# A Low-Resource ECDSA Hardware Implementation Tailored for RFID Tags

Peter Peßl
peter.pessl@student.tugraz.at

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria


Master Thesis

Supervisor: Dipl.-Ing. Dr.techn. Michael Hutter
Assessor: Dipl.-Ing. Dr.techn. Michael Hutter

March, 2014

## Statutory Declaration

*I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.*

Graz, 13.3.2014

Peter Peßl

## Eidesstattliche Erklärung

*Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.*

Graz, 13.3.2014

Peter Peßl

# Acknowledgements

I would like to thank Michael Hutter for his continuous help throughout both my master project and master thesis. He proposed this thesis and gave valuable suggestions. His effort in discussing all my questions and implementation ideas are most appreciated. He also examined this thesis and played a very big part in getting the paper on the low-resource Keccak implementation accepted at the Workshop on Cryptographic Hardware and Embedded Systems (CHES) 2013.

I would also like to thank all other members of the IAIK team, for providing a perfect infrastructure and for all their previous research on ECC, cryptographic-hardware design, and implementation attacks.

Last, but most certainly not least, i would like to thank my parents for their continuous support throughout my academic studies. They helped me to get through all rough parts during all these years.

# Abstract

Radio-Frequency Identification (RFID) technology has gained a lot of attraction over the last years. As one particular application, RFID tags allowing an unforgeable proof of origin are an important aid in the struggle against product counterfeiting. Elliptic Curve Cryptography (ECC) can provide the required secure authentication services and is, due to the small key sizes, more resource friendly than other public-key systems. This makes a successful deployment of ECC to RFID tags mandatory.

Designing cryptographic hardware for RFID tags is a challenging task. Circuit size and power consumption must be minimized in order to attain low production cost and high reading ranges. Most existing low-resource ECC implementations suffer either from large circuit sizes or horrid algorithm runtimes. This thesis aims at tackling these problems.

For this reason, a low-resource hardware implementation of the Elliptic Curve Digital Signature Algorithm (ECDSA) based on the 160-bit prime-field curve `secp160r1` is presented. As a novelty, the Keccak hashing algorithm—selected as the winner of NIST's SHA-3 competition in 2012—is integrated. Various new techniques are deployed to maximize efficiency, e.g., application of a fixed-base comb method and new point-addition formulæ based on co-$Z$ notation provide a considerable speed-up of elliptic-curve operations. The 32-bit datapath contains a 16-bit integer multiplier, which is utilized within a pipelined multiplication scheme. A (very area efficient) single-port RAM macro is used as main storage element.

Especially for easily accessible devices like RFID tags, implementation attacks are a serious threat. Thus, countermeasures aimed at thwarting most common attacks are added. Most prominently, signature generation is performed in a constant and data-independent runtime, thereby making attacks based on power or timing analysis more difficult.

The implementation (including RAM) takes up an area of 63 735 $\mu m^2$ or 12 448 gate equivalents (GEs) in a 130 nm CMOS process technology. A message signature is generated within only 140 000 clock cycles, which makes the design both smaller and significantly faster than previous work. The mean power consumption is 42.7 $\mu W$ at a clock frequency of 1 MHz, thus making it well suitable for passively-powered tags. Interestingly, these figures can compete with published binary-curve implementations, which are considered to be more resource friendly.

**Keywords:** Elliptic Curve Cryptography, Digital Signatures, Hardware Implementation, ASIC, ECDSA, Keccak, SHA-3, RFID.

# Kurzfassung

Radiofrequenz-Identifikationstechnologie (RFID) findet immer breiteren Einsatz. RFID-Tags, die fälschungssichere Herkunftsnachweise erlauben, sind eine große Hilfe im Kampf gegen Produktpiraterie. Elliptische-Kurven-Kryptographie (ECC) stellt die dafür benötigten sicheren Authentifizierungsschemata bereit und ist durch die kleineren Schlüssellängen ressourcenschonender als andere Public-Key-Systeme. Das macht den Einsatz von ECC auf RFID-Tags unausweichlich.

Der Entwurf kryptographischer Hardware für RFID-Tags ist alles andere als trivial. Chipfläche und Leistungsaufnahme müssen minimiert werden um Produktionskosten niedrig und Reichweiten hoch zu halten. Bestehende ECC-Implementierungen weisen entweder eine große Chipfläche oder quälend lange Ausführungszeiten auf. Diese Arbeit nimmt sich dieses Problems an.

In dieser Diplomarbeit wird eine Hardwareimplementierung des Elliptische-Kurven-Digitaler-Signaturalgorithmus (ECDSA) basierend auf der 160-Bit-Primkörperkurve `secp160r1` vorgestellt. Als Neuheit wird der Keccak-Hashalgorithmus verwendet, dieser ging im Jahr 2012 als Gewinner aus dem SHA-3-Wettbewerb hervor. Um eine möglichst hohe Effizienz zu erreichen werden mehrere neuartige Techniken zum Einsatz gebracht. So werden zum Beispiel durch Anwendung einer Kammmethode und neuen Punktadditionsformeln basierend auf Co-$Z$-Notation Kurvenoperationen beschleunigt. Der in den 32-Bit breiten Datenpfad eingebettete 16-Bit-Multiplizierer wird einem Pipeline-Multiplikationsschema verwendet. Ein sehr flächeneffizientes Single-Port-RAM-Makro wird als Speicherelement eingesetzt.

Besonders für leicht zugängliche Geräte wie RFID-Tags sind Implementierungsattacken eine echte Gefahr. Um ebendiese zu verhindern wurden Gegenmaßnahmen implementiert. Besonders hervorzuheben ist hier die konstante und datenunabhängige Laufzeit der Signaturberechnung. Diese erschwert Attacken basierend auf Leistungsaufnahme- und Zeitmessungen.

Die Implementierung (inklusive RAM), benötigt eine Fläche von 63 735 µm$^2$ oder 12 448 Gate-Äquivalente (GEs) in einer 130 nm CMOS-Prozesstechnologie. Die Berechnung einer Signatur benötigt nur 140 000 Taktzyklen, womit dieses Design sowohl kleiner als auch wesentlich schneller als bisherige Arbeiten ist. Die durchschnittliche Leistungsaufnahme beträgt 42.7 µW bei einer Taktfrequenz von 1 MHz, was einen Einsatz in passiven Tags erlaubt. Diese Werte können es durchaus mit Binärkurvenimplementierungen aufnehmen.

**Stichwörter:** Elliptische-Kurven-Kryptographie, Digitale Signaturen, Hardwareimplementierung, ASIC, ECDSA, Keccak, SHA-3, RFID.

# Contents

# Chapter 1

# Introduction

Radio-Frequency Identification (RFID) technology is used in a broad variety of applications, such as supply-chain management, access control, and contactless payment. Basic RFID systems are restricted to simple tag identification, but there is an ever increasing demand for secure services. To satisfy them, elaborate cryptographic schemes need to be applied.

RFID tags are shipped in a very large scale, which makes key distribution in symmetric-key systems extremely difficult. Public-key cryptography solves this problem, it allows to equip each tag with a unique key pair. Elliptic Curve Cryptography (ECC) has much smaller key sizes than comparable public-key systems, which makes it ideal for use on resource-constrained devices like RFID tags.

With all that said, actual deployment of ECC systems on RFID tags is not trivial. Hardware designers must cope with stringent requirements, especially in the case of passively-powered tags. These tags draw their energy from the field of a reader device, which severely limits the available power. Also, to achieve low production, cost the circuit size must be small. Finally, implementers must also consider implementation attacks.

There already exist several low-resource ECC implementations in the wild. Some are (relatively) large and power hungry, which makes them not suitable for such constrained devices. Many others sacrifice computational performance in order to reach their goals; this often leads to horrible runtimes. In order to achieve somewhat short tag-response times, the clock frequency must be increased, which has a severe impact on the power consumption.

This thesis aims at improving this situation. For this reason, a low-resource Elliptic Curve Digital Signature Algorithm (ECDSA) hardware implementation was designed. Goal was to be smaller and considerably faster than previous work. As a first, the Keccak hashing algorithm was evaluated in the context of low-resource ECDSA. The outcome of this work is now presented.

## Organization of the Thesis

This thesis is organized as follows. In Chapter 2, an introduction to ECC and ECDSA is given. Also, efficient algorithms for elliptic-curve operations and finite-field arithmetic are presented. A small part part is dedicated to cryptographic hash functions and the Keccak algorithm. The challenges lying in designing cryptographic hardware for RFID tags are presented in Chapter 3. Additionally, an overview of implementation attacks is given.

In Chapter 4, the requirements of the presented design are stated and some basic design decisions are made. Then, the implementation details are discussed in Chapter 5. It starts with a description of the datapath and then moves on to modular arithmetic, point-scalar multiplication, and modular field inversions. Finally, the implemented countermeasures aimed at thwarting most common implementation attacks are presented. Chapter 6 discusses the used KECCAK architecture and its integration into the design.

In Chapter 7, the detailed implementation results are given. Circuit size, power consumption and algorithm runtime are analyzed and compared to related work. Moreover, further research suggestions are raised. Finally, in Chapter 8 conclusions are drawn.

# Chapter 2

# A Primer on ECDSA

In this chapter, an explanation of the Elliptic Curve Digital Signature Algorithm (ECDSA) is given. First, the very basics of public-key cryptography are discussed in Section 2.1. Section 2.2 then gives a short introduction to Elliptic Curve Cryptography (ECC). ECC implementations require efficient finite-field arithmetic, which is explained in Section 2.3. In Section 2.4, fast elliptic curve point addition formulæ and point-scalar multiplication algorithms are presented. Finally, Section 2.5 discusses the ECDSA and lists the parameters used for this work. Although not directly related to ECC, hashing is an important part of the ECDSA. Thus, Section 2.6 presents the basics of hash functions and gives further details on the KECCAK hashing algorithm.

## 2.1 Public-Key Cryptography

The field of cryptography is concerned with techniques that allow secure and authenticated communication in the presence of an adversary [28]. The typically used communications model is depicted in Figure 2.1. Two entities, called Alice (A) and Bob (B), want to communicate over an unsecure channel. It is assumed that an adversary, called Eve (E), has the capabilities of monitoring and influencing all communications.

There exist two basic types of cryptographic techniques. In *symmetric* schemes, the entities both agree on a secret key that is then used by both parties. Commonly used symmetric algorithms are the Data Encryption Standard (DES), the Advandced Encryption Standard (AES), and symmetric authentication algorithms such as HMAC. Symmetric cryptography suffers from the key-distribution problem, each communicating pair needs to maintain a separate key. In a system with $n$ entities, roughly $n^2$ keys are required.

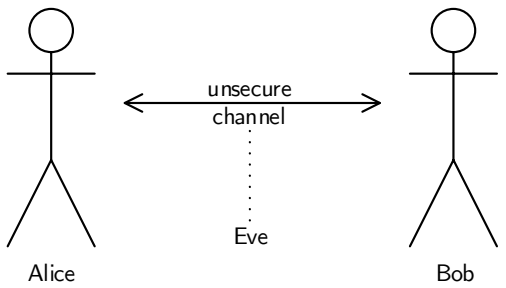



Figure 2.1: Communications model

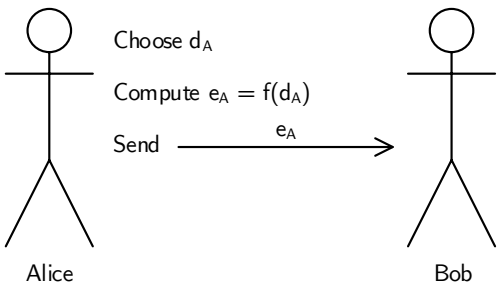



Figure 2.2: Key-pair generation

*Public-key cryptography* (also called asymmetric cryptography) does not suffer from this problem, but is typically more resource consuming. In asymmetric schemes, each entity selects a key pair $(e, d)$, which consists of a *private* key $d$ and a related *public* key $e$. The public key $e$ can be transmitted to other parties (e.g., to Bob) over an unsecure channel like the internet, while the private key $d$ is kept secret. The key-pair generation process is shown in Figure 2.2. The public-key computation must be a one-way function, i.e., deriving the private key only from the public key must be computationally hard.

There exist three main groups of asymmetric schemes, each is based on a different mathematical problem. The integer factorization problem is the foundation of probably the most well-known public-key technique, namely the RSA algorithm. Then there exist techniques based on the hardness of the discrete logarithm problem, such as the ElGamal encryption scheme, the Digital Signature Algorithm (DSA), or the Diffie-Hellmann key agreement. Finally, Elliptic Curve Cryptography (ECC) is based on the elliptic curve version of the discrete logarithm problem, as will be discussed later.

Public-key cryptography can be used for multiple applications, such as encryption, key agreement, or digital signatures. The latter is now discussed in greater detail.

### 2.1.1 Digital Signature Schemes

Digital signatures can be used for message and origin authentication, they can also provide non-repudiation. Some well known examples of digital signature schemes are the Digital Signature Algorithm (DSA) and its elliptic curve analogue ECDSA.

Signature schemes work as follows. The signing entity (A) uses a signature generation algorithm (Sign) with the message $m$ and the private key $d_A$ to compute the signature $s = \text{Sign}(m, d_a)$. After receiving the message and the signature, the verifying party (B) uses a signature verification algorithm ($\text{Verify}(m, s, e_A)$) to validate the signature's authenticity. This operation requires the signers public key, which the verifier must obtain in an authenticated fashion. As only the signer is in possession of $d_A$, the verifier is assured that indeed A signed this message. Also, the signature $s$ is bound to the message, for a different message $m'$ the signature $s$ is not valid.

Digital signatures can be used for RFID-tag authentication services. Figure 2.3 shows a challenge-response authentication protocol which is defined in the ISO 9798-3 standard [36]. The RFID reader first fetches the tag's certificate, it contains the tag's public key $e$ and is signed by a tag issuing party. The reader then validates this certificate, if it is authentic it sends a challenge $c_1$. The tag also chooses a random nonce $c_2$, it then signs the concatenation of both values $c_1||c_2$. Finally, the reader verifies the returned signature.

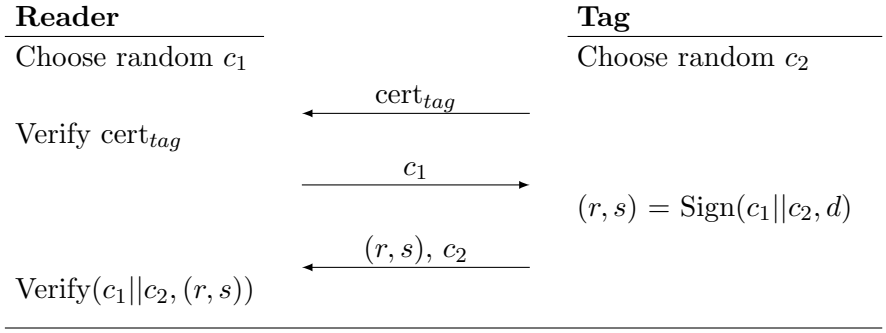



Figure 2.3: Tag authentication protocol

Such tag-authentication mechanisms can be a big aid in the struggle against product counterfeiting, as they allow an (ideally) unforgeable proof of origin. Original products can be fitted with tags, each one featuring its own unique key pair. The tag issuer then signs the public key and stores the certificate on the tag. A successful protocol execution then attests that the tag is indeed in possession of the related private key. Hence, it is proven that the tag was issued by the issuing party, so the product is original. This assumes that the private key is securely stored on the tag, it must not leak in any way.

## 2.2 Elliptic Curve Cryptography

Elliptic curves have long been around and are used for multiple applications. In 1985, Koblitz [45] and Miller [55] independently proposed to use them in asymmetric cryptography schemes, thereby giving birth to Elliptic Curve Cryptography (ECC).

