RUHR-UNIVERSITÄT BOCHUM

Efficient Implementation of a Generic Coprocessor for Elliptic Curve Cryptography on Reconfigurable Hardware

Ariano-Tim Donda

Master's Thesis. October 13, 2015.
Chair for Embedded Security – Prof. Dr.-Ing. Christof Paar
Advisor: Prof. Dr.-Ing. Tim Güneysu

Acknowledgment

I would like to thank the company Rohde & Schwarz SIT GmbH to give me the opportunity to write my Master's thesis there and for the exciting topic.

My special thanks goes to Dr. Simon Hauger and Dr. Torsten Schütze, for the best advice during the work and especially for the assistance in the last weeks before deadline.

Last but not the least, I thank my fiancée Kerstin for supporting me spiritually throughout writing this thesis and my life in general.

Abstract

This project is motivated by the need for a flexible and secure ECC implementation on FPGAs at Rohde & Schwarz SIT GmbH. The coprocessor shall handle elliptic curves over $\mathbb{F}_p, p > 3$, prime with a verifiable pseudo-random prime structure, so not only NIST curves should be possible. Different bit lengths of p must be supported.

One main task is the use of DSPs for fast arithmetic computations as much as possible to save other resources on the FPGA. Particularly, the modular multiplication was examined more detailed. It is solved with the CIOS algorithm, a Montgomery multiplication algorithm design especially for hardware. The hardware design of the CIOS algorithm implemented in Mentens' PhD thesis was the basis for our design.

For secure and efficient implementation of ECC scalar multiplication a special parallelized Montgomery ladder is used which uses projective coordinates in (X, Z) representation.

The coprocessor must be protected/protectable against all kinds of Side-Channel Analysis. Montgomery ladder is used against SPA. Randomized projetive coordinates and prime randomization were supported as a countermeasure against DPA.

We implemented the basic design of Mentens and modified it in various ways: First, the 16×16 multiplier has been replaced by a 17×17 multiplier. Then the full capabilities of multipliers on modern FPGAs has been used, i. e., a 17×24 multiplier. The design and implementation produced with various optimizations regarding the number of DSPs used and the number of pipeline stages. The coprocessor designed provides the improved Montgomery multiplication by Walter, modular addition/subtraction as well as modular inversion.

The designed coprocessor is faster than Mentens' which is the most comparable design. Comparison with other implementations are not easy because of missing either countermeasures against SCA, implementations for curves over $\mathbb{F}_{2^m}$, or approaches that do not allow flexible p.

Statutory Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

I affirm that the digital version is identical to the submitted written version. I hereby agree that the digital version of this submission is used for plagiarism assessment.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Ariano-Tim Donda

Contents

1 Introduction

1.1 Motivation

The topic of this project is the efficient design and efficient implementation of a coprocessor for Elliptic Curve Cryptography (ECC) over finite fields $\mathbb{F}_p$ on Field-Programmable Gate Arrays (FPGAs). Some of the design constraints and feature requests are due to the Rohde & Schwarz SIT GmbH which require a flexible and secure ECC implementation for high-assurance applications. These requirements could not be fulfilled with general designs and IP cores on the COTS (commercial off-the-shelf) market.

Our implementation will be used as part of a larger undisclosed project. The coprocessor should be flexible regarding the bit length. It should allow various implementation techniques including countermeasures against Side-Channel Analysis (SCA) and *should not rely* on special certain structures for the prime p, for example, so-called National Institute of Standards and Technology (NIST) or Solinas primes (Generalized Mersenne primes).

Today, ECC is one of the favored schemes for asymmetric cryptography. ECC key sizes are much smaller than key sizes for traditional schemes like RSA for approximately corresponding security strengths. For example, ECC with 384 bit is roughly equivalent to RSA with 8 192 bit [HMV04].[1] While the number field sieve for factoring is the most efficient method known to attack RSA encryption, there is no known sub-exponential algorithm to attack carefully chosen elliptic curves.

Although ECC has only been established in practical cryptography in the last decade, it has been independently introduced Victor S. Miller in [Mil85] and Neal Koblitz in [Kob87].

1.2 Related Work for Fast Elliptic Curve Multipliers in Hardware

Highly performant implementations often use ECC over the finite field $\mathbb{F}_{2^m}$. However, for high-security applications, especially in Germany, the finite field $\mathbb{F}_p$, $p > 3$, prime is used. Existing work on coprocessors for ECC over $\mathbb{F}_p$ can be summarized as follows:

Orlando and Paar [OP01] designed an ECC coprocessor in $\mathbb{F}_p$. They use a high-radix Montgomery multiplier with Booth re-coding. For reduction, they pre-compute common values. Their architecture uses special prime structures (General Mersenne primes) and,

[1] ECC with 384 bit security level or even with 521 bit can be found in current smart card implementations while RSA 2 048 bit is the upper limit for these cards.

thus, seems not flexible enough for our purposes. No dedicated multipliers/Digital Signal Processing Blocks (DSPs) on the FPGA are used.

The paper [GP08] uses DSPs extensively and, therefore, achieves a very high performance. However, this work is for NIST primes only, too. Another recent paper from the Ruhr University Bochum is [SG14]. Here, the even more special curve *Curve25519* over $\mathbb{F}_p$ with $p = 2^{255} - 19$ is used.

All these approaches do not provide required flexibility.

In [Gui10] a coprocessor for general curves over $\mathbb{F}_p$ has been designed. Its arithmetic unit uses a Residue Number System (RNS) representation. Furthermore, some countermeasures against SCA are built-in.

Mentens [Men07] and Ma et al. [MLPJ13] use Montgomery multipliers to replace RNS. All these papers utilize the DSPs on modern FPGAs for fast multiplication.

Although the performance and area results reported in [Gui10] and its successors are impressive, we decided not to follow a RNS approach, because Montgomery multipliers seem more flexible with respect to varying bit length.

In particular, the PhD thesis of Nele Mentens [Men07] builds an important foundation for our work. Two efficient hardware algorithms were compared in modular multiplication using Montgomery reduction. Presented results and designs of the multipliers serve as a basis for the multipliers in this work.

1.3 Research Aims

The project aim is a flexible and secure ECC implementation on FPGAs with some required design constraints to the coprocessor architecture. The given hardware is a Xilinx 7-Series Kintex XC7K325T and generic curves over $\mathbb{F}_p$ must be supported. The coprocessor must allow efficient randomization for protection against SCA. This includes not only scalar blinding and projective coordinates, but randomization of the prime as well. Brainpool curves serve as an example for general curves with verifiable pseudo-random prime structure. To resist Simple Power Analysis (SPA) the coprocessor should support the use of a Montgomery ladder. The design must be easily modifiable with respect to bit length. In addition, for fast arithmetic DSPs must be adopted where appropriate. On the other hand, other logic resources shall be omitted.

Several different designs varying in area and speed have been implemented and tested. The designs differ in their usage of DSPs and the number of pipeline stages. The final design is comparable to related work. However, it should be noted that flexibility and security is much more important in our design than speed (and area).

A complete implementation of fully protected ECC primitives exceeds the project scope. Further steps would be necessary including a host interface connection from a soft microprocessor core in the FPGA to the coprocessor and, only then, a highly efficient and secure side-channel protected implementation of various ECC primitives and protocols.

1.4 Thesis Structure

This document comprises seven chapters:

Chapter 2 introduces to Elliptic Curve Cryptography and its arithmetics. It is shown how to calculate in groups over elliptic curves and some typical protocols which use ECC. A short description of selected curve parameters and selected SCA countermeasures is included.

Chapter 3 explains the core structure of an FPGA. For the design essential components are described in more detail.

Chapter 4 provides a short overview of the required arithmetics. Especially, techniques for modular multiplication will be considered in detail. The work of Mentens will be analyzed in more detail.

Chapter 5 starts with an overview of the coprocessor designed. First, the complete design is shown and then it is explained in a top-down approach. The focus of the coprocessor is on the Arithmetic Unit (AU) and its usage of multipliers.

In Chapter 6 the designed coprocessor is carefully analyzed and tested. To compare the modular multiplier and the arithmetic unit to existing ECC implementations, it is necessary to define a sequence of required steps for one scalar multiplication utilizing our new coprocessor.

Chapter 7 concludes the thesis, points out open problems, and denotes some further steps.

2 Elliptic Curve Cryptography

Usually, the pyramid in Figure 2.1 is used to show the different levels of ECC. The top is the use of ECC represented by the protocols and algorithms. They define how ECC must be used to avoid obvious attacks. Next level is the key operation of ECC. The complete security of ECC is based on the scalar multiplication of a point P (Discrete Logarithm Problem). It essentially consists of the both general curve operations point addition and doubling. Point addition and point doubling break down to operations in certain finite fields, especially, in our case to modular multiplications, modular additions, etc.

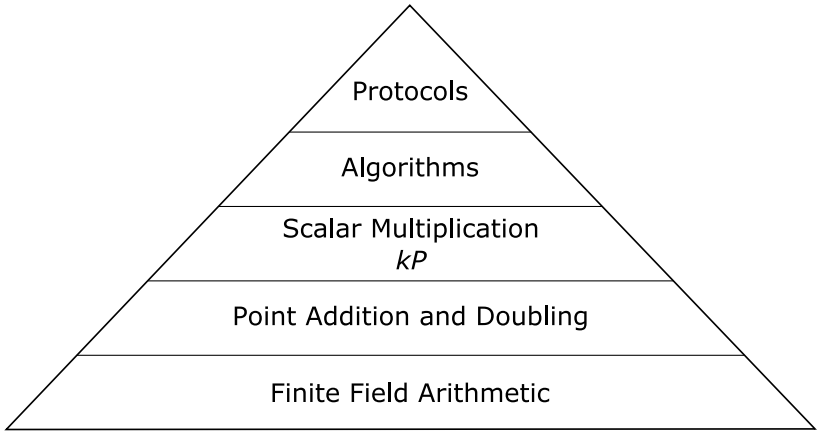

Figure 2.1: Pyramid shows the different elementary levels of using ECC.

The next sections explain the different levels of this pyramid more detailed in a bottom-up approach. Furthermore, different properties of specific curves are explained and also advantages and disadvantages are discussed.

2.1 Finite Fields

A finite field or Galois field is a set of elements with two basic operations. A finite field will be denoted by $\mathbb{F}_q$ where q is the number of elements. Before giving the definition for finite fields we first need the definition of a simpler algebraic structure, Abelian groups. The definitions are from [PP10].

Abelian group: An Abelian group is a set of elements G together with an operation $\circ$ which combines two elements of G. An Abelian group has the following properties:

1. The group operation $\circ$ is closed. It holds that $a \circ b = c \in G$ for all $a, b, c \in G$.
2. The group operation is associative. That is, $a \circ (b \circ c) = (a \circ b) \circ c$ for all $a, b, c \in G$.
3. There is an element $1 \in G$, called the neutral element (or identity element), such that $a \circ 1 = 1 \circ a = a$ for all $a \in G$.
4. For each $a \in G$ there exists an element $a^{-1} \in G$, called the inverse of a, such that $a \circ a^{-1} = a^{-1} \circ a = 1$.
5. A group G is abelian (or commutative) if, furthermore, $a \circ b = b \circ a$ for all $a, b \in G$.

Usually, Abelian groups with the operation + are called *additive Abelian group* and with $\cdot$ are called *multiplicative Abelian group*. While groups only provide one operation finite fields are sets which have two operations. So finite fields must satisfy following further requirements:

Finite field: A finite field $\mathbb{F}_q$ is a set of a finite number of q elements in combination with two operations + and $\cdot$ with following properties:

1. All elements of $\mathbb{F}$ form an additive group with the group operator + and the neutral element 0.
2. All elements of $\mathbb{F}$ form a multiplicative group with the group operation $\cdot$ and the neutral element 1.
3. When two group operations are mixed, the distributivity law holds, i. e., for all $a, b, c \in \mathbb{F}_q : a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

For cryptography, two types of finite fields are mainly used. In $\mathbb{F}_p$ the elements are integers and the order is a prime. The other type of fields is denoted by $\mathbb{F}_{2^m}$ where the elements can be represented as m-bit vectors. This kind of finite field is popular because of efficient arithmetic shortcuts. However, this work deals with a coprocessor design for curves over $\mathbb{F}_p$ with $p > 3$ and p is prime, thus, we use in the following arithmetics of finite fields over general primes.

2.2 Point Addition and Doubling

Current practically relevant public-key algorithms can be categorized into three families: First, algorithms based on integer factorization (like RSA), second, algorithms based on discrete logarithm over finite fields (like DSA), and third, algorithms based on discrete logarithm using elliptic curves (like ECDSA). ECC schemes gain more importance with stricter security requirements. The required bit length for ECC algorithms grows slower than keys for traditional schemes. In the next subsections we explain the functionality of elliptic curve schemes.

2.2.1 Affine Coordinates for Short Weierstrass Curves

The set of all pairs $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ satisfying the equation

$$y^2 = x^3 + ax + b \bmod p, \tag{2.1}$$

where $a, b \in \mathbb{F}_p$ and

$$4a^3 + 27b^2 \neq 0 \bmod p, \tag{2.2}$$

together with the special point $\mathcal{O}$ is called elliptic curve E over $\mathbb{F}_p$ with parameters a, b. The point $\mathcal{O}$ is called point at infinity. It represents the neutral element.

Equation (2.1) is called *Simplified* or *Short Weierstrass Equation*. It is a normal form, i. e., all elliptic curves over $\mathbb{F}_p$ can be represented in this form. It is also possible to define elliptic curves over other finite fields such as $\mathbb{F}_{2^m}$, $m \in \mathbb{N}$, see [Bro10, ANSI-X9.63:11, FIPS-186-4:13], or $\mathbb{F}_{p^m}$, $p > 3$, prime, $m \in \mathbb{N}$, see [ISO-15946-1:02]. However, we consider only curves over large prime fields $\mathbb{F}_p$, $p > 3$, prime are considered.

The set of points of the elliptic curve forms an additive Abelian group with the point at infinity $\mathcal{O}$ as neutral element and the following operations:

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be elements of E and $P, Q \neq \mathcal{O}$. If $x_1 = x_2$ and $y_1 = -y_2$, then $P + Q = \mathcal{O}$, otherwise $P + Q = (x_3, y_3)$ with

$$x_3 = s^2 - x_1 - x_2 \bmod p, \qquad y_3 = s(x_1 - x_3) - y_1 \bmod p, \tag{2.3a}$$

and

$$s = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q, \\ \frac{3x^2 + a}{2y} & \text{if } P = Q. \end{cases} \tag{2.3b}$$

The point at infinity $\mathcal{O}$ is the identity over E, so $P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E$. The operation of $P + Q$ with $P \neq Q$ is named *Point Addition* and if $P = Q$ then it is named *Point Doubling*. The pair $(x, y) \in E$, $x, y \in \mathbb{F}_p$ are called *affine coordinates*.

2.2.2 Projective Coordinates for Short Weierstrass Curves

The point addition defined in (2.3a) and (2.3b) can be implemented with one inversion and three multiplications in $\mathbb{F}_p$, the point doubling with one inversion and four multiplications in $\mathbb{F}_p$. In practice, the affine representation of the elements in E is rarely used, because computations with affine coordinates require many inversions, and there are many special cases ($P = -Q$, one of the points is $\mathcal{O}$) in the addition equation. Such special cases can easily be distinguished, for example, in the power trace. Special cases are prone to SCA and should be avoided.

Fortunately, there are many different equivalent representations of elliptic curves with other coordinates and better properties. Projective coordinates are valuable for secure implementations.

Simple (standard) projective or homogeneous coordinates map a point (X, Y, Z) of the projective plane to the point $(x, y) = (X/Z, Y/Z)$ of the affine plane. Other projective

coordinates which allow efficient implementations are weighted projective or Jacobian coordinates with $(\mathcal{X}, \mathcal{Y}, \mathcal{Z}) \equiv (\mathcal{X}/\mathcal{Z}^2, \mathcal{Y}/\mathcal{Z}^3)$ or Chudnovsky projective coordinates with $(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{Z}^2, \mathcal{Z}^3) \equiv (\mathcal{X}/\mathcal{Z}^2, \mathcal{Y}/\mathcal{Z}^3)$. However, Chudnovsky coordinates need more registers than simple coordinates and Jacobian coordinates are not useful for the scenario in this work, too.

There exists a Montgomery ladder for projective coordinates that requires 19 modular multiplications per bit, see [BJ02]. Later [JY02] showed that such a ladder exists for every Abelian group. Nevertheless, the Montgomery ladder for Jacobian coordinates is not as efficient as the one for projective coordinates. Hence, we focus on simple projective coordinates which are defined below.

Let $P = (\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ with $\mathcal{X}, \mathcal{Y}, \mathcal{Z} \in \mathbb{F}_p$ be a point of the elliptic curve in homogeneous coordinates. If $\mathcal{Z} \neq 0$, the projective point $P = (\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ is equivalent to the affine point $P = (x, y) = (\mathcal{X}/\mathcal{Z}, \mathcal{Y}/\mathcal{Z})$. The point $(0, 1, 0)$ corresponds to the neutral element $\mathcal{O}$. This special coordinate is defined to avoid special routines in case of $P + \mathcal{O}$.

The (non-randomized) transformation from affine to projective coordinates is trivial

$$(x, y) \rightarrow (\mathcal{X}, \mathcal{Y}, 1),$$

while the transformation from projective to affine coordinates requires one inversion and two multiplications.

The short Weierstrass equation (2.1) in projective homogeneous coordinates is given by

$$\mathcal{Y}^2\mathcal{Z} = \mathcal{X}^3 + a\mathcal{X}\mathcal{Z}^2 + b\mathcal{Z}^3 \bmod p. \tag{2.4}$$

With this representation of the points over E there are some benefits:

- Point addition requires 14 modular multiplications and point doubling 11modular multiplications, respectively, see [BL15a, CMO98].
- There are no conditional branches due to the point $\mathcal{O}$.
- There are many representations from one affine point to projective points. This constitutes a natural randomization and is one of the standard countermeasures against SCA such as Differential Power Analysis (DPA) and Timing Analysis (TA) [Cor99].

2.2.3 Reduced Projective Coordinates for Short Weierstrass Curves

In affine coordinates the y-value only represents the sign of the coordinate, its security weight is therefore only one bit. Clearly, it is beneficial to keep any costs small for only one bit. For affine coordinates, this leads naturally to so-called compressed affine points. However, we aim to eliminate $\mathcal{Y}$ in the homogeneous coordinate $(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ from the preceding subsection as well. In the following, there is the derivation of Brier and Joye [BJ02] of the addition of two points P and Q using only the $\mathcal{X}$ and $\mathcal{Z}$ coordinates. Note that the special coordinates and algorithms of point addition and doubling are only used in context with the Montgomery Ladder, see Section 2.3.3.

For $P, Q \in E(\mathbb{F}_p)$ let us introduce the notation

$$\begin{aligned} P = (x_1, y_1) &\equiv (X_1, Y_1, Z_1), & Q = (x_2, y_2) &\equiv (X_2, Y_2, Z_2), \\ P + Q = (x_3, y_3) &\equiv (X_3, Y_3, Z_3), & P - Q = (x_4, y_4) &\equiv (X_4, Y_4, Z_4), \text{ and} \\ 2P = (x_5, y_5) &\equiv (X_5, Y_5, Z_5). \end{aligned}$$

It follows from (2.3a) for $P \neq \pm Q$

$$x_3 = \left(\frac{y_1 - y_2}{x_1 - x_2}\right)^2 - x_1 - x_2, \tag{2.5a}$$

and

$$x_3 = \left(\frac{y_1 - y_2}{x_1 - x_2}\right)^2 - x_1 - x_2 \tag{2.5b}$$

Combination of (2.5a) and (2.5b) gives the following equation for point addition (see [BJ02]):

$$(x_3 + x_4)(x_1 - x_2)^2 = 2(x_1 + x_2)(x_1 x_2 + a) + 4b. \tag{2.6}$$

For x_5 with $P \neq \mathscr{O}$ it follows

$$x_5 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1. \tag{2.7}$$

By substitution of $2y_1$ with (2.4) it follows

$$4x_5(x_1^3 + ax_1 + b) = (x_1^2 - a)^2 - 8bx_1. \tag{2.8}$$

The next step is to switch the representation from affine to projective coordinates and obtain an equation for calculating X_3, Z_3 and X_5, Z_5 using the projective coordinates X_1, Z_1, X_2, Z_2. Let $x_i = {}^{X_i}/_{Z_i}, i = \{1, \ldots, 5\}$. For x_3 follows

$$\begin{aligned} (x_3 + x_4)\left(\frac{X_1}{Z_1} - \frac{X_2}{Z_2}\right)^2 &= 2\left(\frac{X_1}{Z_1} - \frac{X_2}{Z_2}\right)\left(\frac{X_1 X_2}{Z_1 Z_2} + a\right) + 4b, \\ (x_3 + x_4)(X_1 Z_2 - X_2 Z_1)^2 &= 2(X_1 Z_2 + X_2 Z_1)(X_1 X_2 + a Z_1 Z_2) + 4b Z_1^2 Z_2^2, \end{aligned}$$

and finally

$$\begin{aligned} x_3(X_1 Z_2 - X_2 Z_1)^2 = 2(X_1 Z_2 + X_2 Z_1)(X_1 X_2 + a Z_1 Z_2) \\ + 4b Z_1^2 Z_2^2 - x_4(X_1 Z_2 - X_2 Z_1)^2. \end{aligned} \tag{2.9}$$

These reduced projective coordinates are only usable for scalar multiplication kP with a special Montgomery ladder, which is explained later in Section 4.1. Therefore ${}^{X_1}/_{Z_1}$ and ${}^{X_2}/_{Z_2}$ are always the representation of the affine x-coordinate of two points. The

difference of these two points is $P \not\equiv \mathscr{O}$. Without loss of generality we set $\mathcal{Z}_4 = 1$. This assumption simplifies the equations and saves one multiplication. It follows

$$\mathcal{X}_3 = 2\left(\mathcal{X}_1\mathcal{Z}_2 + \mathcal{X}_2\mathcal{Z}_1\right)\left(\mathcal{X}_1\mathcal{X}_2 + a\mathcal{Z}_1\mathcal{Z}_2\right) + 4b\mathcal{Z}_1^2\mathcal{Z}_2^2 - \mathcal{X}_4\left(\mathcal{X}_1\mathcal{Z}_2 - \mathcal{X}_2\mathcal{Z}_1\right)^2 \tag{2.10a}$$

and

$$\mathcal{Z}_3 = \left(\mathcal{X}_1\mathcal{Z}_2 - \mathcal{X}_2\mathcal{Z}_1\right)^2. \tag{2.10b}$$

For the point addition (2.10) 10 modular multiplications are required.

For χ_5 follows

$$4\chi_5\left(\mathcal{X}_1^3\mathcal{Z}_1 + a\mathcal{X}_1\mathcal{Z}_1^3 + b\mathcal{Z}_1^4\right) = \left(\mathcal{X}_1^2 - a\mathcal{Z}_1^2\right)^2 - 8b\mathcal{X}_1\mathcal{Z}_1^3$$

and the following equations for calculating the resulting $\mathcal{X}$ and $\mathcal{Z}$

$$\mathcal{X}_5 = \left(\mathcal{X}_1^2 - a\mathcal{Z}_1^2\right)^2 - 8b\mathcal{X}_1\mathcal{Z}_1^3 \tag{2.11a}$$

and

$$\mathcal{Z}_5 = 4\mathcal{X}_1\mathcal{Z}_1\left(\mathcal{X}_1^2 + a\mathcal{Z}_1^2\right) + 4b\mathcal{Z}_1^4. \tag{2.11b}$$

For the point doubling (2.11) 9 modular multiplications are required.

Often affine coordinates are given as input and required as output. Moreover, it is well known that projective coordinates leak information, see [NSS04]. It is necessary to convert back to affine coordinates after finishing the scalar multiplication kP.

The back transformation of χ_{kP} is equal to simple projective representation. If $\mathcal{Z}_{kP} = 0$, then $kP = \mathscr{O}$. Otherwise it holds

$$\chi_{kP} = \frac{\mathcal{X}_{kP}}{\mathcal{Z}_{kP}} \tag{2.12}$$

which can be calculated with one inversion and one multiplication.

For back transformation of y_{kP} it is assumed that $P = (\chi_1, y_1)$, $\chi_{kP} = {}^{\mathcal{X}_{kP}}/_{\mathcal{Z}_{kP}}$, and $\chi_{(k+1)P} = {}^{\mathcal{X}_{(k+1)P}}/_{\mathcal{Z}_{(k+1)P}}$. With equation (2.3a) it follows for $(k+1)P = kP + P$

$$\chi_{(k+1)P} = \left(\frac{y_{kP} - y_1}{\chi_{kP} - \chi_1}\right)^2 - \chi_{kP} - \chi_1. \tag{2.13}$$

After substitution of the terms y_{kP}^2 and y_1^2 with curve equation (2.1) it follows

$$\begin{aligned}\chi_{(k+1)P}\left(\chi_{kP} - \chi_1\right)^2 &= y_{kP}^2 - 2y_{kP}y_1 + y_1^2 - \left(\chi_{kP} + \chi_1\right)\left(\chi_{kP} - \chi_1\right), \\ &= \chi_{kP}^3 + a\chi_{kP} + b - 2y_{kP}y_1 + \chi_1^3 + a\chi_1 + b - \left(\chi_{kP}^2 - \chi_1^2\right).\end{aligned}$$

Thus,

$$2y_{kP}y_1 = 2b - \chi_{(k+1)P}\left(\chi_{kP} - \chi_1\right)^2 + \left(\chi_1\chi_{kP} + a\right)\left(\chi_1 + \chi_{kP}\right),$$

and finally in projective representation

$$y_{kP} = \frac{2bZ_{kP}^2 Z_{(k+1)P} + Z_{(k+1)P}\left(X_1 X_{kP} + aZ_{kP}\right) - X_{(k+1)P}\left(X_{kP} - X_1 Z_{kP}\right)^2}{2y_1 Z_{kP}^2 Z_{(k+1)P}}. \quad (2.14)$$

Hence with both equations (2.12) and (2.14) it is possible to transform the results of the Montgomery ladder (X_{kP}, Z_{kP}), $(X_{(k+1)P}, Z_{(k+1)P})$ (in projective (X,Z)-form) back to an affine one (x_{kP}, y_{kP}) without loosing any information. The back transformation requires expensive inversions and multiplications. However, this step must be performed only once after a scalar multiplication.

2.2.4 Comparison of the Shown ECC Representations

Every one of the three explained representations of points on elliptic curves has its benefits. Often affine representation is given because projective coordinates leak information [NSS04]. However, the computation in affine representation is expensive because of the required inversion in each point addition and point doubling. Therefore, the projective representation is preferred due to the inversion is only required for back transformation.

The different costs for point addition and doubling are listed in Table 2.1. The costs for projective coordinates are taken from [BL15a, CMO98]. Costs for affine coordinates are taken from equation (2.3), costs for reduced projective coordinates are taken from equations (2.10) and (2.11). I represents the required inversion and M the multiplication respectively. Note that costs for addition/subtraction are neglected because they are not time-consuming operations.

Table 2.1: Overview of costs of point addition and doubling in different coordinate representations.

