware, initial points P_{0_i}, curve parameters a_{24_i}, and k are loaded into the local memory of the ECM units and the memory of the ECM system, respectively. Then, $Q_{0_i} = k * P_{0_i}$ are calculated by corresponding ECM units, and the z-coordinates of points Q_{0_i} are stored in their local memory.

7.1. Hardware and Software Environment

The implementation is designed for the resources available in a Xilinx Virtex-4 SX35 (xc4vsx35-10ff668) FPGA. Roughly spoken, it consists of 15360 logic slices, 192 block RAM elements, and 192 DSP elements. Software development tools are Xilinx ISE 9.2.04i with application version J.40 for par and xst. Simulations are performed by means of ModelSim XE III 6.1e.

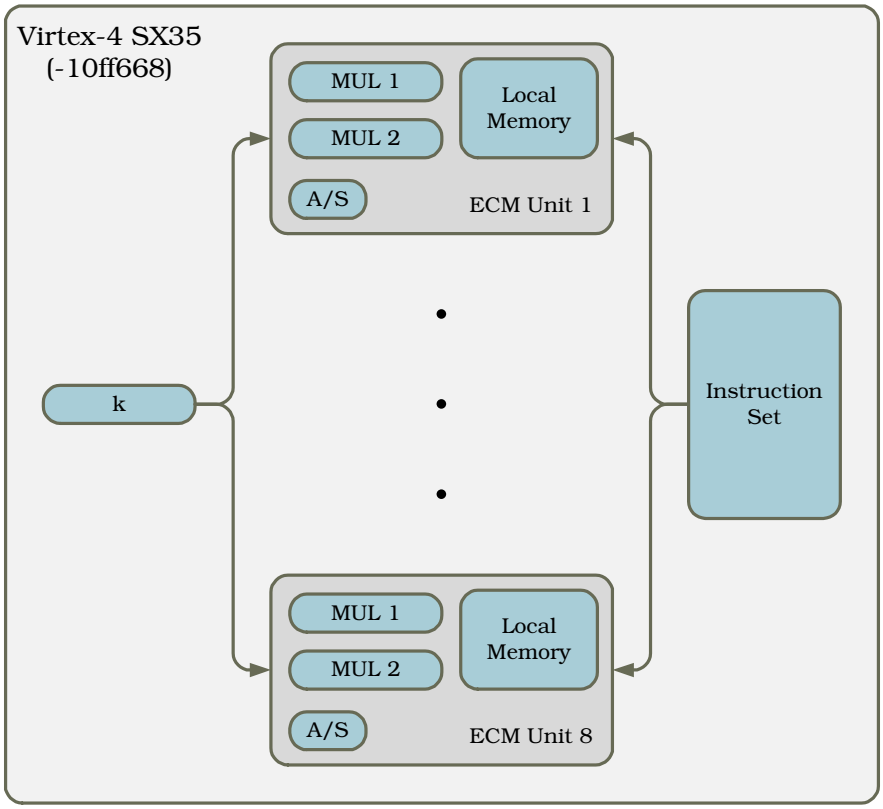

Figure 7.1.: Implemented ECM system

7.2. Instruction

We implemented a relatively complex instruction to keep encoding as flexible as possible. This allows encoding of all different operations possible with two available multipliers and one adder/subtracter. Figure 7.2 shows the different segments of the instruction.

Beginning from left to right, the instruction includes signals which enable the multipliers and the adder/subtracter, set the operation mode of the addition/subtraction unit (i.e., add or subtract), enable storing data in block RAM elements, and specify operation modes of multiplexers. The remaining bits of the instruction encodes addresses of block RAM elements which are connected to the multipliers and the addition/subtraction unit. This is described in detail in Section 7.5. However, the block RAM elements are grouped into three storage blocks. Storage block I and II are connected to the multipliers, and the remaining storage block III is connected to the addition/subtraction unit. For each storage block, the addresses of the parameters A, B and M to each multiplier are encoded. The instruction also

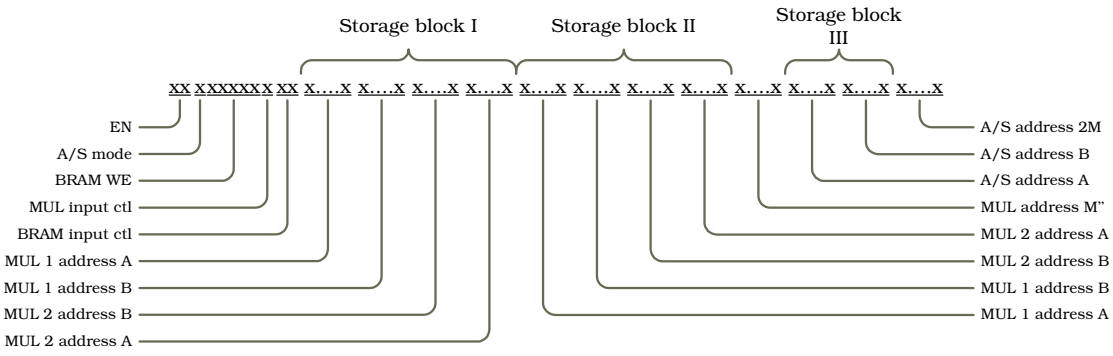

Figure 7.2.: Implemented instruction

encodes the addresses of the parameters A, B and M of the addition/subtraction unit.

Figure 7.3 illustrates a simplified scheme of the realized instruction set. It in-

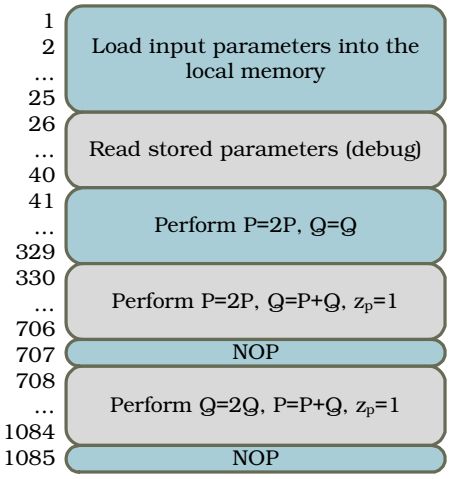

Figure 7.3.: Implemented instruction set

cludes instructions to perform the following operations: loading input parameters into local memory of corresponding ECM units, loading k into the ECM system's memory, performing $2 * P$ and $P \oplus Q$ ($z_p = 1$), and performing $2 * Q$ and $P \oplus Q$ ($z_p = 1$). Hence, this instruction set realizes ECM Phase 1.

7.3. Multiplication Unit

We implemented a Montgomery multiplication unit supporting 151-bit moduli by cascading ten DSP elements with minimal usage of additional logic blocks. Our implementation can be clocked up to 400 MHz (max. device frequency on xc4vsx35-10) due to heavy use of these hardcores.

Figure 7.4 illustrates the multiplier unit with six input and one output signal. All

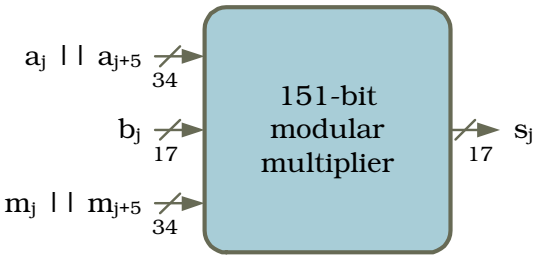

Figure 7.4.: Modular multiplier

parameters are processed block-wise where two different block lengths are applied. Multiplicand A, multiplier B, and modulus M'' are processed 34-bit wise whereas the block length of b_j is 17-bit ($0 \leq j \leq 9$). Other input signals are `CE`, `RST`, and `CLK`. The `CE` signal is used for activating the unit, the `RST` signal resets the unit, and the `CLK` signal is used for clocking tasks. The output signal S is returned in ten blocks each 17-bit long. The overall result (blocks s_0 up to s_9) is calculated and returned after 66 clock cycles.

Figure 7.5 pictures the current implementation of our multiplier in more detail. Ten DSP elements are cascaded by using the internal signals `BCIN/BCOUT` and `PCIN/PCOUT`. The input blocks a_j and m_j are transferred to dedicated DSP elements, more precisely a_0 is routed to DSP 1, a_1 goes to DSP 2, etc. Input blocks b_j are required for all DSP elements but these blocks are applied as input for DSP 1 only. Depending on the actual round of Algorithm 12, b_j is transferred to the next DSP by using the `BCIN/BCOUT` signals. The `PCIN/PCOUT` signals are used to transfer intermediate results to the next DSP element. Other DSP inputs are the integers 0 and 1, intermediate results $S_{i,j}$, and the quotient q_{i-d}. All DSP output signals are connected to two 5x1 multiplexers. Their output is then selected by a subsequent 2x1 multiplexer which transfers the resulting blocks to the unit's output port. The blocks a_j, a_{j+5} of multiplicand A and m_j, m_{j+5} of modulus M'' are transferred to the multiplication unit at the same time which is described in more detail next.

The DSP elements perform operations such that Algorithm 12 is implemented.