One can define point addition and doubling operations on such curves, the set of all curve points then forms an abelian group under the addition. The doubling and addition operations can be used to perform so-called elliptic-curve scalar multiplication (ECSM), i.e., a base point $P$ is multiplied with a scalar $k$. This can be written as

$$Q = kP = \overbrace{P + P + \cdots + P}^{k}.$$

The inverse operation, i.e., finding $k$ with given $P$ and $Q$, is (assumed to be) computationally hard. This so-called Elliptic Curve Discrete Logarithm Problem (ECDLP) is the security foundation of ECC.

ECC implementations can be organized into a hierarchical structure, as depicted in Figure 2.4. Elliptic-curve operations, i.e., addition, doubling, and moreover scalar multiplication, make use of finite-field arithmetic, hence efficient field computation schemes are required. Commonly used types of fields are prime fields $\mathbb{F}_p$ and binary fields $\mathbb{F}_{2^m}$. Then, cryptographic protocols, in this case ECDSA, build upon the point-scalar multiplication.

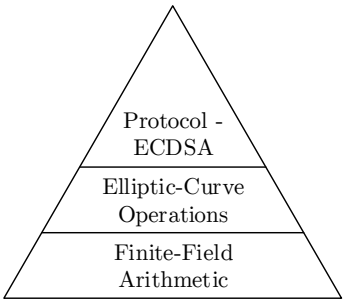


Figure 2.4: Hierarchy of ECC [81]

The major advantage of ECC over schemes reliant on RSA are the considerably smaller key sizes. For example, a 160-bit prime field curve offers a security level of 80 bits. In comparison, a 1024-bit RSA modulus is required to achieve the same level [28]. This makes ECC ideal for low-resource devices, where storage space is absolutely scarce.

## 2.3 Prime-Field Arithmetic

Efficient prime-field arithmetic is absolutely vital for designing fast ECDSA implementations. In this section, first the very basics of prime-field arithmetic are discussed. Then, a short introduction to multi-precision operations is given and common integer multiplication algorithms are discussed. Afterwards, two widespread modular reduction algorithms are presented. Finally, algorithms for computing modular inversions are given.

**Field definition.**
Mathematical *fields* are a set $\mathbb{F}$ with two operations, namely addition ($+$) and multiplication ($\times$). Fields must fulfill following properties [28]:

1. $(\mathbb{F}, +)$ forms an abelian group with an identity denoted by 0.
2. $(\mathbb{F}, \times)$ forms an abelian group with an identity denoted by 1.
3. The distributive law is in place, i.e., $a + (b \times c) = a \times b + b \times c$.

If the number of elements in $\mathbb{F}$ is finite, then it is called a *finite field*. The number of elements $q$ is then called the *order* of the field. The order must be a prime power, i.e., $q = p^m$, where $p$ is a prime (called characteristic) and $m$ is a positive integer. Fields with $m = 1$ are called *prime fields*. Fields with $m \geq 2$ are called *extension fields*, in particular, *binary fields* are extension fields with $p = 2$.

For a prime $p$, the corresponding prime field $\mathbb{F}_p$ contains all integers in the range $\{p-1, \ldots, 2, 1, 0\}$. Addition and multiplication are done modulo $p$, i.e., modular reductions are required to obtain the final result.

### 2.3.1 Multi-Precision Arithmetic

The used primes are typically long, for ECC they range between 160 and up to over 500 bits. So-called *full-precision* logic operates on entire field elements at once. However, due to the large bit sizes this is not an option on low-resource devices. *Multi-precision* arithmetic operates on shorter words, which are obtained by splitting the field element into multiple parts. With a chosen word size $w$, an $n$-bit integer $a$ is split into $t = \lceil n/w \rceil$ equal sized words. This is denoted by $a = (a[t-1], \ldots, a[1], a[0])$.

A modular addition with multi-precision logic is shown in Algorithm 2.1. If the sum is greater than the modulus, it must be reduced modulo $p$. As $c < 2p$, this can be achieved by simple subtraction of $p$.

**Algorithm 2.1:** Multi-precision modular addition

**Input**: $a, b \in \mathbb{F}_p$, modulus $p$
**Output**: $c = a + b \bmod p$

```
1 carry = 0
2 for i = 0 to t − 1 do
3     (carry, c[i]) = a[i] + b[i] + carry
4 end
5 c[t] = carry
6 if c > p then
7     return c
8 else
9     return c − p
```

### 2.3.2 Multi-Precision Multiplication

There exist several different widespread techniques for performing multi-precision multiplication, i.e., multiplying two large integers using lower width multiplications. Two of these, namely *operand-scanning* and *product-scanning* multiplication, are now discussed,

especially in respect to their efficiency in hardware implementations. Not further discussed but still noteworthy are the hybrid method [26], the operand-caching method [35], Karatsuba's method [42], or FFT-based multiplications.

Both operand-scanning and product scanning have a quadratic runtime complexity, exactly $t^2$ partial products must be computed. The sequence of computation however differs. The multiplication process can be illustrated as a rhombus, as seen in Figures 2.5 and 2.6 (with the example of 5-word integers). Each line stands for a single word of operand $a$ or $b$, each dot represents a multiplication. The multiplication sequence is denoted by red arrows, on the left side the structure of the algorithm is shown.

Probably the most simple multiplication technique is the so-called *operand-scanning* multiplication. When implemented using two loops, the outer loop iterates over the words of $a$, while the inner loop loads each word of $b$, multiplies it with the current $a[i]$ and adds the result to the intermediate column value. This process is depicted in Figure 2.5. The intermediate column values are either stored in dedicated registers or in external memory. The main disadvantage of the latter option is the high number of load and store operations of intermediate results.

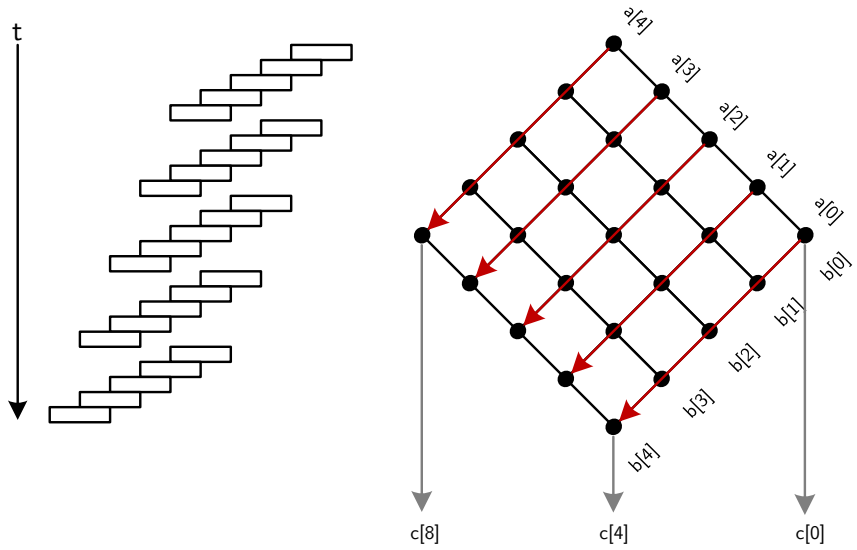



Figure 2.5: Operand-scanning multiplication of 5-word large integers

The *product-scanning* multiplication performs better in this regard. It follows a column-wise approach, i.e., column after column of the final result is computed using the equation $c[i] = (c[i-1] >> 2^w) + \sum_{j,k:j+k=i} a[j]b[k]$ (see Figure 2.6). The partial products can be summed up using a multiply-and-accumulate approach, then no intermediate column results need to be stored or loaded. The downside is the higher number of load operations for fetching the operands.

### 2.3.3 Fast Reduction using Special Primes

Modular reduction after multiplication can be an expensive operation. However, there exist special primes that allow a fast reduction by using only shifting and addition. Such primes, also called pseudo-Mersenne primes, can be written as a short sum or difference of powers of 2.

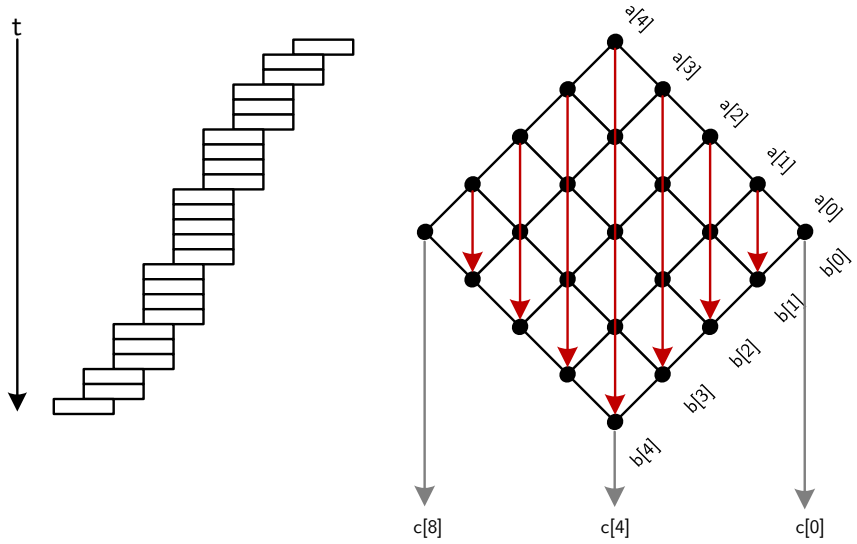



Figure 2.6: Product-scanning multiplication of 5-word large integers

The prime $p$ used in the `secp160r1` curve has this property, as $p = 2^{160} - 2^{31} - 1$. An integer $x > p$ can be reduced as follows. First the integer is split in parts $h, \ell$, with $\ell < 2^{160}$, such that $x = h \cdot 2^{160} + \ell$. Then, by using the equivalence $2^{160} \equiv 2^{31} + 1 \mod p$, one can rewrite $x \equiv h \cdot 2^{31} + h + \ell \mod p$. Hence, reduction is achieved by means of additions and binary shifts (by 31 bits to the left). This process is depicted in Figure 2.7. If the sum is greater than the modulus $p$, another round of reduction must be applied.

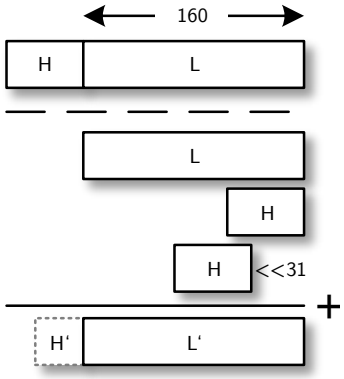



Figure 2.7: Fast reduction in $\mathbb{F}_p$

### 2.3.4 The Montgomery Multiplication

If the modulus does not have any special form, then other reduction algorithms need to be used. Noteworthy examples are the Montgomery multiplication scheme [56] and the Barret reduction [5], the former is now discussed in greater detail.

The Montgomery multiplication performs operations on representatives $\bar{x} = xR \mod N$, with the modulus $N$ and a chosen $R > N$. The set of representatives is from now on also called Montgomery domain. The Montgomery product, denoted by $\text{MonPro}(\bar{a}, \bar{b})$,

is the efficient computation of $\bar{a}\bar{b}R^{-1} \bmod N$. Thus, $\text{MonPro}(\bar{a}, \bar{b}) = \bar{a}\bar{b}R^{-1} \bmod N = abR \bmod N = \bar{c}$, with $c = ab$.

Let $R > N$ and $N' = -N^{-1} \bmod R$. The Montgomery multiplication is then carried out as listed in Algorithm 2.2. It contains both division and modular reduction by the chosen $R$. To make these operations simple, typically $R = 2^w$, where $w$ denotes the processor word size. For a more in-depth description of the algorithm see [56].

**Algorithm 2.2:** Montgomery multiplication scheme [56]

**Input**: $a, b$
**Output**: $\text{MonPro}(a, b) = abR^{-1} \bmod N$

```
1 T = ab;
2 m = (T mod R)N'  mod R;
3 t = (T + mN)/R;
4 if t ≥ N then
5     return t - N;
6 else
7     return t;
8 end
```

The value of $t \equiv abR^{-1} \bmod N$, but is in range $t < 2N$. Thus a conditional subtraction is required to retrieve the final result. This subtraction can be avoided, as first shown by Walter [75, 76]. His method was later analyzed and improved by Hachez and Quisquater [27] and Örs et al. [63]. They modify the algorithm to allow usage of operands $a, b < 2N$, thus the previous result $t$ can be directly used as input for the next multiplication. This is achieved by increasing the lower bound of $R$.

Transforming integers to the Montgomery domain and back is rather expensive, so the Montgomery multiplication scheme is not very efficient for single multiplications. However, the transformation is required only once when part of a larger computation, such as inversion by exponentiation.

**Efficient implementation.**
The Montgomery multiplication method executes two integer multiplications, $ab$ and $mN$, with the operands $a, b$ and the modulus $N$. There exist several possibilities to combine the multiplications and integrate the computation of $m$, as discussed by Koç et al. [46]. The *Finely Integrated Product Scanning* (FIPS) approach is relevant to this work and will be presented.

One simple optimization applies to the computation of $m$. As first noted by Dussé and Kaliski [19], the reduction process proceeds word by word, i.e., one can write
$m[i] = T[i]n'_0 \bmod 2^w$, with $n'_0 = n' \bmod 2^w$ and $w$ the processor word size. This replaces a full-size multiplication with a few single-word ones.

The *Finely Integrated Product Scanning* (FIPS) approach interleaves computation of $ab$ and $mN$ by performing two multiplications in the inner loop, i.e., it switches between multiplication and reduction after each step. Algorithm 2.3 illustrates this approach. acc denotes an accumulator register, $\text{acc}_w$ references the $w$ low-order bits of the accumulator. The final subtraction is not shown.

**Algorithm 2.3:** Finely Integrated Product Scanning (FIPS)

**Input**: a, b
**Output**: t

```
1  s = l/w;
2  for i=0 to s-1 do
3      for j=0 to i-1 do
4          acc = acc + a[j]b[i-j];
5          acc = acc + m[j]N[i-j];
6      end
7      acc = acc + a[i]b[0];
8      m[i] = acc_w n'_0 mod 2^w;
9      acc = acc + m[i]n[0];
10     acc >>= w;
11     for i=s to 2s-1 do
12         for j=i-s+1 to s-1 do
13             acc = acc + a[j]b[i-j];
14             acc = acc + m[j]n[i-j];
15         end
16         t[i] = acc_w;
17         acc >>= w;
18     end
19 end
```

### 2.3.5 Inversion in Prime Fields

In a prime field $\mathbb{F}_p$, the inverse of an element $a$, denoted by $a^{-1} \bmod p$, is an integer that fulfills $a \cdot a^{-1} \equiv 1 \bmod p$.

Probably the most well-known inversion method is the extended Euclidean algorithm. As the name implies, it is an extension to the Euclidean algorithm, which computes the greatest common divisor of two integers. The binary inversion algorithm [28] is a deviation of this method using only shifts and subtractions. Another widely used inversion algorithm is the Montgomery inversion, it is based on the Montgomery multiplication scheme. Both the extended Euclidean algorithm and the Montgomery inversion scheme do not feature a constant, i.e., operand independent, runtime.

Inversion by exponentiation, which is based on Fermat's little theorem, does offer this treat, but is typically slower. Fermat's little theorem (not to be confused with the Fermat's famous last theorem) states that $a^p \equiv a \bmod p$ for each prime $p$ and integer $a$. If $a$ is not a multiple of $p$, then there exists an $a^{-1} \bmod p$. By multiplying both sides of the theorem with $a^{-2} \bmod p$ one gets $a^{p-2} \equiv a^{-1} \bmod p$, i.e., the inverse can be calculated by raising $a$ to the power of $p-2$. The runtime of even the most basic exponentiation algorithms is only dependent on the exponent, which is fixed in this case. Thus, the inverse can be retrieved in constant time. A very basic modular exponentiation algorithm is discussed in the following.