	Coordinates		
Operation	Affine	Projective (X, Y, Z)	Reduced Projective (X, Z)
Point Addition	$1I + 3M$	$14\ M$	$10\ M$
Point Doubling	$1I + 4M$	$11\ M$	$9\ M$

Both projective representations need more multiplications than the affine representation. But one inversion is much more expensive than one multiplication. Hence, the computation in projective representation should be preferred. It is obvious that the calculation with reduced projective representation is more efficient than with simple projective representation. Due to the fact that for reduced projective coordinates the Montgomery ladder given by [FGKS02] must be used, it implies a countermeasure against TA and SPA. This fact is shown later in this work. Therefore, we use the reduced projective representation for computation.

2.3 Scalar Multiplication

Cryptography using elliptic curves is based on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP). It is the main security criteria for ECC and describes the problem to find the factor $d \in \mathbb{N}$ for the given points $P, Q \in E(\mathbb{F}_p)$ with $Q = dP$.

The simplest way to compute the scalar multiplication or point multiplication dP is to add d times the point P. In the following subsections we are providing more efficient algorithms to perform scalar multiplication.

2.3.1 Double-and-Add

Similar to exponentiation in multiplicative groups it is possible to compute the scalar multiplication similar to the square-and-multiply algorithm. Instead of squaring we double points and instead of multiplication we use point addition. Algorithm 2.3.1 presents the Double-and-Add algorithm for scalar multiplication.

Algorithm 2.3.1: Double-and-Add for Scalar Multiplication

Input: elliptic curve E with an elliptic curve point P and a scalar $d = \sum_{i=0}^{n-1} d_i 2^i$ with $d_i \in \{0, 1\}$ and $d_{n-1} = 1$
Output: $T = dP$

```
1 T ← P
2 for i = n − 1 downto 0 do
3 |   T ← T + T
4 |   if d_i = 1 then
5 |   |   T ← T + P
6 return T
```

The algorithm goes through every bit of d. If the bit d_i is 0 point T will be doubled only. Otherwise, if d_i is 1 T will be added with P additionally. It requires n point doublings and on average $n/2$ point additions, where n is the length of scalar d.

2.3.2 Double-and-Add-Always

A SPA is an efficient attack on Algorithm 2.3.1. Due to the increased time complexity of point doubling with addition than point doubling without addition, one can simply extract the key from the power consumption trace. The power consumption should be independent of the current key.

A simple solution to achieve a uniform power consumption is to always double and add for scalar multiplication. This algorithm shown as Algorithm 2.3.2 was already suggested by Coron [Cor99].

One draw back of Algorithm 2.3.2 is that the branch in line 5 may be still extractable in a power trace. Algorithm 2.3.2 requires n point doublings and n point additions.

Algorithm 2.3.2: Double-and-Add-Always for Scalar Multiplication

Input: elliptic curve E with an elliptic curve point P and a scalar $d = \sum_{i=0}^{n-1} d_i 2^i$ with $d_i \in \{0, 1\}$ and $d_{n-1} = 1$
Output: $T = dP$

```
1 T_0 ← P
2 for i = n − 1 downto 0 do
3 |   T_0 ← T_0 + T_0
4 |   T_1 ← T_0 + P
5 |   if d_i = 1 then
6 |   |   T_0 ← T_1
7 return T_0
```

2.3.3 Montgomery Ladder

A better way to circumvent the timing and SPA issues is the use of a Montgomery Ladder as shown in Algorithm 2.3.3.

Algorithm 2.3.3: Montgomery Ladder for Scalar Multiplication

Input: elliptic curve E with an elliptic curve point P and a scalar $d = \sum_{i=0}^{n-1} d_i 2^i$ with $d_i \in \{0, 1\}$ and $d_{n-1} = 1$
Output: $T = dP$

```
1  T_0 ← O
2  T_1 ← P
3  for i = n − 1 downto 0 do
4  |   if d_i = 1 then
5  |   |   T_0 ← T_0 + T_1
6  |   |   T_1 ← T_1 + T_1
7  |   else
8  |   |   T_1 ← T_0 + T_1
9  |   |   T_0 ← T_0 + T_0
10 return T_0
```

At every iteration, we have the same operations (double and add) all results are used without temporary results and only the registers change. So the power trace is uniformly distributed.

Originally, the Montgomery ladder was introduced by Montgomery [Mon87] for so-called Montgomery curves, a special class of elliptic curves. The Montgomery ladder for general short Weierstrass curves is given by [BJ02].

2.4 Cryptographic Protocols and Algorithms

Many protocols like Transport Layer Security (TLS), IPSec, and Vehicle-to-Vehicle communication use cryptography to increase their security in various ways. TLS and IPSec currently use Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signatures and Elliptic Curve Diffie Hellman (ECDH) for authentication purposes. The Vehicle-to-Vehicle standard IEEE P1609.2 uses digital signatures (ECDSA) and hybrid encryption Elliptic Curve Integrated Encryption Scheme (ECIES). In this section three common protocols and algorithms that use ECC are explained.

In case of ECC public parameters have to be known by every participant of the communication protocol. These are the domain parameters $D = \{p, a, b, G, n, h\}$ with

- p is the order of the finite field $\mathbb{F}_p$,
- a and b are the two coefficients that define the elliptic curve E in short Weierstrass form,
- G is the base point which generates a cyclic group $\langle G \rangle$,
- n is the order of the base point G, i. e., the order of the group $\langle G \rangle$, and
- h defines the cofactor with $h = {}^{\#E(\mathbb{F}_p)}/_{n}$, $\#E$ denotes number of points on E.

In special cases, additional parameters that describe the generation process of the elliptic curve can be added to the domain parameters. More restrictions and properties for selected curve parameters are described in Section 2.5.

2.4.1 Elliptic Curve Digital Signature Algorithm (ECDSA)

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a special variant of Digital Signature Algorithm (DSA) which uses ECC. It is standardized, for example, in [FIPS-186-4:13] and [ANSI-X9.62:98].

In ECDSA a key pair consists of a private key d and a public key Q with $Q = dG$ and $d \in_R \{1, \ldots, n-1\}$. After generation of the key pair (d, Q) it is required that the point Q is not the point of infinity. Further, the coordinate values $x_Q, y_Q \in [0, p-1]$ and the point $Q = (x_Q, y_Q)$ must be a point on the curve E.

Algorithm 2.4.1 shows the ECDSA signature generation algorithm and Algorithm 2.4.2 shows the ECDSA signature verification algorithm where H is a cryptographic hash function with appropriate output length and π is a function to convert the x-coordinate to an integer. For signature generation k will be chosen randomly and the point kG will be computed. The x-coordinate of kG is applied as reference value r for the signature. For validation also k must be known. Therefore, k is hidden in s with reference to the private key d. Thus, it was ensured that the signature only can be verified with the corresponding public key Q. The resulting signature is (r, s).

For signature validation the point kG must be computed with s and the public key Q. If all verifications are successful generated $\pi(x_R)$ must be compared with r. If it is equally the algorithm returns *ACCEPT* and *REJECT* otherwise.

Algorithm 2.4.1: ECDSA signature generation

Input: Domain Parameters $D = \{p, a, b, G, n, h\}$, private key d, message m
Output: Signature (r, s)

1 $k \in_R [1, n-1]$
2 $(x_1, y_1) \leftarrow kG$
3 $r \leftarrow \pi(x_1) \bmod n$
4 **if** $r = 0$ **then**
5 go to line 1
6 $s \leftarrow k^{-1}(H(m) + dr) \bmod n$
7 **if** $s = 0$ **then**
8 go to line 1
9 **return** (r, s)

Algorithm 2.4.2: ECDSA signature verification

Input: Domain Parameters $D = \{p, a, b, G, n, h\}$, public key Q, message m, signature (r, s)
Output: Acceptance or rejection of the signature

1 **if** $r, s \notin [1, n-1]$ **then**
2 **return** *REJECT*
3 $w \leftarrow s^{-1} \bmod n$
4 $u_1 \leftarrow H(m)w \bmod n$
5 $u_2 \leftarrow rw \bmod n$
6 $R \leftarrow u_1 G + u_2 Q$
7 **if** $R = \mathcal{O}$ **then**
8 **return** *REJECT*
9 $v \leftarrow \pi(x_R) \bmod n$
10 **if** $v = r$ **then**
11 **return** *ACCEPT*
12 **else**
13 **return** *REJECT*

Note that ECDSA signature generation requires one scalar multiplication with the ephemeral secret scalar k while ECDSA signature verification requires two scalar multiplications or one multi-scalar multiplication with non-secret input.

2.4.2 Elliptic Curve Diffie Hellman (ECDH)

Another application of ECC is the key agreement with Elliptic Curve Diffie Hellman (ECDH). ECDH is a mechanism to establish a shared secret key between two entities. The basic form of Diffie-Hellman key exchange was published by Whitfield Diffie and Martin Hellman in [DH76]. This classical variant, i. e. unauthenticated Diffie-Hellman, is vulnerable against man-in-the-middle attacks. We present the Station-to-Station (STS) protocol, a variant of the Diffie-Hellman key exchange that provides forward secrecy using an ephemeral authenticated ECDH. The method presented was originally introduced in [ANSI-X9.63:11].

Protocol 2.2 constitutes a key agreement protocol. In the following, D are elliptic curve domain parameters, MAC_k is a Message Authentication Code (MAC) algorithm and SIGN_k is the signature generation algorithm. The protocol terminates with failure if any verification fails.

Alice calculates a random point R_A and sends it along to Bob with her identity ID_A. Bob computes an ephemeral random point R_B and is able to generate the shared secret point Z. Then, Bob computes the two keys k_1 and k_2 with key derivation function KDF and the x-coordinate of Z. Besides, he creates a signature s_B with his private key $privKey_B$ and a MAC t_B with secret key k_1. Finally, Bob sends ID_B, R_B, s_B, and t_B to Alice. Now, Alice generates k_1 and k_2, too. She verifies s_B and t_B and sends her signature s_A and MAC t_A to Bob for verification.

After running this protocol Alice and Bob agreed on a shared secret Z, which is derived from both ephemeral public keys R_A and R_B. The complete agreement is authenticated with secret key k_1. The resulting session key is k_2. Each party requires two scalar multiplications for one mutual key agreement using ECDH STS.

2.4.3 Elliptic Curve Integrated Encryption Scheme (ECIES)

The Elliptic Curve Integrated Encryption Scheme (ECIES) is a variant of the ElGamal public-key encryption scheme and has been standardized in [ANSI-X9.63:11] and [ISO-18033-2:06].

Similar to ECDSA one key pair (d, Q) is required with $Q = dG$ and $d \in_R \{1, \ldots, n-1\}$. The key pair shall be verified in the same way as in ECDSA. It is recommended to use a key pair generated for ECIES encryption and decryption only. If ECIES and ECDSA are used, each requires an independent key pair.

ECIES will be used for secure encryption with MAC for integrity assurance. The real bulk encryption is a symmetric-key encryption scheme. Both, ECIES encryption and decryption, are shown in the following algorithms with the following cryptographic primitives: KDF is a key derivation function constructed by a hash function H, ENC and DEC are encryption and decryption functions for symmetric-key encryption scheme,

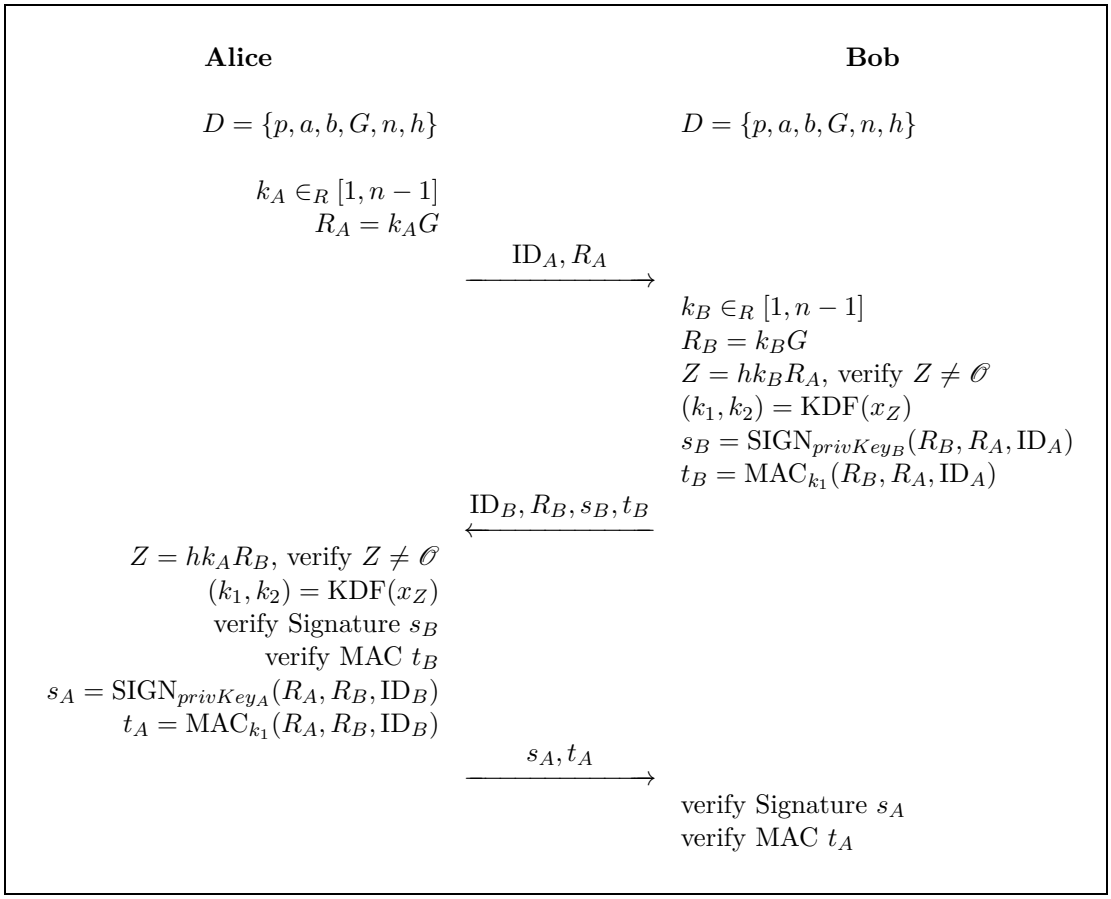

Figure 2.2: ECDH Station-to-Station Protocol.

and MAC is a message authentication code algorithm. With a static public key Q and an ephemeral public key R the KDF derives two keys, k_1 for encryption and decryption and k_2 for MAC. The complete encrypted message consists of the ephemeral public key R, the ciphertext C, and the MAC t. With the own private key d, the ephemeral public key R, and the KDF the receiver is able to derive the keys k_1 and k_2. If the check of the MAC t succeeds, the ciphertext can be decrypted.

Algorithm 2.4.3: ECIES ENCRYPTION

Input: Domain Parameters $D = \{p, a, b, G, n, h\}$, public key Q, plaintext m
Output: Ciphertext (R, C, t)

1 $k \in_R [1, n-1]$
2 $R = kG$ and $Z = hkQ$
3 $(k_1, k_2) = \mathrm{KDF}(x_Z, R)$
4 $C = \mathrm{ENC}_{k_1}(m)$
5 $t = \mathrm{MAC}_{k_2}(C)$
6 **return** (R, C, t)

Algorithm 2.4.4: ECIES DECRYPTION

Input: Domain Parameters $D = \{p, a, b, G, n, h\}$, private key d, ciphertext (R, C, t)
Output: Plaintext m or rejection of ciphertext

1 **if** *R is not valid point of E* **then**
2 **return** *REJECT*
3 $Z = hdR$, $Z \neq \mathcal{O}$
4 **if** $Z = \mathcal{O}$ **then**
5 **return** *REJECT*
6 $(k_1, k_2) = \mathrm{KDF}(x_Z, R)$
7 $t' = \mathrm{MAC}_{k_2}(C)$
8 **if** $t' \neq t$ **then**
9 **return** *REJECT*
10 $m = \mathrm{DEC}_{k_1}(C)$
11 **return** m

Two scalar multiplications are needed for encryption. For the decryption only one scalar multiplication is required because the ephemeral public key R is already given.

2.5 Selected curve parameters

The security of ECC is based on the difficulty of solving the ECDLP. For specific elliptic curves (anomalous, singular, etc.) this problem is too easy to solve. Careful selection of ECC parameters is therefore very important. More recently there has been

discussion about the (in)security of National Institute of Standards and Technology (NIST) and Brainpool curves, see [BCC+14] and [BL15b]. Lochter et al. [LMSS14, LMSS15] discuss requirements for high-assurance and hardware-based ECC.

Currently, various groups like IRTF/CFRG, IETF/TLS WG and W3C/Web Crypto WG are actively discussing the standardization of new curve parameters. One goal is to define curves that are generated by a verifiable, trustworthy process. This is motivated by a loss of confidence in NIST curves. NIST has been the first organization that standardized elliptic curve algorithms for cryptography and parameters for selected curves. Today there are no weaknesses of the NIST curves publically known, but the origin of some constants (seeds for hash input) is not justified [LMSS15].

Another issue of the current selection process is the balance between the competing goals: security, performance, and flexibility. It is desirable to have fast ECC algorithms, but there it is necessary to find a balance between security and performance. In our work we consider the scenario of hostile environments. In such scenarios an "adversary has full access to the implementation and all potential side-channels." [LMSS15] The authors rank security and flexibility over performance.

One important lesson from the current discussion is that countermeasures against SCA must be included into the selection process. Many other groups defined curves for very fast computing due to special algorithms and primes like Generalized Mersenne primes (e. g. for NIST curves). This approach has at least two drawbacks. Firstly, implementations are often restricted to specific primes, i. e., they cannot handle other curves, especially curves with pseudo-random prime structure as e. g. Brainpool curves. Secondly, randomization countermeasures against SCA do not work at all or at least not as expected (e. g. larger blinding factors required compared to random primes, see [SW14]).

In addition to the verifiable generation process of curve parameters, there are some other criteria for generating secure and useful curves. Often the algorithms used for explanation are given in affine coordinates. However, in practice affine coordinates are rarely used for efficient and secure implementations. Even if the curve is defined in short Weierstrass form using affine coordinates, the internal computation uses projective coordinates as $(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ and $(\mathcal{X}, \mathcal{Z})$ [LMSS15].

We assume that the curves used are 'Brainpool-like', i. e., they have the properties given in [Loc05]. Specifically, we assume:

- Curves are given in short Weierstrass form

$$y^2 = x^3 + ax + b \bmod p,$$

- Verifiably pseudo-random prime p,
- cofactor $h = 1$ can be assumed,
- b is a non-square mod p, and
- $p \equiv 3 \bmod 4$ such that square root is easy to compute (point decompression).

The following Brainpool conditions may be fulfilled, but are not guaranteed

- bit length of $n = \#E(\mathbb{F}_p) <$ bit length of p [Loc05],
- $a \equiv -3 \bmod p$ (would allow some performance improvements), and
- twist security.

We are interested in ECC bit lengths of 256 bit, 384 bit, 512 bit, and 521 bit.

Although these requirements seem to be very restrictive, it should be noted that they are common within the high-assurance community, see e. g. [FPRE15], and that the word *generic* is justified by the possibility to compute on general short Weierstrass curves with pseudo-random prime p.

2.6 Selected Side-Channel Analysis Countermeasures

A possible Side-Channel Analysis (SCA) is to analyze the power trace of a cryptographic component. Almost all unprotected implementations for ECC are vulnerable against power analysis. A distinction is made between Simple Power Analysis (SPA) and Differential Power Analysis (DPA). A simple method to avoid SPA is to use a Montgomery Ladder for scalar multiplication. Coron introduced countermeasures against DPA [Cor99]. We describe the important countermeasures in the following subsections.

2.6.1 Timing Analysis

Timing Analysis was one of the first Side-Channel Analysis discovered, see [Koc96]. It exploits run time variations which depend on secret data. Therefore, one of the first countermeasures is to make the run time of an algorithm constant or more special, independent from secret keys. In hardware, this seems relatively easy. Another countermeasure against timing analysis is to add entropy, i. e., to randomize the data, so that the run time varies from invocation to invocation.

Schindler [Sch00] published an attack against RSA with CRT which uses Montgomery multiplication. It exploited the fact of varying run time in Montgomery multiplication due to the final reduction, see Section 4.2. This theoretical attack was implemented by Brumley and Boneh [BB03].

For us, this means usage of the improved Montgomery multiplication (Walter's trick), see Algorithm 4.2.3, usage of a uniform Ladder, see next Section, implementation of time constant formulas and, of course, randomization.

2.6.2 Montgomery Ladder

An SPA allows an attacker to easily extract the private key by inspecting a power trace of a device as shown in Figure 2.3. Usually, the scalar multiplication routine must differ between 1 and 0 of the scalar k for computation of kP. Exactly these variations make the characteristics in a power trace as shown in Figure 2.3. For example, a Montgomery

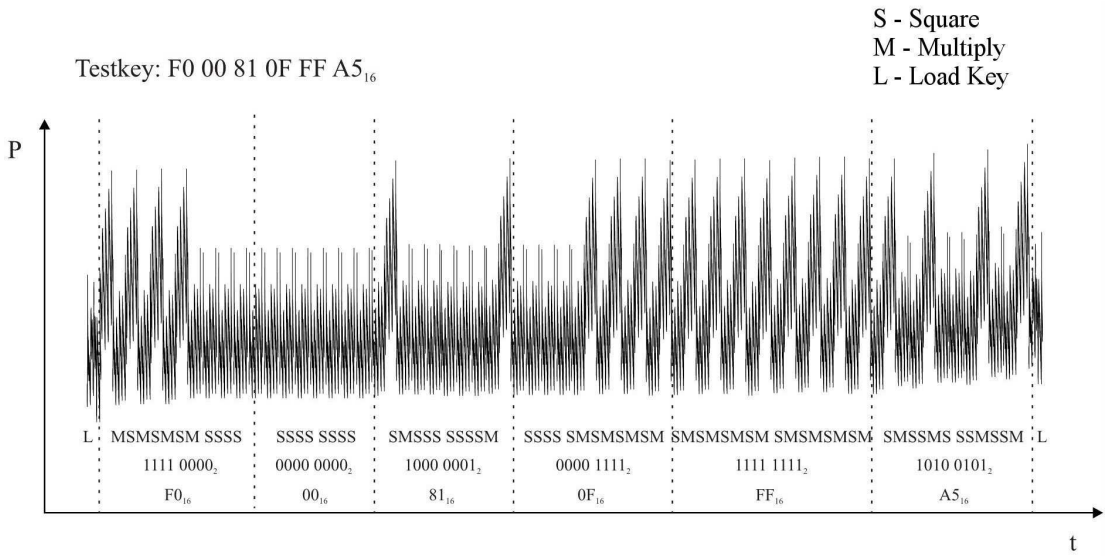

Figure 2.3: SPA attack against the RSA algorithm [Paa13].

Ladder can be used to hide these variations [Wal99]. In a Montgomery ladder every step is similar, no matter if the scalar bit is 0 or 1 (see Section 2.3.3).

2.6.3 Scalar Blinding

Coron suggests in [Cor99] to randomize the scalar in the point multiplication process. The computation of $Q = dP$ works as follows:

1. Select blinding factor $k \in_R \mathbb{N}$ (in practice k should be at least 20 bit of [Cor99], or 32 to 64 bit (current implementations))
2. Compute $d' = d + k \cdot n$, where n is the order of the group $\langle G \rangle >$
3. $Q = d'P$

It is important to change the blinding factor at each new execution of the algorithm.

2.6.4 Randomized Projective Coordinates

One further countermeasures suggested by Coron [Cor99] is the application of random projective coordinates. Computation with projective coordinates can be faster and more efficient compared to affine coordinates. Furthermore, the representation can be used for randomization of the original coordinates. Recall, that the projective point $(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ of an affine point $P = (x, y)$ is given by $P = (x, y) = (\mathcal{X}/\mathcal{Z}, \mathcal{Y}/\mathcal{Z})$. But the projective representation is not unique because

$$(\mathcal{X}, \mathcal{Y}, \mathcal{Z}) = (\lambda\mathcal{X}, \lambda\mathcal{Y}, \lambda\mathcal{Z}) \tag{2.15}$$

for $\lambda \neq 0$.

So for random $\lambda \neq \{0, 1\}$ the projective coordinate of P is a randomization of the affine representation. If required, it is also possible to randomize after every point addition and doubling with a new λ.

2.6.5 Prime Randomization

For some high security requirements it may be necessary to hide the public domain parameters in particular the prime p. A mechanism to achieve such secrecy, which may also be useful for SCA protection, is the randomization of the prime of the base field.

In some smart card coprocessors, e. g. the Crypto2000 coprocessor from Infineon, the modulus has to be transformed to a special structure before any modular operation can be performed.[1] Here, it is natural to randomize the modulus in this step as well [Sch15].

A normal modular multiplication of two numbers works like

$$Z := M \cdot C \bmod N. \tag{2.16}$$

For modulus randomization a random $\mu \in_R \mathbb{N}$ (in practice 32 bit) is selectable and multiplied by N so that

$$N' := N \cdot \mu. \tag{2.17}$$

In practice, the factor μ consists of another non-random factor to transform the modulus N to the desired form N'. The intention is now to compute in larger rings and the original group is hidden for potential attackers. To calculate Z the following two steps are necessary:

$$Z' := M \cdot C \bmod N' \tag{2.18}$$

$$Z := Z' \bmod N \tag{2.19}$$

This countermeasure prevents attacks to infer information on the prime p, e. g. DPA.

[1] The transformed modulus has to start with $011000 \ldots 0xx$ with at least eight zeros following 011.

3 Technical Background

This chapter explains the basics of dedicated reconfigurable hardware used in cryptography. In addition, the given Xilinx 7-Series Kintex XC7K325T FPGA will be considered in detail along with some general information about this family of FPGAs. Because using only general logic resources would be very expensive it makes sense to use specialized units of the FPGA. We present Configurable Logic Blocks (CLBs), DSPs, and Block-RAM (BRAM) in this chapter.

3.1 Motivation and Basics of FPGAs

Cryptographic procedures are time-consuming, because of their intensive computing. Therefore, cryptography is often implemented in dedicated cryptographic hardware. This specialized hardware is developed for high speed applications, in order to avoid cryptography being the bottleneck. Since this hardware does not access to external components directly, most attacks are prevented, aside from very costly and labor-intensive attacks with physical access to these hardware units like microprobing, focused ion beam, or laser cutter.

There are several types for integrated hardware circuits like full-custom static designs or designs on Application Specific Integrated Circuits (ASICs). In this work we focus on Field-Programmable Gate Arrays (FPGAs). Earlier FPGAs consisted of simple multiplexer circuits which were connected to simple logic elements. Modern FPGAs provide many CLBs consisting of slices with Look-Up Tables (LUTs) and Flip-Flops (FFs).

The available hardware is a Xilinx 7-Series Kintex XC7K325T FPGA. This type of FPGA provides 50 950 slices. In addition to slices, FPGAs provide further components for special tasks. Examples for these components are Block-RAM (BRAM), DSP, and Gigabit Transceiver (GTP). The first two components are described in the following subsections. GTPs handle big amount of data with special high speed interfaces like Gigabit Ethernet and PCI Express. These are not required in this work [Xil14c].