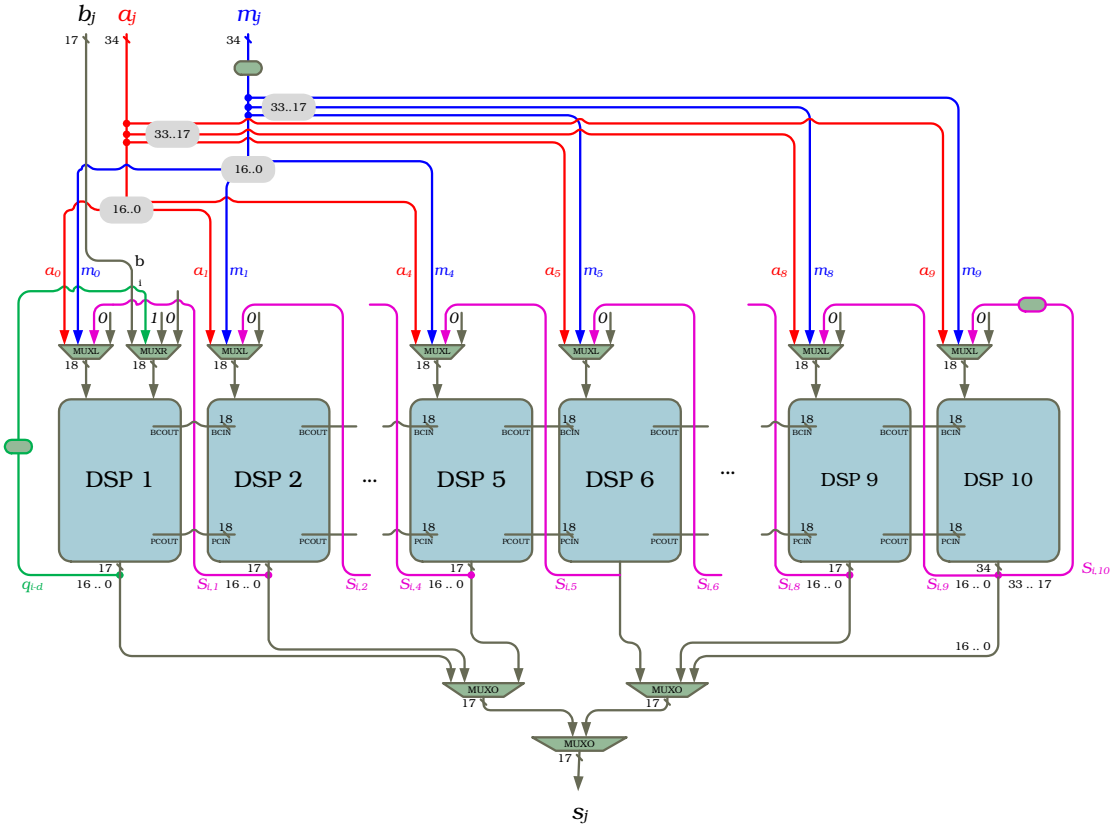

Figure 7.5.: Montgomery multiplier based on DSP elements

Therefore, Table A.1 gives an overview which input blocks and operation modes are needed by DSP 1 and DSP 2 to compute the desired result $S_{11,0}$ and $S_{11,1}$.

DSP elements two up to ten execute uniform instructions with different operation modes which are shifted by one clock cycle from one DSP to another. In Table A.2 the operations of DSP 5 and DSP 6 are described which computes the resulting blocks $S_{11,4}$ and $S_{11,5}$. Note that at clock cycle six, a_0 and a_5 are needed at the same time for different DSP elements. In particular, DSP 1 processes a_0 as input and simultaneously a_5 is processed by DSP 6. The same happens in the next clock cycles, for example, in which DSP 2 needs a_1 at the same time as a_6 is needed by DSP 7. Therefore, the multiplication unit requires 34-bit blocks for input A and M'' where the input block for A contains $a_j \| a_{j+5}$ and the input block for M contains $m_j \| m_{j+5}$.

Finally, Table A.3 contains the operations performed by DSP 9 and DSP 10 which compute the desired resulting blocks $S_{11,9}$ and $S_{11,10}$. Note that all intermediate results $S_{i,j}$ are calculated block-wise with block width of 17-bit. All blocks $S_{11,j}$

concatenated together build the final result S_{n+1}. The last clock cycle is needed such that $S_{11,9}$ is available on the output of the last DSP element while Algorithm 12 ensures that block $S_{11,10}$ contains zero bits only.

From tables A.1, A.2, and A.3 it shows that the DSP operation modes (opmode) reiterate after five clock cycles. The DSP operation mode control logic is implemented as illustrated in Figure 7.6. Both operation modes for DSP 1 and modes

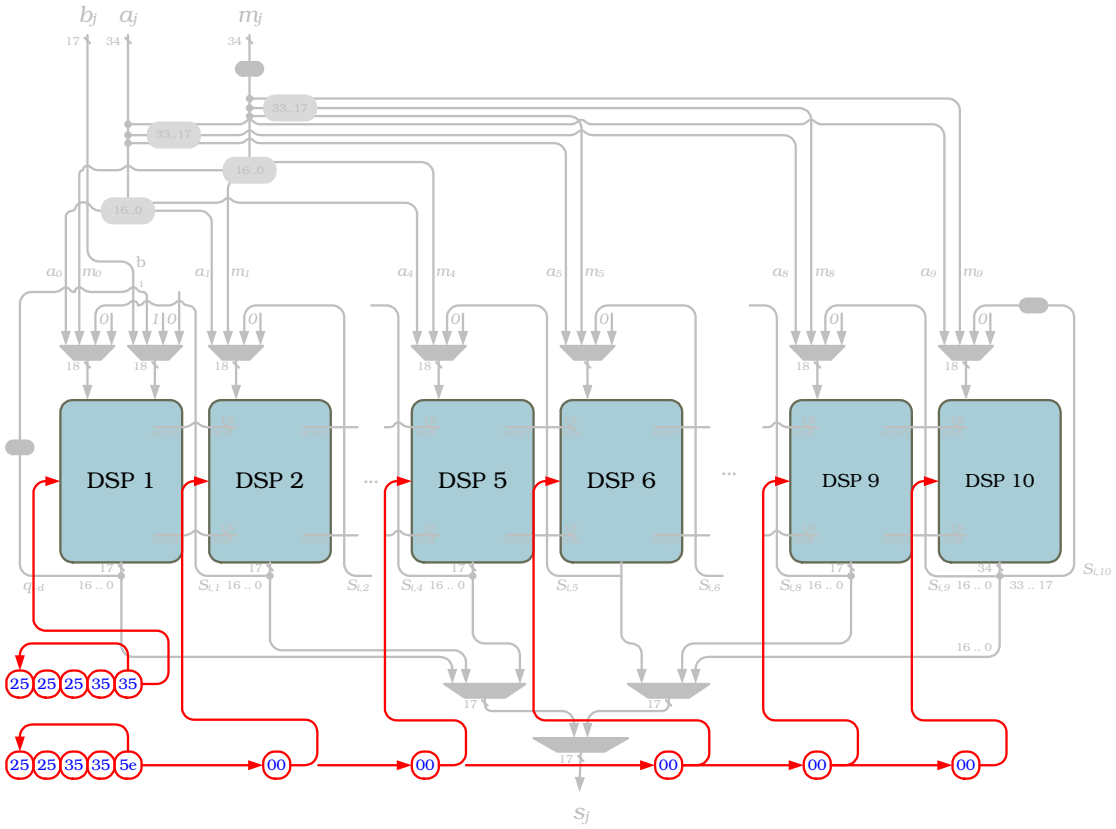

Figure 7.6.: DSP controlling of the multiplier

for the other DSP elements are implemented within the CLBs by shift registers. The output of the shift registers is connected to the internal `OPMODE` signals of the DSP elements. This signal specifies the operation mode of the DSP element in the actual clock cycle. All DSP elements except DSP 1 are started one after another with one clock cycle delay, as well as the `OPMODE` signals.

In Figure 7.7 depicts how the multiplexers are controlled. Since $MUXR$ and all $MUXL$ can be controlled with the same sequence delayed by one clock cycle each, the control path is also implemented within the CLBs as shift register.

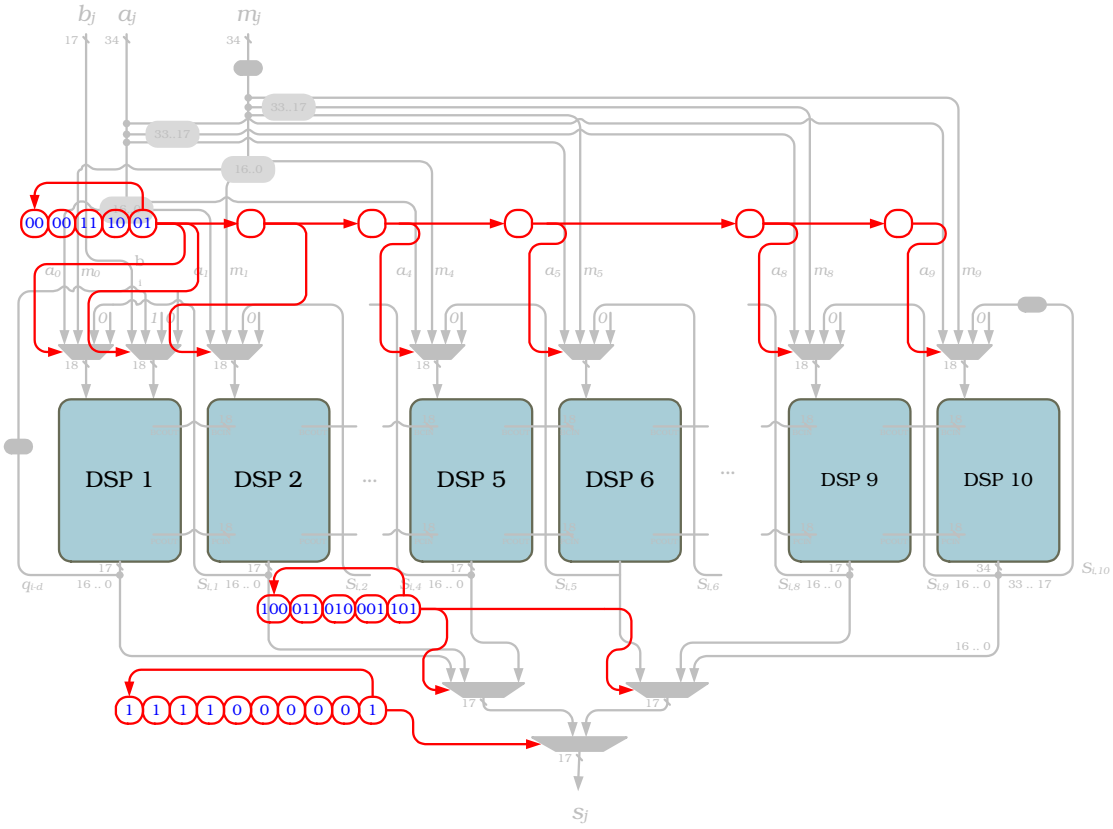

Figure 7.7.: Multiplexer controlling of the multiplier

Figure 7.8 shows a timing diagram of the multiplier unit. At first, the multiplier unit needs to be enabled by the `CE` signal. It it also necessary to reset the unit, before the next multiplication is performed. To perform a multiplication, the corresponding input blocks a_j, b_j and m_j are expected to be available at the input ports. Note that the input blocks of A and M'' contain $a_j \| a_{j+5}$ and $m_j \| m_{j+5}$, respectively. During the multiplication, the inputs $a_j \| a_{j+5}$ and $m_j \| m_{j+5}$ remain unchanged where all blocks b_0 to b_9 are processed. Then, after 56 clock cycles, the first 17-bit block of the result becomes available at the unit's output port. At each following rising edge, the next resulting block is available. From round 11 on, blocks b_j remain zero whereas blocks a_j contain regular inputs. This inputs are processed by the five right most DSP elements because these inputs are still required by the multiplication algorithm. Theoretically, blocks a_0 to a_4 could contain zero data whereas blocks a_5 to a_9 would contain regular inputs. But actually, this is not implemented. After another five clock cycles, input blocks a_j also remain zero. At this time, input blocks m_j are still required for the multiplication