**Modular exponentiation.**
A straight-forward binary multiply-and-square exponentiation (Algorithm 2.4) requires $|e| - 1$ field squarings and HW(e) $- 1$ field multiplications, where $e$ denotes the exponent,

$|e|$ its bit length, and HW($e$) its Hamming weight. The Hamming weight of an integer is defined as the number of its non-zero digits, for a binary representation this is equal to the number of 1s. Sliding window algorithms trade off memory for speed and require a costly precomputation phase. Similarly to comb methods (Section 2.4.5), for a specified window size $w$ they precompute and store all $ax = (1\ a_{w-2} \dots a_0)x$, with $a_i \in \{0, 1\}$ and $0 \leq i < w - 1$. The exponent is then scanned, a multiplication is carried out whenever a window match occurs. The squaring and multiplication sequence is thus computed online and on-the-fly.

**Algorithm 2.4:** Left-to-right binary square-and-multiply exponentiation method

**Input**: $e = (e_{n-1}, \dots, k_0), x$
**Output**: $x^e$
1 $R \leftarrow \mathbf{0}$;
2 **for** *i = n-1 downto 0* **do**
3  $R \leftarrow R^2$;
4  **if** $e_i$ *= 1* **then** $R \leftarrow R \times x$;
5 **end**
6 **return** $R$;

## 2.4 Elliptic-Curve Arithmetic

Elliptic curves are a special form of algebraic curves. An elliptic curve $E$ is usually defined over a field $\mathbb{F}$, e.g., over the real numbers $\mathbb{R}$ or over finite prime or binary fields. This can be denoted by $E/\mathbb{F}$, i.e., the curve $E$ is defined *over* the *underlying* field $\mathbb{F}$. A curve $E$ is typically defined by a short Weierstrass equation

$$E: \ y^2 = x^3 + ax + b \tag{2.1}$$

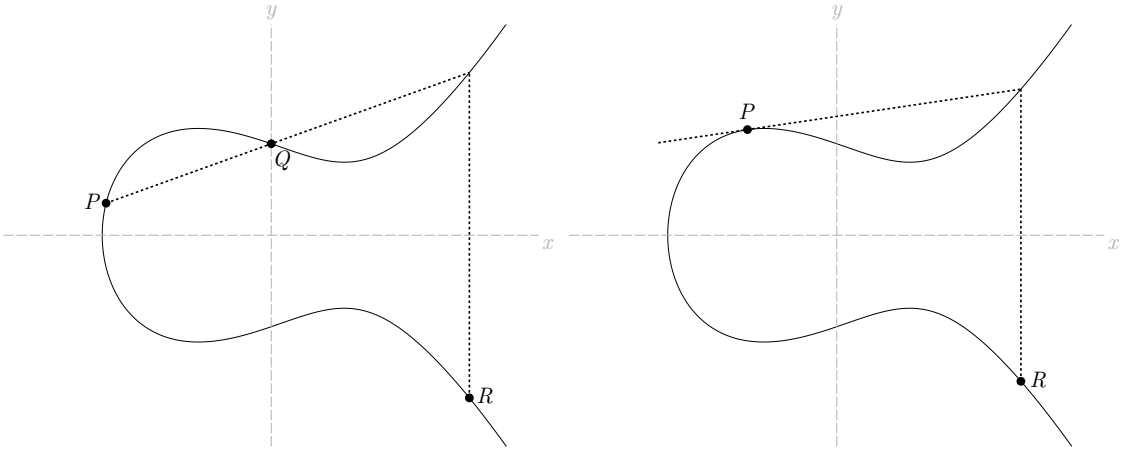
with some $a, b \in \mathbb{F}$. Pairs of $(x, y)$ fulfilling this equation are said to be points $P$ on the curve $E$, the set of all points on a curve is denoted by $E(\mathbb{F})$.

### 2.4.1 Basic Curve Arithmetic

One can define a point-addition operation on elliptic curves using the *cord-and-tangent* rule. This addition can be best explained geometrically.

As illustrated in Figure 2.8a, two points $P, Q$, with $P \neq Q$, can be added with the following method. Draw a straight through both points, this line intersects the curve at a third point. This point is mirrored at the $x$-axis to retrieve the sum $R = P + Q$.

Point doubling is defined as follows. Draw the tangent of the point $P$, just like earlier this line intersects the curve at a third point. This point is mirrored at the $x$-axis to retrieve the double $R = 2P$. This is shown in Figure 2.8b.




(a) Addition $P + Q = R$ (b) Doubling $2P = R$

Figure 2.8: Example of point addition and doubling on curve $y^2 = x^3 - 2x + 3$ over $\mathbb{R}$

For curves in short Weierstrass form, the point doubling operation can be expressed as follows. Let $P = (x_1, y_1) \in E, Q = (x_2, y_2) \in E$ where $P \neq Q$. Then $R = P + Q = (x_3, y_3)$ with

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

The doubling operation is defined as follows. Let $P = (x_1, y_1) \in E$, where $P \neq -P$. Then $R = 2P = (x_3, y_3)$ with

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

The set $E(\mathbb{F})$ forms an abelian group under this addition operation. Groups require a neutral element (an identity), thus the so-called *point at infinity* $\boldsymbol{P_\infty}$ is included. This point is neutral to addition, hence $P + \boldsymbol{P_\infty} = \boldsymbol{P_\infty} + P = P$, for all points $P \in E(\mathbb{F})$.

Points $P$ can be negated by mirroring them at the $x$-axis, i.e., by negating the $y$ coordinate. Hence, with $P = (x, y)$, the negative $-P = (x, -y)$. Note that $P + (-P) = \boldsymbol{P_\infty}$.

### 2.4.2 Projective Coordinates

The doubling and addition formulæ presented in the previous section contain divisions. For prime-field curves this translates to a multiplication with the modular multiplicative inverse of the divisor. Field inversions are expensive in terms of runtime. The use of so-called *projective coordinates* allows to avoid them.

One can represent the two-dimensional points $(x, y)$ using 3 integers $(X, Y, Z)$, called projective coordinates. The equivalence relation $\sim$ of points can be defined as

$$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2) \text{ if } X_2 = \lambda^c X_1, Y_2 = \lambda^d Y_1, Z_2 = \lambda Z_1 \text{ for any } \lambda \in \mathbb{F}_p$$

Both $(X_1, Y_1, Z_1)$ and $(X_2, Y_2, Z_2)$ are then different *representatives* of the same curve point. For each projective point $(X_1, Y_1, Z_1)$ the representative $(X/Z^c, Y/Z^d, 1)$ allows a direct mapping of projective coordinates to affine points, and vice versa.

Out of the set of projective coordinates, homogeneous coordinates with $c = d = 1$ and Jacobian coordinates with $c = 2, d = 3$ are the most widespread. Point-scalar multiplications are then carried out entirely in projective coordinates, in the end a single inversion $Z^{-1}$ is required to retrieve the affine result $(x, y) = (XZ^{-c}, YZ^{-d})$. In the next section, efficient point addition formulæ are presented .They aim at minimizing the number of required field multiplications and squarings.

### 2.4.3 Efficient Addition Formulæ

There exists a broad range of efficient point addition and point doubling formulæ utilizing all sorts of different projective coordinates. The runtime of these algorithms is typically measured in necessary field multiplications (M) and squarings (S), additions and subtractions have a much lower runtime and are hence excluded from a simple time analysis. Another important factor is the minimum number of required field registers, i.e., the minimum memory requirements. Here some caution is required, most authors assume the availability of in-place multiplication. If only out-of-place multiplication is available, the number might be higher.

For a comprehensive list of addition formulæ for all sorts of different curves and projective coordinates, see the *Explicit-Formulas Database* by Bernstein and Lange [6]. For curves in short Weierstrass form and Jacobian coordinates, the fastest listed doubling algorithm requires 3M + 5S. For a mixed Jacobian-affine addition, where one point is represented in Jacobian and the other in affine coordinates, the fastest formula takes 7M + 4S. A doubling-addition $2P + Q$, i.e., performing a doubling immediately followed by an addition, takes 10M + 9S when using these formulæ.

In 2007, Meloni introduced the so-called co-$Z$ addition formulæ [54], which are based on Jacobian coordinates. He noticed that two points $P, Q$ sharing the same $Z$ coordinate can be added in only 5M and 2S (Algorithm 2.5). The key observation is that this addition yields an alternative representation of the input $P$ that shares the $Z$ coordinate with the result $P + Q$, i.e., point $P$ is updated to $P' = (X_1(X_2 - X_1)^2, Y_1(X_2 - X_1)^3, Z(X_2 - X_1))$. Using this method in context of a point-scalar multiplication requires some work, it must be ensured that in each step the added points share the $Z$ coordinate.

**Algorithm 2.5:** Meloni's co-$Z$ addition [54]

**Input**: $P = (X_1, Y_1, Z), Q = (X_2, Y_2, Z)$
**Output**: $P + Q = (X_3, Y_3, Z_3), P' = (B, D, Z_3)$
1 $A = (X_2 - X_1)^2$; $B = X_1 A$;
2 $C = X_2 A$; $D = (Y_2 - Y_1)^2$;
3 $X_3 = D - B - C$;
4 $Y_3 = (Y_2 - Y_1)(B - X_3) - Y_1(C - B)$;
5 $Z_3 = Z(X_2 - X_1)$;

**Co-$Z$ doubling-addition.**
Goundar et al. [25] further analyzed the co-$Z$ addition and introduced a *co-Z doubling-addition*. This operation computes $2P + Q$ and yields an updated representation of the input point $Q$ using 9M and 7S and a minimum of 8 field registers. Using this operation one can construct co-$Z$ versions of some standard ECSM algorithms, like left-to-right signed-digit algorithms or Joye's method.

The co-$Z$ doubling-addition algorithm can be easily adapted to a more general case, where the co-$Z$ requirement of the operand points is dropped and instead the point $Q = (x_2, y_2)$ is given in affine coordinates. Then the first step is to compute a co-$Z$ representation of $Q$, i.e., $Q' = (x_2Z^2, y_2Z^3, Z)$, which takes an additional 3M and 1S. There is no need to retrieve an updated representation of $Q$ at the end of the algorithm, skipping these operations saves 1M and 1S. Hence, the total cost is 11M + 7S.

Longa and Miri [52] followed a different approach, but ended up with very similar results. They observed that a mixed Jacobian-affine addition $P + Q$ (7M + 4S) yields an alternative co-$Z$ representation of $P$. Then it is possible to add $(P + Q) + P = 2P + Q$ using a standard co-$Z$ addition (5M + 2S). When merging these additions one can trade 2S by 1M, thus the total runtime is 11M + 7S. The resultant formulæ are shown in Algorithm 2.6, they can be implemented using only 7 field registers (Algorithm 5.2).

**Algorithm 2.6:** EC doubling-addition according to [52]

**Input**: $P = (X_1, Y_1, Z_1),\ Q = (X_2, Y_2)$
**Output**: $2P + Q = (X_4, Y_4, Z_4)$
1 $\alpha = Z_1^3Y^2 - Y_1;\ \beta = Z_1^2X_2 - X_1;$
2 $X' = 4X_1\beta^2;\ Y' = 8Y_1\beta^3;$
3 $Z' = (Z_1 + \beta)^2 - Z_1^2 - \beta^2;$
4 $\theta = X_3 - X' = 4(\alpha^2 - \beta^3 - 3X_1\beta^2);$
5 $\omega = Y_3 - Y' = \alpha^2 + \theta^2 - (\alpha + \theta)^2 - 16Y_1\beta^3;$
6 $X_4 = \omega^2 - \theta^3 - 2X'\theta^2;$
7 $Y_4 = \omega(X'\theta^2 - X_4) - Y'\theta^3;$
8 $Z_4 = Z'\theta;$
9 **return** $(X_4, Y_4, Z_4)$

Both Longa's scheme and Goundar's (modified) method have a runtime of 11M + 7S. Another thing they have in common is that both compute $(P + Q) + P$, i.e., they replace the doubling operation in $2P + Q$ with a second addition. This has a downside, as it must always be ensured that $P \neq \pm Q$.

### 2.4.4 Point-Scalar Multiplication

The elliptic curve scalar multiplication (ECSM) is the basis of all elliptic curve cryptographic algorithms and protocols. A base point $P$ is multiplied with a scalar $k$ with bit length $m = |k|$, which is denoted by $Q = kP$. The smallest $n$ satisfying $nP = P$ is called the order of the point $P$.

The runtime of point-scalar multiplication algorithms is measured in required additions (A) and doublings (D). It is typically the most expensive operation for cryptographic protocols, so fast and efficient techniques are vital.

The left-to-right binary double-and-add method (Algorithm 2.7) is one of the most simple scalar multiplication algorithms. The intermediate $R$ is first initialized to the point at infinity $\boldsymbol{P_\infty}$. Then the scalar $k$ is scanned from left-to-right, for each digit a doubling is performed, which is followed by an addition of the base point $P$ if the scalar digit equals 1. The expected runtime is $m$D and HW($k$)A.

While very simple, the algorithm has one major flaw. By means of an SCA it is relatively easy to detect if the point addition is carried out, i.e., if the scalar bit $k_i$ equals 1. A single power trace might be sufficient to reveal the entire scalar. The double-and-add

**Algorithm 2.7:** Left-to-right binary double-and-add method

**Input**: $k = (k_{m-1}, \ldots, k_0), P$
**Output**: $Q = kP$
1 $R \leftarrow \boldsymbol{P_\infty}$;
2 **for** $i = m - 1$ *downto 0* **do**
3 $\quad R \leftarrow 2R$;
4 $\quad$ **if** $k_i = 1$ **then** $R \leftarrow R + P$;
5 **end**
6 **return** $R$;

always approach (see Section 3.2.1) tries to solve this problem with dummy operations, but is susceptible to fault attacks.

There exist highly regular algorithms not relying on dummy operations, e.g., Joye's right-to-left algorithm [38] or the Montgomery ladder [40, 57]. Especially the latter is a very widespread scalar-multiplication technique, it is shown in Algorithm 2.8. It has a highly regular structure, which makes it ideal for SCA resistant implementations. The runtime of both Joye's method and the Montgomery ladder is $m$ doublings and additions.

**Algorithm 2.8:** Montgomery ladder [57]

**Input**: $k = (k_{m-1}, \ldots, k_0), P$
**Output**: $Q = kP$
1 $R_0 \leftarrow \boldsymbol{P_\infty}$;
2 $R_1 \leftarrow P$;
3 **for** $i = m - 1$ *downto 0* **do**
4 $\quad \alpha \leftarrow k_i$;
5 $\quad R_{1-\alpha} \leftarrow R_{1-\alpha} + R_\alpha$;
6 $\quad R_\alpha \leftarrow 2R_\alpha$;
7 **end**
8 **return** $R_0$

A possible way of speeding up the ECSM, especially if the base point $P$ is not fixed and some memory is available, are window methods. The most simple window algorithms typically precompute all $aP$, with $0 \leq a < 2^w$ and for a chosen window size $w$, and store these points in memory. They then proceed by scanning the scalar from left to right, $w$ bits at a time are processed by performing $w$ doublings and 1 addition of a precomputed point. When excluding the cost for precomputation, the algorithm requires only $m/w$ point additions. However, the number of point doublings does not decrease when compared to the more simple algorithms.

### 2.4.5 ECSM using Fixed-Base Comb Methods

For algorithms featuring a fixed base point $P$, e.g., ECDSA signature generation, the ECSM can be sped up considerably by applying so-called comb methods. First proposed by Lim and Lee [50], they require an (offline) precomputation of several points that need to be stored in a (read-only) memory.