3.2 Configurable Logic Blocks and Slices

Configurable Logic Blocks (CLBs) are the most important hardware resources of an FPGA. The sub-elements of CLBs are called slices. These slices provides the main logic resources on the FPGA for the implementation of combinatorial circuits. Slices in the same CLB are connected by local routing, for general routing to the connection of other resources a switch matrix provides this access. In the available hardware of Xilinx

7-Series, there are 25 475 CLBs, with two slices per CLB. All information in this section is from [Xil14b].

A slice can be selected and routed independently the other slice in the CLB. Therefore, in practice, resource consumption is given in number of slices. Modern slices provide Look-Up Tables (LUTs), Flip-Flops (FFs), and carry logic. These elements enable advanced capabilities like distributed RAM, shift registers, or multiplexers. In FPGAs of the 7-Series of Xilinx each slice includes, inter alia, 4 6-input LUTs and 8 FFs.

Xilinx 7-Series provides two types of slices, SLICEM and SLICEL. The only difference is that SLICEM supports distributed memory, while SLICEL does not. Figure 3.1 shows a simplified diagram of a SLICEL with the elements LUT, carry logic (CL), and Flip-Flop (FF).

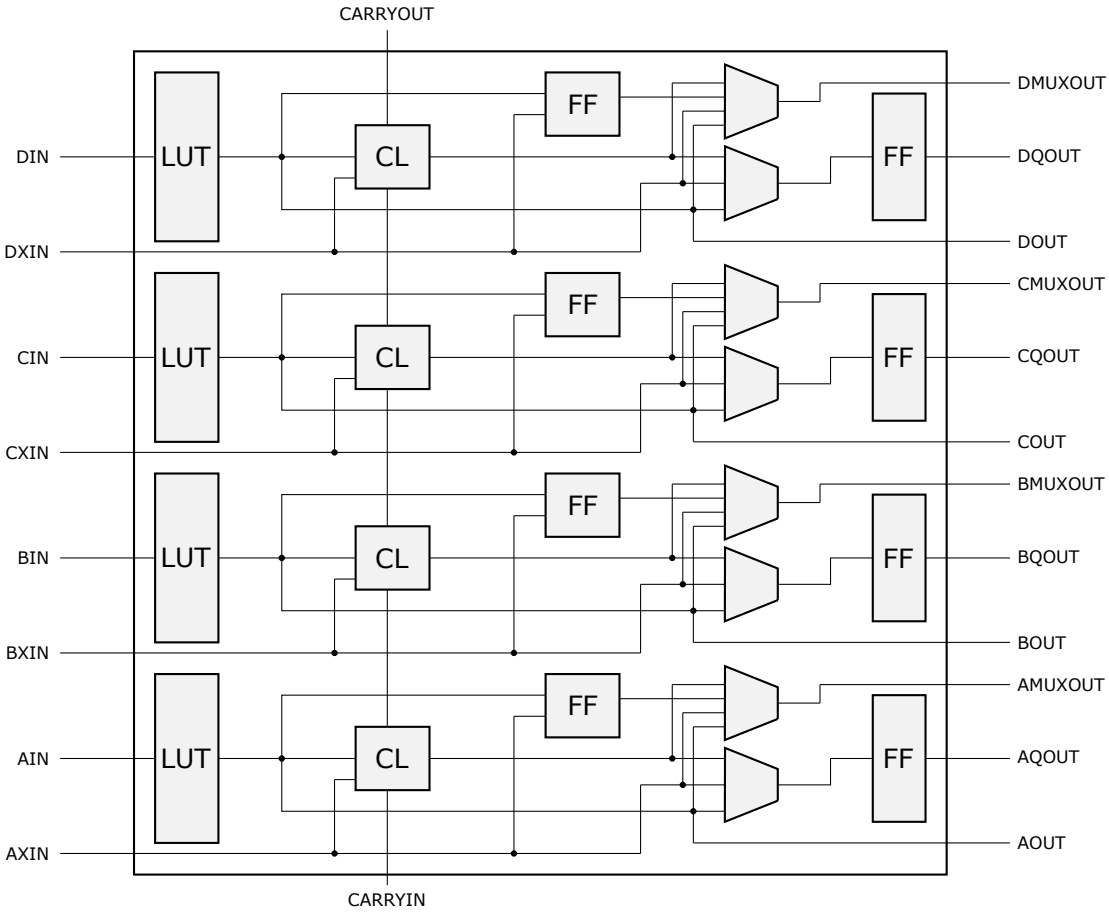

Figure 3.1: Simplified diagram of a SLICEL in XILINX 7-Series.

3.3 Digital Signal Processing Blocks

An essential task of FPGAs is digital signal processing. Especially for this application area, separated processing units are included. As mentioned in Chapter 1, one goal of this work is to use these DSPs efficiently. All information in this section is taken from [Xil14a].

In 7-Series FPGAs of Xilinx DSPs are called DSP48E1. These slices are used for efficient processing of digital signals. Many custom and fully parallel algorithms use binary multipliers and accumulators. To speed up these applications and to safe normal CLB slices DSPs should be used. Furthermore, DSPs consume less power compared to the same computation using CLBs. The XC7K325T offers 840 DSPs.

The basic construction of a single DSP48E1 slice is shown in Figure 3.2. Four input ports of different widths, A with 30 bit, B with 18 bit, C with 48 bit, and D with 25 bit are provided. Fundamental parts are the Arithmetic Logic Unit (ALU), the 25×18 multiplier and the Pre-adder. The ALU can be used as two-input simple logic unit and as three-input adder or subtractor. The three-input adder provides the four following operations which can be selected via *ALUMODE*.

- $Z + X + Y + CIN$
- $Z - (X + Y + CIN)$
- $\text{not}(Z) + X + Y + CIN = -Z + (X + Y + CIN) - 1$
- $\text{not}(Z + X + Y + CIN) = -Z - X - Y - CIN - 1$

The three variables X, Y, and Z correspond to the three multiplexers in the figure which are controlled via *OPMODE* port. The fourth input is *CIN* and stands for carry. Besides the regular inputs it is also possible to select the previous result P in both multiplexers X and Z.

The multiplier receives two signals, 25 bit and 18 bit wide, as input and outputs two partial products with 43 bit width. Both partial products must be added with the ALU for final 48 bit product. Please note that the MSB of both inputs are only useable for signed numbers in Two's complement representation. Therefore, only 24×17 bit are usable for unsigned numbers.

The block *Dual A, D, and Pre-adder* contains an adder which adds A and D before the multiplication. However, only the lowest 25 bit can be used for multiplication. If no Pre-addition and multiplication are needed, both inputs A and B can be cascaded to a 48 bit wide value and selected in X.

Different pipeline stages exist which can be switched on optionally during the implementation phase. Important registers pictured in Figure 3.2 are the final result P, the intermediate result after multiplication M, and the input value C. There are some additional registers in the blocks *Dual A, D, and Pre-adder* and *Dual B Register* where the data paths of A and B can have up to two pipeline stages. D has one pipeline stage and one additional after the Pre-adder.

For short and fast wiring of two DSPs there are special input and output ports marked with a asterisk. They can be connected to adjacent DSP slices and provide an output cascade stream between adjacent DSP slices, e. g., in multi-precision multiplication.

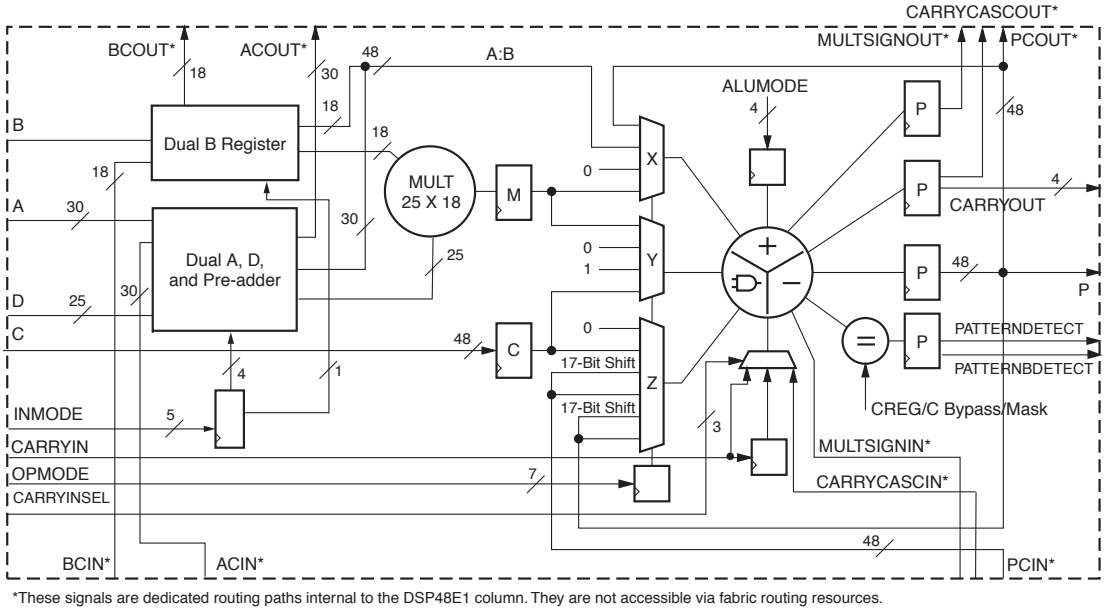

Figure 3.2: Schematic of 7-Series DSP48E1 Slice [Xil14a].

Summarizing, this results in the following fundamental and relevant operations:

- $P = C \pm (B \cdot (D \pm A_{(24 \text{ downto } 0)}) + CARRYIN)$
- $P = P_{\text{old}} \pm (B \cdot (D \pm A_{(24 \text{ downto } 0)}) + CARRYIN)$
- $P = C \pm (A||B)$
- $P = C \pm P_{\text{old}}$
- $P = P_{\text{old}} \pm (A||B)$

3.4 Dedicated Block-RAM

Some applications need to store large amount of data. Therefore, Xilinx implemented dedicated BRAM blocks. These BRAM blocks are denoted by RAMB36E1. All information in this section is taken from [Xil14d].

The available XC7K325T provides 445 each a 36 kbit RAMB36E1. Figure 3.3 gives a schematic overview of a single RAMB36E1 slice. A RAMB36E1 can be used in synchronous True-Dual-Port and Single-Port mode and additionally in asynchronous True-Dual-Port mode. The latter means that that both ports, Port A and Port B can access to the Memory Array independently, also from different clock domains. This makes

it necessary to handle conflict situations in which both ports access the same memory location. The data width can be adjusted, so that each word can be consists of 36 bit, 32 bit data bits and a 4 bit parity. Thus, up to 1 024 words each 36 bit width can be stored in one RAMB36E1.

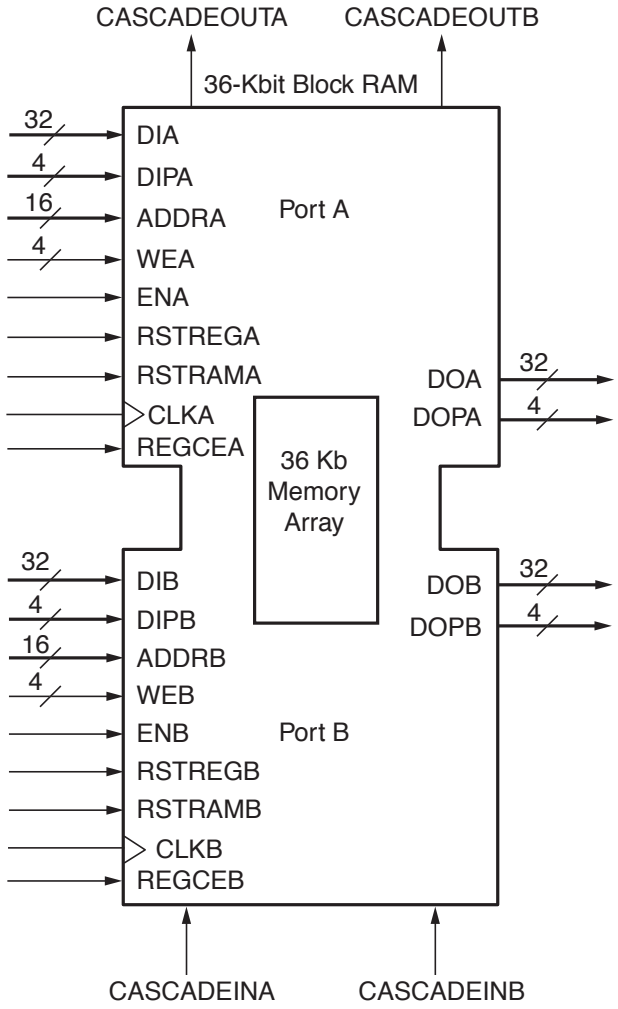

Figure 3.3: Schematic of 7-Series RAMB36E1 Slice [Xil14d].

In the 7-Series of Xilinx FPGAs, some dedicated logic enables to use these BRAM blocks as First-In-First-Out (FIFO), also in dual-clock asynchronous mode. This dedicated logic provides counter, comparator, or status flags for full or empty indication.

4 Elliptic Curve Arithmetics in Hardware

Scalar multiplication is crucial for ECC. We introduced this operation in Section 2.3. In this work, reduced projective coordinate representation are used along with the Montgomery ladder presented by Brier and Joye [BJ02]. We show how the computation can be implemented in a secure and efficient way on reconfigurable hardware.

Point addition and point doubling are based on finite field arithmetic, in particular modular multiplication, modular addition, and modular subtraction. Efficient implementation techniques for these operations are described in this chapter as well.

4.1 Efficient Scalar Multiplication using a Parallel Montgomery Ladder

Section 2.2.3 describes an efficient procedure for point addition and doubling using only $\mathcal{X}$ and $\mathcal{Z}$ coordinates. Since in every step of the Montgomery ladder both, point addition and doubling, have to be performed, these operations can efficiently computed by using a Montgomery ladder for scalar multiplication as proposed by Brier and Joye [BJ02]. Fischer et al. describe this idea for efficient and secure scalar multiplication especially for reduced projective coordinates in [FGKS02]. One step of the Montgomery ladder consists of 19 multiplications, 10 additions, and 4 subtractions. If two arithmetic units are available, a parallel version of this Montgomery ladder can be performed. Parallelizing the algorithm reduces the time complexity to 10 multiplication and 8 addition/subtraction steps.

The approach of Fischer et al. is shown in Algorithm 4.1.1. It shows the parallel computation of $P' = P + Q$ and $Q' = 2Q$ like introduced in equations (2.10) and (2.11). For the computation eight registers are required in addition to three registers holding the domain parameters a and b and the affine difference x_D, the x-coordinate of point $D = P - Q$.

Algorithm 4.1.1 constitutes one step of the parallel Montgomery ladder, called PML-OS. The full Montgomery ladder is shown in Algorithm 4.1.2 (see also Algorithm 2.3.3). At the end of the algorithm in line 7, the projective coordinates of dP and $(d+1)P$ are used to obtain the affine coordinates of dP (see equations (2.12) and (2.14)).

Four essential standard arithmetic operations are required: modular multiplication, modular addition, modular subtraction, and inversion for transformation to affine coordinates. Therefore, these operations must be implemented by the coprocessor.

Algorithm 4.1.1: Parallel Montgomery Ladder — One Step [FGKS02]

Input: $\mathcal{X}_P$, $\mathcal{Z}_P$, $\mathcal{X}_Q$, $\mathcal{Z}_Q$
Output: $\mathcal{X}_{P'}$, $\mathcal{Z}_{P'}$, $\mathcal{X}_{Q'}$, $\mathcal{Z}_{Q'}$
$R0 \leftarrow \mathcal{X}_P$, $R1 \leftarrow \mathcal{Z}_P$, $R2 \leftarrow \mathcal{X}_Q$, $R3 \leftarrow \mathcal{Z}_Q$

(1)	$R6 \leftarrow R2 \cdot R1$	(2)	$R7 \leftarrow R3 \cdot R0$
(3)	$R4 \leftarrow R7 + R6$	(4)	$R5 \leftarrow R7 - R6$
(5)	$R5 \leftarrow R5 \cdot R5$	(6)	$R7 \leftarrow R1 \cdot R3$
(7)	$R1 \leftarrow a \cdot R7$	(8)	$R6 \leftarrow R7 \cdot R7$
(9)	$R0 \leftarrow R0 \cdot R2$	(10)	$R6 \leftarrow b \cdot R6$
(11)	$R0 \leftarrow R0 + R1$	(12)	$R6 \leftarrow R6 + R6$
(13)	$R0 \leftarrow R0 \cdot R4$	(14)	$R1 \leftarrow \mathcal{X}_D \cdot R5$
(15)	$R4 \leftarrow R0 + R6$		
(16)	$R4 \leftarrow R4 + R4$	(17)	$R6 \leftarrow R2 + R2$
(18)	$R4 \leftarrow R4 - R1$	(19)	$R7 \leftarrow R3 + R3$
(20)	$R0 \leftarrow R6 \cdot R7$	(21)	$R1 \leftarrow R3 \cdot R3$
(22)	$R2 \leftarrow R2 \cdot R2$	(23)	$R3 \leftarrow a \cdot R1$
(24)	$R6 \leftarrow R2 - R3$	(25)	$R7 \leftarrow R2 + R3$
(26)	$R1 \leftarrow R1 + R1$		
(27)	$R2 \leftarrow b \cdot R1$	(28)	$R7 \leftarrow R7 \cdot R0$
(29)	$R1 \leftarrow R2 \cdot R1$	(30)	$R0 \leftarrow R0 \cdot R2$
(31)	$R6 \leftarrow R6 \cdot R6$		
(32)	$R6 \leftarrow R6 - R0$	(33)	$R7 \leftarrow R7 + R1$

$R4 \leftarrow \mathcal{X}_{P'}$, $R5 \leftarrow \mathcal{Z}_{P'}$, $R6 \leftarrow \mathcal{X}_{Q'}$, $R7 \leftarrow \mathcal{Z}_{Q'}$

Algorithm 4.1.2: Parallel Montgomery Ladder

Input: elliptic curve E together with an elliptic curve point $P = (\mathcal{X}_P, \mathcal{Z}_P)$ and a scalar $d = \sum_{i=0}^{n-1} d_i 2^i$ with $d_i \in \{0, 1\}$ and $d_{n-1} = 1$, $\mathcal{X}_D$
Output: dP

```
1 Q ← 𝒪
2 for i = n − 1 downto 0 do
3     if d_i = 1 then
4         Q, P ← PML-OS(P, Q)
5     else
6         P, Q ← PML-OS(P, Q)
7 P = (x_dP, y_dP) ← ProjectiveToAffine(P, Q)
8 return (x_dP, y_dP)
```

4.2 Modular Multiplication – Montgomery Multiplication

Recall, scalar multiplication based on point addition and doubling and it require computations in finite field arithmetic like modular multiplication, i. e., $Z = X \cdot Y \bmod M$ with $0 \leq X, Y < M$ and $M > 0$ (M is assumed to be odd for our applications). If the intermediate product $Z' = X \cdot Y$ is computed first afterwards reduced by the modulus M, then the intermediate result Z' has double register length compared to its operands. So an interleaving step might be appropriate.

The naive computation of modular reduction $Z = Z' \bmod M$, the iteratively subtracting the divisor M until the result is less than M, is very expensive.

We choose the Montgomery Multiplication for fast and efficient modular multiplication in hardware.

Montgomery [Mon85] introduced an efficient method in software and hardware for modular multiplication without trial division. With the Montgomery multiplication the dividend is always a multiple of the divisor. This procedure, the Montgomery Reduction $\mathrm{MRed}(Z')$, is shown in Algorithm 4.2.1.

Let $R > M$ with $\gcd(R, M) = 1$. The Montgomery Reduction computes $Z \cdot R^{-1} \bmod M$ for an input $0 \leq Z' < M \cdot R$. The inputs to the algorithm are the product of two integers Z', the modulus M, a pre-computed constant $M' := -M^{-1} \bmod R$, and the radix R. $\mathrm{MRed}(Z')$ uses the radix R for easy division in line 2 only by shifting the complete safe word. Therefore it is helpful to choose R as a power of the base, usually two for binary systems.

Algorithm 4.2.1: Basic Montgomery Reduction $\mathrm{MRed}(Z')$

Input: Z', R, M, and M'
Output: $Z \cdot R^{-1} \bmod M$

```
1 U = Z' · M' mod R
2 Z = (Z' + U·M') / R
3 if Z ≥ M then
4     return Z − M
5 else
6     return Z
```

Of course, the output of the basic Montgomery reduction includes R^{-1} which is not needed. It is recommended to convert both multipliers before multiplication. This results in the following operation chain for a modular multiplication $X \cdot Y \bmod M$:

1. $\mathrm{MRed}(X \cdot R^2) = XR = \widetilde{X}$
2. $\mathrm{MRed}(Y \cdot R^2) = YR = \widetilde{Y}$
3. $\mathrm{MRed}(\widetilde{X} \cdot \widetilde{Y}) = XYR = \widetilde{XY}$
4. $\mathrm{MRed}(\widetilde{XY} \cdot 1) = XY$

The forward and reverse transformation are only necessary once. At the start we transform it and at the end we reverse it back to the new output value. It should be noted that $\widetilde{X} = X \cdot R \bmod M$ can be seen as a M-residue of $0 \leq X < M$, i. e., as a unique representative of X.

The usage of $\mathrm{MRed}(Z')$ is not the most efficient way to multiply two numbers. $\mathrm{MRed}(Z')$ simplifies the reduction but not the multiplication itself. Algorithm 4.2.2 combines integer multiplication and Montgomery reduction in a single function. Single words of the second multiplicand Y with length b will be processed iteratively in line 2. It follows that the division in line 4 is a right shift by b bits. At the end of the algorithm, if $Z \geq M$, it is necessary to subtract M once and return the result Z.

Algorithm 4.2.2: Word-Level Montgomery Product [Men07]

Input: $M = (M_{n-1}, \ldots, M_0)_{2^b}$, $X = (X_{n-1}, \ldots, X_0)_{2^b}$, $Y = (Y_{n-1}, \ldots, Y_0)_{2^b}$, with $0 \leq X, Y < M$, $R = 2^{n \cdot b}$, $\gcd(M, 2^b) = 1$, and $M' = -M^{-1} \bmod 2^b$
Output: $(X \cdot Y \cdot R^{-1}) \bmod M$

1 $Z = (Z_n, \ldots, Z_0)_{2^b} \leftarrow 0$
2 **for** $i = 0$ **to** $n - 1$ **do**
3 $\quad U_i \leftarrow ((Z_0 + X_0 \cdot Y_i) \cdot M') \bmod 2^b$
4 $\quad Z \leftarrow (Z + X \cdot Y_i + M \cdot U_i)/2^b$
5 **if** $Z \geq M$ **then**
6 $\quad Z \leftarrow Z - M$
7 **return** Z

Algorithm 4.2.2 is an efficient way for modular multiplication. However, the conditional statement in line 6 takes additional time and, furthermore, makes the algorithm vulnerable to SCA, especially TA. Therefore it is useful to modify this algorithm as first proposed by Walter in [Wal99]. The modified algorithm is shown in Algorithm 4.2.3. Instead of the conditional subtraction, the loop will pass one more time, thus saving this conditional step. The consequence is one more word Y_n which can filled with zeros and one more loop is required. But eliminating the conditional step reduces complexity and increases the robustness against SCA. The additional loop is negligible.

Summarized, there are no additional steps needed for transformation to Montgomery form, only $R^2 \bmod M$ is required. The transformation has to be done only at the begin and at the end of the computation. The improved version of Montgomery multiplication is secured against timing attacks. In contrast to normal modular multiplication additional registers are needed to store M', Y_n, and Z_n each with a length of b bit. The Improved Montgomery Multiplication Algorithm 4.2.3 has been chosen as one of the algorithms to be implemented for this work.

Algorithm 4.2.3: IMPROVED MONTGOMERY PRODUCT MontMul(X, Y, M) [Men07]

Input: $M = (M_{n-1}, \ldots, M_0)_{2^b}$, $X = (X_{n-1}, \ldots, X_0)_{2^b}$, $Y = (Y_n, \ldots, Y_0)_{2^b}$, with $0 \leq X, Y < 2 \cdot M$, $R = 2^{(n+1)\cdot b}$, $\gcd(M, 2^b) = 1$, and $M' = -M^{-1} \bmod 2^b$
Output: $(X \cdot Y \cdot R^{-1}) \bmod M$

```
1 Z = (Z_n, ..., Z_0)_{2^b} ← 0
2 for i = 0 to n do
3 |   U_i ← ((Z_0 + X_0 · Y_i) · M') mod 2^b
4 |   Z ← (Z + X · Y_i + M · U_i)/2^b
5 return Z
```

4.3 Modular Adder and Subtractor

Our ECC arithmetic requires modular addition and subtraction. Let $0 \leq X, Y < M$ and consider the operations

$$Z' = X + Y, \qquad Z = Z' \bmod M,$$
$$Z' = X - Y, \qquad Z = Z' \bmod M.$$

Algorithm4.3.1 shows the implemented routine for modular addition. It holds, the intermediate result Z' is in maximum $2M - 2$. Therefore, only one conditional subtraction with M is required for final result $Z \in [0, \ldots, M-1]$. If $X = 0$ and $Y = M - 1$ the difference Z' is in minimum $-(M-1)$. This negative result requires a addition with M to be positive.

Algorithm 4.3.1: MODULAR ADDITION ModAdd(X, Y, M)

Input: X, Y, M, with $0 \leq X, Y < M$
Output: Z

```
1 Z ← X + Y
2 if Z ≥ M then
3 |   return Z − M
4 else
5 |   return Z
```

For modular subtraction, Algorithm 4.3.2 shows a similar routine. In worst case, $X = 0$ and $Y = M - 1$ which results $Z = M - 1$. The conditional addition with M avoid the return of a negative number.

Algorithm 4.3.2: Modular Subtraction ModSub(X, Y, M)

Input: X, Y, M, with $0 \leq X, Y < M$
Output: Z

```
1 Z ← X − Y
2 if Z < 0 then
3     return Z + M
4 else
5     return Z
```

4.4 Inversion

The transformation from projective to affine coordinates requires at least two inversions, one for x and one for y. In $\mathbb{F}_p$ the multiplicative inverse can be computed by the Extended Euclidean Algorithm (EEA) or by Fermat's Little Theorem. If no further hardware should be used, then Fermat's Little Theorem is favored. Another point in favor of Fermat is possible leakage of side-channel information by the EEA.

Let $M = p$, $p > 3$, prime and $1 < X < M$. It holds

$$X^{-1} \equiv X^{M-2} \bmod M. \tag{4.1}$$

Due to the one-to-one correspondence to rings for Montgomery multiplication it holds

$$X^{-1}R \equiv X^{M-2}R \bmod M \tag{4.2}$$

where R is defined as introduced in Section 4.2.

The straightforward way to implement the exponentiation in Fermat's Little Theorem is square-and-multiply algorithm. It is advised to avoid using square-and-multiply algorithm to avoid SCA vulnerabilities. In our case $M = p$ is a public domain parameter and so it is not necessary to avoid it. Algorithm 4.4.1 shows the square-and-multiply algorithm for Fermat's little theorem.