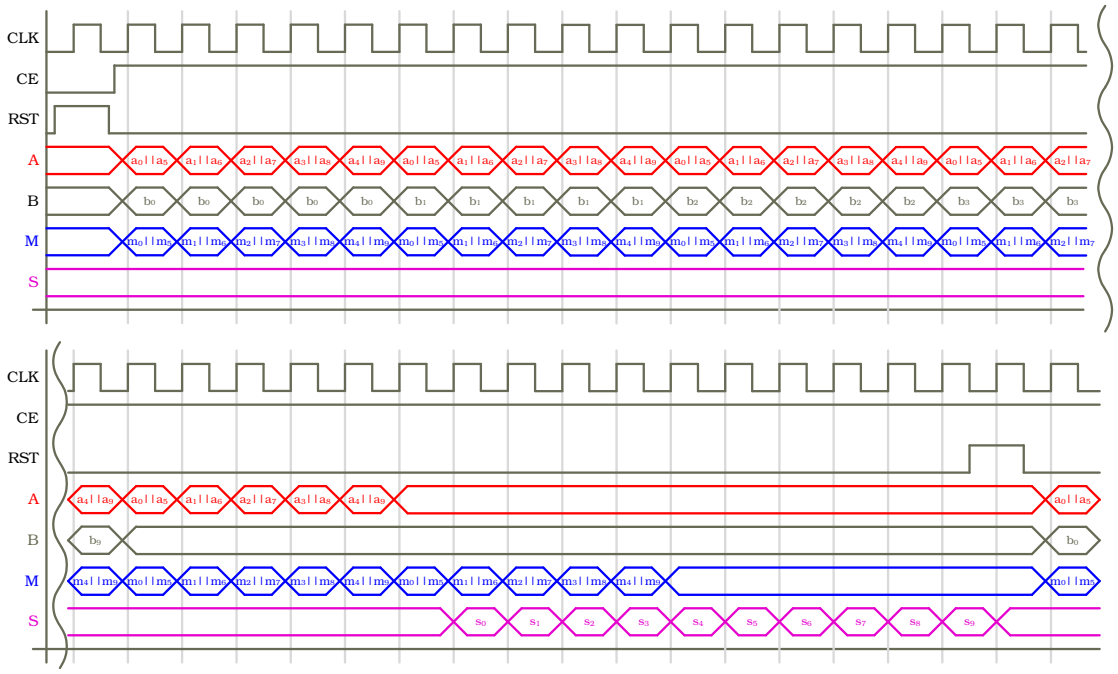

Figure 7.8.: Timing diagram of the multiplier

while the first resulting blocks s_i become available at the multiplier's output port. After another five clock cycles, also input blocks m_j contain zero data while the remaining resulting blocks s_i are computed. In total, after 66 clock cycles, the multiplication of A and B is computed.

The advantage over previous implementations [38] is the intensive use of available DSP elements encapsulating computationally challenging modular integer operations in dedicated hardcores instead of common control logic to benefit from the FPGA's full potential. The maximal possible frequency of the Virtex-4 SX35 hardware is 400 MHz (for speedgrade 10). This is also the frequency the DSP elements can be triggered. Since these DSP elements are used intensively, the Montgomery multiplication implementation consists of a minimum of general logic blocks which are usually the bottle neck of fast implementations due to long propagation paths.

7.4. Addition/Subtraction Unit

Usually, the number of clock cycles needed to perform a modular addition/subtraction is not of highest importance since the modular multiplier is much more complex and typically the actual bottle neck. Therefore, we implemented Algorithm 13 in a way that two operations (i.e., addition or subtraction) can be performed concurrently with one multiplication. This goes in line with the optimizations of the ECM operations described previously in Section 5.4.

The addition/subtraction unit illustrated in Figure 7.9 consists of two DSP elements and supports 170-bit moduli. All parameters, the inputs A and B, and the output S are expressed in 34-bit blocks. Furthermore, the `SUB_MODE` signal selects

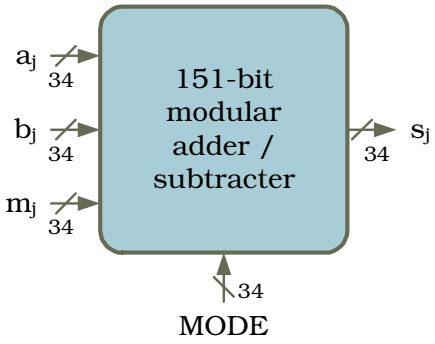

Figure 7.9.: Modular adder/subtracter

the operation mode, i.e. addition or subtraction, `CE` is used for activating and `RST` for resetting the unit, and the `CLK` signal is used for clocking tasks. The result S is calculated and returned within 24 clock cycles.

The implementation of our adder/subtracter is depicted in Figure 7.10. The left most DSP processes input parameters A and B where the 34-bit summand A is split into two 17-bit blocks which are transferred to the 17-bit DSP's internal signals `A` and `B`. The 34-bit internal signal C is used to load summand B into the DSP. Depending on the `SUB_MODE` signal ($0 \hat{=}$ "add" and $1 = \hat{=}$ "subtract"), DSP 1 calculates $a_j \pm b_j \pm c_{j-1}$ (see Algorithm 13) where c_{j-1} is a carry/borrow bit, depending on the operation mode. The result is routed to DSP 2 by using the high speed connection between cascaded DSP elements. The second DSP loads the modulus $2\tilde{M}$ by transferring it to the internal signals `A` and `B` of DSP 2, and calculates $a_j \pm b_j \pm c_{j-1} \mp m_j \mp (c_{j-1} \oplus \tilde{c}_{j-1})$ (see Algorithm 13) where $\tilde{c}_j$ is a borrow/carry bit, which depends on the current `SUB_MODE` signal. Intermediate blocks s_j and s'_j are fed into a shift register until the second DSP calculates s'_4. Next, it can be determined whether S or S' is the correct result by analyzing the

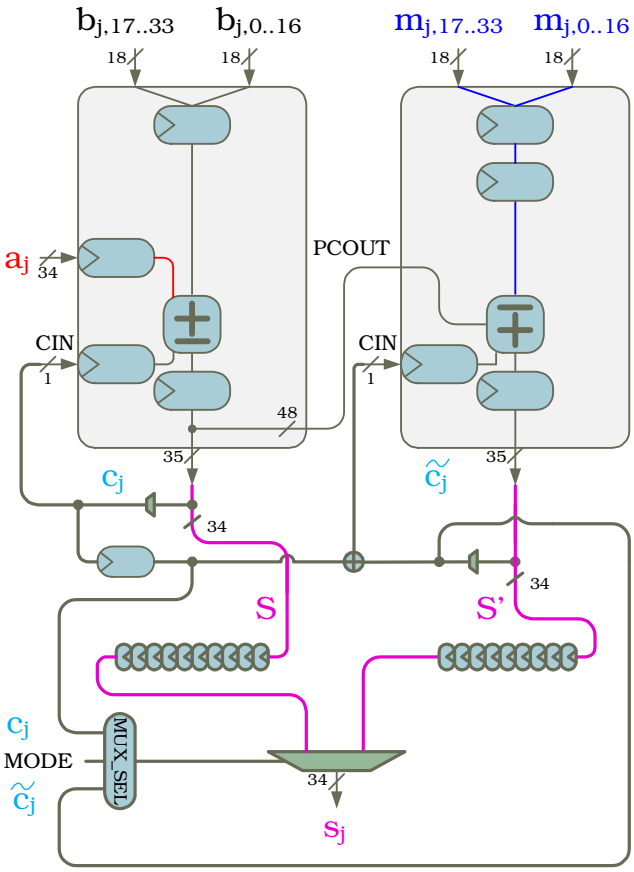

Figure 7.10.: Modular addition/subtraction unit based on DSP elements

last borrow bit of DSP 1 or DSP 2, depending on the operation mode. In case of addition, if borrow bit $\tilde{c}_4$ of DSP 2 occurs, i.e. $A + B - M < 0$, then S is used as output. For subtraction, borrow bit c_4 is tested. If it is equal to 1, i.e. $A - B < 0$, then S' is used as output, otherwise S. Note that the second DSP must wait a clock cycle until it can determine $c_{j-1} \oplus \tilde{c}_{j-1}$ since input blocks are loaded with one clock cycle delay.

Figure 7.11 shows a timing diagram of the modular addition/subtraction unit. It is necessary to enable and reset the unit first before two values can be added or subtracted. It it also necessary to reset the unit, directly before the next operation is performed. After resetting and selecting the operation mode, a_j, b_j, and m_j are loaded 34-bit block-wise every second clock cycle. Then, after 11 clock cycles the first 34-bit block of the result is available at the unit's output port. The other four resulting blocks are returned after another two clock cycles, and in total, 24 clock cycles are needed to calculate and return the correct result.