Basic comb methods, e.g., [14], proceed as follows. Let $w$ be the chosen width of the comb and $\ell = \lceil m/w \rceil$. Then one needs to precompute

$$[\alpha_{w-1}, \ldots, \alpha_1, \alpha_0]P = \alpha_{w-1}2^{(w-1)\ell}P + \cdots + \alpha_{w-1}2^{\ell}P + \alpha_{w-1}P$$

for all possible bit strings $(\alpha_{w-1}, \ldots, \alpha_1, \alpha_0)$ and store all $2^w - 1$ points in a (read-only) memory. This precomputation is very expensive, so it can only be done offline and for a fixed base point.

For the actual scalar multiplication, comb methods rearrange the scalar in a matrix, with $w$ equal sized parts forming the rows of length $\ell$. This is shown in Figure 2.9, where the length $m$ of the scalar is 160 and the width $w$ is set to 4. The scalar matrix is then processed column-wise from left-to-right in a simple double-and-add fashion. Processing time is cut by a factor of $w$ when compared to a standard multiplication algorithm. Note that also the number of doublings is decreased.

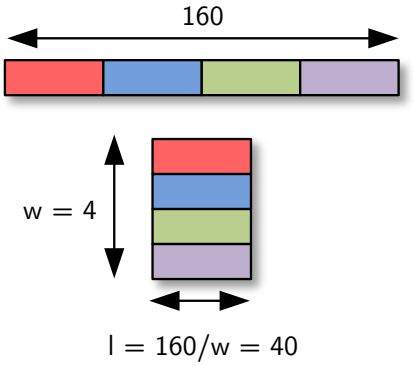



Figure 2.9: Rearranging the scalar for comb methods

Comb methods are somewhat similar to window algorithms, both require a precomputation of several points and process $w$ bits of the scalar $k$ at once. However, there are also some major differences. Comb methods rearrange the scalar $k$ into a matrix and then process its columns, window methods simply scan the scalar from left to right. For window methods the number of doublings does not decrease, however, precomputation is much cheaper and can be done online. Hence, window methods are also applicable to non-fixed base protocols, e.g., ECDSA signature verification.

Just like the binary double-and-add algorithm, the simple comb method is not secure against SCA. If all bits of a column are equal to 0, then the point addition needs to be skipped. This behavior is easily detectable in an SPA and reveals $w$ bits of the scalar per occurrence. The comb method of Hedabou et al. [29] (Algorithm 2.9) thwarts this attack scenario and also helps saving storage space. It uses the *Zeroless Signed Digit* (ZSD) recoding scheme [25] for representing the scalar, i.e., the scalar is represented using only digit values of 1 and -1. All-zero columns can obviously not occur when using a *zeroless* recoding scheme, thus the addition step is always executed and the side channel is eliminated.

**A detailed view on the Zeroless Signed Digit (ZSD) recoding.**
The ZSD representation is based on the observation that $1 = 2^w - \sum_{i=0}^{m-1} 2^i, \forall w > 0$. The binary representation of the scalar is $k = \sum_{i=0}^{m-1} k^i 2^i$, with $k_i \in \{0, 1\}$ and $n$ the bit length, and let $k$ be odd ($k_0 = 1$). Based on the mentioned observation each $b$-bit block

**Algorithm 2.9:** Hedabou's comb method [29]

**Input**: $k = (k_{n-1}, \ldots, k_0), P, w$
**Output**: $Q = kP$

$l \leftarrow \lceil n/w \rceil$;
$\kappa \leftarrow ZSD(k)$;
$R \leftarrow \mathbf{0}$;
**for** $i = l - 1$ *downto 0* **do**
  **for** $j$ *= 0 to w-1* **do** $\alpha_i = \kappa_{j \times l + i}$;
  $s = \alpha_{w-1}$;
  $R \leftarrow 2R + s \times [1, s\alpha_{w-2}, \ldots, s\alpha_0]P$;
**end**
**return** $R_0$

of $000\ldots01$ can be replaced by $b$ signed bits $1\bar{1}\bar{1}\ldots\bar{1}\bar{1}$, with $\bar{1} = -1$. Thus, the ZSD representation of $k = \sum_{i=0}^{m-1} \kappa_i 2^i, \kappa_i \in \{1, -1\}$ is defined as [25]:

$$\kappa_i = \begin{cases} 1 & \text{if } k_{i-1} = 1 \\ -1 & \text{if } k_{i-1} = 0 \end{cases} \qquad \text{for } 0 \leq i \leq n-2$$
$$\kappa_{m-1} = 1$$

This representation can be obtained by simply shifting $k$ to the right and reinterpreting all 0 bits as (-1). The hardware cost of obtaining the ZSD representation is therefore almost zero.

Apart from the added security against SCA, usage of the ZSD representation has another big advantage. A simple trick helps slashing the number of stored precomputed points in half. The set of all precomputed points is $[\alpha_{w-1}, \ldots, \alpha_0]P, \forall \alpha_i \in \{1, -1\}$. When selecting a designated sign bit, e.g., the column MSB $\alpha_{w-1}$, the precomputed points can be grouped into pairs of $\{Q, -Q\}$, with $Q = [1, \alpha_{w-2}, \ldots, \alpha_0]P$. A point $Q$ on the elliptic curve can be negated by simply negating its $y$ coordinate, i.e., if $Q = (x_1, y_1)$ then $-Q = (x_1, -y_1)$. Hence, it suffices to store only half the points, the negative counterparts are computed on the fly. The memory requirements are decreased to $2^{w-1}$ points, which is roughly half of the $2^w - 1$ points needed for traditional comb methods.

**Co-$Z$ point-addition formulæfor comb methods.**
Comb methods require so-called point doubling-additions, i.e., point operations of form $R = 2P + Q$. In Section 5.6, two schemes based on Meloni's co-$Z$ addition formulæ [54] were presented. Both feature a runtime of 11M + 7S, which is only a small improvement over non co-$Z$ methods. A higher speed-up might be expected, especially when considering the potential of co-$Z$ arithmetic.

The core observation behind the co-$Z$ addition $P + Q$ is that it provides an alternative representation of $P$, this representation $P'$ features a common $Z$ coordinate with the result. Hence, $P$ is *updated* to $P'$, which allows an immediate readdition of $P'$. To fully utilize this fact, adapted point-scalar multiplication algorithms are required. They have in common that in each step one operand point is updated to a co-$Z$ representation.

This makes such schemes incompatible with comb methods, which select the added point $Q$ out of the set of precomputed points stored in ROM. Updating just this one point with a new $Z$ coordinate is pointless, as a (potentially) different point is added in the next iteration. Thus, implementers must stick to the slower doubling-addition formulæ.

## 2.5 The Elliptic Curve DSA

The *Elliptic Curve Digital Signature Algorithm* is the elliptic curve variant of the *Digital Signature Algorithm*. It is standardized by, e.g., ANSI [2], FIPS [61] and by the Standards for Efficient Cryptography Group (SECG) [16].

**The `secp160r1` domain parameters.**
Before starting any computations, the signer and verifier must agree on domain parameters $T = \{p, a, b, G, n, h\}$. These parameters describe an elliptic curve $E$ over a finite field $\mathbb{F}_p$, also included is a base point $G$ with prime order $n$. The cofactor $h$ is defined as $|E|/n$, i.e., the ratio of total points on the curve to the order of the base point. Prime field curves are typically given in short Weierstrass form $y^2 = x^3 + ax + b$.

In this thesis the `secp160r1` parameters defined by the *Standards for Efficient Cryptography Group* (SECG) in [15, 16] are used. Note that, due to their relatively low security level, 160-bit curves have been dropped from version 2.0 of SEC 2 [17]. However, the targeted application does not have high security requirements.

According to the Standards for Efficient Cryptography 2 [16] the `secp160r1` parameters are defined by the sextuple $T = (p, a, b, G, n, h)$ with the following values.

The finite field $\mathbb{F}_p$ is defined by the 160-bit prime

$$\begin{aligned} p &= \texttt{0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 7FFFFFFF} \\ &= 2^{160} - 2^{31} - 1 \end{aligned}$$

The elliptic curve $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ is defined by

$$\begin{aligned} a &= \texttt{0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 7FFFFFFC} \\ b &= \texttt{0x1C97BEFC 54BD7A8B 65ACF89F 81D4D4AD C565FA45} \end{aligned}$$

The base point $G = (G_x, G_y)$ is defined as

$$\begin{aligned} G = (&\texttt{0x4A96B568 8EF57328 46646989 68C38BB9 13CBFC82,} \\ &\texttt{0x23A62855 3168947D 59DCC912 04235137 7AC5FB32)} \end{aligned}$$

The prime order $n$ of the base point $G$ and its cofactor $h$ are

$$\begin{aligned} n &= \texttt{0x1 00000000 00000000 0001F4C8 F927AED3 CA752257} \\ h &= 1 \end{aligned}$$

The prime $p$ is a so called pseudo-Mersenne prime, it can be written as a sum of some powers of 2. This allows the usage of the fast reduction algorithm described in Section 2.3.3. Note that $a \equiv -3 \bmod p$, this allows a faster doubling operation.

**Key-pair generation.**
After choosing domain parameters $T$, the signer has to generate his key pair. The private key $d \in [1, n-1]$ is a scalar and has to be chosen randomly in a secure, i.e., non-predictable and secret, fashion. The corresponding public key $Q$ is then computed as $Q = dP$.

Reversing this process, i.e., computing $d$ with a given $Q$, is exactly the ECDLP. It is essential that the domain parameters $T$ are chosen in a way that this problem is hard to solve.

**Signature generation.**
The ECDSA signature generation algorithm (Algorithm 2.10) computes the $2|n|$ bit signature pair $(r, s)$. First the message is hashed using a cryptographic hash function H, if the output is longer than $|n|$ it needs to be truncated. Then the base point $G$ is multiplied with a secret and random nonce $k$, the (affine) $x$ coordinate of the result is used as signature value $r$. The nonce then needs to be inversed and is multiplied with $(e + rd) \bmod n$ to retrieve the second signature part $s$.

**Algorithm 2.10:** ECDSA signature generation

**Input**: Domain Parameters $T = \{p, a, b, G, n, h\}$, private key $d$, message $m$
**Output**: Signature $(r, s)$
1 Compute hash $e = \mathrm{H}(m)$, truncate to bit length of $n$
2 Select random $k \in [1, n-1]$
3 Compute $(x, y) = kG$
4 Compute $r = x \mod n$
5 **if** $r = 0$ **then** goto 2
6 Compute $s = k^{-1}(e + rd) \mod n$
7 **if** $s = 0$ **then** goto 2
8 **return** *(r,s)*

The signature algorithm requires computation in two different prime fields. The elliptic curve $E$ is defined over $\mathbb{F}_p$, the computation of $s$ is carried out in $\mathbb{F}_n$.

The scalar $k$ must be secret, random, and it must not be reused. For a known nonce $k$ one can simply compute the private key $d = r^{-1}(ks - e) \mod n$. If the scalar is reused for signatures $(r_1, s_1), (r_2, s_2)$ of two different messages $m_1, m_2$, the key can be recovered by computing $k = (s_1 - s_2)^{-1}(e_1 - e_2) \mod n$.

It is also crucial to use a secure cryptographic hash function H, as explained in Section 2.6.1.

**Signature verification.**
The signature verification algorithm shown in Algorithm 2.11 verifies if a given pair $(r, s)$ is a valid signature for a message $m$ and a public key $Q$. First it must be checked if $r, s \in [1, n-1]$, if this is not the case the signature must be rejected. Then the message is hashed and two point multiplications are carried out. Note that the public key $Q$ is used as base point for the second multiplication $u_2Q$. The signature is accepted if the final $x$ coordinate is equal to the signature value $r$.

**Algorithm 2.11:** ECDSA signature verification

**Input**: Domain Parameters $T = \{p, a, b, G, n, h\}$, public key $Q$, message $m$, signature $(r, s)$
**Output**: Signature acceptance or rejection
1 Verify that $r, s \in [1, n-1]$, otherwise reject
2 Compute hash $e = \mathrm{H}(m)$, truncate to bit length of $n$
3 Compute $w = s^{-1} \mod n$
4 Compute $u_1 = ew \mod n$, $u_2 = rw \mod n$
5 Compute $(x, y) = u_1G + u_2Q$
6 **if** $r \equiv x \mod n$ **then** accept **else** reject

It is now shown that this algorithm is correct, i.e., the signature pair was indeed computed by a legitimate signer for message $m$. According to [28], one can rearrange $s \equiv (e + rd) \mod n$ to

$$k \equiv s^{-1}(e + dr) \equiv s^{-1}e + s^{-1}dr \equiv we + wdr \equiv u_1 + u_2 d \mod n \tag{2.2}$$

Then $(x, y) = u_1 G + u_2 Q = u_1 G + (u_2 d)G = kG$, hence $x = r$. □

## 2.6 Keccak - SHA-3

This section briefly describes cryptographic hash algorithms in general and the Keccak hash function in detail. Although not directly related to elliptic curve cryptography, a short introduction is necessary as hashing is an essential part of the ECDSA.

### 2.6.1 Cryptgraphic Hash Functions

Hash functions map an arbitrary length input to a fixed-size hash value, i.e., $h = \mathrm{H}(m)$, where $m$ stands for the input (also called message), $h$ for the hash value (also called message digest) and H for the used hash function.

Cryptographic hash functions must fulfill 3 main requirements:

**Preimage Resistance.** Given a hash value $h$ it should be infeasible to compute an $m$ such that $\mathrm{H}(m) = h$.

**Second Preimage Resistance.** Given a message $m_1$ and $h = \mathrm{H}(m_1)$, it should be infeasible to compute a second preimage $m_2 \neq m_1$ with $\mathrm{H}(m_2) = h$.

**Collision Resistance.** It should be infeasible to find two messages $m_1 \neq m_2$ such that $\mathrm{H}(m_1) = \mathrm{H}(m_2)$.

Possible applications of cryptographic hash functions include data integrity checks, password verification and pseudorandom number generation. In context of the ECDSA, the hash is used to map the input message to a fixed-length value in a secure fashion. If the hash function is not preimage resistant, then an adversary can forge a signature by determining a preimage of an already signed message hash and can then simply copy the signature. Collisions can also be used for forgery, an adversary can compute $m_1 \neq m_2$ with $\mathrm{H}(m_1) = \mathrm{H}(m_2)$ and then ask the signer to sign $m_1$, the computed signature is also valid for $m_2$.

Examples of widely used cryptographic hash functions are MD5, SHA-1, or the SHA-2 algorithm family (SHA-224/256/384/512). As most other hash functions, they are based on the Merkle-Damgård hash construction and thus feature a fixed output length $n$. The generic attack complexity of such hashes is $n$ bits of preimage security and $n/2$ bits of collision resistance.

### 2.6.2 The Hash Crisis and SHA-3 Competition

In recent years, it was shown that some widely used cryptographic hash functions have serious weaknesses. The MD5 algorithm appears to be completely broken in terms of collision resistance, as demonstrated by Wang and Yu [78]. In 2009, Stevens et al. were able to successfully forge X.509 certificates using MD5 collisions [73], proving that the use of MD5 is a real security threat. In 2005, Wang et al. presented a theoretical attack on SHA-1 [77], which allows to find collision in $2^{63}$ operations instead of the ideal $2^{80}$. Recently, Stevens presented an attack that should allow finding collisions in $2^{61}$ operations [72]. Due to these findings the SHA-1 is now considered to be broken. No attacks on the SHA-2 algorithm family (full versions) are known to date. However, the algorithmic similarity to SHA-1 and MD5 raises some concerns.

In light of these events the NIST announced an open competition, similar to the successful AES contest, to develop a new hash function named SHA-3. Out of the 51 first

round candidates, five algorithms made it into the final round. In October, 2012 NIST announced the KECCAK algorithm as the winner of the competition. Selected versions of the algorithm will be incorporated into the Secure Hash Standard (SHS) [60] as SHA-3.