Algorithm 4.4.1: Inversion in $\mathbb{F}_p$ — Fermat's little theorem

Input: XR, $M = (M_{n-1}, \ldots, M_0)_{2^b}$, with $X < M$
Output: $Z = X^{-1}R \bmod M$

```
1 Z ← 1
2 for i = n − 1 downto 0 do
3     Z ← Z · Z mod M
4     if M_i = 1 then
5         Z ← Z · XR mod M
6 return Z
```

4.5 Efficient Multiplier in FPGA using DSPs

In the previous sections, general algorithms for required operations have been presented and explained. Modular addition and modular subtraction are easy to implement in hardware. However, the efficient implementation of Montgomery Multiplication is the most difficult problem in this work. Mentens [Men07] already meets all requirements and so it is the most important reference for this work.

Mentens uses Montgomery multiplication for efficient reduction and therefore she investigates some existing hardware designs of [KAK96], mainly Separated Operand Scanning (SOS) and Coarsely Integrated Operand Scanning (CIOS). Due to the improved results reported CIOS are preferred for multiplication [KAK96].

CIOS is a modification of SOS by integrating the reduction step into the multiplication loop. So for reduction, only the length of one word will be shifted instead of the complete length of M. This is possible by changing the position of the loops in both algorithms SOS and CIOS. Figure 4.1 shows how CIOS works. In the following T takes the role of Z in Algorithm 4.2.2 and 4.2.3. In every iteration i it is necessary to compute $U_i = (T_0 + X_0 \cdot Y_i) \cdot M'$, the important part of Montgomery Multiplication for easy division by shifting only. The normal multiplication happens by multiplying X step by step with one word of Y. The complete product T, after every iteration, is the sum of both products XY_i and MU_i. Last step of iteration is shifting by one word. After all iterations are done, T is the final result of MontMul(X, Y, M).

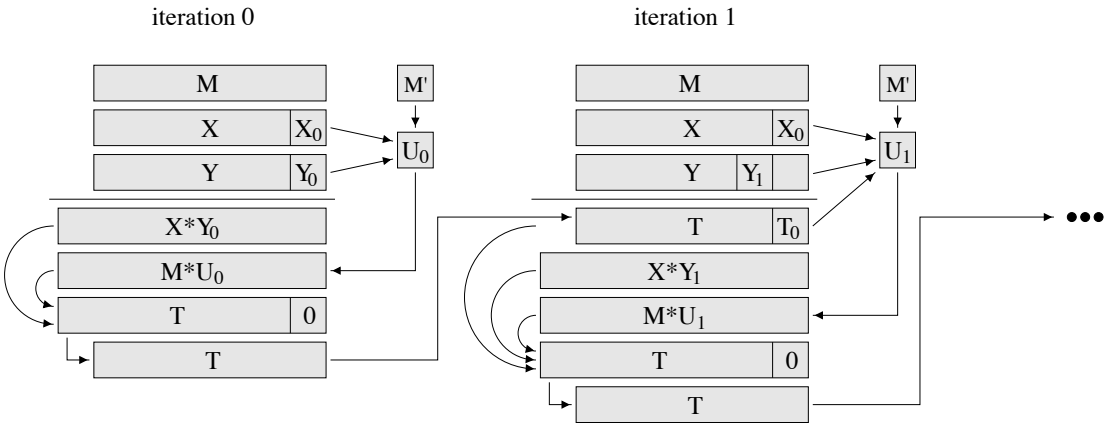

Figure 4.1: Schematic execution of parallelized CIOS algorithm by Mentens [Men07, Figure 3.2].

For better understanding of this schematic presentation, Algorithm 4.5.1 gives a pseudo code of CIOS, see [Men07, Algorithm 8]. Note, that in [Men07, Algorithm 8] only the basic Montgomery multiplication is shown, not the improved Montgomery multiplier by Walter [Wal99]. In this case, carry-safe representation is used which requires the final step FINALADD(C, S). The outer loop is equivalent to the single iterations in Figure 4.1. The first and second inner loop compute the multiplication $X_j \cdot Y_i$ and $M_j \cdot U$ for each word. The third inner loop is equivalent to a right shift by one word of S.

Algorithm 4.5.1: Variation of Coarsely Integrated Operand Scanning (CIOS) method for Montgomery Multiplication with integration of Improved Montgomery Multiplication and Carry-Safe Addition by Mentens [KAK96].

Input: $M = (M_{n-1}, \ldots, M_0)_{2^b}$, $X = (X_n, \ldots, X_0)_{2^b}$, $Y = (Y_n, \ldots, Y_0)_{2^b}$, with $0 \leq X, Y < 2 \cdot M$, $R = 2^{(n+1)\cdot b}$, $\gcd(M, 2^b) = 1$, and $M' = -M^{-1} \bmod 2^b$

Output: $(X \cdot Y \cdot R^{-1}) \bmod M$

1 $S = (S_{n+1}, \ldots, S_0)_{2^b} \leftarrow 0$
2 $C = (C_n, \ldots, C_0)_{2^b} \leftarrow 0$
3 **for** $i = 0$ **to** n **do**
4 **for** $j = 0$ **to** n **do**
5 $(C_j, S_j) \leftarrow X_j \cdot Y_i + S_j + C_j$
6 $U \leftarrow ((S_0 + C_0) \cdot M') \bmod 2^b$
7 $(C_0, S_0) \leftarrow U \cdot M_0 + S_0$
8 **for** $j = 1$ **to** $n - 1$ **do**
9 $(C_j, S_j) \leftarrow U \cdot M_j + S_j + C_{j-1}$
10 $(C_n, S_n) \leftarrow S_n + C_{n-1}$
11 $S_{n+1} \leftarrow C_n$
12 **for** $j = 0$ **to** n **do**
13 $S_j \leftarrow S_{j+1}$
14 $S_{n+1} \leftarrow 0$
15 $T \leftarrow \text{FINALADD}(C, S)$
16 **return** T

Based on this algorithm, MENTENS created a hardware architecture as shown in Figure 4.2, see [Men07, Figure 3.10]. Most important parts are the four blocks I', II', III', and IV', whereby I' and III' are similar. These two blocks consists of $(n+1)$ respectively n $b \times b$ multipliers. The output in carry-safe representation always is represented by double lines. Block I' is equivalent to the multiplication $X_j \cdot Y_i$ in line 5 and Block III' to the multiplication $U \cdot M_j$ in line 9 of Algorithm 4.5.1. In block II', U is calculated and can be mapped to line 6. Until now, all additions in first and second inner loops were neglected. MENTENS separated all additions and combines them into block IV'. It is specially designed 6-2 carry-safe adder with 6 inputs and 2 outputs. Also the reduction step in line 13 is included in this box, but in hardware only wiring is required for shifting. Outputs are one carry and one safe which must be added as shown in line 15. This final addition is not shown.

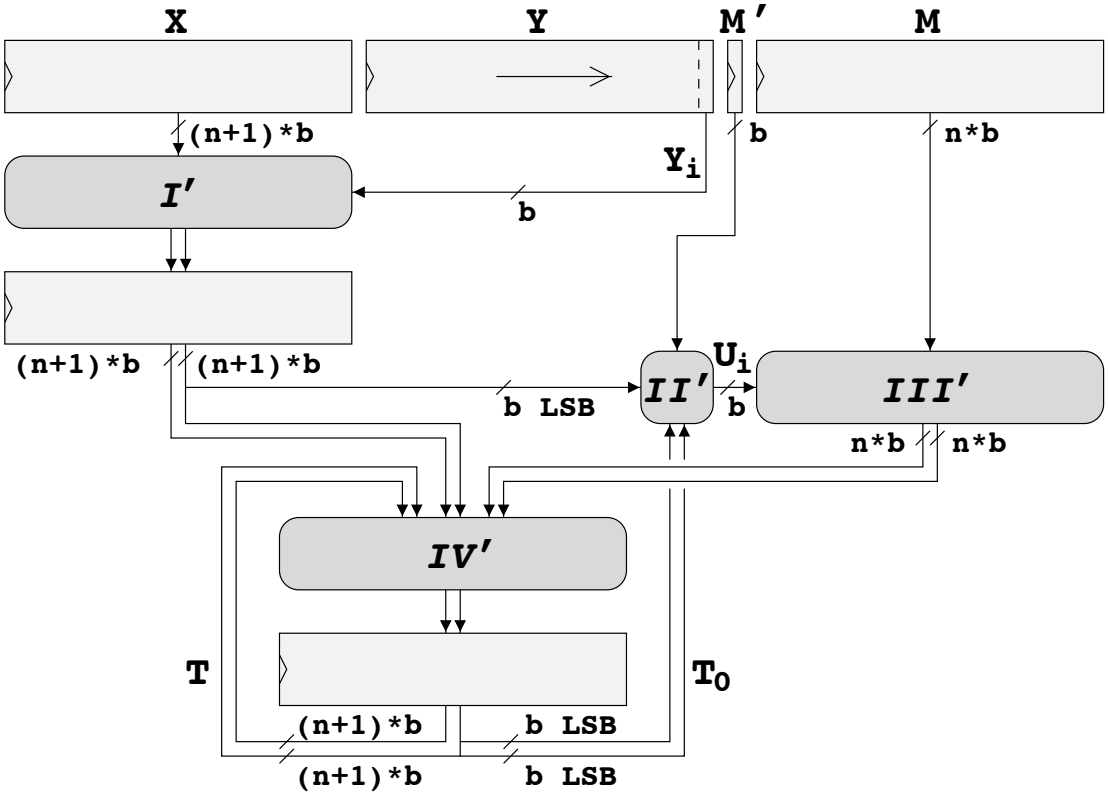

Figure 4.2: Architecture of CIOS multiplier by Mentens [Men07, Figure 3.10].

MENTENS implemented this algorithm on a Xilinx Virtex XC2VP30-7FF1152 FPGA of the 2-Series and used its DSPs to implement the multipliers. This series of Xilinx includes 18×18 multipliers in the DSPs, but the two MSB bits are reserved for overflow and unsigned representation. Therefore the DSPs are used as 16×16 multipliers. The special 6-2 carry-safe adder is implemented in normal logic because only simple operations are required. MENTENS implemented this multiplier without any pipeline stages inside the DSPs which led to a low clock rate relatively.

5 Design and Implementation of the Coprocessor

This chapter covers the design and the implementation of the coprocessor. We summarize all requirements and aims of the final design. After that the designed coprocessor and its units are presented like data memory, instruction buffer, and Arithmetic Unit (AU). The design is presented in a top down approach. The coprocessor core is the AU with its Montgomery Multiplier. This CIOS Multiplier is described more precisely. Furthermore, several variations of the CIOS Multiplier were designed and implemented which each are described as well.

5.1 Design Requirements

The main task of the coprocessor is the fast execution of Montgomery multiplication and also modular addition and subtraction. But an environment is required to use these calculation units. The coprocessor will be connected in subsequent works to a soft microprocessor core which controls the ECC operations. Several requirements are given for the coprocessor design.

The coprocessor must provide an asynchronous mode. This means that the inner part of the coprocessor runs with a frequency different from the part outside of the coprocessor. The internal part is called *internal clock domain* and the part outside is called *external clock domain*. Therefore, all connections must provide the frequency of the external clock domain, independently of the internal clock domain frequency.

Storage for data is required. The data storage must be connected internal for provide data and storage for the AUs and external for data exchange to the microprocessor. Another storage is required for control data. It must store the control data in a queue and provide them by and by to the internal control unit. Asynchronous mode for the connection from the external clock domain of both storage units is mandatory.

It is recommended to use the parallelized Montgomery ladder shown in Algorithm 4.1.1. Therefore, two independent AUs are required.

5.2 Coprocessor

This section explains the overall design of the coprocessor. The particular elements are described in the following sections.

Besides the fact that the coprocessor includes arithmetic elements, three issues must be solved:

1. How can the data be exchanged between microprocessor and coprocessor?
2. How can the coprocessor be controlled?
3. How is the problem of different clock domains of microprocessor and coprocessor to solve?

All these issues can be solved by using BRAM. It stores the data and can be connected via two autonomous port, each in different clock domains. The idea for controlling the coprocessor is the use of opcodes. The opcodes will be written into a FIFO and it handle the queue procedure. Because of FIFO support of the BRAM with asynchronous True-Dual-Port mode it can be used for the opcodes as well.

The coprocessor is designed for flexible bit lengths up to 1 024 bit for M. The generic parameter for bit length must be set before synthesis. The coprocessor is split into two clock domains shown in Figure 5.1 (dotted line). Three elements communicate with the soft microprocessor core, the BRAM, the FIFO, and the Error Reg.

With the BRAM, the processor can read and write directly into the memory that are used internally. The coprocessor provides a data bus of 32 bit for data-in and data-out. Single registers and their 32 bit words can be addressed via the *bram_addra* port. The BRAM contains 16 registers with fix size of 1 024 bit but only the lowest bits, which are set with the generic parameter, are used. Thus 9 bit are required to address all single 32 bit words. More details for the BRAM follow in Section 5.3.

The processor controls the coprocessor through a FIFO. Opcodes of 32 bit will be written into the FIFO which are processed internally. The signal *fifo_ready* signals the processor if the coprocessor is idle.

It must be possible to check the coprocessor for errors, for example invalid opcodes, so *error_status* indicates if any errors happened.

Internally there are three main blocks, both AUs and the Finite State Machine (FSM). The internal data path between AUs and BRAM is 256 bit wide. To reduce data path delays a register between AUs and BRAM is introduced. The output of the AU is size of M. Both outputs are defined as two further results $Z1$ and $Z2$. Direct loading $Z1$ and $Z2$ into an AU allows the use of the full data width to reduce clock cycles. To store an output of one AU a multiplexer splits this output to words of 256 bit. All internal elements are controlled by the FSM, which executes the opcodes obtained by the FIFO. It controls them with its control lines, which are represented as gray lines.

5.3 Data Memory

The data memory is the central point for exchange of data between coprocessor and microprocessor. Firstly, the required memory must be evaluated.

The parallel Montgomery ladder as shown in Algorithm 4.1.1 is one requirement of the design. The algorithm requires eight longwords plus further three longwords for values a, b, and χ_D. The modulus M is required for modulo operation. For Montgomery multiplications R^2 and M' are required as well. Two further registers with fixed values 0 and

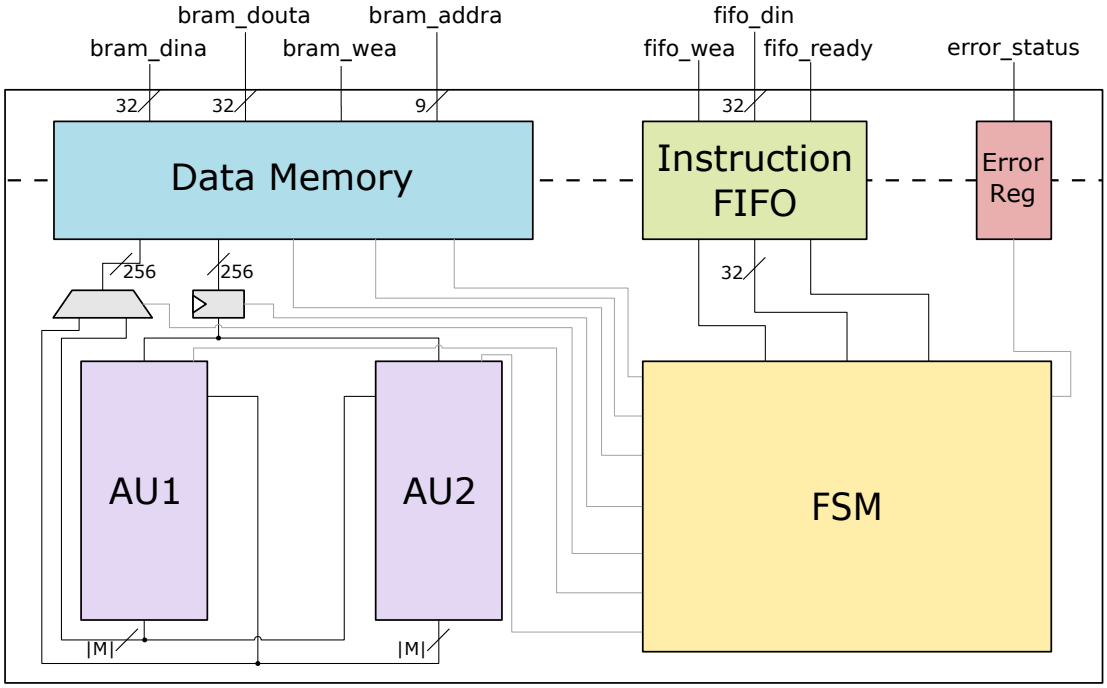

Figure 5.1: Architecture of designed coprocessor.

1 for special cases are desired. In sum 16 longwords[1] are required for a straightforward non-optimized implementation of this design. Note, in the following the storage of one longword is called register. These 16 registers are labeled with $R0$ to $R15$.

A RAMB36E1 has a data width of 32 bit, normally 36 bit but 4 bit are only for parity. The internal data width of the coprocessor is 256 bit, therefore, for fast access to one complete 256 bit word it is necessary to use more than one BRAM at the same time. In this case eight blocks are required.

All registers[2] are stored in these eight BRAM. In maximum configuration of bit length of M one register requires 1 024 bit. Eight RAM36E1 slices with each 36 kbit are clearly overdone for the in maximum required 16 kbit. It is defined, that in every bit length configuration one register has fixed size of 1 024 bit, and only the lower significant bits are used for smaller M.

Figure 5.2 shows a schematic design of the complete BRAM element. For simplified representation all signals are connected to the single RAMB36E1 slices via the data bus shown. Both interfaces of inner and outer domain are almost identical with one exception. The internal data width is 256 bit. Therefore, 16 registers with each 4 words must be addressed, so 6 bit are required for addressing. In the external domain only

[1] Normally, 17 longwords are required if also y_P should be calculated as shown in equation (2.14). This fact was overlooked in final implementation. However, it can compensated by the fact, that both registers of M and M' are unused after they loaded into MREG and M'REG of both AUs.

[2] $R0 = 0$ and $R1 = 1$ are special fictive registers which are loaded directly by simple logic into the input registers of the AUs for saving clock cycles. We ignore this fact for easier description of the design.

32 bit can be read and written, thus 9 bit are required here for addressing. The three further bits control both multiplexers.

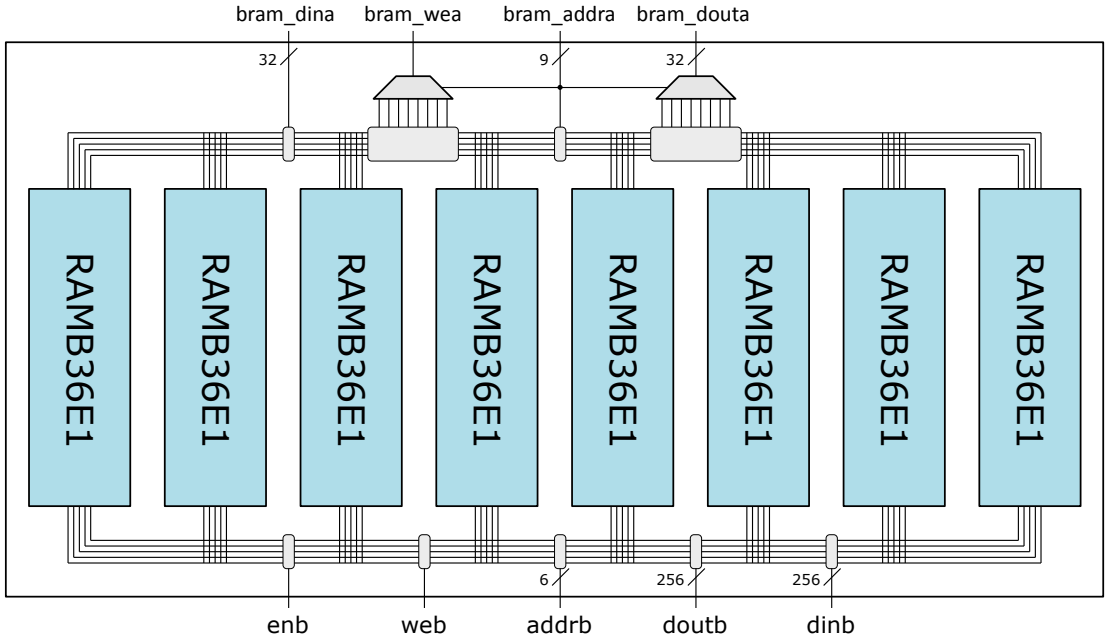

Figure 5.2: Architecture of BRAM consists of eight parallel RAMB36E1 slices.

5.4 Coprocessor Control

The coprocessor is instructed by opcodes which are written into the instruction FIFO and interpreted by the Finite State Machine (FSM). In case of errors the Error Reg indicates them to the microprocessor. In the following, these different units are described in more detail.

First of all, we define the opcodes that are stored in the FIFO and interpreted and executed in the Finite State Machine (FSM). The following points must be observed:

- 8 different operations are needed: NOP, MontMul, ModAdd, ModSub, Add, Load, Store, Reset.
- 2 AUs must be controlled separately.
- All registers $R0$ to $R15$ and also $Z1$ and $Z2$ must be addressable.

Considering the following requirements the scheme shown in Figure 5.3 is defined. The opcodes consists two subopcodes each of 2 Byte size. Every subopcode is divided into 4 parts. The operation byte indicates the operation (load, store, MontMul, ModAdd, etc.). Both, reg1 and reg2, have different meanings depending on the operation. For example, load reg1 to reg2 or multiply reg1 with reg2. The flag byte is used for distinguishing

between regular registers, from `0x0` to `0xF`, and result registers $Z1$ and $Z2$. A detailed description of the opcode design is attached in Appendix B.

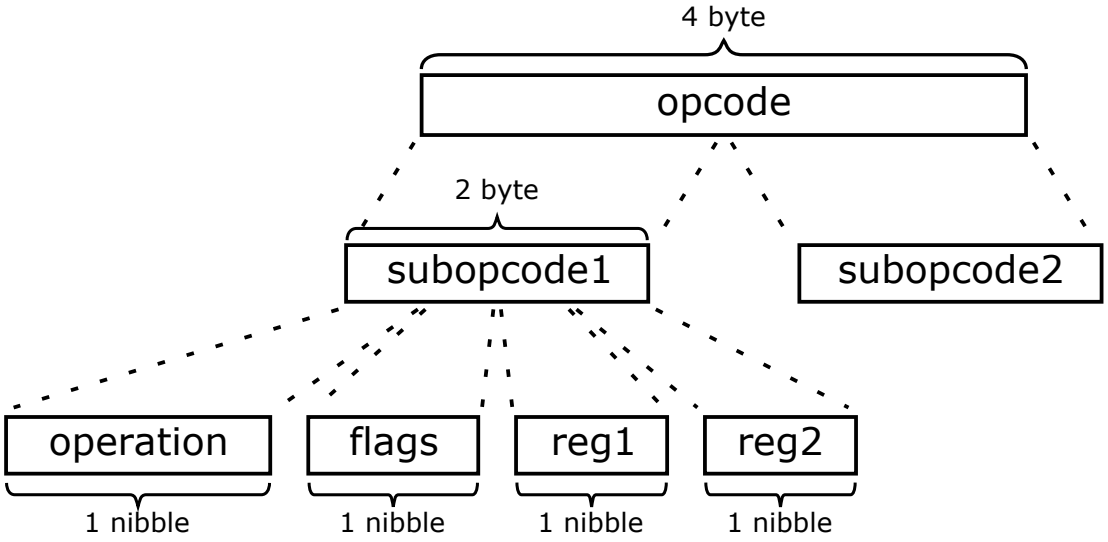

Figure 5.3: Schematic presentation of an opcode and its parts. The opcode is parted into two subopcodes, one for AU1 and one for AU2..

These opcodes are stored and handled by the FIFO. BRAMs of Xilinx provide built-in FIFO support. Bit length of one opcode is 32 bit, thus one FIFO is sufficient. Two signals are for storage capacity warnings, one for the outer domain to signal the FIFO is empty and one for internal domain to signal the FSM for further existing opcodes. The FSM calls with a separate signal for a new opcode if the last one is executed.

We define rules to process the opcodes:

- The order of execution is always subopcode1 before subopcode2.
- Per opcode it is only possible to instruct one AU. Subopcode1 is always for AU1 and subopcode2 always for AU2. However, if only one AU is needed, also NOP operations are supported.
- It is possible to instruct different operations for both AUs, e. g., load $R4$ to XREG of AU1 and store $Z2$ of AU2 to $R6$.
- If both tasks are arithmetic tasks, both AUs start simultaneously after loading of all operands.

The basic processing steps of the FSM are shown in Figure 5.4. Firstly, check the first sub-opcode. Depending on the operation, the different steps are executed. In the case of an invalid subopcode the status bit of Error Reg is set to 1. After execution of Subopcode1 the second part is checked and executed just like the first one. If both subopcodes are arithmetic operations, both AUs receive the start signal simultaneously. Otherwise, the respective AU receive the start signal immediately after loading both

operands. After execution of both subopcodes the FSM idles until all load, store, and arithmetic calls are finished, the next opcode is received.

The possibility to signalize errors must be given. We realize it with Error Reg. For the outer domain it shows the status and it can be reset. In the inner domain this block is controlled by the FSM. The FSM sets the Error Reg to 1 in case of receiving wrong opcodes.

Because Error Reg is reachable from two different clock domains it is not sufficient to use only one register. To avoid this problem three cascaded registers must be used but the first register is clocked by the internal clock and the other two by the external clock. This realizes a dual rank synchronizer which synchronizes the internal and external section.

5.5 Arithmetic Unit

The core element of this coprocessor is the Arithmetic Unit (AU). It must provide all operations which are described in Chapter 4. These fundamental operations are modular multiplication, which is solved with Montgomery multiplication, modular addition, and modular subtraction. Additionally, simple addition is needed in order to sum carry and safe of Montgomery multiplication result.

These operations are designed and implemented as shown in Figure 5.5. The FSM controls the AU. It is triggered by the start signal from the outer FSM and the given opcode. The three input registers, X and M with length $|M| = |X|$ and Y with length $|Y|$, are used for both elementary operations $X \pm Y = Z \bmod M$ and $\mathrm{MontMul}(X, Y, M)$ which uses also M'. The input register of M' needs just 17 bit. The output register Z is suggested within Mod Add/Sub. The input registers are loaded from the outer FSM. There are three methods to load these input registers. Usually the input is load from the data memory into one of these registers. The outer FSM controls the multiplexer and the enable-signals of the FFs because of handling the internal data width of the coprocessor of 256 bit. The second and the third ways are similar. In one case the own Z register and in the other case the Z register of the other AU can be loaded directly in full bit length. This method allows a faster computation, since the result are directly fed into the next step without the need to save them in the data memory. The AU is completely controlled by the FSM which itself receives control signals from the outer FSM. The FSM also sends the finish signal if the AU completed the computation. The FSM controls which registers are used for input in each arithmetic block. Both arithmetic blocks *MontMul* and *Mod Add/Sub* are explained in more detail in the further subsections.