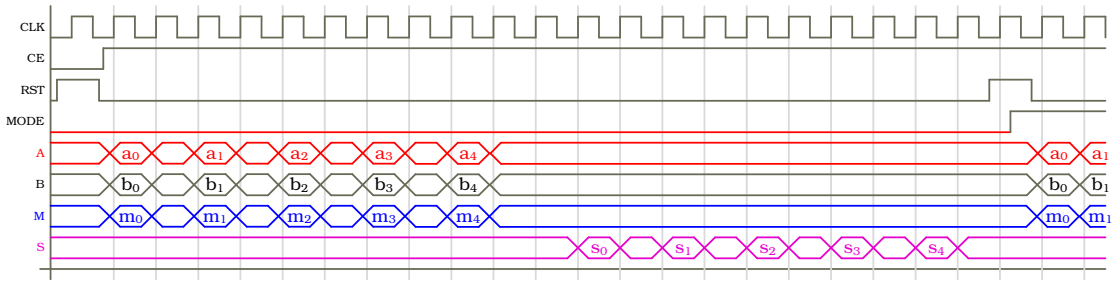

Figure 7.11.: Timing diagram of the adder/subtracter

7.5. Storage Management

The two multipliers and the addition/subtraction unit are connected to several block RAM elements in which input parameters, intermediate results, and final results are stored. The corresponding data path are illustrated in Figure 7.12. The

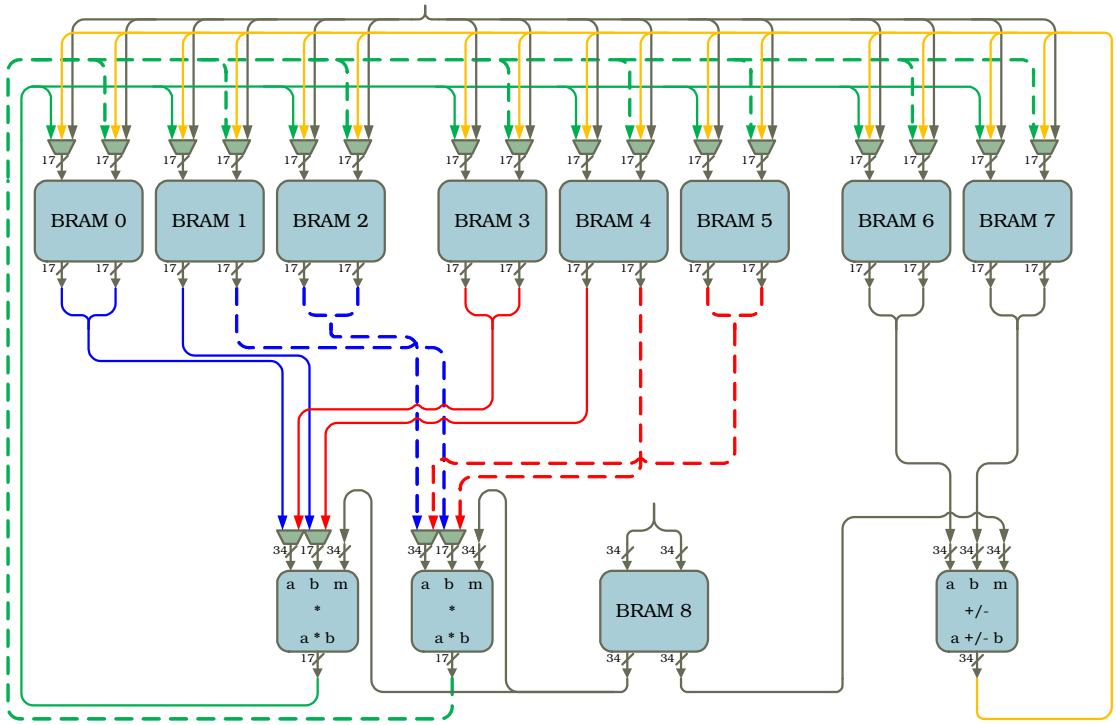

Figure 7.12.: Local memory and arithmetic components of one ECM unit

block RAM elements which are dedicated to the two multipliers (BRAM 0 - 5) are grouped into two storage blocks, namely storage block I and II. Each group consists

of three block RAM elements and is connected to both multipliers. The left most block RAM elements of storage block I and II (BRAM 0 and 3) contain the input parameter A for the first multiplier, block RAM 1 and 4 contain input parameter B of both multipliers, and block RAM 2 and 5 contain input parameter A of the second multiplier. Each left side port is connected to the first multiplier whereas the right side port is connected to the second multiplier. A second storage block is necessary since the addition/subtraction unit needs to store data into the block RAM elements dedicated to the multipliers while the multipliers are reading input data from them. Storage block III consists of two block RAM elements (BRAM 6 and 7) which are dedicated to the addition/subtraction unit. It contains input parameters A and B for the addition/subtraction unit. Block RAM 8 is dedicated for storing both moduli M'' and $2\tilde{M}$ and is therefore connected to each of the three operation units.

Since multiplication results are required to perform further multiplications or additions/subtractions, the output of the two multipliers is connected to all storage blocks (BRAM 0-7). A selection of the storage blocks in which the multiplier's outputs are stored can be configured by the instruction. All combinations of storage blocks I, II, and III are possible options. The same is true for the output of the addition/subtraction unit.

Next, three examples are described going into more detail of the storage management of our ECM unit. First, loading parameters into the local memory is described. Then, an addition is performed which is followed by a multiplication.

The following instructions encode the load operation for the x-coordinate of the initial point P_0, the moduli used for additions/subtractions $2\tilde{M}$, and the z-coordinate of P_0.

```
1   -- load x_p and 2M into BRAM
    "00" & '0' & "--101" & "010" & x"------------" & x"------------" & x"0b0a0b0a0"
    "00" & '0' & "--101" & "010" & x"------------" & x"------------" & x"0d0c0d0c2"
    "00" & '0' & "--101" & "010" & x"------------" & x"------------" & x"0f0e0f0e4"
5   "00" & '0' & "--101" & "010" & x"------------" & x"------------" & x"111011106"
    "00" & '0' & "--101" & "010" & x"------------" & x"------------" & x"131213128"

    -- load z_p into BRAM
    "00" & '0' & "--100" & "010" & x"------------" & x"------------" & x"15141514a"
10  "00" & '0' & "--100" & "010" & x"------------" & x"------------" & x"17161716a"
    "00" & '0' & "--100" & "010" & x"------------" & x"------------" & x"19181918a"
    "00" & '0' & "--100" & "010" & x"------------" & x"------------" & x"1b1a1b1aa"
    "00" & '0' & "--100" & "010" & x"------------" & x"------------" & x"1d1c1d1ca"
```

The instructions in line 2-6 encode the addresses of x_p and the modulo $2\tilde{M}$ at which cells these values are stored. It is also encoded that only storage block III and BRAM 8 are targets for a write operation. The other instructions determine the addresses of z_p which are also used for storage block III. Table A.4 shows the corresponding content of the block RAM elements of storage block III after the loading sequence.

The next set of instructions perform a modular addition of x_p and z_p, $a_1 = x_p +$

$z_p \pmod{2\tilde{M}}$.

```
1  -- start addition: a_1 = x_p + z_p
   "00" & '0' & "--000" & "001" & x"------------" & x"------------" & x"0c0a16140"
   "00" & '0' & "--000" & "001" & x"------------" & x"------------" & x"0c0a16140"
   "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"100e1a182"
5  "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"100e1a182"
   "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"0b12151c4"
   "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"0b12151c4"
   "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"0f0d19176"
   "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"0f0d19176"
10 "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"13111d1b8"
   "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"13111d1b8"

   -- delay
   "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"fffffffa"
15 "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"fffffffa"
   "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"fffffffa"

   -- start subtraction: s_1 = x_p - z_p;
   "01" & '0' & "--000" & "001" & x"------------" & x"------------" & x"0c0a16140"
20
   -- start write a_1 into BRAM
   "00" & '0' & "-1000" & "001" & x"------------" & x"201e201e201ea" & x"0c0a16140"
   "01" & '1' & "-1000" & "001" & x"------------" & x"201e201e201ea" & x"100e1a182"
   "01" & '1' & "-1000" & "001" & x"------------" & x"242224222422a" & x"100e1a182"
25 "01" & '1' & "-1000" & "001" & x"------------" & x"242224222422a" & x"0b12151c4"
   "01" & '1' & "-1000" & "001" & x"------------" & x"1f261f261f26a" & x"0b12151c4"
   "01" & '1' & "-1000" & "001" & x"------------" & x"1f261f261f26a" & x"0f0d19176"
   "01" & '1' & "-1000" & "001" & x"------------" & x"232123212321a" & x"0f0d19176"
```

Therefore, instructions in lines 2-11 encode reading the input parameters while in each clock cycle the blocks a_j and b_j are added corresponding to Algorithm 13. After three delay cycles (lines 14-16), the output is stored 34-bit wise in storage block II at the addresses `0x1E -- 0x27`. In line 19, the subtraction $x_p - z_p$ is started while a_1 is being stored. Figure 7.13 illustrates the connections between the addition/subtraction unit and involved block RAM elements. Note that in this example block RAM elements 0-2 are not used, neither for loading parameters nor for storing results. Furthermore, unused connections between components like the block RAM elements 0-5 and multiplier units are masked out.

The following example deals with the execution of an addition while two multiplications are performed simultaneously. Corresponding instructions are listed below.

```
1  -- address delay
   "00" & '0' & "--000" & "001" & x"------------" & x"------------" & x"34323e3c0"
   "00" & '0' & "--000" & "001" & x"------------" & x"------------" & x"34323e3c0"

5  -- mul: m_1 = a_1 * a_1; m_2 = s_1 * s_1; add: a_2 = x_q + z_q;
   -- b0
   "01" & '0' & "-0000" & "100" & x"------------" & x"1f1e1e2829281" & x"383642402"
   "11" & '0' & "-0000" & "100" & x"------------" & x"21201e282b2a3" & x"383642402"
   "11" & '0' & "-0000" & "100" & x"------------" & x"23221e282d2c5" & x"333a3d444"
10 "11" & '0' & "-0000" & "100" & x"------------" & x"25241e282f2e7" & x"333a3d444"
   "11" & '0' & "-0000" & "100" & x"------------" & x"27261e2831309" & x"3735413f6"

   -- b1
   "11" & '0' & "-0000" & "100" & x"------------" & x"1f1e202a29281" & x"3735413f6"
15 "11" & '0' & "-0000" & "100" & x"------------" & x"2120202a2b2a3" & x"3b3945438"
   "11" & '0' & "-0000" & "100" & x"------------" & x"2322202a2d2c5" & x"3b3945438"
   "11" & '0' & "-0000" & "100" & x"------------" & x"2524202a2f2e7" & x"fffffffa"
   "11" & '0' & "-0000" & "100" & x"------------" & x"2726202a31309" & x"fffffffa"

20 -- b2
   "11" & '0' & "-0000" & "101" & x"------------" & x"1f1e222c29281" & x"fffffffa"
   -- start subtraction: s_2 = x_q - z_q;
   "11" & '0' & "-0000" & "101" & x"------------" & x"2120222c2b2a3" & x"34323e3c0"
   -- write a_2 into storage block I while multiplier is reading data from storage block II
25 "10" & '0' & "10000" & "101" & x"343234323432" & x"2322222c2d2c5" & x"34323e3c0"
```