Please note that at the time of writing the SHA-3 standardization process is still ongoing, the KECCAK algorithm was not yet incorporated into the SHS. Derivations from the KECCAK contest submission [10] are possible and likely. The KECCAK-$f$ permutation will most likely stay unchanged, however, the values of the tunable parameters $r$ and $c$ are subject to change. For recent events surrounding the standardization process visit the KECCAK website [43] or the NIST web page on the topic [59].

### 2.6.3 The KECCAK Algorithm

This section gives a brief overview of the KECCAK hashing algorithm, for a more detailed explanation please see the KECCAK reference [9]. Additionally, the implementation overview [11] explains different implementation techniques. For the use in the SHA-3 standard, the KECCAK authors suggested values for the parameters state size and capacity [10]. A departure from these values is absolutely possible and, although (possibly) not standard conform anymore, allows, e.g., the construction of light-weight hashes with a state size of 400 or 200 bits. Also, the security level and throughput can be tweaked by changing the security parameter $c$.

#### The Sponge Construction

In contrast to many existing hash functions typically based on the Merkle-Damgård construction, KECCAK uses the relatively new sponge construction [8]. This construction is based on iterated application of a fixed-size permutation $f$ on a state of size $b$. The $b$-bit state is split into two parts of size $r$ (rate) and $c$ (capacity), respectively. The KECCAK authors suggested to set $b = 1600$ and $c = 2n$ for SHA-3, where $n$ equals the desired output size in bits [10]. KECCAK-$f$ is used for the state permutation function $f$.

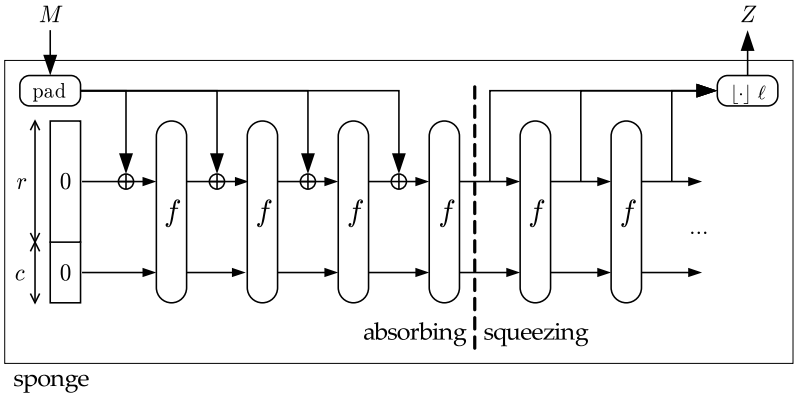



Figure 2.10: The sponge construction[1]

The hashing process is illustrated in Figure 2.10. The input message is padded to a length that is a multiple of the rate $r$, then it is cut into equal-sized blocks. During the absorbing stage, the message blocks are XORed with the $r$ low-order bits of the state,

[1]This picture has been taken from the Sponge function website [7] and is available under the Creative Commons Attribution License.

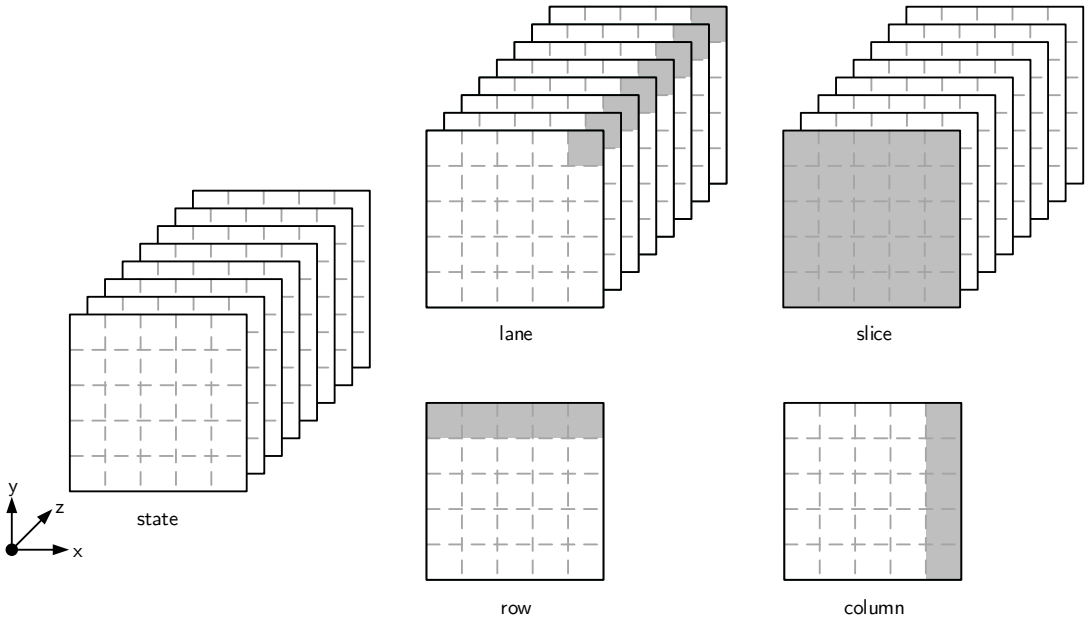



Figure 2.11: KECCAK pieces of state

each block-XOR operation is followed by an application of the permutation $f$. After having absorbed the whole message, the sponge switches to squeezing mode. During squeezing, the lower $r$ bits of the state are used as output, followed again by applications of $f$ when $n > r$.

Unlike the Merkle-Damgård construction the sponge construction allows to tweak the security level by simply choosing an appropriate value for the security parameter $c$. The sponge offers a security level of $2^{c/2}$ bit for both preimage and collision security, unless an easier generic attack, e.g., a birthday attack on a truncated output, is possible [8]. The suggested $c = 2n$ hence results in $n$ bit preimage security but $n/2$ bit collision security, which is the complexity of a generic birthday attack on the (truncated) output. Choosing $c = n$ matches both security levels.

### The KECCAK-$f$ Permutation

The KECCAK-$f$ state permutation is defined for state sizes of $b = 25 \times 2^\ell$, with $0 \leq \ell \leq 6$, the different instances are denoted by KECCAK-$f[b]$. The two largest permutation functions are KECCAK-$f[1600]$ and KECCAK-$f[800]$.

The permutation function organizes the $b$ bit state in a three-dimensional matrix with dimension $5 \times 5 \times w$, with $w = 2^\ell$. The names of the different parts of state are shown in Figure 2.11. A *lane* consists of $w = 2^\ell$ bit of constant $x$ and $y$ coordinates. A *slice* consists of 25 bits of constant $z$ coordinate, it can be further split into 5 *rows* or 5 *columns*.

KECCAK-$f$ is an iterated function consisting of a total of $12 + 2\ell$ rounds, each round consists of five different state mappings:

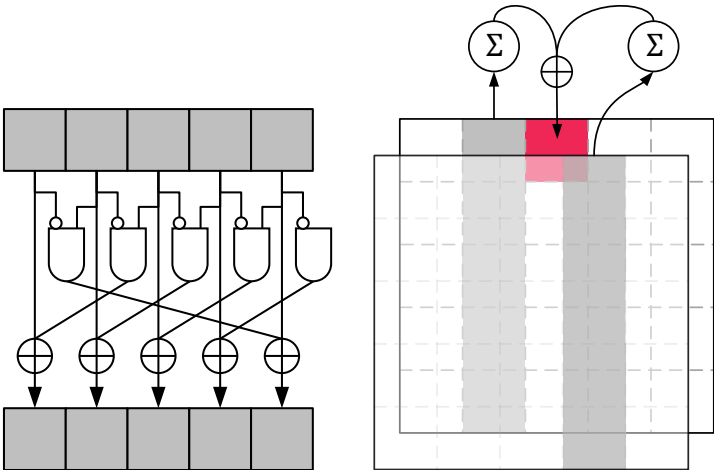
**$\theta$** The column parity of two nearby columns is computed and added (XORed) to each bit of the current column (Figure 2.12a).

**$\rho$** Each lane is rotated by a specified offset.

$\boldsymbol{\pi}$ The lanes are transposed in a fixed pattern.

$\boldsymbol{\chi}$ The 5 bits of each row are combined using logical ANDs and inverters, the result is added back to the row (Figure 2.12b).

$\boldsymbol{\iota}$ A $w$-bit round constant is added to lane[0,0].




(a) $\chi$ (b) $\theta$

Figure 2.12: KECCAK $\chi$ and $\theta$ transformation

For a more thorough explanation of the KECCAK algorithm and a listing of all constants, e.g., the rotation offset and round constants, please see the KECCAK reference [9]. Suggested values for the state size and capacity are given in the KECCAK contest submission [10].

# Chapter 3

# Secure Hardware for RFID Tags

In this chapter, a brief introduction to Radio-Frequency Identification (RFID) technology is given and the challenges that arise by the inclusion of strong cryptography on tags are discussed. Also, an overview of implementation attacks is given in Section 3.2.

## 3.1 RFID

Radio-Frequency Identification (RFID) is a contactless communication technology, it consists of three main parts: tags, readers, and a back-end system. Tags are basically microchips with an attached antenna, they are mass produced and must be very cheap. They can communicate with the reader device through an electromagnetic field. Tags can be powered by batteries (*active tags*) or draw the required energy from the field of the reader (*passive tags*). The reader communicates with the back-end system, which typically hosts some sort of database with tag data.

RFID systems are used in a broad variety of applications, such as access control, contactless payment, and logistics. Also, it is the base for the aspiring Near Field Communication (NFC) technology, which is now commonly used in banking cards and mobile phones. While most RFID tags are very simple and only store their own ID, some applications have high security demands. Adding strong cryptography to tags enables services such as proof of origin, tag and message authentication, and confidentiality.

### 3.1.1 Designing Hardware for Tags

Designing cryptographic hardware for (passive) RFID tags is a challenging task. Implementers must cope with stringent area and power requirements, while trying to satisfy the ever increasing security demands. The most basic design principles and requirements for RFID tags are now listed.

**Power consumption.**
Passive powered tags draw their energy from the field of a reader, internal capacitors are used as energy buffers. The available power depends on multiple factors, such as distance to the reader, antenna size, and operating frequency. Tags operating in the HF band (13.56 MHz) are inductively coupled, they have relatively short reading ranges but can draw several milliwatts of power, which is more than enough for most devices. UHF systems operating at roughly 900 MHz, such as EPC Gen2 tags, boast reading rangers of several meters. However, their power is severely limited, only a few microwatts can be

drawn from the electrical far field of the reader. Thus, UHF systems must cope with an available power roughly 1 000 times lower than that of HF systems [66, 68].

**Chip area and costs.**
RFID tags are typically produced in very high volumes. Their production cost is directly related to circuit size, minimizing area requirements is thus a prime goal. This is reflected by the basic RFID design principle *few gates and many cycles*, as stated by Weis [79]. In the past, several authors have given circuit size and cost estimations. Early figures were given by Sarma [69, 70, 71] and Weis [79] in 2003. They predicted that the production cost might fall under 5 dollar cents, with an available circuit area of up to 15 kGEs (but only 2 000 gates for cryptographic purposes). Advances in process technology will surely increase these figures, as noted by Feldhofer and Wolkerstorfer [20]. Also, slightly more expensive tags might feature a bigger area.

**Speed.**
Performance is often sacrificed in order to achieve low area requirements, this leads to high algorithm run- and tag response times. However, many applications demand a tag response in a limited amount of time. The ISO/IEC 14443-4 standard allows response times of up to 5 seconds, but there might be higher performance demands depending on the application.

## 3.2 Implementation Attacks on Cryptographic Hardware

The maths behind cryptography is often beautiful and elaborate, so breaking modern algorithms is nearly impossible. However, even the most sophisticated schemes are of no use if not implemented securely, i.e., if secret information is leaked. So-called implementation attacks make use of this fact, they target the specific implementation instead of the executed algorithm.

They can be grouped in two main categories. In an *active attack*, an attacker somehow interferes with on-device computations. One prominent example are fault attacks, where an attacker induces a computation error during device operations. In the case of *passive attacks*, the device is not actively manipulated. Instead, an attacker tries to deduce information on secret material by extensively monitoring some properties of the device, e.g., its execution time or power consumption.

One particular group of passive attacks, namely *Side-Channel Analysis* (SCA), has received a lot attention. SCA attacks can be based on the observation of the device's execution time [47], power consumption [48], or EM radiation [22, 65]. Recently, Genkin et al. [23] demonstrated that even the emanated sound might leak information. They were able to retrieve secret RSA keys with the help of microphones placed in a distance of up to 4 meters to the cryptographic device.

Power analysis techniques have gained the most interest over the last years. Introduced in 1999 by Kocher et al. [48], they can be applied to both symmetric and asymmetric schemes [18]. The basic setup of power analysis is as follows. The device's power consumption is measured over time, e.g., with the help of a shunt resistor and a digital oscilloscope, then stored and analyzed. In this context, the term *power trace* refers to the set of measurements taken during a single algorithm execution. Kocher described two analysis techniques, namely *Simple Power Analysis* (SPA) and the more powerful *Differential Power Analysis* (DPA). They are now shortly described, for a more thorough

explanation of power analysis please see [53]. Afterwards, an overview of fault attacks is given.

**Simple Power Analysis.**
According to Kocher, SPA involves directly interpreting the power measurements. For an SPA, an attacker usually has only a few power traces available, also he must posses detailed knowledge of the inner workings of the attacked implementation. For unprotected devices, the key might by revealed by means of a simple visual trace inspection or by template matching, as shown in [53]. Consider a device implementing the simple binary left-to-right ECC point-scalar multiplication (Algorithm 2.7). The power traces of point addition and point doubling might differ considerably, allowing an attacker to distinguish them. If two consecutive point-doubling operations are detected, then the point-addition step was skipped, which means that the scalar bit was set to 0. Hence, a single trace might be enough to reveal the entire scalar $k$.

A possible countermeasure against SPA-based attacks is to eliminate all data dependencies in the control path, e.g., by using highly-regular algorithms. Then attackers must resort to using data-dependent leakage, which is usually much weaker and harder to exploit than operation-dependent leakage.

**Differential Power Analysis.**
DPA-based attacks are more sophisticated and considered to be more powerful than SPA. They exploit variations in the power consumption stemming from different processed data values. These differences are usually very small, much smaller than that of different instructions. Thus, DPA requires a large amount of traces. In exchange, no detailed knowledge of the implementation is necessary.

They proceed as follows [53]. First an intermediate of the cryptographic algorithm must be chosen, this intermediate must be a function $f(d, k)$, where $d$ is known non-constant value (e.g., a part of the plain or ciphertext), and $d$ is a part of the key. The second step is to record many traces and store them alongside the used values of $d$. Then, the attacker must calculate hypothetical intermediate values for each possible value of $k$. All these intermediate values are afterwards mapped to hypothetical power consumption values, which requires a power model. The outcome of the attack strongly depends on the quality of this model, so some knowledge of the device is beneficial. In the final step, the hypothetical power values are compared to the measured ones. The key guess resulting in the highest correlation is most likely to be correct.

There exist several techniques aimed at thwarting DPA-based attacks. One possibility is to avoid computations of form $f(d, k)$, which might be achieved by algorithmic rearrangements or by randomization of the computed values.

**Fault attacks.**
In fault attacks, an adversary tries to induce a computation error (a fault) during execution of a cryptographic algorithm. The (potentially) faulty algorithm output is then used to deduce information on the key. Typical means of fault injection are clock glitches, power spikes, and optical methods [4].

A prime example of these schemes is the fault attack on RSA implementations utilizing the Chinese remainder theorem. This theorem allows a speed-up of private key operations, such as signing and decryption, but enables key retrieval when inducing a fault at the correct time [84]. Another instance are *safe-error* attacks, which were introduced by Yen [82, 83] and are applicable to implementations utilizing dummy operations. They are

further explained in the next section, with the example of the double-and-add always algorithm.