5.5.1 Modular Multiplier MontMul

This subsection describes the designed Montgomery multiplier or rather different versions of Montgomery multipliers. Generally, it is an adaption of the CIOS multiplier of the design in Nele Mentens dissertation as described in Section 4.5. Therefore, the changes from her design to this elaborated design are explained first. After that, the further changes and modifications are presented.

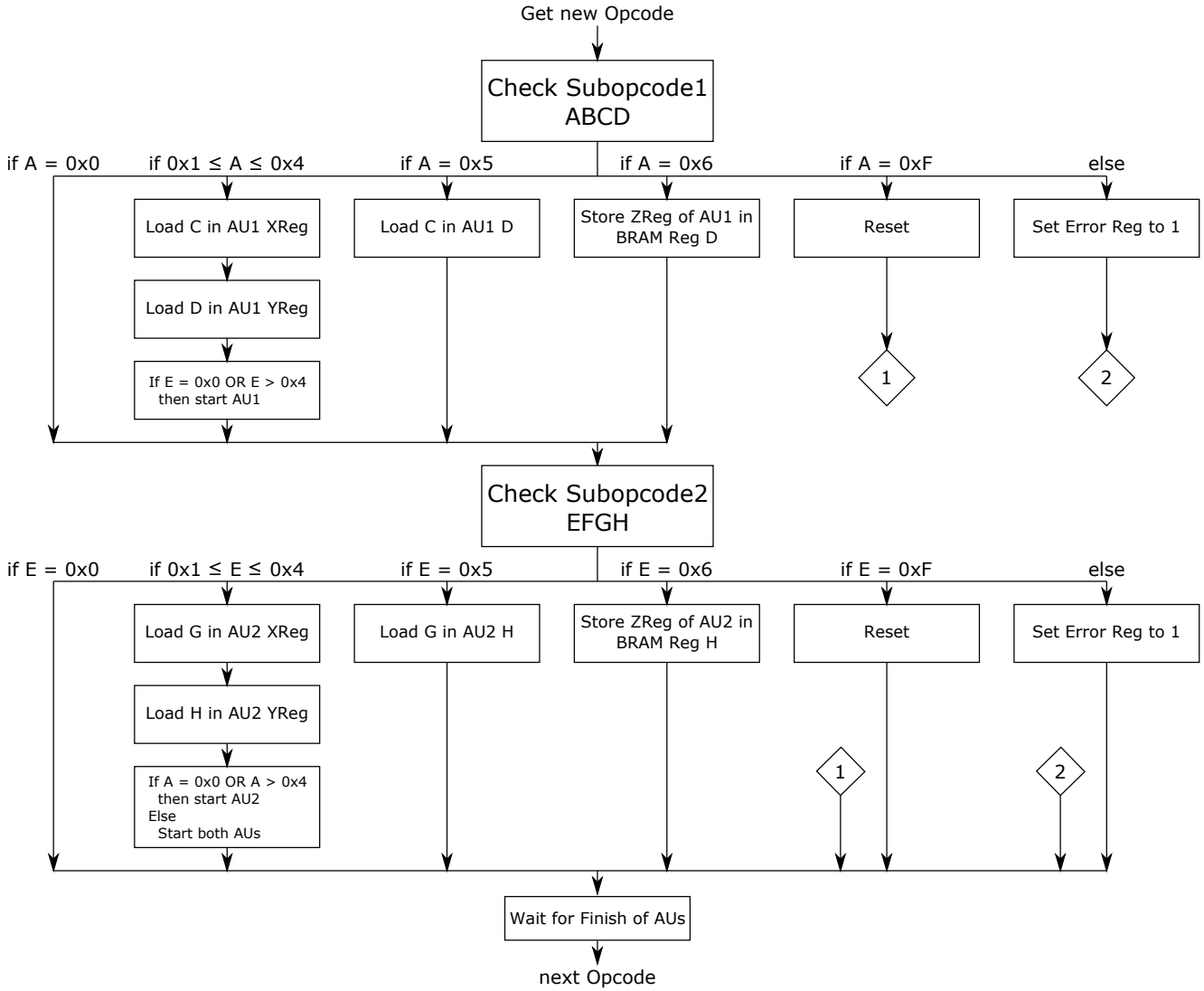

Figure 5.4: Single processing step of the coprocessor FSM for execution of one opcode.

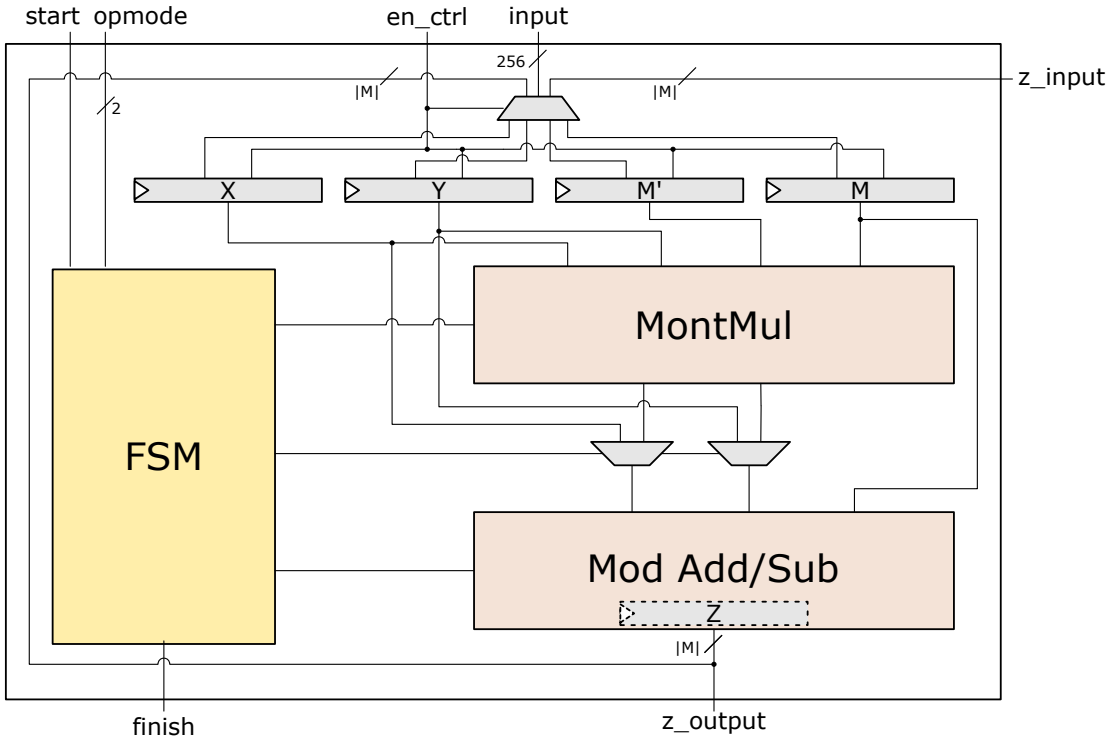

Figure 5.5: Architecture design of an Arithmetic Unit. It consists mainly of two core functions, Montgomery multiplication and modular addition/subtraction.

5.5.1.1 MontMul using 17×17 Multiplication

In CIOS design by MENTENS there are two different types of blocks. I', II', and III' consist of DSPs and IV' consists of simple logic slices for a special 6-2 Carry-Safe Adder (CSA). The following list shows all changes for the adaption from MENTENS' design to the first design step.

- In our FPGA 17×24 multipliers are opposed to the 16×16 multipliers in MENTENS' design. So in a first step we use the DSPs as 17×17 multipliers. It saves two iterations of the outer loop of the CIOS algorithm if $|M| = 544$ bit. Later also the full available bit length are used.
- In MENTENS' design it is possible to calculate with $M = (M_{n-1}, \ldots, M_0)_{2^b}$, $X = (X_n, \ldots, X_0)_{2^b}$, and $Y = (Y_n, \ldots, Y_0)_{2^b}$. In this work it was predefined that $X, Y < M$. Therefore, a single word of X can be saved ($X = (X_{n-1}, \ldots, X_0)_{2^b}$) resulting in less hardware consumption. This does not apply for Y, because of the implementation of the improved version of Montgomery multiplication.
- MENTENS' design has a separate CSA using simple logic slices. One aim of this work is to use DSPs as much as possible and, instead, save other resources of the FPGA. Therefore, the pre- and post-adders inside of the DSPs are used.
- In the given CIOS hardware design hardly any pipeline stages are used. Our first approach in this project is to use almost all pipeline stages inside of the DSP.

Figure 5.6 shows the first adaption of the basic design. It is similar to the basic CIOS design but the separate addition block IV' is integrated in both Multiply-Then-Add (MTA) blocks, also named with I and III. The Add-Then-Multiply (ATM) block II is similar to II' in MENTENS' work. The here designed CIOS multiplier is closer to Algorithm 4.5.1. The complete schematic design corresponds to the outer loop. Y_i is selected with the multiplexer. MTA_1 matches the first inner loop of the algorithm, the calculation of U matches ATM, and the second inner loop is identical to MTA_2. The output is in carry-safe representation and is summed up externally of this multiplier.

For better understanding of correct sorting of all carry and safe words Figure 5.7 shows one execution of the outer loop of CIOS. The upper part corresponds to MTA_1. Here, X and Y_i are multiplied and $S_{IIIprev}$ and $C_{IIIprev}$ are added. The result of this calculation is passed to MTA_2. Here, U and M are multiplied and added to the previous result. The lowest significant safe word of the resulting sum consists of zeros, thus, the complete safe can be shifted by one word as required for Montgomery multiplication. The outcomes are one safe and one carry which are used in the next loop or, if all loops are finished, they are passed to final addition.

A MTA block is split into several columns of 17 bit words. Every column consists of a post-add multiplier. The ATM block consists of one pre-add multiplier. Both, pre- and post-add multiplier, are shown in Figures 5.8 and 5.9. The MTA block consists of single post-add multipliers and one post-add multiplier consists of two DSPs. The first DSP calculates

$$\text{multiplicand1} \cdot \text{multiplicand2} + \text{summand1} = PC_{out/in}$$

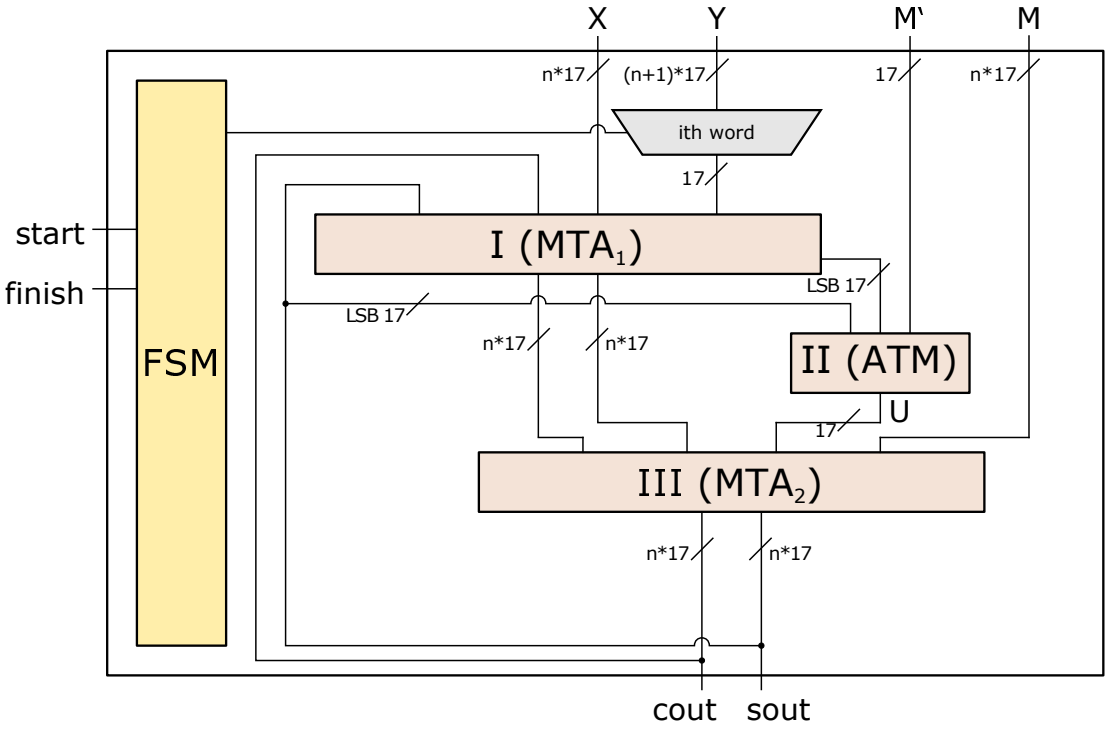

Figure 5.6: Architecture Design of Montgomery Multiplier using 17×17bit Multipliers.

and the second one calculates

$$PC_{out/in} + \text{summand2} = \text{carry}||\text{safe}.$$

As described in Section 3.3 it is only possible to add one further summand in DSP48E1 if multiplication is used, since partial products are used in the DSP. Therefore a second DSP is required for the addition of summand2. Computing U follows the reverse principle. The summands are added and then multiplied with multiplicand1. For U only the lowest 17 bit are required. Following equation is implemented by the pre-add multiplier

$$(\text{summand1} + \text{summand2}) \cdot \text{multiplicand1} = \text{safe} \bmod 2^{17}.$$

The upper 17 bit are not used. Please note, that the resource optimization measures (e. g. using multiple DSPs) are not in the scope of this project. We aim to increase the use of DSPs and, thereby, keep other resources free. Some possible optimization measures are described in following sections.

One thing is not mentioned so far about the post-add multiplier. The $interim_result$ indicated in Figure 5.8 as a dotted line is a shortcut required for saving one clock cycle in one loop run. This fact can be recognized in Figure 5.10. The red area corresponds to ATM, the yellow area corresponds to MTA_1, and the green area corresponds to MTA_2. For the calculation of U it is necessary to add the lowest safe word of $X_0 \cdot Y_i$ and add this result with lowest carry and safe word of the last loop. The pre-adder handles only

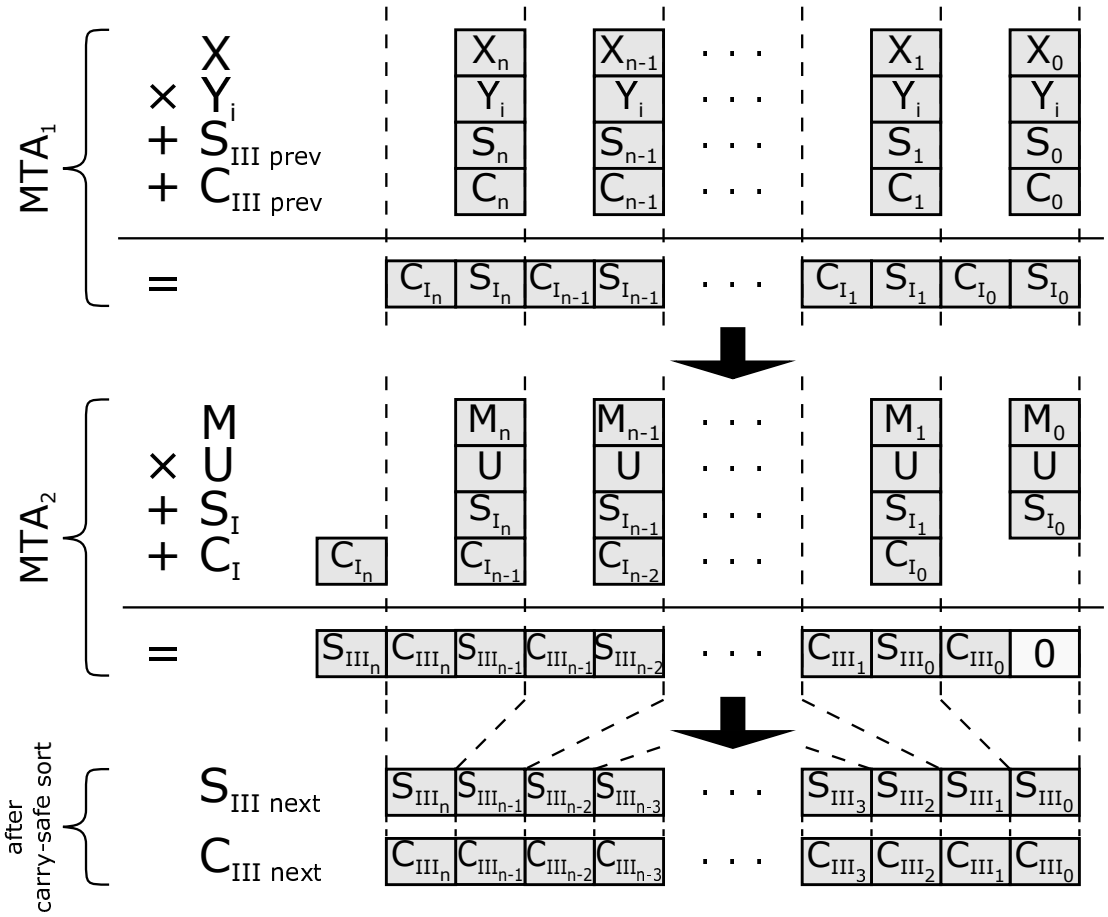

Figure 5.7: Schematic execution of one loop of CIOS algorithm. The upper part corresponds to MTA_1 and the part in the middle to MTA_2. The bottom shows carry and safe after reduction step.

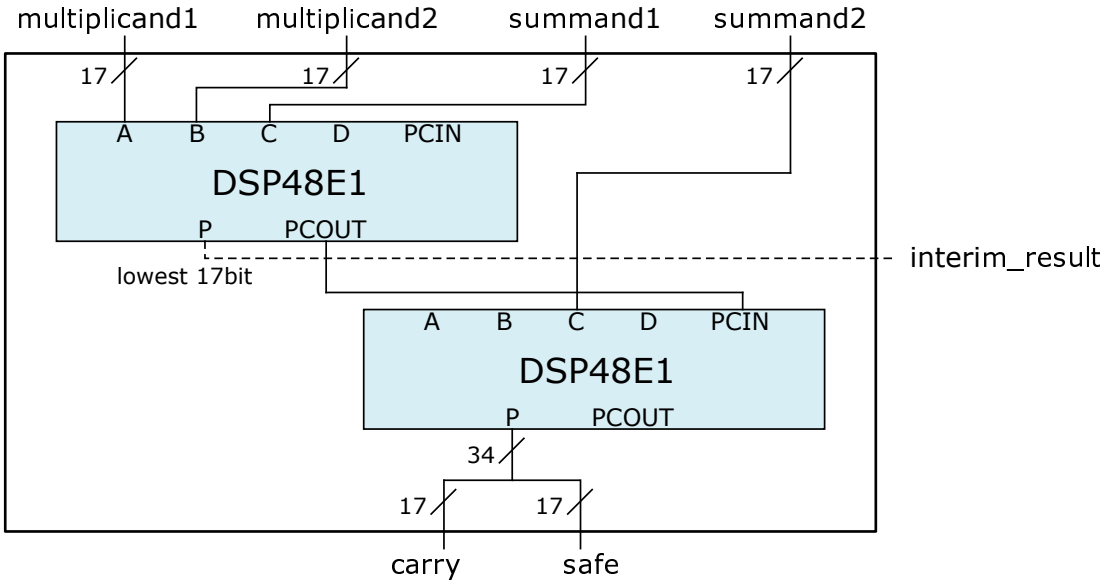

Figure 5.8: Schematic design of one 17×17bit multiplier with post-addition.

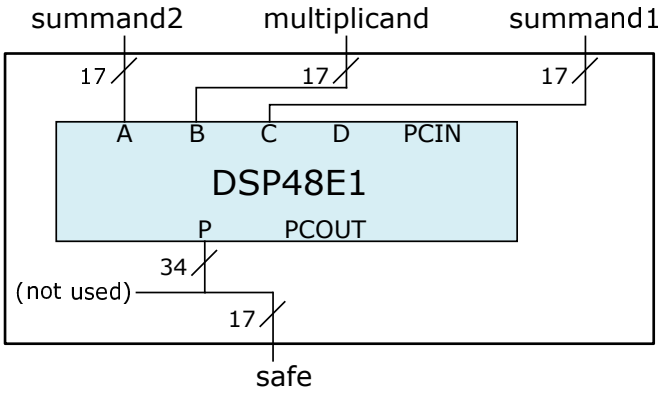

Figure 5.9: Schematic design of one 17×17bit multiplier with pre-addition.

two operands and therefore, the first pre-addition $(X_0 \cdot Y_i) + safe_{last}$ occurs within the MTA_1. The interim result is passed to ATM. The pre-adder sums this interim result with $carry_{last}$ at the same time as in the post-add multiplier.

Generally, one loop iteration requires ten pipeline stages. They are presented in the figure by the registers in the respectively pipeline stage. Both registers after a multiplication step are intended to represent the partial products in register M of DSP48E1. In Figure 5.10 eleven pipeline stages are shown but stage0 and stage9 carry data simultaneously. Thus, Montgomery multiplication (without final addition) requires $1 + (n + 1) \cdot 9$ clock cycles.

5.5.1.2 MontMul using 17×24 Multiplication

In the last subsection we adapted MENTENS' design to the current requirements and hardware. For easier modification the multiplier inside of a DSP48E1 is only used as 17×17 bit multiplier. In the next step the full available bit lengths are to be utilized. The DSP48E1 provides a multiplier with 17×24 bit operand lengths for unsigned numbers. At first glance, two ways are possible to use the varying operand sizes:

Word length of Y_i and M' is 24 bit and X and M are split into words of length 17 bit.
This modification would result in a more efficient utilization of the multipliers in the DSP48E1 with the same hardware usage. Furthermore, Y is split in less words of 24 bit length instead of 17 bit which results in less iterations that are needed and thus less time is required. This benefits would be nice but, unfortunately, it is not possible to use this procedure by checking the functionality because of the following problem.

In the last step before reduction the following operation is done:

$$M_0 \cdot U + S_{I_0} = C_{III_0} || S_{III_0}.$$

M_0 has length 17 bit and U and S_{I_0} have a length of 24 bit. This results 24 bit safe with 17 bit carry. Because of Algorithm 4.2.3, it is necessary to shift by b bit. In this scenario M' has length 24 bit and therefore $b = 24$. However, after the last post-add multiplier it is not possible to shift by 24 bit because only 17 zeros are present. Consider the following example computation given:

$$\begin{aligned} X &= \texttt{0x244451111} = \{\texttt{0x12222}, \texttt{0x11111}\}_{17}, \\ Y_0 &= \texttt{0x000001}, \\ M &= \texttt{0x333333333} = \{\texttt{0x19999}, \texttt{0x13333}\}_{17}, \text{ and} \\ M' &= -M^{-1} \bmod 2^{24} = \texttt{0x000005}. \end{aligned}$$

MTA_1 calculates

$$\begin{aligned} C_{I_0} = \texttt{0x00000} || S_{I_0} = \texttt{0x011111} &\leftarrow X_0 \cdot Y_0 + S_{III_0\,prev} + C_{III_0\,prev} \text{ and} \\ C_{I_1} = \texttt{0x00000} || S_{I_1} = \texttt{0x012222} &\leftarrow X_1 \cdot Y_0 + S_{III_1\,prev} + C_{III_1\,prev}. \end{aligned}$$

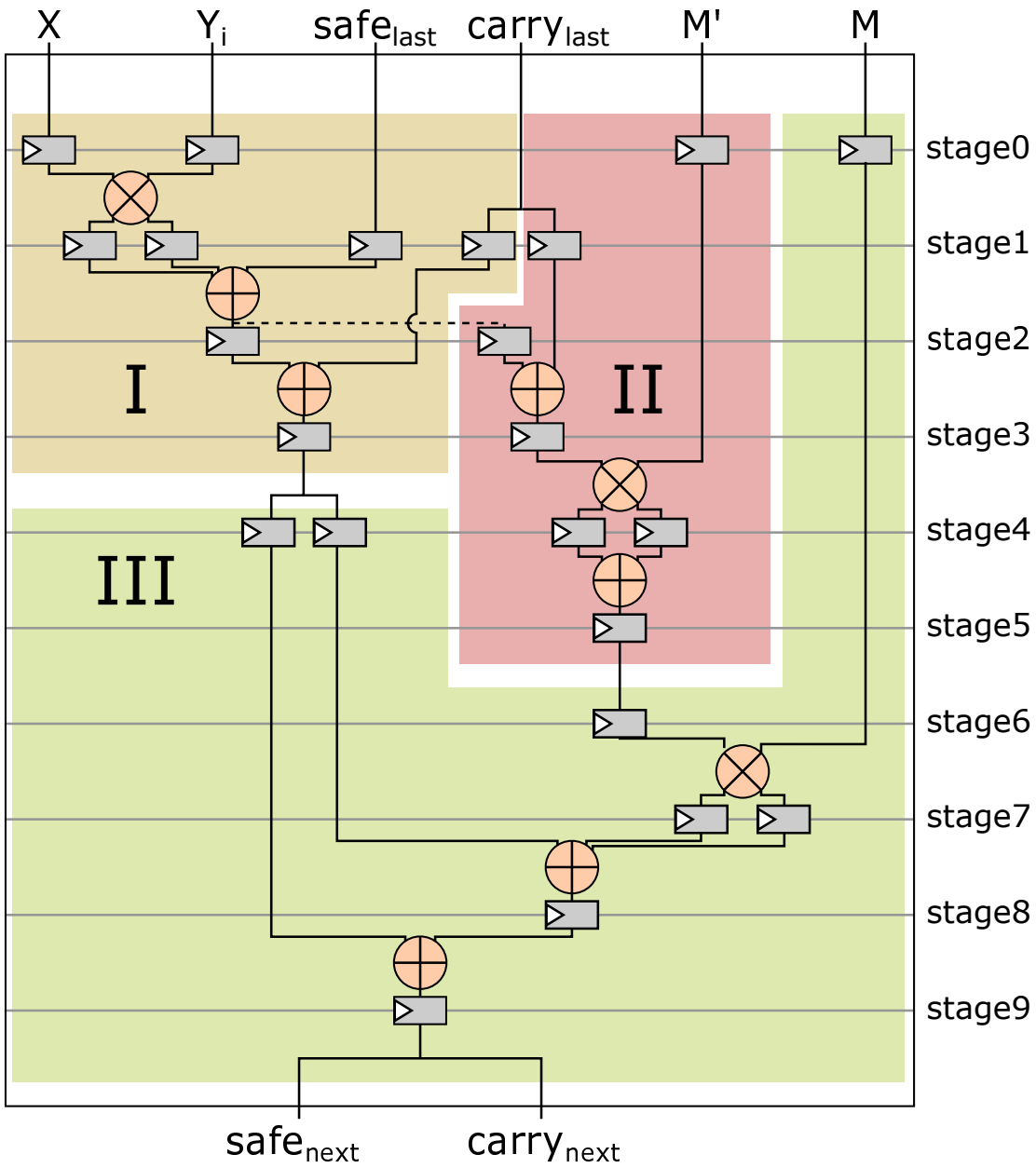

Figure 5.10: Overview of the pipeline stages of Montgomery multiplier using 17×17 bit multipliers.