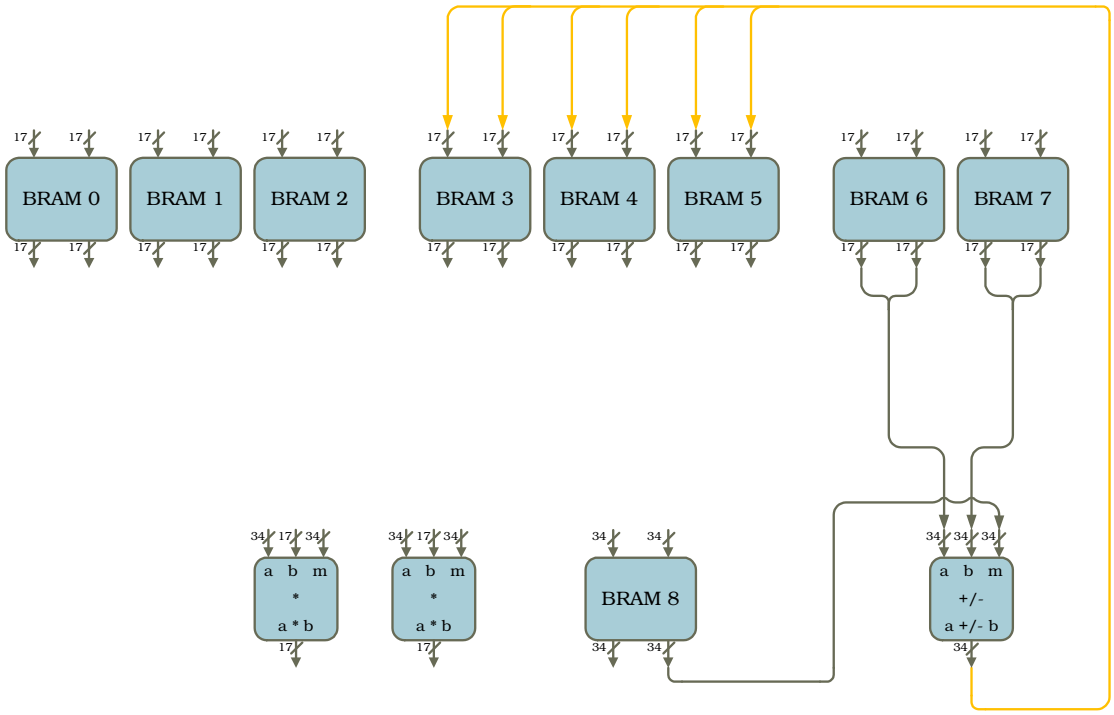

Figure 7.13.: Modular addition and involved components

```
    "11" & '1' & "10000" & "101" & x"343234323432" & x"2524222c2f2e7" & x"383642402"
    "11" & '1' & "10000" & "101" & x"383638363836" & x"2726222c31309" & x"383642402"

    -- b3
30  "11" & '1' & "10000" & "101" & x"383638363836" & x"1f1e242e29281" & x"333a3d444"
    "11" & '1' & "10000" & "101" & x"333a333a333a" & x"2120242e2b2a3" & x"333a3d444"
    "11" & '1' & "10000" & "101" & x"333a333a333a" & x"2322242e2d2c5" & x"3735413f6"
    "11" & '1' & "10000" & "101" & x"373537353735" & x"2524242e2f2e7" & x"3735413f6"
    "11" & '1' & "10000" & "101" & x"373537353735" & x"2726242e31309" & x"3b3945438"
35
    -- b4
    "11" & '1' & "10000" & "101" & x"3b393b393b39" & x"1f1e263029281" & x"3b3945438"
    "11" & '1' & "10000" & "101" & x"3b393b393b39" & x"212026302b2a3" & x"----------"
    "11" & '1' & "-0-00" & "101" & x"------------" & x"232226302d2c5" & x"----------"
40  "11" & '1' & "-0-00" & "101" & x"------------" & x"252426302f2e7" & x"----------"
    "11" & '1' & "-0-00" & "101" & x"------------" & x"2726263031309" & x"----------"

    ...

45  -- b9
    "10" & '0' & "-0-00" & "100" & x"------------" & x"1f1e273129281" & x"----------"
    "10" & '0' & "-0-00" & "100" & x"------------" & x"212027312b2a3" & x"----------"
    "10" & '0' & "-0-00" & "100" & x"------------" & x"232227312d2c5" & x"----------"
    "10" & '0' & "-0-00" & "100" & x"------------" & x"252427312f2e7" & x"----------"
50  "10" & '0' & "-0-00" & "100" & x"------------" & x"2726273131309" & x"----------"

    -- b = 0
    "10" & '0' & "-0-00" & "100" & x"------------" & x"1f1efffff29281" & x"----------"
    "10" & '0' & "-0-00" & "100" & x"------------" & x"2120ffff2b2a3" & x"----------"
55  "10" & '0' & "-0-00" & "100" & x"------------" & x"2322ffff2d2c5" & x"----------"
    "10" & '0' & "-0-00" & "100" & x"------------" & x"2524ffff2f2e7" & x"----------"
    "10" & '0' & "-0-00" & "100" & x"------------" & x"2726ffff31309" & x"----------"

    -- b = 0, a = 0
60  "10" & '0' & "---00" & "100" & x"------------" & x"------------1" & x"----------"
    "10" & '0' & "---00" & "100" & x"------------" & x"------------3" & x"----------"
    "10" & '0' & "---00" & "100" & x"------------" & x"------------5" & x"----------"
    -- write m_1 and m_2 into storage blocks II and III
```

```
    "10" & '0' & "-1100" & "100" & x"------------" & x"1e281e281e287" & x"1e281e28a"
65  "10" & '0' & "-1100" & "100" & x"------------" & x"202a202a202a9" & x"202a202aa"

    -- b = 0, a = 0, m = 0
    "10" & '0' & "-1100" & "100" & x"------------" & x"222c222c222ca" & x"222c222ca"
    "10" & '0' & "-1100" & "100" & x"------------" & x"242e242e242ea" & x"242e242ea"
70  "10" & '0' & "-1100" & "100" & x"------------" & x"263026302630a" & x"26302630a"
    "10" & '0' & "-1100" & "100" & x"------------" & x"1f291f291f29a" & x"1f291f29a"
    "10" & '0' & "-1100" & "100" & x"------------" & x"212b212b212ba" & x"212b212ba"
    "10" & '0' & "-1100" & "100" & x"------------" & x"232d232d232da" & x"232d232da"
    "10" & '0' & "-1100" & "100" & x"------------" & x"252f252f252fa" & x"252f252fa"
75  "10" & '0' & "-1100" & "100" & x"------------" & x"273127312731a" & x"27312731a"
```

After two delay cycles in which addresses for addition parameters are preloaded, the multiplications and the addition are started. The multipliers use different input parameters. In particular, the first multiplier processes a_1 stored at addresses `0x1E -- 0x27` which is used for the multiplier input A and B whereas the second multiplier processes s_1 stored at addresses `0x28 -- 0x31`. In line 25, the multipliers are reading data blocks from storage block II while the addition/subtraction unit stores data into storage block I. Obviously, the addition/subtraction unit cannot use storage block II for writing results back to memory since the current input parameters which are used by the multipliers would be erroneously overwritten. After line 64, all multiplication results are stored into storage blocks II and III. The connections between the arithmetic units and involved block RAM elements while two multiplications and one addition are performed is pictured in Figure 7.14. Note that also in this example unused connections between components such as the block RAM elements 0-2 and multiplier units are masked out.

In Tables A.5, A.6, and A.7), the address ranges of the intermediate results and the corresponding storage blocks are listed for point doubling and a single step of the Montgomery ladder ($k_i = 0$ and $k_i = 1$). Note that x_{2P} is always stored at addresses `0x1E -- 0x27`, z_{2P} at addresses `0x28 -- 0x31`, x_{P+Q} at addresses `0x32 -- 0x3B`, and z_{P+Q} at addresses `0x3C -- 0x45` of storage block III.

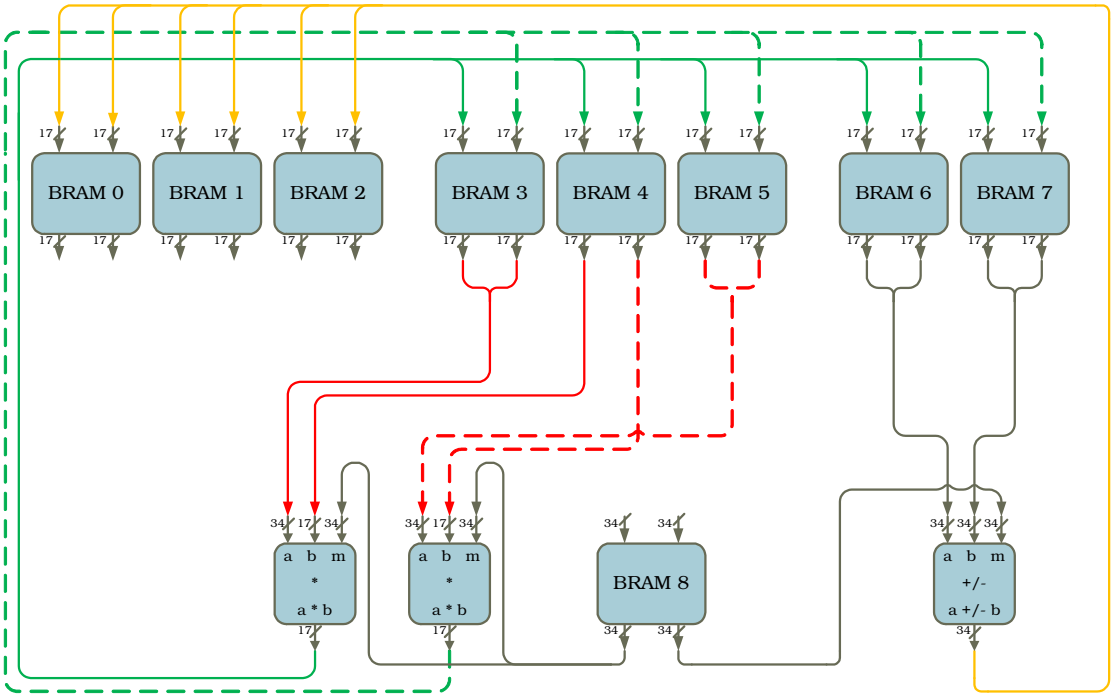

Figure 7.14.: Several operations and involved components

8. Implementation Results and Comparisons

In this section, we present our implementation results which are obtained by sufficient hardware simulations (including place & route). On the one hand, we present our results regarding the implementation of low-level components (i.e., modular multiplier and addition/subtraction unit). On the other hand, we compare our achievements with the results obtained by [13, 14] in which we expect the fastest FPGA implementation of the elliptic curve method is described.