### 3.2.1 Attacks and Countermeasures for ECC Implementations

This section describes attacks and possible countermeasures specific to ECC implementations. Attacks usually aim to identify the scalar $k$ used in a point-scalar multiplication, so secure ECSM schemes are a necessity.

The most simple scalar-multiplication algorithm, namely the binary left-to-right approach shown in Algorithm 2.7, requires conditional branching. This makes it susceptible to SPA and timing attacks, as already mentioned earlier.

To thwart this attack scenario, Coron [18] introduced the double-and-add always algorithm (Algorithm 3.1). It simply executes a point addition regardless of the scalar bit $k_i$, i.e., a dummy operation is performed if $k_i = 0$. Safe-error attacks can exploit this fact, as noted by Yen et al. [83]. An attacker might induce a fault during a single point-addition operation, he then checks the algorithm output for correctness, e.g., by validating the signature in the case of ECDSA. If the output is still valid and the glitch did not affect the result, then the scalar bit must have been 0, otherwise it was 1.

**Algorithm 3.1:** Left-to-right binary double-and-add always method [18]

**Input**: $k = (k_{n-1}, \ldots, k_0), P$
**Output**: $Q = kP$
1 $R \leftarrow \mathbf{0}$;
2 **for** $i = n - 1$ *downto 0* **do**
3 $\quad R \leftarrow 2R$;
4 $\quad$ **if** $k_i = 1$ **then** $R \leftarrow R + P$;
5 $\quad$ **else** $R_x \leftarrow R + P$;
6 **end**
7 **return** $R$;

There exist highly regular point-multiplication schemes not reliant on dummy operations, such as the Montgomery ladder (Algorithm 2.8) and Joye's right-to-left algorithm [38]. Both feature a constant runtime and control flow, thus they provide a natural protection against SPA and fault attacks.

Basic fixed-base comb methods (Section 2.4.5) suffer from the same fate as the (unmodified) binary double-and-add approach. If all column bits evaluate to 0, then the point-addition step is skipped. Although the probability of all-zero columns decreases with higher comb widths, they are still a threat. In Hedabou's comb method (Algorithm 2.9), the scalar $k$ is first recoded using the *Zeroless Signed Digit* (ZSD) scheme, which represents an odd integer using only digits $\kappa_i \in \{-1, 1\}$. Then a point addition is required in each iteration.

**DPA.**
ECC protocols using a fixed scalar $k$, such as the Elliptic Curve Integrated Encryption Scheme (ECIES) encryption scheme or the Elliptic Curve Diffie-Hellmann (ECDH) key agreement, are potentially susceptible to DPA attacks. According to [24], three methods are considered to be effective countermeasures and are widely deployed: Randomized Projective Coordinates [18], Randomized Curve Isomorphisms, and Random Field Iso-

morphisms [39]. Especially the first is considered to be a very cheap but still effective method. When using Randomized Projective Coordinates (RPC), the (affine) coordinates of the base point are first transformed into a projective representation with a random $Z$ coordinate. As both representations are equivalent, the scalar-multiplication outcome is unchanged. However, the processed values differ in each ECSM execution, thereby thwarting DPA.

**RPA and ZPA.**
The RPA (Refined Power-Analysis Attack) [24] and the ZPA (Zero-Value Point Attack) [1], both presented in 2003, are based on the occurrence of certain points that lead to zeros in the execution path. The three mentioned anti-DPA countermeasures do not prevent these attacks, other means of protection have to be found. Both attacks require a fixed secret scalar $k$ and a user-selectable ECSM base point. Neither is the case with the ECDSA, so they are not applicable here. Example of susceptible algorithms are again the ECIES and ECDH.

**Attacks on partial nonce leaks.**
ECDSA requires secrecy of two values: the private key $d$ and the nonce $k$ (the scalar). A cryptographic nonce is a *number used only once*, it is typically required to thwart replay attacks. In 2001, Howgrave-Graham and Smart [31] demonstrated that even partial nonce leaks are dangerous. They were able to retrieve a 160-bit ECDSA private key given only 8 bits of the nonce $k$ for 30 signatures. Recently, Liu and Nguyen [51] improved on these results, they reduced the number of required nonce bits to only 2. Mulder et al. [58] also presented results, they were able to deduce the key with a 5-bit nonce leak and 4000 signatures. Moreover, they mounted a real-life attack of a smart card running ECDSA. The nonce bits were revealed using a template-matching approach (SPA) on the implemented binary inversion algorithm. As stated in Section 2.3.5, this algorithm includes data-dependent branches and loops, which makes it susceptible to SPA.

# Chapter 4

# Requirements and Design Space Exploration

Before diving into the details of the ECDSA hardware implementation, it is important to discuss the imposed requirements as well as some fundamental design choices. In this chapter, first the detailed requirements and goals are presented in Section 4.1. Then, in Section 4.2 some basic design considerations are discussed. Finally, in Section 4.3 a brief overview of the implementation is given.

## 4.1 Requirements and Goals

Goal of this thesis is to improve the state of the art in low-resource hardware implementations of the Elliptic Curve Digital Signature Algorithm (ECDSA). For this reason, an Application Specific Integrated Circuit (ASIC) coprocessor dedicated to signature generation must be designed and the results presented.

The targeted application are low-cost passively-powered RFID tags. Such tags draw the energy required for computation from the field of a reader, hence a low power consumption is required to ensure high reading ranges. Moreover, tags are mass produced, so the circuit size must be minimized in order to keep production cost low.

There already exist several low-resource ASIC implementations of the ECDSA. However, many of these proposed designs are troubled by either large area requirements or excruciating runtimes. Performance is often sacrificed in favor of small circuit area, thus following the RFID design principle *few gates and many cycles* as suggested by Weis [79]. However, there exist applications requiring fast tag response times, a runtime way above the one second mark might not be acceptable. Moreover, a low cycle count allows decreasing the operating frequency for non time-critical computations, thereby drastically reducing the power consumption. For these reasons, the implemented design should be considerably faster, but still smaller than comparable implementations. To achieve these goals novel techniques, such as fixed-base comb methods and faster point-addition formulæ, need to be deployed. Both methods decrease the runtime of the elliptic curve scalar multiplication, which is typically the most expensive operation in protocols relying on ECC.

Another main goal is to achieve security against the state of the art of implementation attacks. Countermeasures thwarting known attacks must be implemented, most promi-

nently the implementation must be able to sign data in an absolute constant (i.e., data independent) runtime.

The implementation needs to be based on the SECG-standardized curve `secp160r1`, which is defined over a 160-bit prime field. As a novelty, the KECCAK hashing function should be integrated. This is a first, as most other ECDSA implementations either rely on SHA-1/SHA-2 or do not feature hashing at all. The impact of KECCAK integration on area and power requirements should be evaluated.

The implementation must be fully-fledged, i.e., all parts of signature generation must be supported. The only exception is random number generation, implementation of a secure pseudorandom number generator is out of scope of this work.

These requirements are now summed up in bulletin form.

- ECDSA coprocessor for signature generation
- Fully fledged, but no integrated random number generation
- Smaller and considerably faster than previous work
- Achieve speed-up with fixed-base comb methods and new addition formulæ
- Secure against the state of the art of SCA and other implementation attacks
- Based on SECG-standardized curve `secp160r1` [15]
- Integration of KECCAK hashing algorithm

Some of these points need to be discussed in greater detail.

**Implementation features.**
ECDSA signatures can be used within several authentication protocols, e.g., embedded in identification schemes to allow entity authentication or in signature schemes to offer message authentication. On-tag verification of signatures is typically not required. Hence, the focus is put upon designing a small signature-only core. The implementation is designed purely as a coprocessor, its sole purpose is signature computation. For use in real-life RFID tags additional logic for protocol handling and for the RF frontend is required.

The implementation of a true or pseudorandom number generator (RNG) is out of scope of this thesis. RNGs can be based on, e.g., hash functions, block and stream ciphers, or dedicated algorithms. They require a truly random number, called the *seed*, as an initial value. Acquiring this seed value is a difficult task, as true random number generators are difficult to implement and external sources might not be trustworthy.

**The `secp160r1` curve.**
The `secp160r1` domain parameters define an elliptic curve over a 160-bit prime field. While hardware implementations of binary-field curves are generally smaller and faster [81], they typically lack signing capabilities. ECDSA requires prime-field arithmetic regardless of the chosen curve type, addition of prime-field units more than obliterates the initial area advantage of binary-field curves (see [81] for an example). Moreover, they are often excluded in security standards, e.g., there are no binary-field curves in NSA's Suite B cryptography standards.

In general an $n$ bit curve has a security level of $n/2$ bits [28]. Due to the relatively low security level of 80 bits, all 160-bit curves have been dropped from the current version 2.0 of the *Recommended Elliptic Curve Domain Parameters* by the SECG [17]. However, the targeted application—very low-cost RFID tags—does typically not require high security

levels, achieving a small circuit area and hence lowering the cost is often more important. The runtime also benefits from the use of smaller curves.

The `secp160r1` curve is defined over a 160-bit pseudo-Mersenne prime $p$, the group order $n$ of the fixed base point $G$ is 161 bits long. Starting from now, $p$ and $n$ refers to these specific values, $\mathbb{F}_p$ and $\mathbb{F}_n$ denotes the respective prime fields.

**Hashing - KECCAK.**
For the hashing operation the KECCAK algorithm, which was selected as the winner of the SHA-3 competition held by the National Institute of Standards and Technology (NIST), is used. The selected instance is KECCAK[r=640, c=160]. The Digital-Signature Standard (DSS) [61] specifies that an algorithm specified in the Secure Hash Standard (SHS) [60] is to be used for ECDSA. The standard includes SHA-1 and the SHA-2 algorithm family. At the time of writing, the winner of the SHA-3 competition was already chosen, however, the standardization process is still ongoing and it remains open if the used KECCAK instance will be part of the standard. Absolute standard conformity is not a necessity for this work, so usage of a non-standard hash is acceptable.

SHA-1, with its output length of 160 bits, seems to be predestined for usage with a 160-bit curve. Longer hashes, like those produced by the SHA-2 family, have to be truncated, so the use of such typically more complex algorithms does not increase the security level. However, concerns over the security of SHA-1 were raised in the past. Another driving factor in the choice of the hashing algorithm is the differentiation from previous ECDSA implementations using SHA-1 [33, 44, 80]. Integration of KECCAK into ECDSA is a novelty, so a first analysis of area and power impact can be given.

The KECCAK algorithm family offers a tunable hashing algorithm. Due to the use of the sponge construction the output length is variable, the security level can be influenced by choosing an appropriate capacity $c$. The parameters are chosen to fit the selected ECDSA domain parameters, i.e., both the output length and the capacity are set to 160 bits, which results in a security level of 80 bits for preimage and collision security. Recall that hash algorithms based on the Merkle-Damgård construction, such as SHA-1 and SHA-2, offer $n$ bits of preimage security and $n/2$ bits of security against collisions, with $n$ being the length of the message digest. A preimage security vastly higher than collision resistance is not needed, lowering it to $n/2$ by setting $c = n$ speeds up the hashing process.

The chosen parameters result in a state size of 800 bits, which is the second biggest specified KECCAK state size right after 1600 bits. The KECCAK authors proposed to use the 1600-bit state version in the SHA-3 standard, but smaller state sizes allow faster hashing while using less memory for state storage. The circuit area of hardware implementations also benefits from smaller state sizes.

## 4.2 Basic Design Considerations

In this section, some core ideas of the hardware design are presented. They act as a basis for the implementation presented in Chapter 5.

**Storage type.**
Elliptic curve operations as well as the KECCAK algorithm require a lot of storage space. While memory requirements are kept relatively low by selecting a 160-bit curve, they still amount to well over 1 kbits. Previous work often utilized some sort of synthesized dual-port memory block (e.g., [30, 33, 44]), which results in extremely high area requirements. It was thus chosen to use a single-ported SRAM macro as the main storage element, such

specialized macros typically require much less resources than standard-cell based memory blocks. A single SRAM cell consists of 6 transistors and thus requires 1.5 GEs, in contrast, even the most simple flip-flop requires 5.5 GEs (in the chosen process). While there also exist dual-port RAM macros, they require roughly twice the area of single-ported versions.

Using macros introduces a technology dependency, porting the design to other, e.g. more modern, process technologies requires switching the macro. However, this does not affect the computation core and is an acceptable price to pay for the area savings. To keep things simple, the clock frequency of the RAM is set equal to that of the computation core. The choice of a single-ported memory has severe impacts on the overall design. Concurrent read and write operations are not possible, also only a single data word can be fetched at each cycle. In order to, still attain a high speed, the implementation core has to fully utilize the bus width while keeping memory wait cycles to an absolute minimum.

Constants required for computation are stored in a dedicated ROM, some later listed decisions inflate the memory requirements to almost 3 kbits. Despite this high number, the ROM is based on standard cells instead of a macro. The much higher memory density of ROMs brings down the hardware cost to an acceptable level, modern hardware synthesizers can implement such ROMs in less than 1 kGEs.

**Datapath and memory width.**
An important decision to make is that of the datapath and memory width. RAM and ROM bus interface should obviously have a width equal to that of the datapath, then fetched operands can be immediately processed. A full-precision solution, i.e., a 160-bit datapath capable of processing an entire field element at once, might be suitable for high-speed implementations, for a low-cost application the extremely high hardware cost immediately disqualifies this approach. A multi-precision solution is a much better fit for a low-resource implementation. Typical datapath sizes include 8, 16, 32 and 64 bits, other (less conventional) widths were not considered. Here no detailed runtime or hardware cost analysis is carried out, instead the choice is based on simple estimations and arguments.

A 64-bit datapath, while without doubt a very fast option, is too costly in terms of area and power. Also, 160 is not a multiple of 64, making implementation even more complex. On the opposite side of the spectrum, an 8-bit datapath is very area friendly but comes with an excruciating performance penalty. After eliminating these two possibilities, the options 16 and 32 are left.

The majority of a signing operation's runtime is typically spent for modular multiplications or squarings. The runtime of basic multi-precision multiplication algorithms has a quadratic complexity[1], i.e., slashing the word size in half increases the runtime by a factor of four. Also, the lack of a dual-ported memory needs to be compensated somehow, e.g., by choosing a wider bus. These considerations led to the choice of a 32-bit datapath and memory interface.

**Integer multiplier.**
At the core of all multi-precision multiplication schemes, an integer multiplication unit is found, the presented design also needs to incorporate such a multiplier in its datapath. Its width needs to be discussed separately. Setting the multiplier width equal to the rest of the datapath, i.e., include a $32 \times 32$ bit multiplier, seems tempting, but has severe shortcomings. The hardware complexity (gate count) of a basic $n \times n$ bit integer multiplier is quadratic, it requires roughly $n^2$ AND gates and adders. This makes multipliers

[1] Sub-quadratic multiplication algorithms like Karatsuba's technique were not considered because it usually requires more hardware resources than quadratic algorithms.

extremely costly, too costly to include a 32-bit instance in a low-resource design. Also, due to the absence of a dual-ported RAM, the memory is not fast enough to fully utilize the multiplier. Two memory cycles are required for fetching both multiplication operands while the actual multiplication takes only one, i.e., the multiplier is idling half the time. When including RAM-setup and result-write cycles into this calculation the outcome gets even worse.

For these reasons it was decided to include a $16 \times 16$ bit integer multiplier. The chosen bus width favors 32-bit multiplications, so the smaller instance is used to compute $32 \times 32$ bit multiplications during the course of multiple, or more exactly four, cycles. Operand load and result write cycles can be streamlined into this multi-cycle process, this ensures maximum multiplier utilization. When compared to a dedicated 32-bit multiplier this scheme reduces the hardware cost by a factor four, while the runtime increases by a factor of less than two.