U is calculated by

$$U = \texttt{0x015555} \leftarrow (X_0 + S_{III_0\,prev} + C_{III_0\,prev}) \cdot M' \bmod 2^{24}.$$

Eventually, MTA_2 calculates

$$\begin{aligned} C_{III_0} = \texttt{0x00199} \quad || \quad & S_{III_0} = \texttt{0x9A0000} \leftarrow M_0 \cdot U + S_{I_0}, \\ C_{III_1} = \texttt{0x00222} \quad || \quad & S_{III_1} = \texttt{0x21EEEF} \leftarrow M_1 \cdot U + S_{I_1} + C_{I_0}, \text{ and} \\ & S_{III_2} = \texttt{0x000000} \leftarrow C_{I_1}. \end{aligned}$$

The next step in Algorithm 4.2.3 would be shift safe by 24 bit, but S_{III_0} consists only of 17 zeros. This is caused by the fact that the lowest 24 bit of M must be multiplied by U to allow correct shifting.

Therefore, this variant can not be implemented.

Word length of Y_i and M' is 17 bit and X and M are split into words of length 24 bit. Unlike the previous procedure there is no time advantage using this method because there still requires the processing of $n+1$ words to process Y. Fewer DSPs are needed instead, since X and M are split into fewer words. For better overview the number of 24 bit words is defined as

$$\widetilde{n} = \left\lceil \frac{\left\lceil \frac{|M|+17}{17} \right\rceil \cdot 17}{24} \right\rceil.$$

The reduction needs a shift by 17 bit because M' has length of 17 bit which follows $b = 17$. In case of 17×17 multipliers the shift causes dropping the lowest safe word. This simple solution does not work because one safe word has length 24 bit. The consequence of this is the fact that the complete result after MTA_2 must be resorted. Figure 5.11 shows a solution. The upper part shows the output after MTA_2. The shift causes the top 24 bit to be the next safe and the lowest 17 bit of the next higher safe to be the next carry. In the next loop MTA_1 must calculate the following operation in maximum

$$\begin{aligned} X_0 \cdot Y_1 + S_{III_0\,prev} + C_{III_0\,prev} &= \texttt{0x1FFFF} \cdot \texttt{0xFFFFFF} + \texttt{0xFFFFFF} + \texttt{0xFFFF80} \\ &= \texttt{0x20000FDFF80} \\ &\rightarrow 18 \text{ bit carry}||24 \text{ bit safe}. \end{aligned}$$

In the next iteration the carry consists of 18 bit. This would cause the result to grow with each iteration. Figure 5.11 shows a strategy to avoid iteratively growing results. The highest bit is the 42nd bit which will be moved as LSB to the next above carry word. This is filled up with zeros, thus no complications are possible.

This second approach of using 17×24 multipliers has been realized. The changes shown have been implemented into the new design. The resulting design is shown in Figure 5.12 with the following changes relative to the 17×17 design:

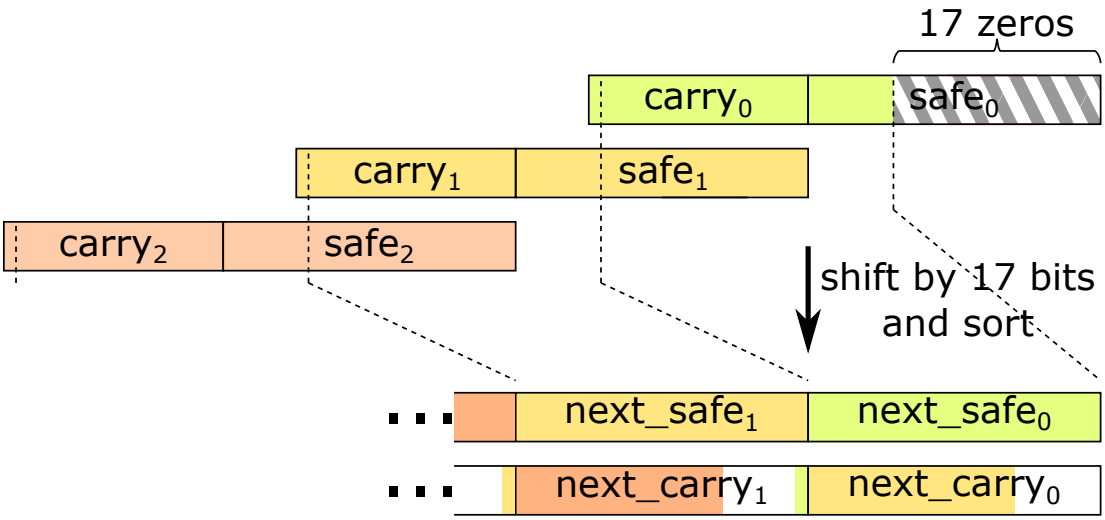

Figure 5.11: Schematic of shifting and sorting after MTA_2 in 17×24 Version.

- All DSPs use their multiplier with full provided lengths 17×24.
- X and M are split into $\tilde{n}$ words of 24 bit. Thus, less DSPs are necessary in both MTAs.
- Generally, safe has 24 bit and carry has 18 bit size. After MTA_2 the block *Carry-Safe Sort* moves all carry and safe words as shown in Figure 5.11.

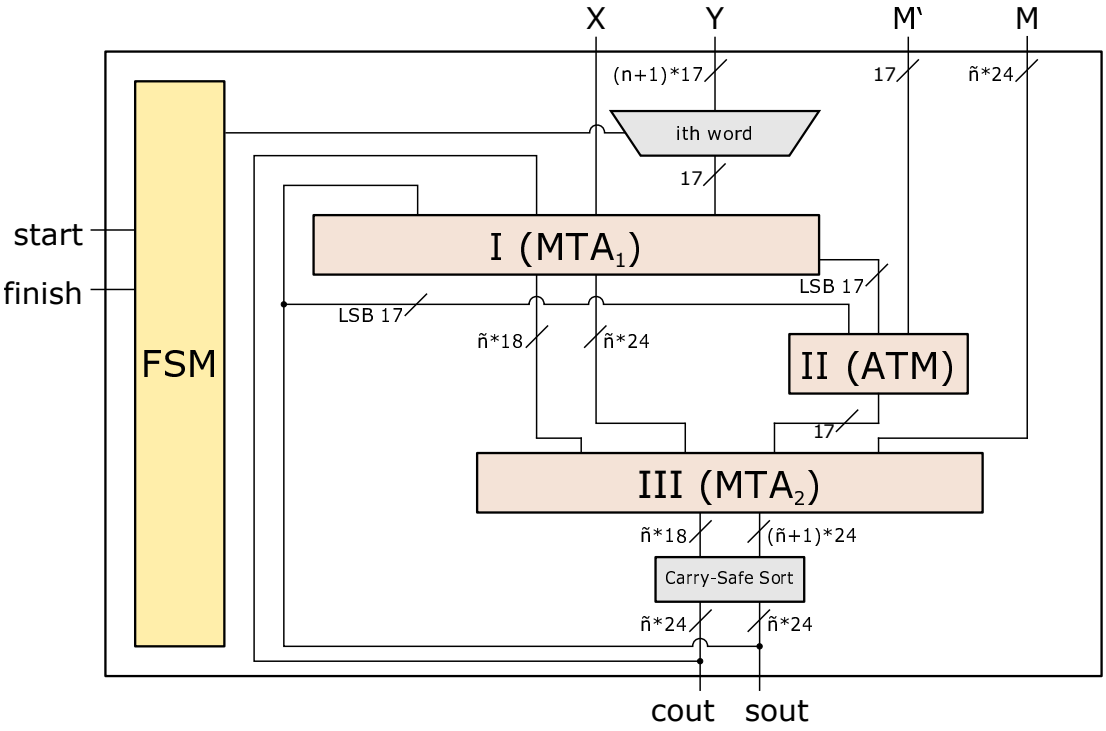

Figure 5.12: Schematic Design of Montgomery Multiplier using 17×24 Multipliers with Maximum Number of DSPs.

5.5.1.3 More Efficient Usage of DSPs in Time

Previously, no consideration was taken to save DSPs. The CIOS algorithm was exactly placed in hardware as given. One reason is the aim to use DSPs as much as possible instead of other hardware resources. Almost all DSPs are unused for most of the execution time. Therefore, some ideas for reducing the number of DSPs are given.

Multiplexing of MTA_1 and MTA_2: Both MTA blocks are identical and not used simultaneously. For the computation we can rearrange the computation onto a single block instead of using a two blocks sparsely. The new MTA block is used twice in the computation chain. The multiplexing for MTA input is controlled by the FSM. Using this measure halves the number of DSPs used.

Multiplexing DSPs within MTA: Building on the recently presented modification it is possible to halve the number of used DSPs again. Inside of one post-add multiplier two DSPs are used. It is possible to use only one DSP48E1 by using the internal loop of P[3]. After the first addition with summand1, P will be reused and added with summand2. This further reduction of used DSPs requires multiplexer for the input.

Multiplexing of MontMul and FinalAdd: In the actual design of the AU as shown in Figure 5.5 separate elements are defined for Montgomery multiplication and for modular addition/subtraction. Reducing the number of used DSPs can be achieved by combining both elements. One possible idea is to modify the MTA block so that it can be used also for modular addition and modular subtraction. More multiplexing units are necessary.

The first two presented modifications were implemented and tested. The third modification was not taken into account in this work. First tests have shown that the modification slows the multiplier significantly.

5.5.1.4 Higher Throughput by Removing Pipeline Stages

Normally, one aim of a new hardware design is to reduce the critical path as much as possible to obtain a higher clock rate. Here, however, it makes sense to omit some pipeline stages and use a lower clock rate instead. The reason is that there is a serious imbalance of combinatoric delay in pipeline stages. Mentens, too, uses no pipeline stages in her design for the part that is equivalent to the outer loop of CIOS.

Therefore, another implemented and tested modification is the omitting of pipeline stages as shown in Figure 5.13. It is similar to Mentens design. Only one pipeline stage from the register after multiplication of MTA_1 down to the result register of MTA_2 is configured.

In case of using one single MTA block also omitting of pipeline stages has been tested. Because the single MTA block has to compute twice for one iteration, at least two pipeline states are required.

Alternatively, balancing of pipeline stages would be possible but this results no advantages for throughput.

5.5.2 Adder and Subtractor

The second main element of the AU is one unit for calculation of simple addition, modular addition, and modular subtraction. Modular additions and modular subtractions are required for scalar multiplication. The simple addition is needed for the final addition of carry and safe after Montgomery multiplication.

Both modular variants are implemented as defined in Section 4.3. For modular addition both summands will be added and the result stored in an external register. Then

[3] P is the result register of a DSP48E1, see also Section 3.3.

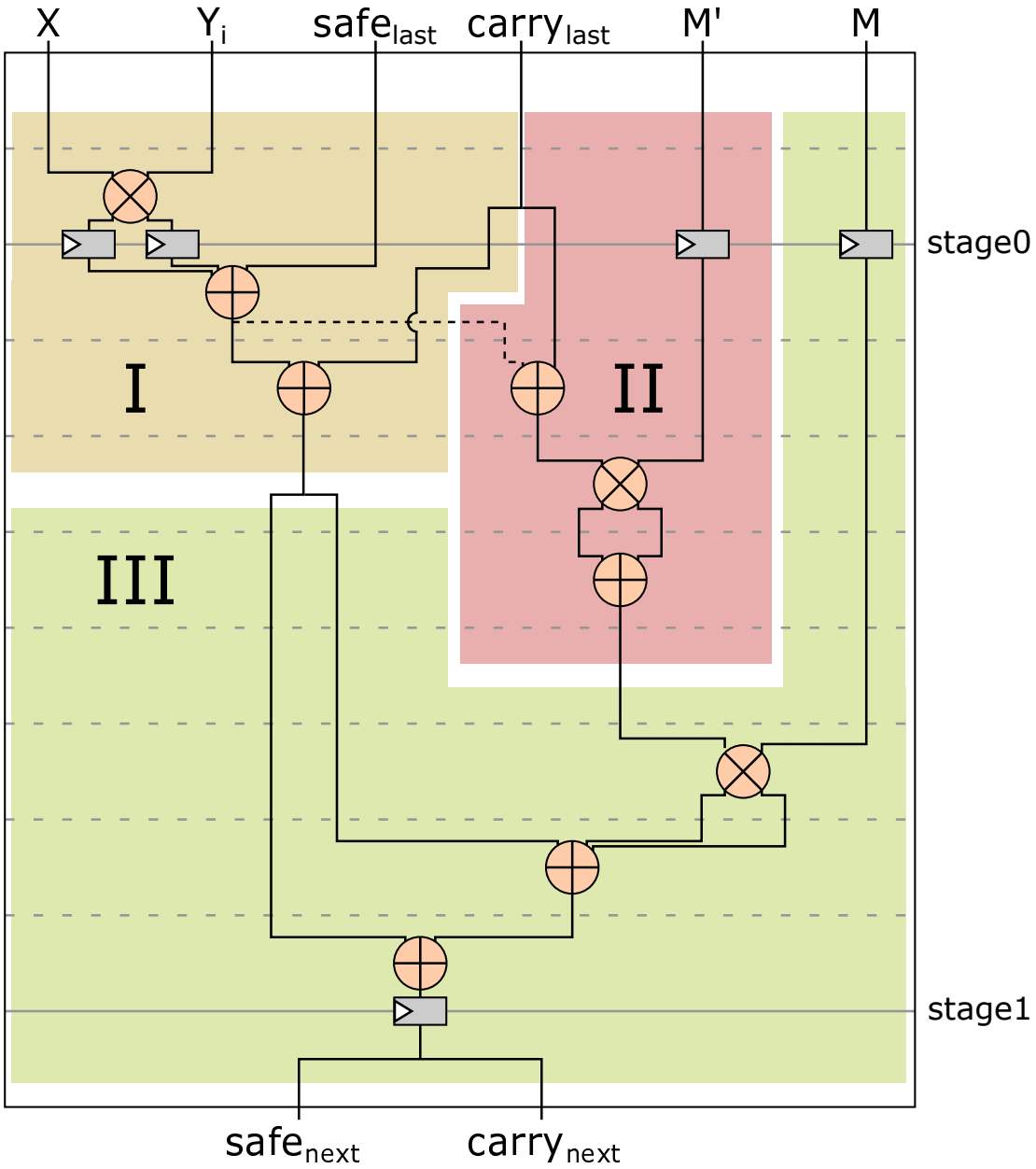

Figure 5.13: Overview over the pipeline stages of Montgomery multiplier using 17×24 bit multipliers with only one pipeline stage.

this result is subtracted by M which corresponds to the conditional subtraction. The FSM considers the most significant bit to verify a negative result after subtraction. The DSPs use two's complement representation, thus, the most significant bit is 1 if the number is negative. If this result is positive, it is output, otherwise the first result. The situation is similar for modular subtraction. If the result of the subtraction is negative the conditional addition with the modulus must be performed. Otherwise, the result can be output.

Figure 5.14 shows the structure of the designed unit. The inputs of the DSPs are controlled by the FSM. One DSP48E1 provides the addition/subtraction of two 48 bit operands plus one further carry bit. Therefore, all DSP48E1 receive 48 bit words as operands. The carry will be passed via the dedicated cascading routing path *CARRYCASCOUT* and *CARRYCASCIN*. The result for modular addition has to be be saved for the conditional step. Therefore, one separate register is required. In the DSPs only the P register is enabled. This seems unusual but the critical path was never met in this unit in all tested AU variants. Thus, clock cycles can be saved. Which register will output is controlled by the FSM. Please note, the final register, depending on the fact which register will output, is the Z register as introduced in Section 5.5. This fact is also indicated in the Figure 5.5 of the complete AU.

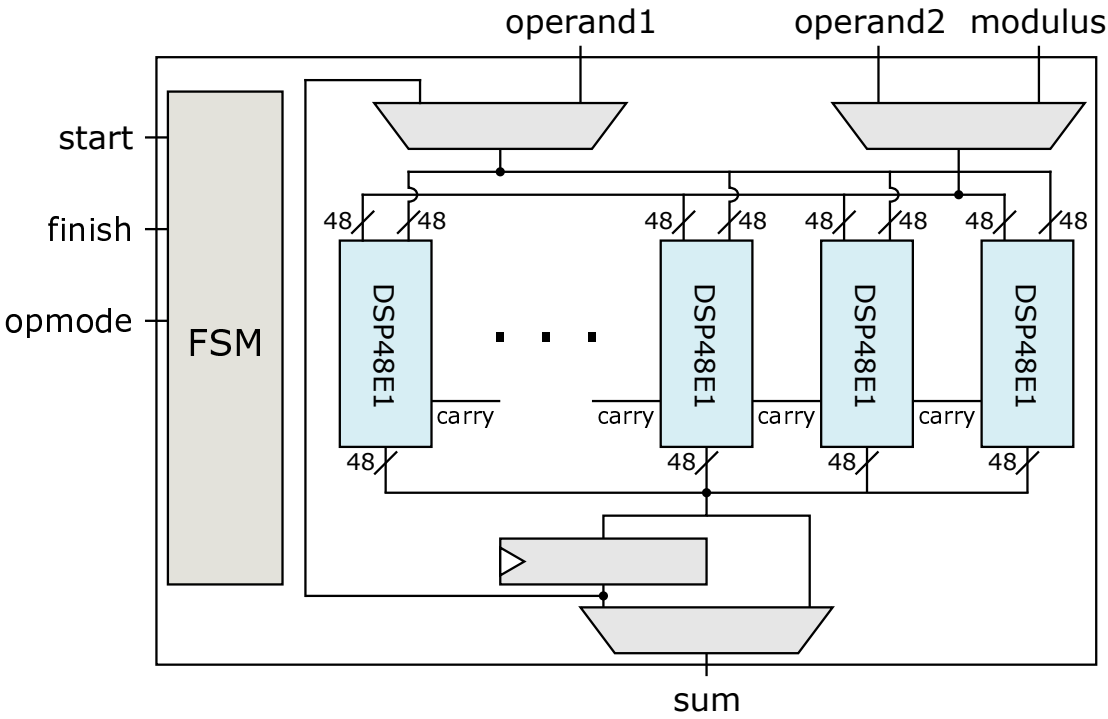

Figure 5.14: Architecture Design of the Addition Unit for Straight Addition and Modular Addition/Subtraction.

6 Evaluation

In this chapter the various designs of the CIOS multiplier and the complete coprocessor are carefully evaluated and compared with previous existing work. The coprocessor has been completely designed, implemented, and validated. The interface of the coprocessor for the soft microprocessor core was not part of this work. Thus, all results about complete ECC scalar multiplication are calculated (using various assumptions) and are not experimentally measured. All timing and resource values are given from Vivado Design Suite 2015.2 of Xilinx after place and route.

Since the most important part of the coprocessor is the CIOS multiplier, this element is evaluated first. After that the results for timing and resource consumption of the actual coprocessor are given.

One further note to the bit lengths tested. Due to the requirements by Rohde & Schwarz SIT GmbH for high assurance, particularly larger bit lengths with additional randomization factor are tested. For randomization length, a random value of 32 bit is considered as proposed in Section 2.6.5. In Section 2.5 the interesting bit lengths of 256, 384, 512, and 521 bit are defined.[1] This leads to the following bit lengths of 288, 416, and 544 which are evaluated in the next sections.

It should be noted that in most publications shorter bit lengths, i. e., 160 or 224 bit without randomization are used. Furthermore, all ECC primitives are implemented there as well. This should be considered when comparing area and timing.

6.1 Simulation and Experimental Validation in Hardware

We implemented different variants of the designed multiplier and coprocessor. All variants were simulated with Aldec Riviera-PRO 2015.06 and examined for correct computation by comparison with a self-implemented reference implementation in a Python script. The described designs were placed and routed with the Vivado Design Suite 2015.2 on the Xilinx KC705 evaluation board which features the Kintex-7 XC7K325T FPGA. After place and route, one step of the parallelized Montgomery ladder was examined on the evaluation board with a debug tool provided by Vivado Design Suite. The correct computation of the coprocessor and correct functionality of memory and all coprocessor control units were validated. The computation outputs were examined by comparison with a self-implemented reference implementation in a Python script, as well.

A validation of one complete ECC scalar multiplication was not possible because of missing ECC primitives. The implementation of these primitives and the interface of the coprocessor for the microprocessor were not part of this work.

[1] For 521 bit (NIST P521), a shorter randomization factor of 23 bit is assumed.

6.2 Timing and Resource Consumption of CIOS Multipliers

Main task of the coprocessor is scalar multiplication which requires many modular multiplications. These modular multiplications are the time consuming tasks in the designed coprocessor. Therefore, this section considers the timing and resource consumption of the different CIOS multipliers implemented.

Section 5.5.1 presented different options to design the Montgomery multiplier in hardware. The following list shows all implemented and tested versions:

⟨0⟩: 17×17 multiplier with Max Number of DSPs and all pipeline stages,

⟨1⟩: 17×24 multiplier with Max Number of DSPs and all pipeline stages,

⟨2⟩: 17×24 multiplier with Max Number of DSPs and one pipeline stage,

⟨3⟩: 17×24 multiplier with Average Number of DSPs and all pipeline stages,

⟨4⟩: 17×24 multiplier with Average Number of DSPs and two pipeline stages, and

⟨5⟩: 17×24 multiplier with Low Number of DSPs and all pipeline stages.

The first implementation ⟨0⟩ is almost an identical implementation of Mentens' design of CIOS multiplier but using 17×17 multiplications as introduced in Section 5.5.1.1. All other tested variants use 17×24 multiplications. The different options are explained in the following:

- *Max Number of DSPs* corresponds to the design shown in Figure 5.6 and Figure 5.12, respectively, without any reduction optimization of the number of DSPs.
- *Average Number of DSPs* indicates that both MTA blocks are consolidated into one block.
- *Low Number of DSPs* further reduces the number of DSPs from the *Average Number of DSPs* option by additional saving of DSPs by multiple use of one DSP inside the MTA block.
- *All pipeline stages* means, that all pipeline stages are enabled as shown in Figure 5.10.
- *One pipeline stage* and *two pipeline stages* are variants in which pipeline stages have been removed as suggested in Figure 5.13. Option ⟨2⟩ is implemented with only one pipeline stage and ⟨4⟩ with two pipeline stages, after each pass of the MTA block.

Variant ⟨3⟩ is an adaptation of ⟨1⟩ by multiple use of DSPs. Variant ⟨5⟩ is an adaptation of ⟨3⟩. Variant ⟨2⟩ is equivalent to ⟨1⟩ but with less pipeline stages, variant ⟨4⟩ to ⟨3⟩ respectively.

All results of timing and resource consumption are given in Table 6.1. All multipliers ⟨0⟩ to ⟨5⟩ were evaluated for bit lengths 288, 416, and 544. *Max. Freq.* gives the theoretical

maximum frequency, extracted of the critical path, for the particular multiplier. *Latency* reports the time for one complete Montgomery multiplication as shown in Algorithm 4.5.1. It is calculated by the maximum frequency and the number of required clock cycles. The other four columns reflect the resource consumption of DSPs, LUTs, FFs, and logic slices. For a graphical overview of all values, Figures 6.1, 6.2, and 6.3 show diagrams of timing and resource consumption. Further diagrams can be found in Appendix C.

As hypothesized, the change from 17×17 to 17×24 multiplication does not only safe DSPs, but also improves the multiplier's performance slightly. Comparing $\langle 0 \rangle$ and $\langle 1 \rangle$ illustrate this finding. Both variants require the same number of clock cycles, but by reducing the number of DSPs the critical path was reduced, too. This results in a higher maximum frequency.

The relationship for timing between design and bit length is very similar in each design. There are some significant differences between the design variants. If all pipeline stages are used, variant $\langle 5 \rangle$ is slightly faster for larger bit lengths. It is particularly interesting that both variants with less pipeline stages are clearly faster in all bit lengths for the same level of hardware resources than their equivalents with all pipeline stages. This results of a wide variety of the critical data paths in full pipelined versions. Variants $\langle 2 \rangle$ and $\langle 4 \rangle$ reduce the execution time by at least 50 % in comparison to their counterparts $\langle 1 \rangle$ and $\langle 3 \rangle$.

In variant $\langle 1 \rangle$ to $\langle 3 \rangle$ the number of used DSPs almost halves. The adaption from $\langle 3 \rangle$ to $\langle 5 \rangle$ halves almost as well. Both adaptations achieve a reduction of used DSPs, but an increase of the number of used LUTs. This applies particularly to the first adaption. Figure 6.3 shows the relationship between used LUTs or FF and required slices. There is an imbalance between FFs per slice and LUTs per slice of variants $\langle 0 \rangle$, $\langle 1 \rangle$, and $\langle 2 \rangle$. These design variants consist mainly of arithmetics, which are computed by DSPs, and registers. This explains the untypical imbalance. The other design variants needs logic for multiplexing. The required logic increases the number of required LUTs and thus normalize the relationship.

Overall, design $\langle 2 \rangle$ is the fastest. If DSPs are to be saved, then $\langle 4 \rangle$ is the preferred variant. If no low clocked variant is sought, the design must find a compromise between usage of DSPs and other resources.

6.3 Conditions for the Evaluation of the Coprocessor

Since ECC primitives and the interface of the coprocessor for the microprocessor are not available, the assumptions for estimating the timing of scalar multiplication must be defined. The coprocessor can only be controlled by opcodes but for scalar multiplication in ECC different operations have to be performed. The opcode is given by two concatenated subopcodes with the following definition. For convenience the single operations are abbreviated:

$$\circ_1 || \circ_2 \text{ , where } \circ_1, \circ_2 \in \{\mathrm{N}, \mathrm{M}, \mathrm{A}, \mathrm{S}, \mathrm{L}\}$$

with N = NOP, M = MontMul, A = ModAdd or ModSub, S = Store, L = Load. For simplicity addition and subtraction are subsumed.

Table 6.1: Overview of hardware consumption and latency of the different implemented CIOS multipliers (different bit lengths). Maximum frequencies are extracted from timing analyzer of Vivado. Latency is computed from the provided frequency and the number of clock cycles required.