8.1. Low-level Components

The results of the implemented modular multiplication unit are presented in Table 8.1. The implementation is optimized such that both word width k and delay parameter d, which are parameters of the DSP-optimized multiplication algorithm, remain constant (i.e., $d = 0$ and $k = 17$). The multiplication unit takes a multi-

Multiplier properties	
Input A Input B Input M'' Output S_{n+d+2}	$0 \leq A < 2^{170}$ $0 \leq B < 2^{170}$ $0 \leq M'' < 2^{151}$ $0 \leq S_{n+d+2} < 2^{170}$
#DSPs	$\delta = \left\lceil \frac{h+k(d+1)+2}{k} \right\rceil = 10$
#Rounds	$r = n + d + 1 = 11$, $n = \delta$
Max. clock freq.	400 MHz
Result after	$5 \times r + \delta + 1 =$ 66 cycles = 165 ns

Table 8.1.: Results of the implemented modular multiplier

plicand A, a multiplier B, and a modulus dependent value M'' as inputs. Since the Montgomery multiplication computes values greater than the current modulus,

the inputs A and B for our multiplier are limited to 170-bit values. Input A and M'' are provided in 34-bit blocks whereas input B is fed in by 17-bit blocks. The result S is also returned in 17-bit blocks. The multiplier handles h-bit moduli with $h = 151$. The number of DSP elements is equal to the number of blocks n. The implementation of this multiplier can be operated at a maximum clock frequency of 400 MHz. The result $S_{n+d+2} \equiv A \cdot B \cdot R^{-1} \pmod{M}$ is provided after $n + 1$ rounds. Hence, the result is available after 66 clock cycles which is equivalent to 165 ns.

The results of our implemented modular addition/subtraction unit are listed in Table 8.2. The modular adder/subtracter is implemented by means of two DSP elements. This number remains constant even when higher bit lengths are used. Input parameters A, B, and $2\tilde{M}$ are of size 2^{170} such that operands calculated by

Adder/Subtracter properties	
Input A Input B Input $2\tilde{M}$ Output S	$0 \leq A < 2^{170}$ $0 \leq B < 2^{170}$ $0 \leq 2\tilde{M} < 2^{170}$ $0 \leq S < 2^{170}$
#DSPs	2
#Input blocks	$n = 5$
Max. clock freq.	400 MHz
Result after	$2 \cdot (n \times \#DSPs) + 4 =$ 24 cycles = 60 ns

Table 8.2.: Results of the implemented modular addition/subtraction unit

our modular multiplier can be processed. Note that $2\tilde{M}$ is the modulus used for addition/subtraction tasks. This number is a multiple of the origin modulus M such that results calculated modulo $2\tilde{M}$ are congruent to values modulo M. The implemented addition/subtraction unit can be operated at 400 MHz as well as our multiplier unit. The result is available after 24 clock cycles which is equivalent to 60 ns. Thus, at least two additions/subtractions can be performed during the time needed for one multiplication.

8.2. Comparisons

Our 151-bit ECM system is realized by the described low-level components and provides the execution of Phase 1 of the elliptic curve method. In Table 8.3, the achieved results of our ECM system are compared with the implementation results

obtained by [13, 14] considering a modulus of the same bit length. We attain

	Our results #clock cycles	[13, 14] # clock cycles
Modular addition	24	31[1]
Modular multiplication	66	167
$2 * P$	287	n/a
$2 * P$ and $P \oplus Q$	377	947

Table 8.3.: Results of our ECM system for Phase 1

an improvement in all elementary operations which are required for performing Phase 1 (i.e., modular addition/subtraction, modular multiplication, point doubling on elliptic curves, and one step of the Montgomery ladder ($2 * P$ and $P \oplus Q$) for $z_p = 1$). Concerning these operations, our main improvement relies on the efficient implementation of the multiplication unit. Considering the number of clock cycles, this low-level component is approximately 2.5 times faster than the multiplier implemented by [13, 14]. Since one step of the Montgomery ladder basically consists of five consecutive multiplications, this significant improvement also results into a seriously reduced number of clock cycles needed to perform one step of the Montgomery ladder ($947 \rightarrow 377$).

In order to get a realistic approximation of the running time of the ECM hardware, we perform place & route which maps the design on the FPGA device and calculates timings and signal delays. Consequently, our ECM system achieves the results described in Table 8.4. Note that the frequencies of our multiplier and adder/subtracter can only be reached if such a unit is solely implemented on the FPGA. However, we also attain better results for all elementary components considering the maximum frequency. In total, our ECM system performs Phase 1 about 3.2 times faster than the hardware implemented by [13, 14].

8.3. Phase 2 Estimate

Regarding to Algorithm 10 (Phase 2 of the elliptic curve method), we estimate the running time of Phase 2 without its complete implementation. Therefore, we introduce new parameters T_1, T_2, T_3, n_{primes}, and M_N. T_1 denotes the time which is needed for one step of the Montgomery ladder in Phase 2 ($2P$ and $P \oplus Q$ with

[1] Since [13, 14] gives no formula for the number of clock cycles of a modular addition, we linearly approximate this number for 151-bit moduli by $\frac{198-bit}{151-bit} = 1.311$.

	Our results	[13, 14]
Modular adder/ subtracter	400 MHz	n/a
Modular multiplier	400 MHz	n/a
One ECM unit	195 MHz	135 MHz
ECM system (all ECM units)	174 MHz	n/a
Max. number of ECM units	8	4[2]
Time for Phase 1 per ECM unit	2.987 ms	9.7 ms

Table 8.4.: Results of our ECM system for Phase 1 after place & route

$z_p \neq 1$), T_2 denotes the running time of a point addition ($R \oplus Q$), the time T_3 is needed for one Montgomery multiplication, n_{primes} is the number of ones in a precomputed table called prime_table, and M_N is the number of possible values for m. Table 8.5 outlines these parameters.

Elementary operations and parameters of Phase 2		
$2P$ and $P \oplus Q$ with $z_p \neq 1$	T_1	445
$R \oplus Q$	T_2	315
Mod. multiplication	T_3	66
Number of primes	n_{primes}	4361
Number of different m	M_N	267

Table 8.5.: Parameters required for estimating the running time of Phase 2

In Table 8.6, we estimate the running time of Phase 2 where $B_1 = 960$, $B_2 = 57000$, and $D = 210$ are required parameters. The presented formulas, which are also used by [13, 14], are obtained by analyzing Algorithm 10. Furthermore, we assume that the frequency of the Phase 2 implementation remains unchanged to our current implementation. As expected, the time needed to perform Phase 2 ($3,17$ ms) is similar to the running time of Phase 1 (2.987 ms).

[2] Since [13, 14] used a high-level Virtex-4 VLX200 FPGA, we divide the actual number of ECM units by 5.8 to become comparable. This factor is determined by $\frac{\#\text{Slices in Virtex-4 VLX200}}{\#\text{Slices in Virtex-4 SX-35}} = \frac{89088}{15360}$ and expect that this is a fair approximation.

Operation	Timing formulas	#clock cycles
Precomputations		
$j * Q$	$T_{jQ} = 2T_1 + (\lfloor D/4 \rfloor - 2)T_2$	16640
$D * Q$	$T_{DQ} = \lceil \log_2(D+1) \rceil T_1$	3560
$M_{MIN} * DQ$	$T_{M_{MIN}DQ} = \lceil \log_2(M_{MIN}+1) \rceil T_1$	1335
Computations		
mDQ	$T_{mDQ} = (M_N - 2)T_2$	83475
$\prod d$	$T_d = (\lceil 1,5 * n_{primes}) \rceil)T_3$	431772
Phase 2 estimate		
Phase 2 in total	$T_{Phase\ 2} = T_{jQ} + T_{DQ} + T_{M_{MIN}DQ} + T_{mDQ} + T_d$	536782
Time $[ms]$	@174 MHz	3,17

Table 8.6.: Execution time estimate of Phase 2

9. Conclusion and Future Work

We have efficiently implemented the elliptic curve method on reconfigurable hardware and therefore used new strategies for the realization of high-speed modular arithmetic operations. The optimization of these fundamental operations basically relies on the intensive use of DSP elements integrated in modern FPGA devices. We have achieved improvements in the running time and the maximal frequency of the hardware-based ECM. Most importantly, we have obtained a significantly higher performance of the multiplier in time and frequency. Based on these improvements, we have achieved a significant gain in performance of the elliptic curve method at least for Phase 1.

Since the cost for the ECM hardware is required for determining the cost-benefit ratio, we need to consider this in our comparisons. In Table 9.1, we compare two ECM implementations where [13, 14] uses an high-cost Virtex-4 FPGA whereas we choose a middle-cost model of the same FPGA family. As we can see, we obtain

	[13, 14]	Our results	Ratio
FPGA	XC4VLX200-11	XC4VSX35-10	
#ECM units per FPGA	27	8	
Cost of one FPGA[1]	7564	468	
Max. freq.	135 MHz	174 MHz	
#instructions for Phase 1	1302125	519750	
Time for Phase 1	9,7 ms	2,987 ms	
#Phase 1 ops. per sec.	2783	2666	
#Phase 1 ops. per sec. per $100	36	569	15.8

Table 9.1.: Comparison of two Virtex-4 based implementations

[1]Prices of a single FPGA taken from http://www.em.avnet.com in April 2008.

a significant improvement factor of 15.8 in the number of Phase 1 operations per second per $100 when using Virtex-4 hardware. Basically, this relies on the intensive use of DSP elements, similarly to our implementation.