**Speeding up the ECSM with comb methods.**
While a low area and power consumption are the primary goals of this work, the runtime, often a weakness of other low-resource designs, should not be neglected. The elliptic curve scalar multiplication (ECSM) is by far the most time consuming part of the ECDSA, so it is essential to cut computation time here. Comb methods (Section 2.4.5) allow a drastic speed-up, thus it was decided to incorporate them into the design.

For a chosen comb width $w$ the achieved speed up is linear, i.e., the original computation time is divided by $w$. The performance gain is paid by high memory requirements, roughly $2^w$ points on the curve need to be precomputed and stored in a ROM. A trade-off between the linear speed-up and the exponential storage requirements needs to be found.

**New elliptic curve addition formulæ.**
In 2007, Meloni introduced new elliptic curve addition formulæ later dubbed as co-$Z$ addition formulæ [54]. They offer a speed-up over other methods and led to a fair amount of further research. Their disadvantage is that they require modified scalar multiplication algorithms. A refinement of co-$Z$ arithmetic and several point-multiplication algorithms were presented by Goundar et al. [25]. In [34], Hutter et al. presented a co-$Z$ based version of the Montgomery ladder algorithm. Longa and Miri [52] presented a fast co-$Z$ doubling-addition scheme, which can also be used in more tradition scalar-multiplication algorithms. The presented formulæ should be evaluated in the context of comb methods, if a suitable algorithm is found then they should be incorporated into the design.

**Security against SCA.**
A chain is only as strong as its weakest link, so implementing secure algorithms without considering implementation attacks is grossly negligent. The presented implementation must be secure against the state of the art of implementation attacks, ranging from simple and differential power analysis to fault attacks. One essential part in securing the design is to assure an absolute constant runtime and operation flow, no data dependency should exist in the control path.

This requirement restricts the freedom of algorithm choice on several occasions. The most prominent example is the computation of modular multiplicative inversions (cf. Section 5.7). Two widely used algorithms, namely the binary inversion and the Montgomery inversion algorithm, do not feature a constant runtime, and thus can not be used. It needs to be resorted to other, typically slower, algorithms. For this thesis inversions should be performed by utilizing Fermat's little theorem, i.e., by means of modular exponentiation. This approach can easily be executed in constant time, but is relatively slow.

The point-scalar multiplication is another potential pitfall. The runtime of the most basic comb methods is dependent on the random and secret nonce $k$. There exist alterations of the comb method that eliminate this dependency. More details are given in Section 5.5.

Finally, modular reduction must also be performed in constant time. The prime $p$ defined in `secp160r1` is a pseudo-Mersenne prime, it allows usage of a fast-reduction mechanism. The order $n$ of the base point $G$ is not of any special form, hence the fast-reduction algorithm is not applicable and a more general approach must be found. The Barret reduction scheme relies on conditional additions and subtractions, which conflicts with the constant runtime requirement. The Montgomery multiplication scheme can be modified to fulfill this important requirement, thus it was decided to use it for computations in $\mathbb{F}_n$.

**Power-saving measures.**
There exist two simple generic techniques that allow a reduction of power consumption, namely *clock gating* and *operand isolation*. Both methods aim to reduce switching activity, which is the main contributor to power consumption in CMOS technology. For clock gating, registers are only clocked if new values are to be stored. So-called clock-gating cells disable the clock signal for inactive registers, thereby reducing their dynamic power consumption to zero. The second technique, operand isolation, can be applied to combinational blocks. If an output of such a block is not needed in the current cycle, the input is set to a constant value. Both techniques should be applied.

## 4.3 Implementation Overview

Figure 4.1 depicts the overall structure of the implementation. Alongside the ECDSA computation core, it contains a 1 504-bit RAM and a 2 976-bit ROM, both featuring a wordsize of 32 bits. The core can be split into three major parts, a controlling block, a datapath, and an AMBA APB interface.

The AMBA Advanced Peripheral Bus (APB), a standardized yet simple interface implemented according to [3], allows communication of an outside entity, i.e., a bus master, with the core. The functionality of the bus includes controlling the core, querying its current status, writing the message and finally retrieving the signature pair $(r, s)$. The width of the read and write data buses is left open in the specification, for this implementation a width of 8 bits was chosen.

The datapath contains separate modules for elliptic curve and Keccak computations, both share a common register file. The controlling block is split up into multiple sub-controllers. The multiplication controller steers the process of modular multiplication in both finite fields $\mathbb{F}_p$ and $\mathbb{F}_n$ and implements two different reduction algorithms. The elliptic curve doubling-addition (EC-DA) controller houses the program code needed for the execution of the chosen elliptic curve addition formulæ. The SHA controller, used for the hashing operation, is taken from previous work (cf. [64]) and slightly modified to accommodate to the differing security parameters. Finally, the top-level controller oversees the signing process by operating the sub-controllers as well as providing the logic needed for field inversion, amongst other things.

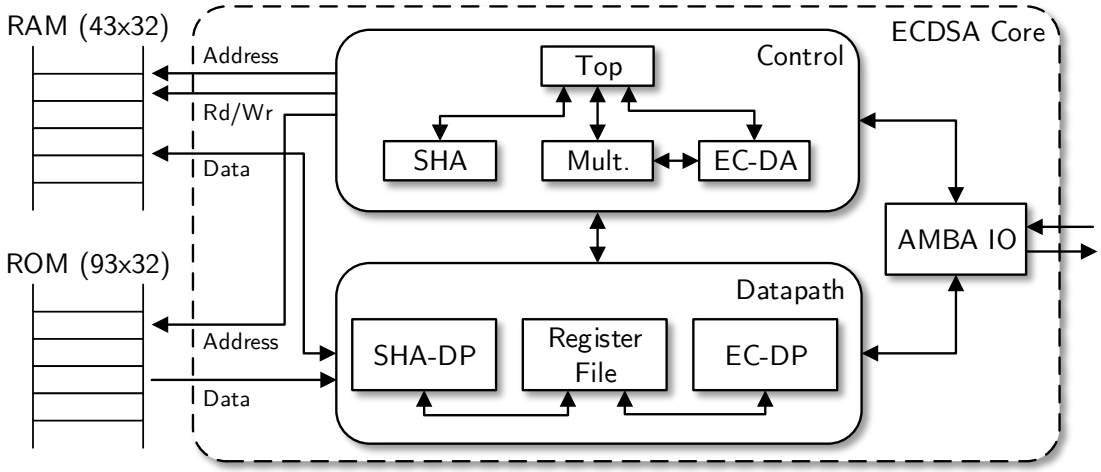



Figure 4.1: Top-level view of the ECDSA core

## 4.4 Summary

In this chapter, the fundamental implementation requirements and goals were presented. A low-resource core capable of generating ECDSA signatures should be implemented, it should be both smaller and considerably faster than previously proposed designs. As a novelty, the KECCAK hashing algorithm should be incorporated. Also, security against modern implementation attacks must be achieved. Then some core design ideas were discussed. A 16-bit multiplier should be embedded into the 32-bit datapath. The integration of techniques such as fixed-base comb methods and new point-addition formulæ should allow a drastic speed-up of computations. Finally, a brief overview of the design was given.

# Chapter 5

# Elliptic-Curve Modules

After having outlined the basics of elliptic curve cryptography in Chapter 2 and discussing the requirements and the most important design choices in Chapter 4, it is now time to dive into the implementation details. In this chapter, the detailed hardware design used for signature computation (with the exception of hashing) is discussed. First the basic datapath is presented, then the focus shifts to the higher-level parts of the ECDSA.

To get the big picture, some corner stones of the implementation are summed up below.

- 32-bit datapath
  - 67 bit accumulator allowing multiply-accumulate operation
  - $16 \times 16$ bit multiplier used in a pipelined product-scanning multiplication
  - Dedicated fast-squaring operation
  - Dual-field ($\mathbb{F}_p$, $\mathbb{F}_n$) support
- $47 \times 32$ bit single-ported SRAM for storage (i.e., 1 504 bits)
- $93 \times 32$ bit ROM for storing of constants (i.e., 2 976 bits)
- Hybrid finite-state machine and microcontroller-like controlling, split into multiple dedicated controllers

The remainder of this chapter is organized as follows. In Section 5.1, the layout of both RAM and ROM is shown, also the exact contents of the ROM are detailed. In Section 5.2, the datapath is presented and it is shown how basic arithmetic operations, e.g., addition, multiplication or squaring, are performed. In Sections 5.3 and 5.4, the efficient implementation of modular arithmetic in the fields $\mathbb{F}_p$ and $\mathbb{F}_n$ is discussed. Using this foundation the implementation of the more higher-level aspects of ECDSA are presented. Section 5.5 is mainly concerned with selecting an appropriate value for the comb method. All comb methods require so-called *doubling-additions*, a fast and efficient implementation of this operation is outlined in Section 5.6. In Section 5.7, optimized algorithms for computing modular multiplicative inversions are presented. The implementation details of the top-level controller, which is in charge of overseeing the signing process, are given in Section 5.8. Protecting the implementation from all sorts of side-channel and other implementation attacks is a major goal, in Section 5.9 applied countermeasures are presented and possible remaining attack points are given.

## 5.1 Memory Organization

This section describes the memory layout of both RAM and ROM. Due to the choice of the ECDSA domain parameters most stored elements feature a bit length of 160 bits. RAM as well as ROM is partitioned into so-called *field registers*, most field register are comprised of five 32-bit words. They are denoted with $\mathbf{R_x}$, where $x$ is the index of the register. The name *field register* stems from the fact that they are used to store elements out of the fields $\mathbb{F}_p$ and $\mathbb{F}_n$.

The 1 504-bit RAM consists of $47 \times 32$ bit words organized into 9 field registers. The lower registers $\mathbf{R_0}$ to $\mathbf{R_7}$ are 160 bits long while the up-most $\mathbf{R_8}$ features 224 bits, its extra bits are required for the fast-reduction process in $\mathbb{F}_p$ (Section 5.3). The RAM register allocation, i.e., which value is stored in which register, depends on the respective operation. For details please refer to the following sections and the program code listed in Appendix B.

The ROM stores $93 \times 32$ bit words partitioned into 19 field registers, which sums up to a total of 2 976 bits. The contents of the ROM are pictured in Figure 5.1. It contains the precomputed points for the comb method, constants needed for the Montgomery multiplication scheme and the Montgomery representation of the private key $d$. The key was added to the (hardcoded) ROM for the sake of simplicity, in a real life scenario it needs to be moved to a writable memory block, e.g., an EPROM. The comb width $w$ was chosen to be 4, thus $2^{w-1} = 8$ points need to be precomputed and dumped into ROM. Points are stored in affine coordinates, so two 160-bit field registers are needed to store both the $x$ and $y$ coordinates. Reducing storage requirements to only a single coordinate per point is not possible, in contrast to the Montgomery ladder algorithm there exists no $x$-coordinate only version of the comb method. The coordinates of each point $P_n$ are stored in the ROM registers $\mathbf{R_{2n}}$ ($x$ coordinate) and $\mathbf{R_{2n+1}}$ ($y$ coordinate), respectively. The constant $R^2 \bmod n$ is needed for transformations into the Montgomery domain (Section 5.4.4) and is stored in $\mathbf{R_{16}}$. The 96-bit $\mathbf{R_{17}}$ stores the 3 low-order words of the modulus $n$, the upper words are not stored as they are either zero or one. Finally, the private key $d$ (in Montgomery representation $dR \bmod n$) is kept in ROM register $\mathbf{R_{18}}$.

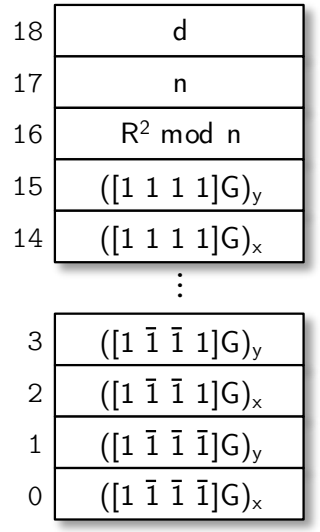



Figure 5.1: ROM contents (address on left denotes the field register). $\bar{1} = (-1)$

All 160-bit field registers, with the exception of the key register, are kept at the lower part of the respective address space to allow easy addressing. The address adder depicted in Figure 5.2 multiplies the 5-bit register index with 5 and adds the 3-bit word offset to retrieve the wanted address. The same address signal is fed to both RAM and ROM, thus only a single address adder is needed. Concurrent operations on RAM and ROM are not possible.

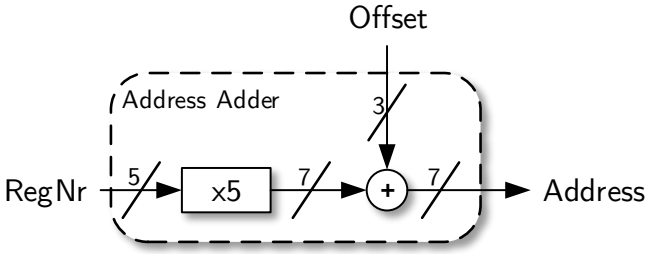



Figure 5.2: Address adder

## 5.2 The Datapath and Basic Operations

This section describes the overall structure of the datapath and tries to explain how the most basic arithmetic operations, e.g., multi-precision addition and multiplication, are carried out. The later sections then describe how these operations are used to build complex algorithms.

The datapath, shown in Figure 5.3, is comprised of a multiplication (left) and an accumulation path (right). A 67-bit accumulator register (ACC) alongside a 67-bit adder, which allows adding an integer to the accumulator, form an accumulation unit. Multiplexers allow shifting the current adder output by 32 bits to the right. The lower 32 bits of the adder output are fed to the RAM, a write-enable signal determines if the current output is actually written. Writing is usually, but not always, activated alongside the shifting operation.

The accumulation operand can be selected from the inverted or non-inverted RAM output, the multiplication result or (a 32-bit part of) the constant modulus $p$. Thanks to the highly regular modulus $p$—only a single bit is zero, all others are one—it is possible to hardcode this constant without any noteworthy area gain. The selected operand is then routed through two configurable shifters, the first one can shift its input up to 3 bits to the left and thus produces a 35-bit output, while the second shifter can shift this result by either 0, 16, or 32 bits to the left.

The multiplication part of the datapath consists of two multiplication operand registers A, B and a $16 \times 16$ bit integer multiplier producing a 32-bit output. The operand registers are made up of 16-bit chunks, A consists of 3 parts A0 to A2 while B consists of 2 parts B0 and B1. The operand registers are used to perform a pipelined multiplication, which will be explained in Section 5.2.2.

### 5.2.1 Basic Arithmetic Operations

Before diving into more complex matter the most basic operations shall now be discussed. Due to the chosen domain parameters (160-bit curve) all integers processed are 160 bits

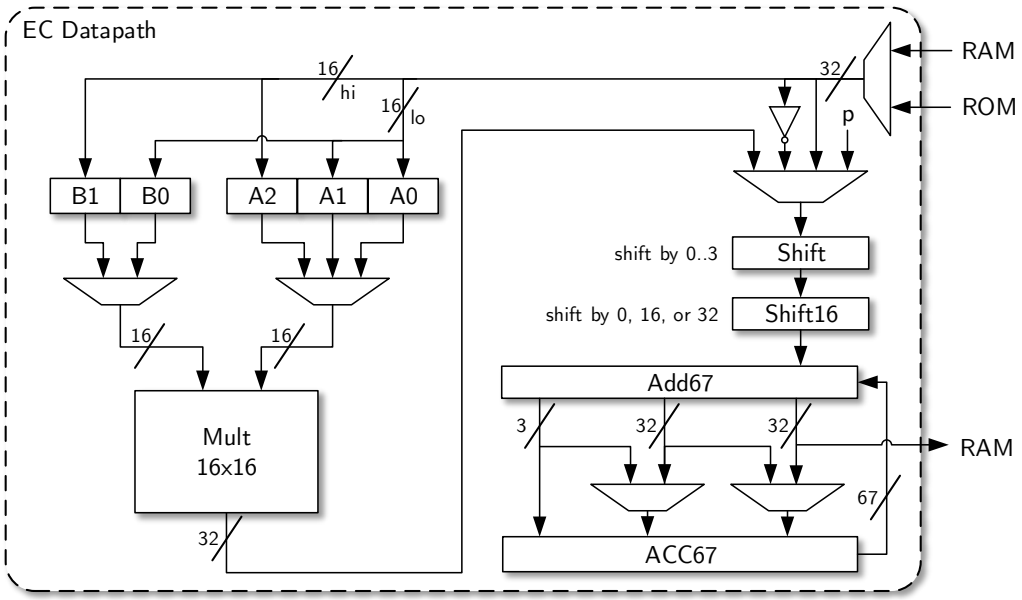



Figure 5.3: ECDSA datapath

long. Both the RAM interface and the basic datapath have a width of 32 bits, thus each integer has to be split into five 32-bit memory words.