Design No.	Bit Length	Max. Freq. [MHz]	Latency [ns]	# DSPs	# LUTs	# FFs	# Slices
⟨0⟩	288	339.9	497.2	77	116	898	250
	416	304.8	800.6	112	139	1 285	413
	544	295.9	1 047.8	143	168	1 672	474
⟨1⟩	288	383.0	441.3	57	117	898	186
	416	300.1	813.0	84	187	1 285	430
	544	315.8	981.8	107	185	1 672	545
⟨2⟩	288	88.4	226.3	57	111	891	246
	416	89.4	324.4	84	178	1 276	424
	544	86.9	414.3	107	180	1 660	419
⟨3⟩	288	295.0	572.9	33	1 127	898	448
	416	315.1	774.5	48	1 520	1 285	727
	544	303.1	1 022.7	61	1 889	1 672	929
⟨4⟩	288	130.2	299.5	33	1 008	901	385
	416	128.6	435.4	48	1 470	1 289	693
	544	122.8	570.1	61	1 849	1 678	820
⟨5⟩	288	352.4	479.6	20	1 384	898	475
	416	351.6	693.9	29	1 907	1 285	831
	544	340.5	910.5	37	2 414	1 672	946

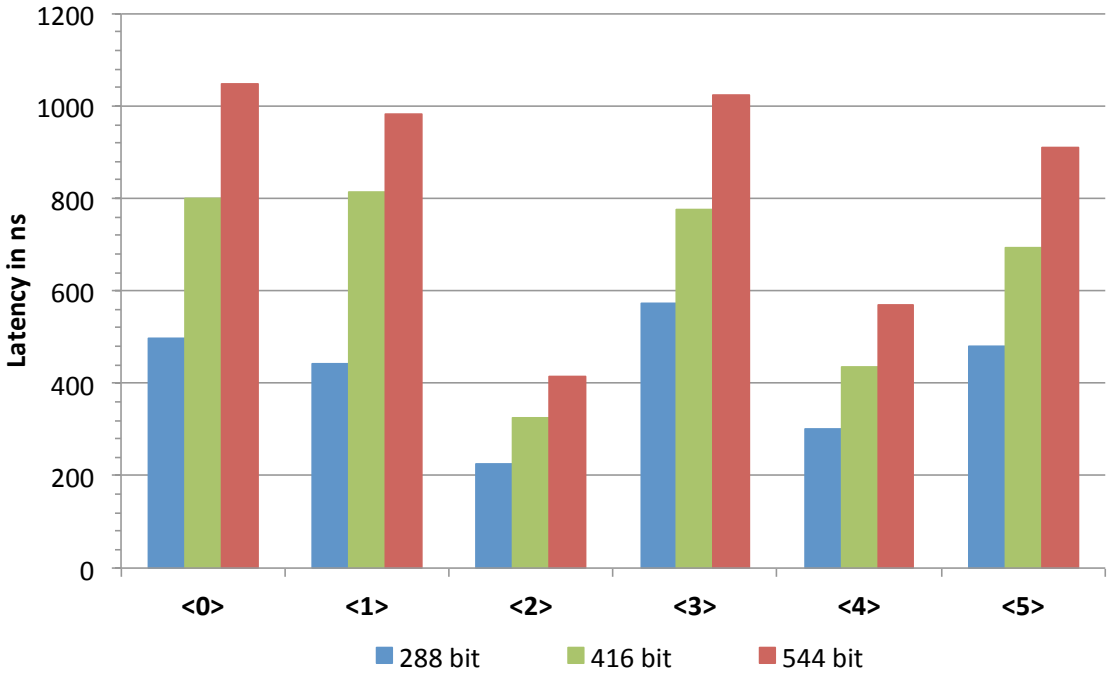

Figure 6.1: Bar chart for illustration of the latencies of the different multiplier variants. Every version has been tested with 288, 416, and 544 bit.

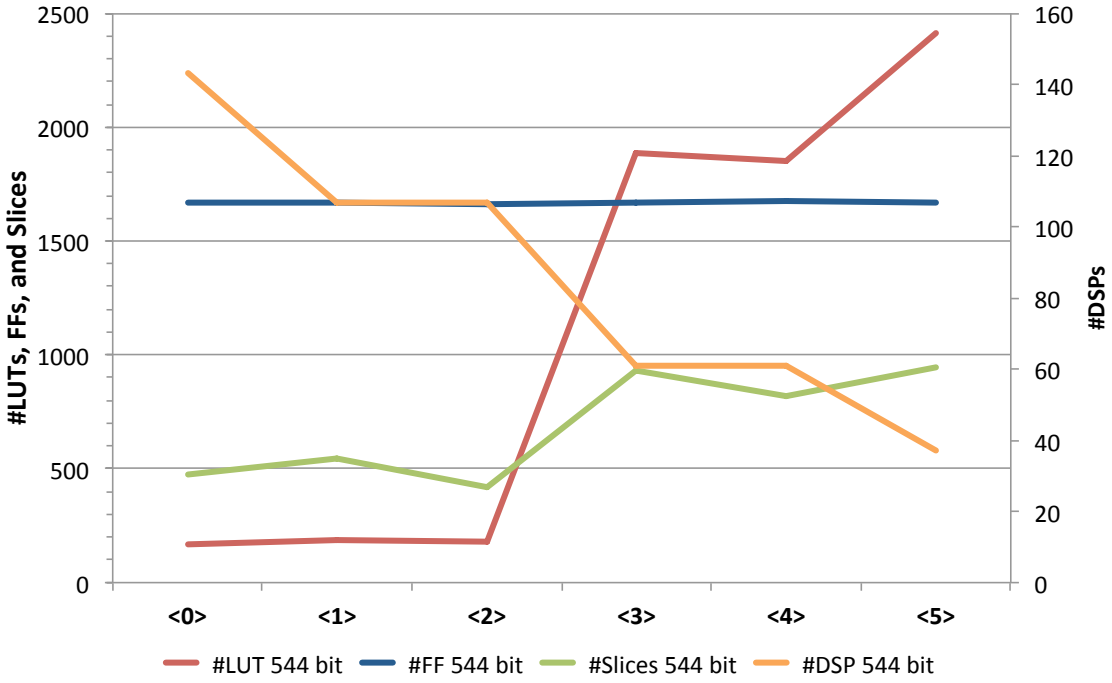

Figure 6.2: Line graph shows the resource consumption of all designed multipliers (544 bit).

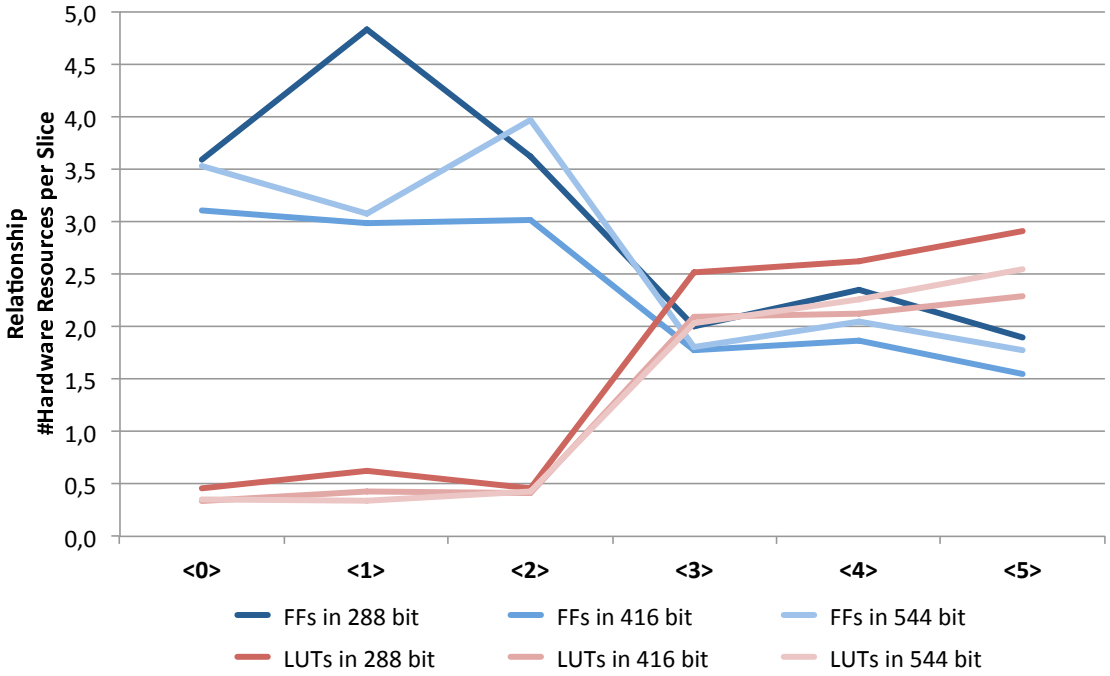

Figure 6.3: Line graph shows the relationship between used slices and used FFs and LUTs of all designed multipliers (different bit lengths).

The most important ECC operation is scalar multiplication, i. e., get the affine coordinate of kP. Which coordinates are really required differs between the various algorithms. For ECDSA signatures both coordinates are required. In other cases like ECIES or ECDH only the x-coordinate might be required. Thus, in this evaluation a distinction is made for both cases.

Our goal is to give an estimate for the number of arithmetic operations, especially modular multiplication, for a complete scalar multiplication using our coprocessor.

In the following there is an overview of the individual steps which are required for a complete point multiplication kP with $P = (x, y)$. Input and output are always affine coordinates. The input of the coprocessor are

- $P = (x, y)$ affine together with modulus M of curve,
- randomized projective coordinate z, and
- M' and R^2 for Montgomery Multiplication.

We assume that all required values are already stored in BRAM.

1. **Convert x and z to Montgomery representation**
   The first step is the conversion into Montgomery representation so that the Montgomery multipliers can be used. It is required to load M and M' into both AUs and after that to convert with

$$xR = \text{MontMul}(x, R^2, M) \quad \text{and} \quad zR = \text{MontMul}(z, R^2, M).$$

1 LL, 1 MM, and 1 SS are required.

2. **Convert affine x_1R, and if required yR, to Projective Montgomery X_1R (and Y_1R)**
   After that, the conversion of x_1 is required with

   $$X_1 = x_1 \cdot Z \bmod M.$$

   In the simplest case Z is 1. Because of randomization Z must be a random value with $1 < Z < M$. So here 1 MN and 1 SN are needed. Note, if y value is also required instead of the NOP, subopcode y can be converted into yR nearly for free in the second AU (1 MM and 1 SS).

3. **Montgomery ladder**
   Now all essential preconditions are fulfilled to calculate the Montgomery ladder like Algorithm 4.2.3. Depending on the different bit lengths the execution time varies, the bit length of k, usually in the order of the bit length of M (Hasse's Theorem), indicates the number of loop iterations. We obtain the number of operations by counting the steps in Algorithm 4.1.1. In general, 9 MM, 1 MN, 6 AA, 2 AN, 4 SS, and 10 SN are necessary for one iteration.

4. **Convert x_1R, and if required yR, back to affine representation**
   The conversion of X_1R consists of one simple multiplication and one expensive inversion. Since M is not a secret value the inversion are calculated by using Fermat's Little Theorem and square-and-multiply algorithm

   $$(ZR)^{-1} = (ZR)^{M-2} \bmod M.$$

   Then x_1R can be calculated by

   $$x_1R = X_1R \cdot (ZR)^{-1} \bmod M.$$

   For uniformly distributed bits this requires on average $1.5 \cdot |M|$ times MN in addition to another MN.

   If yR is required, Formula (2.14) must be calculated, which needs an inversion. This inversion is possible to compute almost without additional cost, because one of the AU idles during the inversion of ZR. Therefore timing is similar to the first variant, only some further operations are needed for the remaining calculation of Formula 2.14. This process needs approximately $1.5 \cdot |M| \times$MM+5 MM+3 AA+8 SS operations.

5. **Reverse transformation to x_1, and if required y, from Montgomery representation**
   The reverse transformation of a Montgomery representation needs up to two multiplication operations.

   $$x_1 = \text{MontMul}(x_1R, 1, M) \quad \text{and if } y \text{ is needed} \quad y = \text{MontMul}(yR, 1, M)$$

   So 1 MN and 1 SN or 1 MM and 1 SS respectively are required.

These steps form the basis for the evaluation of the timing estimates. Totally,

$$|k|\cdot(9\cdot\mathrm{MM}+1\cdot\mathrm{MN}+6\cdot\mathrm{AA}+2\cdot\mathrm{AN}+4\cdot\mathrm{SS}+10\cdot\mathrm{SN})+1.5\cdot\mathrm{MM}+1\cdot\mathrm{LL}+9\cdot\mathrm{MM}+3\cdot\mathrm{AA}+11\cdot\mathrm{SS}$$

operations are estimated in total for a complete scalar multiplication.

6.4 Timing and Resource Consumption of Coprocessor

Now the timing of a scalar multiplication using the coprocessor can be examined. The coprocessor is implemented once. Only the CIOS multiplier is replaced for evaluation. Thus, all resources of the coprocessor remain the same in all cases. The different multipliers are $\langle 1\rangle$ to $\langle 5\rangle$ from Section 6.2. Multiplier $\langle 0\rangle$ was not continued for implementation into the final coprocessor, due to its different word lengths and better timing results and resource consumption of its counterpart $\langle 1\rangle$.

All results of timing and resource consumption are given in Table 6.2 with variants of the used CIOS multiplier $\langle 1\rangle$ to $\langle 5\rangle$. Tested bit lengths are 288, 416, and 544, as well. *Max. Freq.* gives the theoretical maximum frequency, extracted of the critical path, for the particular coprocessor. *Latency* reports the duration of one complete scalar multiplication as defined in Section 6.3. It results from the maximal frequency and the required clock cycles in column *#Clock Cycles*.We differentiate between a scalar point multiplication with calculation of x and y, and calculation of only x. The other four columns reflect the resource consumption of DSPs, LUTs, FFs, and logic slices. For a graphical overview of all values, Figures 6.4, 6.5, and 6.6 show our results as graphs. Further diagrams can be found in Appendix C.

Our results show two key findings. Firstly, there is only a minor difference in latency between the five alternative multipliers when calculating x only or x and y. Thus, the overhead of recovering the y-coordinate can be neglected. On the other hand, the advantage of removing pipeline stages is considerably lower, especially for 288 bit. Only the other two bit lengths benefit more from removing of pipeline stages. If variants with all pipeline stages are compared, variant $\langle 1\rangle$ is slightly faster for 288 bit compared to variants $\langle 3\rangle$ and $\langle 5\rangle$. For 544 bit the difference is more significant. If variants with lower clock frequencies are sought, variants with smaller bit lengths do not differ as strongly as larger bit lengths. For example, variant $\langle 4\rangle$ with 288 bit (2 343.3 µs) is only around 32 % faster than its equivalent $\langle 3\rangle$ (3 434.6 µs), and around 20 % faster than $\langle 1\rangle$ (2 938.1 µs), the fastest variant with all pipeline stages. For 544 bit it results 45 % faster than $\langle 3\rangle$ and 42 % faster than $\langle 1\rangle$.

The resource consumption for the complete coprocessor shows the same behavior as only for the CIOS multipliers. The number of FFs differs slightly between variants with all and with less pipeline stages. The required overhead of LUTs to control the coprocessor shows an interesting behavior. While the number of LUTs per slice increase significantly, the results for CIOS multipliers only decrease. Almost all LUTs are associated to the coprocessor design for variants $\langle 0\rangle$ and $\langle 1\rangle$. As mentioned, it increases if variants of CIOS multipliers are used with reduced numbers of DSPs.

Table 6.2: Overview of hardware consumption and latency of the complete coprocessor with different multipliers (different bit lengths). Maximum frequencies are extracted from timing analyzer of Vivado. The number of clock cycles for one complete point multiplication is estimated as shown in Section 6.3. Latency is computed from the provided frequency and the number of clock cycles required. Note, both clock cycles and latency, are given once for point multiplication with needed calculating of y and once without calculating y.

Design No.	Bit Length	Max. Freq. [MHz]	Latency x only [µs]	Latency x and y [µs]	# Clock Cycles x only	# Clock Cycles x and y	# DSPs	# LUTs	# FFs	# Slices
	288	248.5	2 938.1	2 958.1	730 155	735 111	116	3 929	2 798	1 432
⟨1⟩	416	202.1	7 072.8	7 107.8	1 429 721	1 436 785	168	5 445	3 770	1 938
	544	222.5	10 447.5	10 492.0	2 324 254	2 334 144	214	6 966	4 844	2 349
	288	89.6	2 505.7	2 549.2	224 551	228 449	116	3 837	2 284	1 681
⟨2⟩	416	81.3	4 601.3	4 669.6	374 245	379 793	168	5 364	3 154	2 305
	544	80.8	7 118.1	7 216.6	575 018	582 970	214	7 416	3 922	2 439
	288	212.6	3 434.6	3 458.0	730 155	735 111	68	6 344	2 798	2 104
⟨3⟩	416	205.4	6 959.9	6 994.3	1 429 721	1 436 785	96	8 848	3 770	2 663
	544	212.4	10 944.9	10 991.5	2 324 254	2 334 144	122	11 813	4 844	3 425
	288	121.3	2 343.3	2 376.4	284 239	288 263	68	5 628	2 296	1 822
⟨4⟩	416	121.2	4 123.4	4 170.7	499 929	505 659	96	8 047	3 182	2 676
	544	130.0	6 012.1	6 075.0	781 598	789 781	122	10 400	3 934	3 256
	288	225.3	3 241.2	3 263.2	730 155	735 111	42	6 039	2 798	1 630
⟨5⟩	416	215.4	6 638.2	6 671.0	1 429 721	1 436 785	58	8 459	3 770	2 379
	544	198.2	11 728.2	11 778.1	2 324 254	2 334 144	74	10 902	4 844	3 327

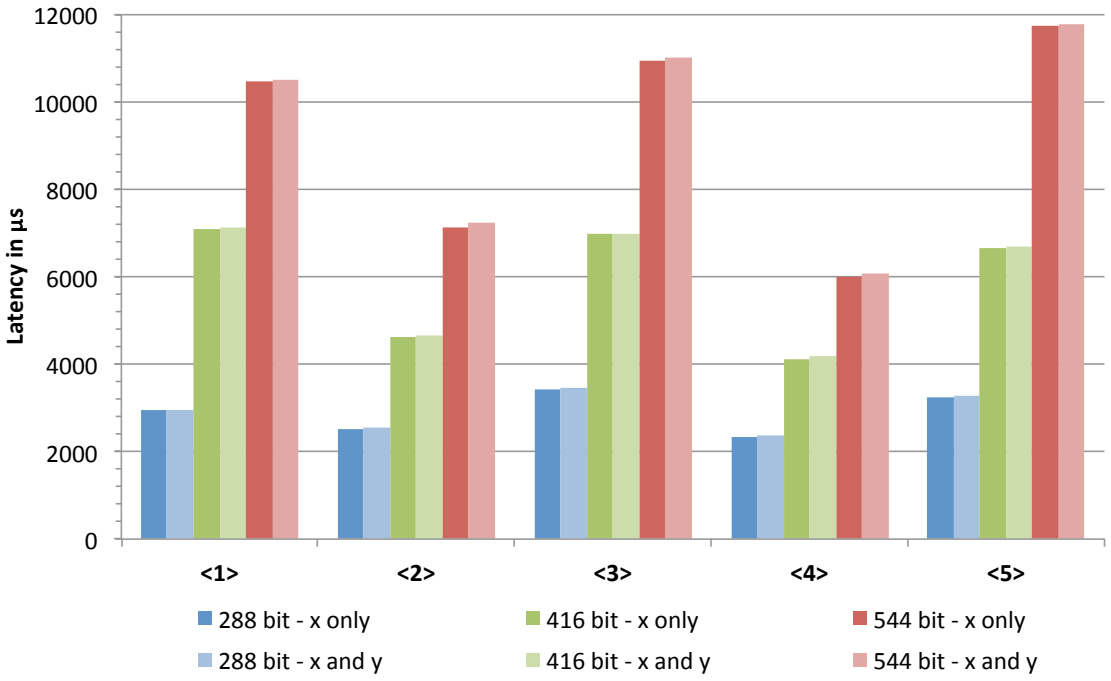

Figure 6.4: Bar chart for illustration of the latencies of the coprocessor with different multiplier variants used. Every version has been tested with 288, 416, and 544 bit.

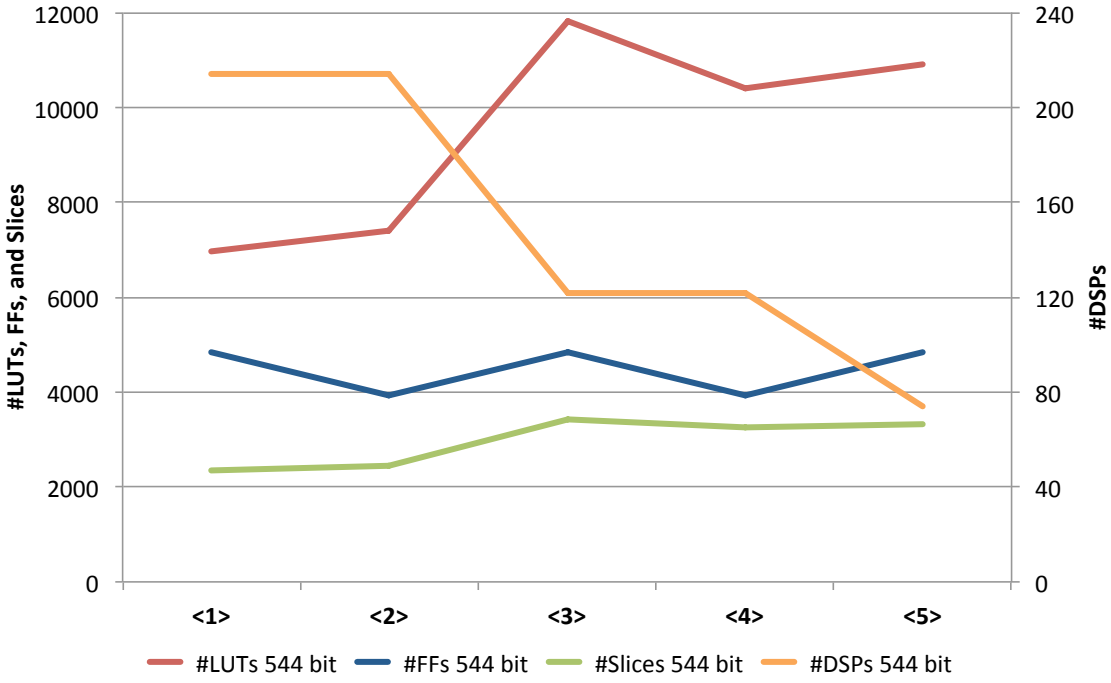

Figure 6.5: Line graph shows the resource consumption of the coprocessor with different multiplier variants used (544 bit).

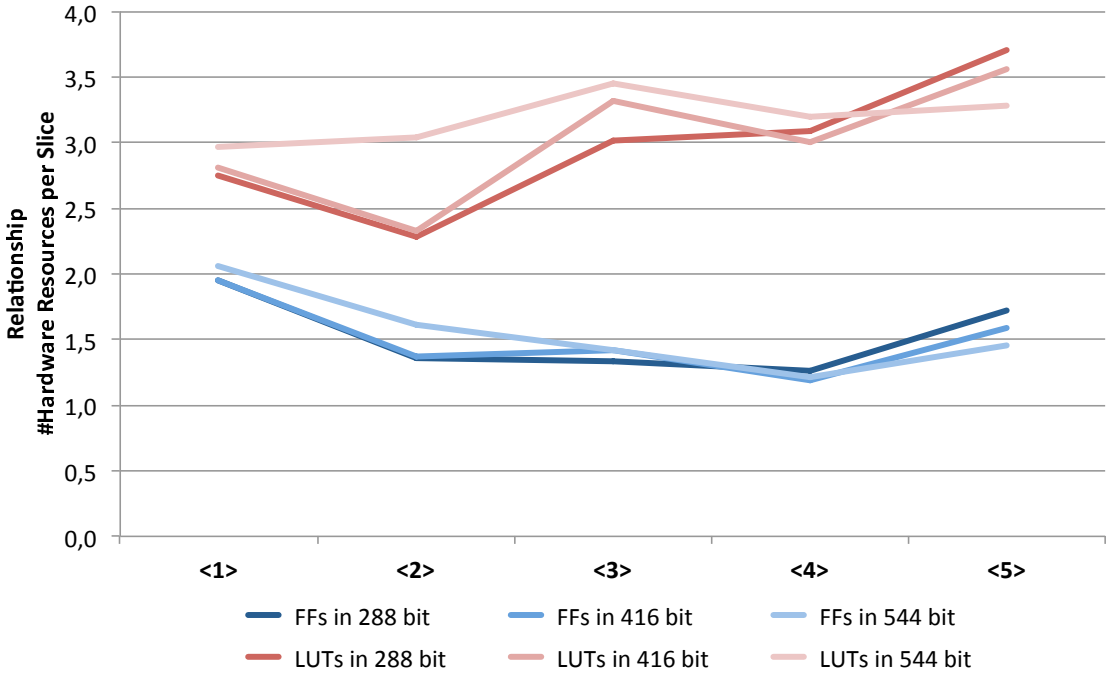

Figure 6.6: Line graph shows the relationship between used slices and used FFs and LUTs of the coprocessor with different multiplier variants used (different bit lengths).

There is no simple answer to which variant is the preferred one and depends on the use case. If a low clock rate is acceptable then ⟨4⟩ is preferable. Otherwise, it depends on the number of DSPs available. Variant ⟨1⟩ with the maximum number of DSPs is slightly faster than ⟨3⟩ and ⟨5⟩. When there are less DSPs available, variants with fewer DSPs must be selected.

One note to these results. The real implementation may differ slightly, because some special cases were neglected. For example, there are no distinctions made in modular addition and modular subtraction for conditional subtraction and addition of M.

6.5 Comparison to Existing Work

For a more precise evaluation it is helpful to compare with existing work. Three publications have been selected for comparison which have followed similar requirements and aims. Comparable works are [Men07], [Gui10], and [MLPJ13]. Other papers are not listed because of missing either countermeasures against SCA or implementations for curves over $\mathbb{F}_{2^m}$.

Table 6.3 reports some reference values. The first four lines with ⟨1⟩, ⟨2⟩, ⟨3⟩, and ⟨4⟩ are different variants of the design in this work. Our design uses 288 bit instead of 256 bit due to 32 bit randomization buffer. Furthermore, comparing to other designs, we assume the logic for ECC primitives is not implemented in the coprocessor. Only the

control by opcodes is provided. This should be considered when comparing latencies and area consumption.

The table shows that the resource consumption of logic units in our design is lower or equivalent to the other comparable works. The usage of DSPs is much higher which results from the motivation to use many DSPs than other resources as much as possible. Please note, the comparison for resource consumption with [Gui10] is limited because the work is based on a FPGA of different manufacturer. Frequencies of variants with less pipeline stages $\langle 2 \rangle$ and $\langle 4 \rangle$ are comparable to [Men07]. The frequency of the high clocked variants are also comparable to [Gui10] and [MLPJ13].

To compare timings of our design with Mentens' design, we assume that the latency for scalar multiplication grows cubic with the bit length. Thus, we get estimate latencies of $2.94\,\text{ms} \times (256/288)^3 = 2.06\,\text{ms}$ for $\langle 1 \rangle$, 1.76 ms for $\langle 2 \rangle$, 2.41 ms for $\langle 3 \rangle$, and 1.64 ms for $\langle 4 \rangle$. Except for $\langle 3 \rangle$, all variants of our design are faster than the reported results in [Men07].

The comparison with the other investigations shows that our coprocessor design is significantly slower. The reason for this is the vastly different arithmetic implementations. In [Gui10] RNS multipliers are implemented unlike desired here (flexibility) and in [MLPJ13] they use a windowing method for scalar multiplication. The windowing method requires precomputations and further RAM which was undesirable for our use case.

Table 6.3: Comparison between the designed coprocessor in this work and other existing designs.