Since [13, 14] concludes that low-cost FPGA devices have the best cost-benefit ratio, we also compare our results to a low-cost FPGA implementation used by [13, 14] which has the best properties regarding the number of Phase 1 operations per second per $100. In Table 9.2, we see that our implementation is just slightly faster but with no significant improvement. For this comparison, we choose the

	[13, 14]	Our results	Estimate
FPGA	XC3S5000-5	XC4VSX35-10	XC3SD3400A-4
#ECM units per FPGA	10	8	5
Cost of one FPGA[1]	153	468	156
Max. freq.	100 MHz	174 MHz	109 MHz[2]
#instructions for Phase 1	1302125	519750	519750
Time for Phase 1	13,1 ms	2,987 ms	4,8 ms
#Phase 1 ops. per sec.	763	2666	1041
#Phase 1 ops. per sec. per $100	498	569	667

Table 9.2.: Comparison of ECM systems considering the best cost-benefit ratio

Spartan 3 FPGA instead of the Spartan 3E because, in contrast to [13, 14], this device attains the best cost-benefit ratio considering our underlaying FPGA prices (e.g., #Phase 1 operations per second per $100 for the Spartan 3E is 387[1]). Nevertheless, the estimates regarding new low-cost FPGA devices like the Spartan 3A DSP indicate that on the one hand, FPGA devices supporting high-performance DSP elements are getting cheaper and on the other hand, the number of Phase 1 operations per second per $100 may increase considerably (498 → 667). Assuming that in the near future the cost for FPGA devices supporting DSP elements keeps falling, DSP-based implementations will become more important.

[1] Prices of a single FPGA taken from http://www.em.avnet.com in April 2008.

[2] The maximum frequency is estimated by a linear approximation of our results since DSP elements of Spartan 3A DSP FPGAs with speedgrade 4 can be triggered by 250 MHz only whereas DSP elements of Virtex-4 FPGAs with speedgrade 10 can be triggered by 400 MHz. The linear factor is calculated by $\frac{400\text{MHz}}{250\text{MHz}} = 1.6$ such that the estimated frequency is $\frac{174\text{MHz}}{1.6} \approx 109\text{MHz}$.

Our future work includes adapting our implementation to other bit lengths in order to accelerate the general number field sieve as well as possible. For that purpose, the low-level operation units must be modified. This is essentially the adaptation of the number of DSP elements in the case of the multiplier and the modification of the output control in the case of the addition/subtraction unit. Furthermore, we aim to complete the implementation of Phase 2 to realize an entirely hardware-based ECM. This task requires at least an adaptation of the storage management and an enlargement of the instruction set. Another important goal is integrating our ECM systems in the special purpose hardware COPACOBANA which should enable realizing a fine-granulated network of ECM systems with best possible support of the GNFS sieving step. Its realization includes at least the development of a framework in which the ECM systems are embedded. Such a framework handles the I/O communication between the ECM systems and the controller of the COPACOBANA that is connected to a host system.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 70–77, Los Alamitos, CA, 1999. IEEE Computer Society Press.

[2] T. Blum and C. Paar. High-radix montgomery modular exponentiation on reconfigurable hardware. *IEEE Transactions on Computers*, 50(7):759–764, 2001.

[3] R. P. Brent. Some integer factorization algorithms using elliptic curves. *Australian Computer Science Communications 8*, pages 149–163, 1986.

[4] R. P. Brent. Factorization of the tenth and eleventh Fermat numbers. Technical Report TR-CS-96-02, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, February 1996.

[5] R. P. Brent. Factorization of the tenth Fermat number. *Mathematics of Computation*, 68(225):429–451, 1999.

[6] H. Cohen and G. Frey, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, pages 407–449. Discrete Mathematics and its Applications. Chapman & Hall/CRC, 2006. K.H. Rosen, éditeur de la collection.

[7] S. Contini. Factorworld web page. on web. `http://www.crypto-world.com/FactorWorld.html`.

[8] G. M. de Dormale and J. Quisquater. High-speed hardware implementations of elliptic curve cryptography: A survey. *Journal of System Architecture*, 53(2-3):72–84, 2007.

[9] P. de Fermat. Oeuvres 2, 256, 1894.

[10] G. de Meulenaer, F. Gosset, G. M. de Dormale, and J. Quisquater. Integer Factorization Based on Elliptic Curve Method: Towards Better Exploitation of Reconfigurable Hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM07)*. IEEE Computer Society Press, 2007.

[11] B. Dixon and A. K. Lenstra. Massively parallel elliptic curve factoring. *Lecture*

Notes in Computer Science, 658:183–193, 1993.

[12] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, M. Šimka, and C. Stahlke. An effcient hardware architecture for factoring integers with the Elliptic Curve Method. In *1st Workshop on Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2005*, pages 51–62, Paris, France, February 24–25, 2005.

[13] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi. Implementing the elliptic curve method of factoring in reconfigurable hardware. In *CHES*, pages 119–133, 2006.

[14] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi. Implementing the elliptic curve method of factoring in reconfigurable hardware. 2006.

[15] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[16] M. Joye and S. Yen. The montgomery powering ladder. In B.S. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 291–302, 2002.

[17] T. Kleinjung. Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024 bit integers. SHARCS 2006, 2006.

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

[19] N. Koblitz. *A course in number theory and cryptography*. Graduate Texts in Mathematics. Springer, 1998.

[20] N. Koblitz, A. J. Menezes, Y. Wu, and R. J. Zuccherato. *Algebraic aspects of cryptography*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[21] C. K. Koc, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms — assessing five algorithms that speed up modular exponentiation, the most popular method of encrypting and signing digital data. *IEEE Micro*, 16(3):26–33, 1996.

[22] M. Kraïtchik. *Théorie des nombres, Tome 2*. Gauthier-Villars, Paris, 1926.

[23] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with copacobana - a cost-optimized parallel code breaker. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2006.

[24] A. K. Lenstra, H. W. Lenstra Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In *ACM Symposium on Theory of Computing*, pages 564–572, 1990.

[25] H. W. Lenstra. *Annals of mathematics*. Factoring integers with elliptic curves. 1987.

[26] C. McIvor, M. McLoone, and J. V. McCanny. FPGA montgomery multiplier architectures - a comparison. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 279–282, Washington, DC, USA, 2004. IEEE Computer Society.

[27] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Norwell, MA, USA, 1994. Foreword By-Neal Koblitz.

[28] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

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

[30] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

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

[32] P. L. Montgomery. *An FFT extension of the elliptic curve method of factorization*. PhD thesis, Los Angeles, CA, USA, 1992.

[33] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *ARITH '95: Proceedings of the 12th Symposium on Computer Arithmetic*, page 193, Washington, DC, USA, 1995. IEEE Computer Society.

[34] J. Pelzl, M. Šimka, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer, and C. Paar. Area–time efficient hardware architecture for factoring integers with the elliptic curve method. *IEE Proceedings - Information Security*, 152(1):67–78, 2005.

[35] C. Pomerance. A tale of two sieves. *AMS*, pages 1473–1485, 1996.

[36] D. Shanks. On Maximal Gaps between Successive Primes. *Mathematics of Computation*, 18:646–651, 1964.

[37] J. H. Silverman. *The Arithmetic of Elliptic Curves*. Number 106 in Graduate Texts in Mathematics. Springer, 1986.

[38] D. Suzuki. How to maximize the potential of FPGA resources for modular

exponentiation. In *CHES*, pages 272–288, 2007.

[39] A. F. Tenca and C. K. Koc. A scalable architecture for montgomery multiplication. In *Cryptographic Hardware and Embedded Systems*, pages 94–108, 1999.

[40] M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer, and C. Paar. Hardware factorization based on elliptic curve method. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 107–116, Napa Valley, California, 2005. IEEE Computer Society Press.

[41] P. Zimmermann. Integer factoring records web page. on web. `http://www.loria.fr/~zimmerma/records/factor.html`.

[42] P. Zimmermann and B. Dodson. 20 years of ECM. In Florian Hess, Sebastian Pauli, and Michael E. Pohst, editors, *ANTS*, volume 4076 of *Lecture Notes in Computer Science*, pages 525–542. Springer, 2006.

A. Appendix

A.1. Fermat's and Kraïtchik's ideas in more detail

As mentioned in the Chapter 1, the main ideas behind modern factoring algorithms are based on suggestions proposed by Fermat (1607-1665) [9] and Kraïtchik (1882-1957) [22].

Fermat expressed composite integers as a difference of two squares of the form

$$n = p \cdot q = x^2 - y^2$$

where all given numbers are positive integers. This equation holds true for all odd and composite integers n since the product of two odd numbers $p = (2s + 1)$ and $q = (2t + 1)$ also results in an odd number $(2s + 1)(2t + 1) = (2k + 1)$ with $k = 2st + s + t$. Hence, if we assume that $n = p \cdot q$ is odd, then p and q are also odd. We also know that adding and subtracting two odd numbers results in an even integer. Hence, two numbers can be constructed such that $x = \frac{p+q}{2}$ and $y = \frac{p-q}{2}$ where x and y remain integers without a residue. Then, the difference of the corresponding squares is described as follows:

$$x^2 - y^2 = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2 = \frac{p^2 + 2pq + q^2}{4} - \frac{p^2 - 2pq + q^2}{4} = p \cdot q.$$

Consequently, factors of n can be determined easily by computing

$$p = (x + y) \text{ and } q = (x - y)$$

which holds true since $(x^2 - y^2) = (x + y)(x - y)$ is satisfied for all x and y.

Kraïtchik extended Fermat's idea by two new suggestions. Firstly, Kraïtchik was also looking for differences of squares which are multiples of n such that

$$k \cdot n = k \cdot p \cdot q = x^2 - y^2$$

where k is a positive integer. In other words, Kraïtchik's was looking for quadratic congruences satisfying

$$x^2 \equiv y^2 \pmod{n}.$$

If suitable x^2 and y^2 are determined, number n can easily be factored again. Assuming $(x \mp y)$ is not equal to $(n \cdot k)$ which means that $(x+y)$ and $(x-y)$ are non-trivial factors of n. Then, p and q must be factors of $(x+y)$ and $(x-y)$ since $(k \cdot p \cdot q) = (x+y)(x-y)$ is true under the given assumption. Hence, p and q can be determined easily by computing

$$p = \gcd(x+y, n) \text{ and } q = \gcd(x-y, n).$$

Kraïtchik secondly proposed a method for finding such quadratic congruences which is described as follows:

a) For $x_i = 1, 2, \ldots$ compute $x_i^2 \equiv Q_i(x_i) \pmod{n}$, where $Q_i(x_i) = x_i^2 - n$

b) Factorize $Q_i(x_i)$, e.g. by trial division

c) Determine a combination of factorized $Q_i(x_i)$ which multiplicated results in a square $v^2 = \prod_{i=1}^{k} Q_i(x_i)$; use factors of $Q_i(x_i)$ for that

d) Multiply the x_i of the corresponding combination to $u^2 = \prod_{i=1}^{k} x_i^2$, where $u^2 \equiv v^2 \bmod n$

This method is applicable since the equation

$$u^2 = \prod_{i=1}^{k} x_i^2 = x_1^2 \cdots x_k^2 \equiv (x_1^2 - n) \cdots (x_k^2 - n) \equiv \prod_{i=1}^{k} Q_i(x_i) \equiv v^2 \pmod{n}$$

is true.