	Curve	Device	# Logic Units	# DSPs	Frequency [MHz]	Delay [ms]
$\langle 1 \rangle$	288 any	Kintex-7	1 432 Slices	116	248	2.94
$\langle 2 \rangle$	288 any	Kintex-7	1 681 Slices	116	89	2.50
$\langle 3 \rangle$	288 any	Kintex-7	2 104 Slices	68	212	3.43
$\langle 4 \rangle$	288 any	Kintex-7	1 822 Slices	68	121	2.34
[Men07]	256 any	Virtex-2 Pro	3 529 Slices	67	67	2.35
[Gui10]	256 any	Stratix II	9 177 ALM	157	157	0.68
[MLPJ13]	256 any	Virtex-5	1 725 Slices	37	291	0.38

7 Conclusion

The aim of this work was the design and implementation of an FPGA-based arithmetic coprocessor for scalar multiplication in elliptic curves. The coprocessor can also be used to perform general arithmetic operations, like modular multiplication, addition, and subtraction. However, it was designed especially to be used in Elliptic Curve Cryptography (ECC). Different design constraints and design principles had to be met:

- Flexibility and security is much more important than speed and area.
- The coprocessor shall handle elliptic curves over $\mathbb{F}_p$, $p > 3$, prime with a verifiable pseudo-random prime structure, so not only NIST curves should be possible, but for example also Brainpool curves.
- Different bit lengths of p must be supported.
- Use DSPs to avoid other resource consumption in the FPGA.
- Design should be protected/protectable against all type of SCAs. Especially, the use of an additional randomization buffer should be possible.

All of these design constraints have been satisfied.

The design uses reduced projective coordinates with $(\mathcal{X}, \mathcal{Z})$ representation. This has several advantages: No inversion is required for point addition and doubling, only the back transformation from projective to affine coordinates requires two modular inversions. In contrast to the standard projective representation $(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ only two rather than three coordinates are used for computation. This saves registers and arithmetic operations.

Usage of reduced projective coordinates is very closely related to the Montgomery ladder by Brier and Joye [BJ02] for curves in short Weierstrass form. A Montgomery ladder is a countermeasure against Timing Analysis (TA) and Simple Power Analysis (SPA). More specifically, the Montgomery ladder for parallel scalar multiplication by Fischer et al. [FGKS02] was used for the design of the coprocessor. This parallel version requires only the time for 10 modular multiplications and utilizes the resource of the FPGA more efficiently.

The coprocessor consists mainly of data memory, instruction FIFO, and Arithmetic Units (AUs). BRAM is used for data exchange between processor and coprocessor. The coprocessor is controlled by opcodes which the coprocessor writes into the FIFO. It organizes the queue of opcodes and dequeues the first opcode to the FSM. Both, BRAM and FIFO, can operate in asynchronous mode. Thus, the coprocessor can operate with a separate clock frequency. Two AUs are used for a parallel variant of scalar multiplication

by Fischer et al. [FGKS02]. Each AU consists of a modular multiplier and a modular adder/subtractor. All computations are performed by using DSPs.

Modular multiplications are time-consuming. Therefore, one main task of the work was to design an efficient modular multiplier. Due to the demand to support flexible bit lenghts, Montgomery multiplication computed with the CIOS algorithm by Koç et al. [Koç95] is used. Mentens [Men07] implemented the improved variant of the CIOS algorithm, see [Wal99]. This improved Montgomery multiplication is more robust with respect to Timing Analysis. Therefore, Mentens hardware design of the improved CIOS algorithm was an important reference for this work.

Compared to Mentens' PhD thesis, a newer version of FPGA was used. Therefore, the design was modified for better utilization of DSPs, e. g. use 17×24 rather than 16×16 multiplication. Further, modifications to this basic design like multiple use of DSPs (multiplexing) and reduction of pipeline stages were realized and evaluated.

The coprocessor was evaluated with different variants of designed CIOS multipliers: The first variant of the CIOS multiplier uses 17×17 multiplication in the DSP, very similar to Mentens' design. The next step was to exploit the features of new DSPs by using 17×24 multiplication in the DSP. Building on this, further optimizations are evaluated with 288 bit, 416 bit, and 544 bit. These bit lengths are not typical for ECC. They are caused by an additional 32 bit buffer, usable, for example, for the randomization of the prime p. Comparing all variants, the modifications with less pipeline stages are up to 55 % faster than their equivalents with all pipeline stages, due to imbalanced delay in the pipeline stages.

In comparison with the work of Mentens our design is faster. The comparison is not trivial because we use 288 bit instead of 256 bit. If we assume that the latency for scalar multiplication grows cubically with the bit length, it can be estimated that our simplest design $\langle 1 \rangle$ requires 2.06 ms[1] for 256 bit. Mentens' design for 256 bit requires 2.35 ms. The comparison of resource consumption is difficult too, because Mentens implemented a complete coprocessor with all ECC primitives. Our coprocessor currently provides only the required finite field arithmetic for ECC. Instead of a large FSM it was decided by Rohde & Schwarz SIT GmbH to use a soft microprocessor core which is more flexibly adjustable to changing requirements.

Other efficient designs often use precomputation [MLPJ13] or other efficient arithmetic procedures for modular multiplication, like RNS in [Gui10]. These designs do not satisfy the required flexibility.

For future work, some possible improvements are proposed. Setting of pipeline stages has been done based on intuition. For better timing results they must be set more precisely. Especially the variants with low clock frequency and higher bit length will benefit particularly. There is still some room for optimization of the Finite State Machine (FSM) of the coprocessor by more efficient opcode processing.

Following our work, a host interface from a soft microprocessor core to this coprocessor has to be implemented. This will be followed by the implementation of various ECC primitives and protocols in software.

[1] $2.94\,\text{ms} \times (256/288)^3$

A Acronyms

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ATM	Add-Then-Multiply
AU	Arithmetic Unit
BRAM	Block-RAM
CIOS	Coarsely Integrated Operand Scanning
CLB	Configurable Logic Block
CSA	Carry-Safe Adder
DPA	Differential Power Analysis
DSA	Digital Signature Algorithm
DSP	Digital Signal Processing Block
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
EEA	Extended Euclidean Algorithm
FF	Flip-Flop
FIFO	First-In-First-Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GTP	Gigabit Transceiver
LUT	Look-Up Table
MAC	Message Authentication Code
MTA	Multiply-Then-Add
NIST	National Institute of Standards and Technology
RNS	Residue Number System
SCA	Side-Channel Analysis
SOS	Separated Operand Scanning
SPA	Simple Power Analysis

STS	Station-to-Station
TA	Timing Analysis
TLS	Transport Layer Security

B Explanation of Coprocessor Opcodes

One opcode consists of two subopcodes with each 2 Byte as shown in Figure 5.3 on page 43. The following rules are defined for the different operations. Note, if a Byte is symbolized with `X` that this Byte will ignored. R␣ presents a regular R register and Z␣ a result register.

NOP:

[`0x0XXX`] – No operation.

Montgomery Multiplication:

[`0x10VW`] – MontMul(R`V`, R`W`) with `V,W` $\in \{0, \ldots, 0xF\}$.
[`0x11VW`] – MontMul(R`V`, Z`W`) with `V` $\in \{0, \ldots, 0xF\}$ and `W` $\in \{1, 2\}$.
[`0x14VW`] – MontMul(Z`V`, R`W`) with `V` $\in \{1, 2\}$ and `W` $\in \{0, \ldots, 0xF\}$.
[`0x15VW`] – MontMul(Z`V`, Z`W`) with `V,W` $\in \{1, 2\}$.

Modular Addition:

[`0x20VW`] – R`V` + R`W` mod MReg with `V,W` $\in \{0, \ldots, 0xF\}$.
[`0x21VW`] – R`V` + Z`W` mod MReg with `V` $\in \{0, \ldots, 0xF\}$ and `W` $\in \{1, 2\}$.
[`0x24VW`] – Z`V` + R`W` mod MReg with `V` $\in \{1, 2\}$ and `W` $\in \{0, \ldots, 0xF\}$.
[`0x25VW`] – Z`V` + Z`W` mod MReg with `V,W` $\in \{1, 2\}$.

Modular Subtraction:

[`0x30VW`] – R`V` − R`W` mod MReg with `V,W` $\in \{0, \ldots, 0xF\}$.
[`0x31VW`] – R`V` − Z`W` mod MReg with `V` $\in \{0, \ldots, 0xF\}$ and `W` $\in \{1, 2\}$.
[`0x34VW`] – Z`V` − R`W` mod MReg with `V` $\in \{1, 2\}$ and `W` $\in \{0, \ldots, 0xF\}$.
[`0x35VW`] – Z`V` − Z`W` mod MReg with `V,W` $\in \{1, 2\}$.

Simple Addition:

[`0x40VW`] – R`V` + R`W` with `V,W` $\in \{0, \ldots, 0xF\}$.
[`0x41VW`] – R`V` + Z`W` with `V` $\in \{0, \ldots, 0xF\}$ and `W` $\in \{1, 2\}$.
[`0x44VW`] – Z`V` + R`W` with `V` $\in \{1, 2\}$ and `W` $\in \{0, \ldots, 0xF\}$.
[`0x45VW`] – Z`V` + Z`W` with `V,W` $\in \{1, 2\}$.

Load:

[0x50V0] – Load RV to XReg with V $\in \{0, \ldots, 0xF\}$.
[0x54V0] – Load ZV to XReg with V $\in \{1, 2\}$.
[0x50V1] – Load RV to YReg with V $\in \{0, \ldots, 0xF\}$.
[0x54V1] – Load ZV to YReg with V $\in \{1, 2\}$.
[0x50V2] – Load RV to MReg with V $\in \{0, \ldots, 0xF\}$.
[0x54V2] – Load ZV to MReg with V $\in \{1, 2\}$.
[0x50V3] – Load RV to M'Reg with V $\in \{0, \ldots, 0xF\}$.
[0x54V3] – Load ZV to M'Reg with V $\in \{1, 2\}$.

Store:

Note, it is not possible to store in special registers R0 and R1!
[0x6XXV] – Store ZReg to RV with V $\in \{2, \ldots, 0xF\}$.

Reset:

[0xFXXX] – All registers inside of both AUs are reseted. Other subopcode is ignored.

Examples:

[0x242C5022] – AU1 calculates Z2 + R12 mod M and AU2 loads R2 into its MReg.
[0x0000600F] – AU1 do nothing and AU2 store ZReg to R15.
[0x10231511] – AU1 calculates MontMul(R2, R3) and AU2 calculates MontMul(Z1, Z1).

C Further Diagrams for Evaluation

This chapter shows some further diagrams of evaluation of hardware consumption in chapter 6.

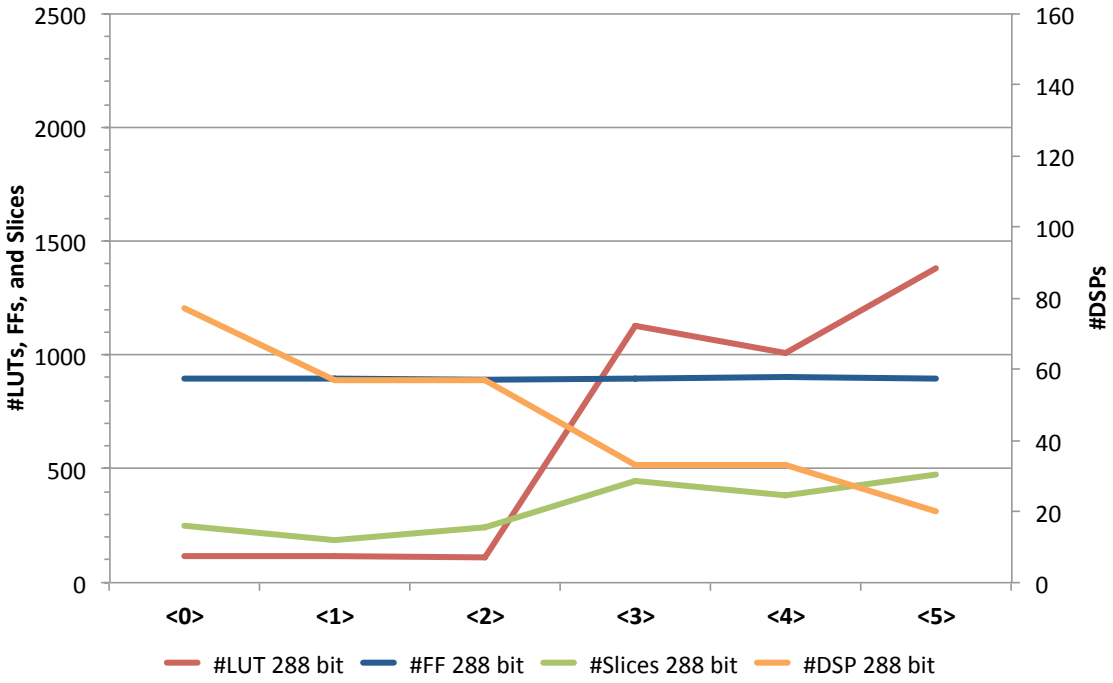

Figure C.1: Line graph shows the resource consumption of all designed multipliers (288 bit).

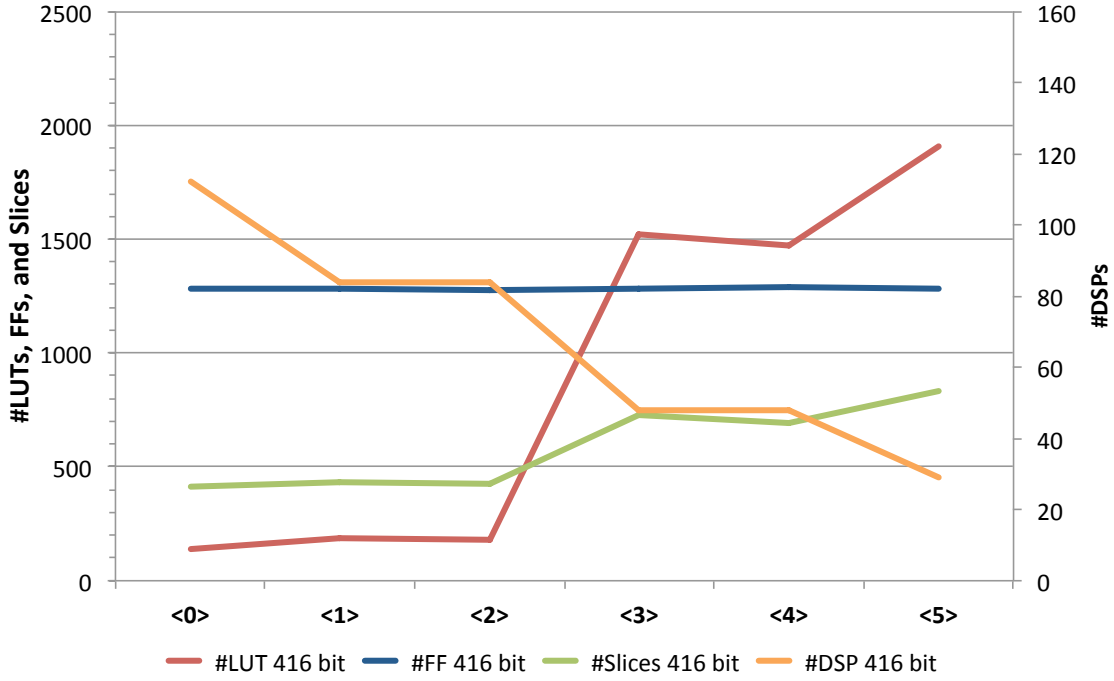

Figure C.2: Line graph shows the resource consumption of all designed multipliers (416 bit).

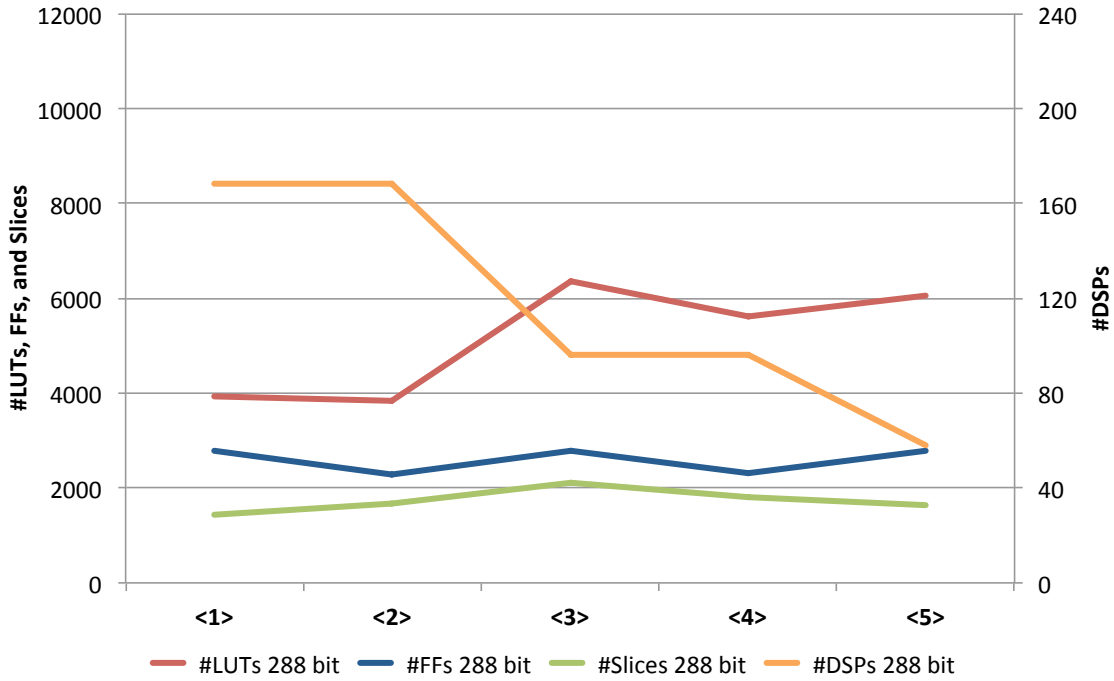

Figure C.3: Line graph shows the resource consumption of the coprocessor with different multiplier variants used (288 bit).

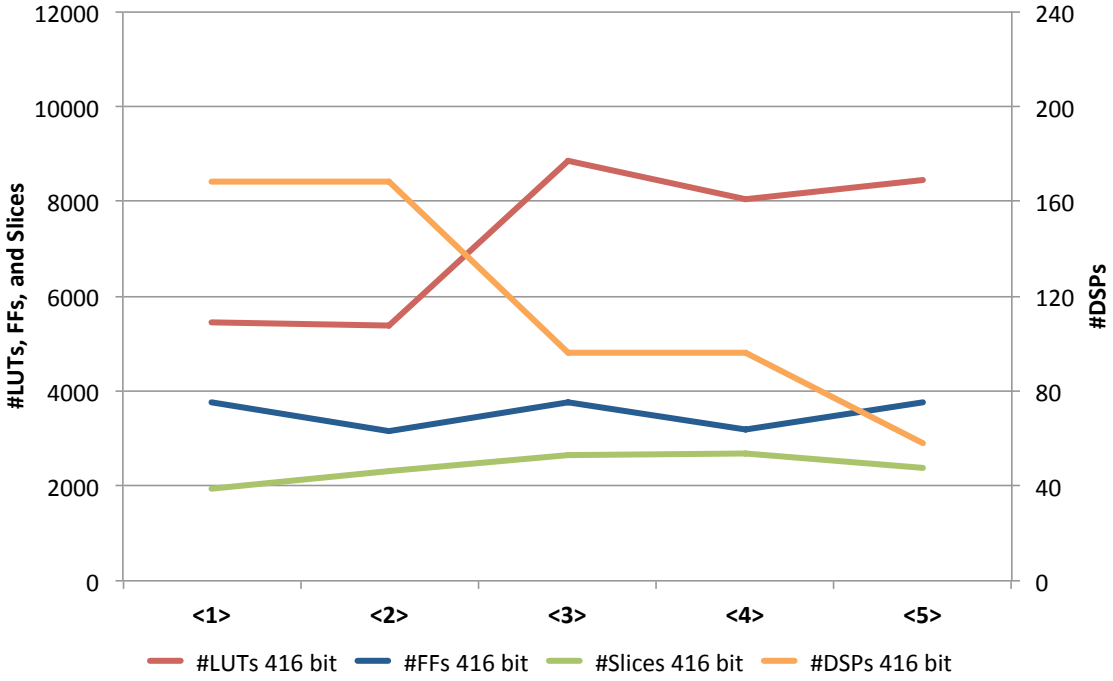

Figure C.4: Line graph shows the resource consumption of the coprocessor with different multiplier variants used (416 bit).

List of Figures

List of Tables

List of Algorithms

Bibliography

[ANSI-X9.62:98] ANSI X9.62-1998. Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). Standard, American National Standards Institute, Washington, D.C., United States, 1998.

[ANSI-X9.63:11] ANSI X9.63-2011. Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography. Standard, American National Standards Institute, Washington, D.C., United States, 2011.

[BB03] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[BCC+14] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: a white paper for the black hat. *IACR Cryptology ePrint Archive*, 2014:571, 2014.

[BJ02] Éric Brier and Marc Joye. Weierstraß Elliptic Curves and Side-Channel Attacks. In David Naccache and Pascal Paillier, editors, *Proceedings of PKC 2002*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer Berlin Heidelberg, 2002.

[BL15a] Daniel J. Bernstein and Tanja Lange. Explicit-Formulas Database. `https://www.hyperelliptic.org/EFD/`, 2015. accessed 19 May 2015.

[BL15b] Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. `http://safecurves.cr.yp.to/`, 2015. accessed 10 Juni 2015.

[Bro10] Daniel R. L. Brown. SEC 2: Recommended Elliptic Curve Domain Parameters. Standard, Certicom Research, 2010.

[CMO98] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In *Advances in Cryptology - ASIACRYPT '98, International Conference on the Theory and Applications of Cryptology and Information Security, Beijing, China, October 18-22, 1998, Proceedings*, pages 51–65, 1998.

[Cor99] Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, pages 292–302, 1999.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[FGKS02] Wieland Fischer, Christophe Giraud, Erik Woodward Knudsen, and Jean-Pierre Seifert. Parallel scalar multiplication on general elliptic curves over F_p hedged against Non-Differential Side-Channel Attacks. Cryptology ePrint Archive, Report 2002/007, 2002. `http://eprint.iacr.org/`.

[FIPS-186-4:13] FIPS PUB 186-4. Digital Signature Standard (DSS). Standard, National Institute of Standards and Technology, Gaithersburg, MD, United States, 2013.

[FPRE15] Jean-Pierre Flori, Jérôme Plût, Jean-René Reinhard, and Martin Ekerå. Diversity and transparency for ECC. *IACR Cryptology ePrint Archive*, 2015:659, 2015.

[GP08] Tim Güneysu and Christof Paar. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In E. Oswald and P. Rohatgi, editors, *Proceedings of CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 2008.

[Gui10] Nicolas Guillermin. A High Speed Coprocessor for Elliptic Curve Scalar Multiplications over $\mathbb{F}_p$. In S. Mangard and F.-X. Standaert, editors, *Proceedings of CHES 2010*, volume 6625 of *Lecture Notes in Computer Science*, pages 48–64. Springer-Verlag, 2010.

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

[ISO-15946-1:02] ISO/IEC 15946-1:2002. Information technology — Security techniques — Cryptographic techniques based on elliptic curves — Part 1: General. Standard, International Organization for Standardization, Geneva, Switzerland, 2002.

[ISO-18033-2:06] ISO/IEC 18033-2:2006. Information technology — Security techniques — Encryption algorithms — Part 2: Asymmetric ciphers. Standard, International Organization for Standardization, Geneva, Switzerland, 2006.

[JY02] Marc Joye and Sung-Ming Yen. The Montgomery Powering Ladder. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 291–302, 2002.

[KAK96] Çetin Kaya Koç, Tolga Acar, and Burt Kalisky. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[Koç95] Çetin Kaya Koç. RSA Hardware Implementation. Technical report, RSA Laboratories, August 1995. Version 1.0.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 104–113, 1996.

[LMSS14] Manfred Lochter, Johannes Merkle, Jörn-Marc Schmidt, and Torsten Schütze. Requirements for Standard Elliptic Curves, Position paper of the ECC Brainpool. IACR Cryptology ePrint Archive, Report 2014/832, September 2014.

[LMSS15] Manfred Lochter, Johannes Merkle, Jörn-Marc Schmidt, and Torsten Schütze. Requirements for Elliptic Curves for High-Assurance Applications. NIST Workshop on ECC Standards, June 2015.

[Loc05] Manfred Lochter. *ECC Brainpool Standard Curves and Curve Generation v. 1.0*. Bundesamt für Sicherheit in der Informationstechnik (BSI), Bonn, Germany, 2005.

[Men07] Nele Mentens. *Secure and Efficient Coprocessor Design for Cryptographic Applications on FPGA*. PhD thesis, Katholieke Universiteit Leuven, June 2007.

[Mil85] Victor S. Miller. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, pages 417–426, 1985.

[MLPJ13] Yuan Ma, Zongbin Liu, Wuqiong Pan, and Jiwu Jing. A High-Speed Elliptic Curve Cryptographic Processor for Generic Curves over $GF(p)$. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *Proceedings of SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 421–437. Springer-Verlag, 2013.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[Mon87] Peter L Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[NSS04] David Naccache, Nigel P. Smart, and Jacques Stern. Projective Coordinates Leak. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 257–267, 2004.

[OP01] Gerardo Orlando and Christof Paar. A Scalable $GF(p)$ Elliptic Curve Processor Architecture for Programmable Hardware. In Ç. K. Koç, D. Naccache, and Chr. Paar, editors, *Proceedings of CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 348–363. Springer-Verlag, 2001.

[Paa13] Christof Paar. Implementation of cryptographic schemes 1. Script of Lecture of Chair for Embedded Security from Ruhr University Bochum, 2013.

[PP10] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2010.

[Sch00] Werner Schindler. A Timing Attack against RSA with the Chinese Remainder Theorem. In *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, pages 109–124, 2000.

[Sch15] Torsten Schütze. Personal communication, 2015-05-12. Rohde & Schwarz SIT GmbH, 2015.

[SG14] Pascal Sasdrich and Tim Güneysu. Efficient Elliptic-Curve Cryptography Using Curve25519 on Reconfigurable Devices. In Diana Goehringer, Marco Domenico Santambrogio, João M. P. Cardoso, and Koen Bertels, editors, *Proceedings of ARC 2014, Reconfigurable Computing: Architectures, Tools, and Applications – 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014*, volume 5154 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, 2014.

[SW14] Werner Schindler and Andreas Wiemers. Power attacks in the presence of exponent blinding. *J. Cryptographic Engineering*, 4(4):213–236, 2014.

[Wal99] Colin D. Walter. Montgomery's Multiplication Technique: How to Make It Smaller and Faster. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, pages 80–93, 1999.

[Xil14a] Xilinx, Inc., San Jose, CA, United States. *7 Series DSP48E1 Slice – User Guide UG479*, November 2014. v1.8.

[Xil14b] Xilinx, Inc., San Jose, CA, United States. *7 Series FPGAs Configurable Logic Block – User Guide UG474*, November 2014. v1.7.

[Xil14c] Xilinx, Inc., San Jose, CA, United States. *7 Series FPGAs GTP Transceivers – User Guide UG482*, November 2014. v1.8.

[Xil14d] Xilinx, Inc., San Jose, CA, United States. *7 Series FPGAs Memory Resources – User Guide UG473*, November 2014. v1.11.