A.2. Tables

	DSP 1				DSP 2			
Cycle	Input		Opmode	Output	Input		Opmode	Output
1	a_0	b_0	0x25				Idle	
2	m_0	q_0	0x35		a_1	b_0	0x5e	
3	$S_{0,1}$	1	0x35		m_1	q_0	0x35	
4			0x25		$S_{0,2}$	1	0x35	
5			0x25				0x25	
6	a_0	b_1	0x25	$S_{1,0}, q_1$			0x25	
7	m_0	q_1	0x35		a_1	b_1	0x5e	$S_{1,1}$
8	$S_{1,1}$	1	0x35		m_1	q_1	0x35	
9			0x25		$S_{1,2}$	1	0x35	
10			0x25				0x25	
11	a_0	b_2	0x25	$S_{2,0}, q_2$			0x25	
12	m_0	q_2	0x35		a_1	b_2	0x5e	$S_{2,1}$
...								
46	a_0	b_9	0x25	$S_{9,0}, q_9$			Idle	
47	m_0	q_9	0x35		a_1	b_9	0x5e	
48	$S_{9,1}$	1	0x35		m_1	q_9	0x35	
49			0x25		$S_{9,2}$	1	0x35	
50			0x25				0x25	
51	0	0	0x25	$S_{10,0}, q_{10}$			0x25	
52	m_0	q_{10}	0x35		0	0	0x5e	$S_{10,1}$
53	$S_{10,1}$	1	0x35		m_1	q_{10}	0x35	
54			0x25		$S_{10,2}$	1	0x35	
55			0x25				0x25	
56	0	0	0x25	$S_{11,0}, q_2$			0x25	
57	0	0	0x35		0	0	0x5e	$S_{11,1}$

Table A.1.: Operation modes for DSP 1 and DSP 2 required by the multiplier

	DSP 5				DSP 6			
Cycle	Input		Opmode	Output	Input		Opmode	Output
1			`Idle`				`Idle`	
2			`Idle`				`Idle`	
3			`Idle`				`Idle`	
4			`Idle`				`Idle`	
5	a_4	b_0	`0x5e`				`Idle`	
6	m_4	q_0	`0x35`		a_5	b_0	`0x5e`	
7	$S_{0,5}$	1	`0x35`		m_5	q_0	`0x35`	
8			`0x25`		$S_{0,6}$	1	`0x35`	
9			`0x25`				`0x25`	
10	a_4	b_1	`0x5e`	$S_{1,4}$			`0x25`	
11	m_4	q_1	`0x35`		a_5	b_1	`0x5e`	$S_{1,5}$
12	$S_{1,5}$	1	`0x35`		m_5	q_1	`0x35`	
...								
46	m_4	q_8	`0x35`		a_5	b_8	`0x5e`	
47	$S_{8,5}$	1	`0x35`		m_5	q_8	`0x35`	
48			`0x25`		$S_{8,6}$	1	`0x35`	
49			`0x25`				`0x25`	
50	a_4	b_9	`0x5e`	$S_{9,4}$			`0x25`	
51	m_4	q_9	`0x35`		a_5	b_9	`0x5e`	$S_{9,5}$
52	$S_{9,5}$	1	`0x35`		m_5	q_9	`0x35`	
53			`0x25`		$S_{9,6}$	1	`0x35`	
54			`0x25`				`0x25`	
55	0	0	`0x5e`	$S_{10,4}$			`0x25`	
56	m_4	q_{10}	`0x35`		0	0	`0x5e`	$S_{10,5}$
57	$S_{10,5}$	1	`0x35`		m_5	q_{10}	`0x35`	
58			`0x25`		$S_{10,6}$	1	`0x35`	
59			`0x25`				`0x25`	
60	0	0	`0x5e`	$S_{11,4}$			`0x25`	
61	0	0	`0x35`		0	0	`0x5e`	$S_{11,5}$

Table A.2.: Operation modes for DSP 5 and DSP 6 required by the multiplier

	DSP 9				DSP 10			
Cycle	Input		Opmode	Output	Input		Opmode	Output
1			`Idle`					
2			`Idle`				`Idle`	
3			`Idle`				`Idle`	
4			`Idle`				`Idle`	
5			`Idle`				`Idle`	
6			`Idle`				`Idle`	
7			`Idle`				`Idle`	
8			`Idle`				`Idle`	
9	a_8	b_0	`0x5e`				`Idle`	
10	m_8	q_0	`0x35`		a_9	b_0	`0x5e`	
11	$S_{0,9}$	1	`0x35`		m_9	q_0	`0x35`	
12			`0x25`		$S_{0,10}$	1	`0x35`	
...								
46	$S_{7,9}$	1	`0x35`		m_9	q_8	`0x35`	
47			`0x25`		$S_{8,10}$	1	`0x35`	
48			`0x25`				`0x25`	
49	a_8	b_8	`0x5e`	$S_{8,8}$			`0x25`	
50	m_8	q_8	`0x35`		a_9	b_8	`0x5e`	$S_{8,9} \| S_{8,10}$
51	$S_{8,9}$	1	`0x35`		m_9	q_8	`0x35`	
52			`0x25`		$S_{8,10}$	1	`0x35`	
53			`0x25`				`0x25`	
54	a_8	b_9	`0x5e`	$S_{9,8}$			`0x25`	
55	m_8	q_9	`0x35`		a_9	b_9	`0x5e`	$S_{9,9} \| S_{9,10}$
56	$S_{9,9}$	1	`0x35`		m_9	q_9	`0x35`	
57			`0x25`		$S_{9,10}$	1	`0x35`	
58			`0x25`				`0x25`	
59	0	0	`0x5e`	$S_{10,8}$			`0x25`	
60	m_8	q_{10}	`0x35`		0	0	`0x5e`	$S_{10,9} \| S_{10,10}$
61	$S_{10,9}$	1	`0x35`		m_9	q_{10}	`0x35`	
62			`0x25`		$S_{10,10}$	1	`0x35`	
63			`0x25`				`0x25`	
64	0	0	`0x5e`	$S_{11,8}$			`0x25`	
65	0	0	`0x35`		0	0	`0x5e`	$S_{11,9} \| S_{11,10}$
66	0	0	`0x35`		0	0	`0x35`	

Table A.3.: Operation modes for DSP 9 and DSP 10 required by the multiplier

Port B address	Content of memory cell		Port A address
0x27			0x26
0x25			0x24
0x23			0x22
0x21			0x20
0x1F			0x1E
0x1D	z_{p_9}	z_{p_4}	0x1C
0x1B	z_{p_8}	z_{p_3}	0x1A
0x19	z_{p_7}	z_{p_2}	0x18
0x17	z_{p_6}	z_{p_1}	0x16
0x15	z_{p_5}	z_{p_0}	0x14
0x13	x_{p_9}	x_{p_4}	0x12
0x11	x_{p_8}	x_{p_3}	0x10
0x0F	x_{p_7}	x_{p_2}	0x0E
0x0D	x_{p_6}	x_{p_1}	0x0C
0x0B	x_{p_5}	x_{p_0}	0x0A
0x09			0x08
0x07			0x06
0x05			0x04
0x03			0x02
0x01			0x00

Table A.4.: Content of block RAM 6 and 7

2P		
Intermediate results	Storage block	Addresses
a_1	II	0x1E -- 0x27
s_1	II	0x28 -- 0x31
m_1	II/III	0x28 -- 0x31
m_2	II/III	0x1E -- 0x27
s_3	II	0x32 -- 0x3B
m_6	III	0x46 -- 0x4F
x_{2P}	III	0x1E -- 0x27
a_4	II	0x28 -- 0x31
z_{2P}	III	0x28 -- 0x31

Table A.5.: Addresses of intermediate results while performing $2 * P$

2P and $P \oplus Q$		
Intermediate results	Storage block	Addresses
a_1	I/II	`0x1E -- 0x27`
s_1	I/II	`0x28 -- 0x31`
a_2	I	`0x32 -- 0x3B`
s_2	I	`0x3C -- 0x45`
m_1	II/III	`0x28 -- 0x31`
m_2	II/III	`0x1E -- 0x27`
s_3	II	`0x32 -- 0x3B`
m_3	III	`0x1E -- 0x27`
m_4	III	`0x32 -- 0x3B`
a_3	I	`0x1E -- 0x27`
s_4	I	`0x28 -- 0x31`
m_6	III	`0x32 -- 0x3B`
x_{2P}	III	`0x1E -- 0x27`
a_4	II	`0x28 -- 0x31`
m_8	II	`0x1E -- 0x27`
x_{P+Q}	III	`0x32 -- 0x3B`
z_{2P}	III	`0x28 -- 0x31`
z_{P+Q}	III	`0x3C -- 0x45`

Table A.6.: Addresses of intermediate results while performing $2 * P$ and $P \oplus Q$

2Q and $P \oplus Q$		
Intermediate results	Storage block	Addresses
a_1	I/II	`0x1E -- 0x27`
s_1	I/II	`0x28 -- 0x31`
a_2	I	`0x32 -- 0x3B`
s_2	I	`0x3C -- 0x45`
m_1	II/III	`0x28 -- 0x31`
m_2	II/III	`0x1E -- 0x27`
s_3	II	`0x32 -- 0x3B`
m_3	III	`0x1E -- 0x27`
m_4	III	`0x32 -- 0x3B`
a_3	I	`0x1E -- 0x27`
s_4	I	`0x28 -- 0x31`
m_6	III	`0x1E -- 0x27`
x_{2P}	III	`0x32 -- 0x3B`
a_4	II	`0x28 -- 0x31`
m_8	II	`0x1E -- 0x27`
x_{P+Q}	III	`0x1E -- 0x27`
z_{2P}	III	`0x3C -- 0x45`
z_{P+Q}	III	`0x28 -- 0x31`

Table A.7.: Addresses of intermediate results while performing $2 * Q$ and $P \oplus Q$