The most basic operation is a simple addition of two integers. Computation starts with the least significant word 0. This lowest-order word of the first operand is fetched from memory and added to the accumulator, which is assured to have a value of zero at the beginning of the operation. This is then repeated for the second operand. The addition result (adder output) is shifted by 32 bits to the right and the lowest 32 bits are written to the RAM. Everything above those 32 bits is now stored in the lower bits of the accumulator. This process is repeated for all other words of the integers.

Subtraction is very similar to addition. The two's complement representation of the subtrahend, which is retrieved by inversion and addition of 1, is added to the minuend. In the subsequent reduction step a possible negative result would need to be treated differently than a positive one, such differing behavior should be avoided as it poses as a potential side channel. To ensure that the result of a modular subtraction is always positive, a multiple of the modulus $p$ is added to the difference. Note that, due to working in $\mathbb{F}_p$, this does not change the outcome of the operation.

The first shifter allows shifting each operand a configurable number of up to 3 bits to the left before adding it to the accumulator, this is equal to a multiplication with a power of two.

### 5.2.2 Pipelined Multiplication

Modular multiplication is both the most time and area consuming operation needed for ECDSA, thus an efficient multiplication scheme is vital. Starting from a very basic integer multiplier algorithms for fast and efficient modular multiplication and squaring were built and are now discussed.

At the very center of all multi-precision multiplication algorithms is an integer multiplier. To illustrate the structure of such a multiplier, a simple 4 by 4 bit instance is

shown in Figure 5.4, it computes the 8-bit product $p = a \times b$ of two 4-bit integers $a, b$. A single-bit multiplication corresponds to an AND combination of the two operand bits. To perform an $n \times n$ bit multiplication each bit of operand $a$ needs to be combined with each bit of operand $b$, hence a total of $n^2$ AND gates are required. The partial products are summed up with simple adder chains. This implementation features a 16 by 16 bit multiplier, which computes the 32-bit product of two 16-bit operands and needs $n^2 = 256$ AND and $n(n-1) = 240$ adder gates.

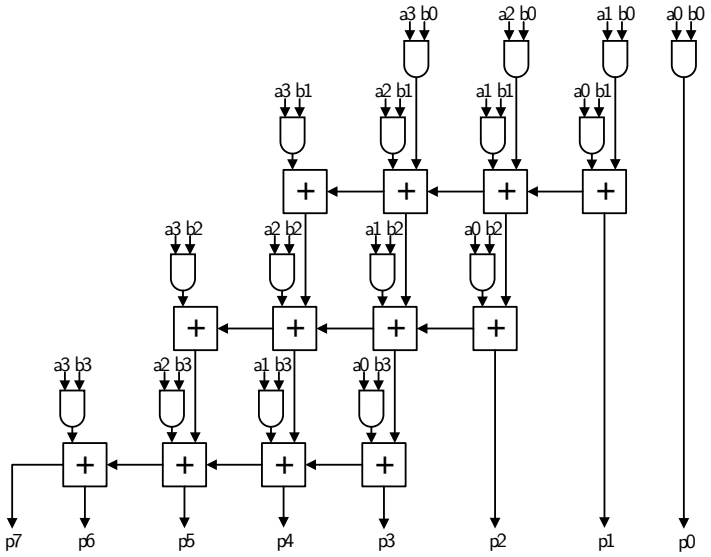
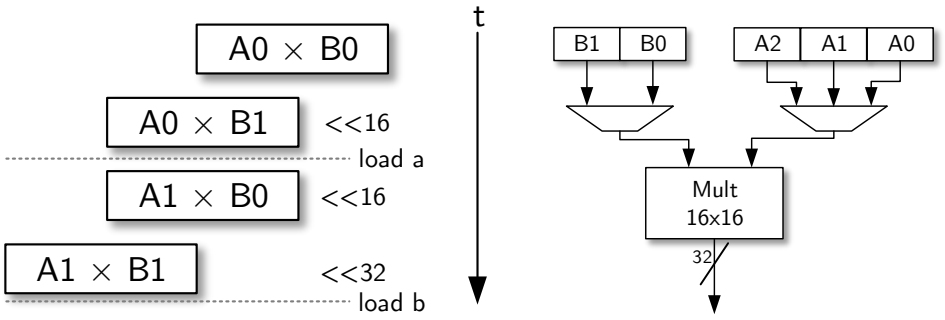



Figure 5.4: $4 \times 4$ integer multiplier

To fully utilize the 32-bit single-port memory interface, $32 \times 32$ bit multiplications are computed with the help of the 16-bit multiplier. A 32-bit multiplication takes four cycles and is done with a simple school book multiplication algorithm, as illustrated in Figure 5.5a. Both 16-bit chunks of the first operand (A0 and A1) are multiplied with both chunks of the second operand (B0 and B1). The 32-bit partial products are first shifted accordingly using the second shifter and then added to the accumulator, thus utilizing the multiply-and-accumulate (MAC) functionality of the datapath. More sophisticated multiplication algorithms might speed up the process considerably, e.g., the Karatsuba multiplication [42] would reduce the number of cycles needed for a 32-bit multiplication to three. However, the area overhead would be significant, which is why the shown school book multiplication scheme was chosen.

Prior to the start of the 32-bit multiplication, both operands need to be loaded into the designated operand registers, they are then selected using multiplexers (see Figure 5.5b). Dedicated operand load cycles would slow down the multiplication process considerably, thus the operands for the next 32-bit multiplication are fetched during execution of the current one. As denoted by dotted lines in Figure 5.5a, operand $b$ is replaced at the end of the fourth cycle, operand A after the second cycle. However, the 16 high-order bits of $a$ (A1) are still needed in the third and fourth multiplication cycle, for this reason there are three 16-bit operand registers for $a$. The 16 high-order bits of operand $a$ are alternately stored in the operand registers A1 and A2. If the value of register A1 is still needed, the high-order bits of the loaded operand are written to A2, and vice versa.




(a) Instruction flow of a single 32 bit pipelined multiplication

(b) Multiplier with operand registers

Figure 5.5: Overview over the pipelined multiplication

This scheme is an example of a two-stage pipeline, the operand is fetched in the first stage while the actual multiplication is performed in the second stage.

### 5.2.3 Multi-Precision Multiplication

Two multi-precision multiplication techniques were discussed in Section 2.3.2, namely the *operand-scanning* and the *product-scanning* multiplication. The product-scanning approach (Figure 2.6) was chosen for this work, it offers several advantages over the operand-scanning technique. It works exceptionally well in combination with a multiply-and-accumulate unit, which is also used by the pipelined multiplication approach presented earlier. The number of operand load cycles is higher when compared to operand-scanning, however, they are streamlined into the pipelined multiplication process.

The 160-bit multiplication operands are split into five 32-bit words, thus a total of $5^2 = 25$ partial products need to be computed and summed up. Using the presented pipelined multiplication scheme, this can be done in a minimum of $25 \times 4 = 100$ cycles.

Some additions and alterations are made to adopt the multiplication algorithm to modular multiplication. Modular multiplication in $\mathbb{F}_p$ is discussed in Section 5.3.1, multiplication in $\mathbb{F}_n$ using the Montgomery multiplication scheme and Integrated Product Scanning in Section 5.4.

### 5.2.4 Fast Squaring

The squaring operation is equal to a multiplication with identical operands. However, there exists a very easy and cheap way of speeding up the process. Due to the commutative property of multiplication, i.e., $a[i] \times a[j] = a[j] \times a[i], \forall i \neq j$, and hence $a[i] \times a[j] + a[j] \times a[i] = 2(a[i] \times a[j])$, one can skip computation of roughly half the partial products. This is shown in Figure 5.6, grey dots can be skipped and the number of necessary partial products is reduced to 15. The doubling operation, needed whenever $i \neq j$ (blue dots), is equivalent to a binary shift to the left by one bit, which is achieved with the first shifter in the shifter chain. The hardware cost of this simple optimization is almost zero, while it speeds up the processes of both field inversion and point-scalar multiplication tremendously.

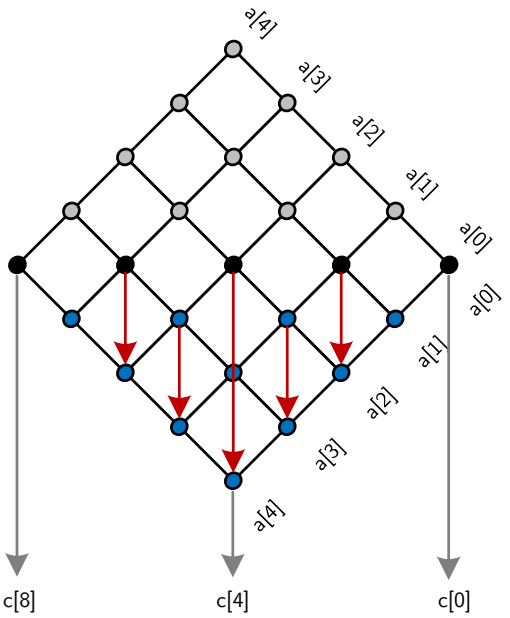



Figure 5.6: Fast squaring with product scanning

### 5.2.5 The Multiplication Controller

The multiplication process is steered by the dedicated multiplication controller. It is implemented using a finite-state machine approach, the FSM including all states and possible transitions is depicted in Figure 5.7. During execution of the states InitLoadA and InitLoadB the multiplication pipeline is initialized, i.e., the operands of the first 32-bit multiplication are fetched from memory and stored in the operand registers. The actual pipelined multiplications are then carried out in the Mult state. In WriteAcc the current accumulator content is written to RAM and the accumulator value is shifted to the right. The states RedAdd, Reduction0, and Reduction1 are used for reduction in $\mathbb{F}_p$, AccM and SubN are part of the Montgomery multiplication scheme for performing computations in $\mathbb{F}_n$.

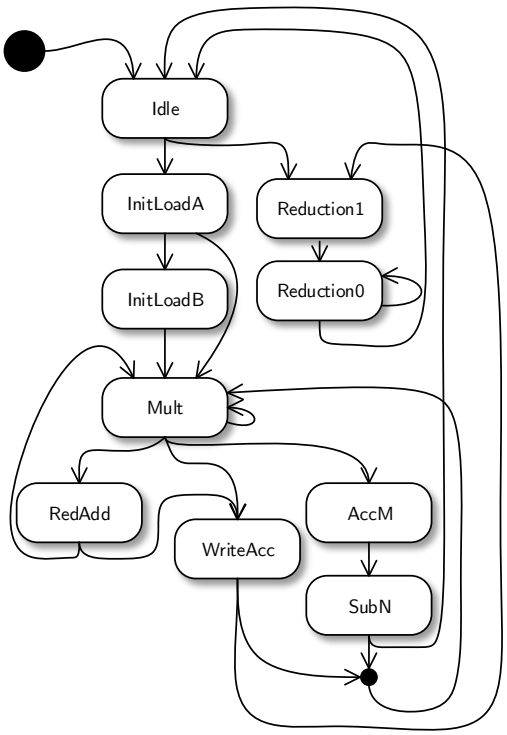



Figure 5.7: Multiplication controller FSM

## 5.3 Modular Arithmetic in $\mathbb{F}_p$

An elliptic curve is defined over an underlying field, which in the case of the `secp160r1` curve is a finite prime field $\mathbb{F}_p$ with a 160-bit prime $p$. This section presents the implemented modular-arithmetic algorithms.

The prime $p$ is a so called pseudo-Mersenne prime that permits fast reduction (Section 2.3.3). Recall that an integer $x > p$ can be reduced by splitting it in $x = h \cdot 2^{160} + \ell$ and then computing $\ell + h + (h << 31)$, i.e., reduction is achieved using shifts by 31 bits and additions. 31 is not a multiple of the word size 32, thus disallowing shifting by simple addressing. Instead the datapath is modified to allow addition of a 32-bit word, i.e., a word of $h$, in two different locations of the accumulator concurrently. As pictured in Figure 5.8, an additional single-bit adder is necessary for handling the overlapping bit 31. A dual-digit carry addition is avoided by storing the carry in a dedicated register instead of propagating it to the next adder in the chain. After advancing to the next multiplication column, this carry bit is added to the LSB of the accumulator.

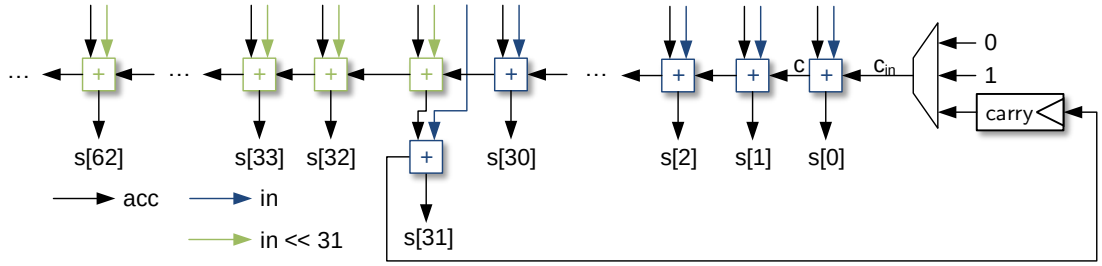



Figure 5.8: Structure of the adder

The fast reduction algorithm is not only used in multiplication, but also for the basic operations addition, subtraction, and shifting. The alternative reduction method of conditional subtraction of the modulus $p$ (Algorithm 2.1) does not offer a constant runtime, it is also difficult to implement whenever the result range exceeds $2p$.

Reduction works as follows. The unreduced operation result is first stored in the 224-bit RAM register $\mathbf{R_8}$. Then word after word of the lower 160 bits ($l$) is fetched and added to the accumulator, whenever necessary a word of $h$ is added in two places using the presented adder structure.

### 5.3.1 Modular Multiplication

Modular multiplication is the most time critical part of an ECSM, thus it is important to design a highly efficient multiplication algorithm. The minimal time spent for a single 160-bit multi-precision multiplication is already established to $25 \times 4 = 100$ cycles, it is now necessary to minimize the cycle count for modular reduction.

The simplest approach is to keep multiplication and reduction separate, i.e., to calculate the 320-bit product and to reduce it afterwards. However, this has several problems. First, storing the intermediate result needs lots of memory, and second, it is slow due to the additional memory access cycles.

A more efficient approach is now presented. The reduction is integrated into the multiplication process, this gives a time and storage space advantage. As seen in Figure 5.9,

multiplication is performed in two phases. First only the upper columns 5 to 8 are processed, the 160-bit result $H$ is stored in RAM register $\mathbf{R_8}$. Then multiplication continues with the lower columns. After having finished a column the appropriate word of $H$ is added twice using the aforementioned adder structure. This produces $L + H + (H << 31)$, where $L$ is the 195 bit product of the lower five columns. The 195-bit sum is again stored in $\mathbf{R_8}$, which for this purpose is the only one longer than 160 bits. Finally, another round of reduction is performed, i.e., the upper part of the intermediate is added twice to the lower-order bits, this produces the final 160-bit output.

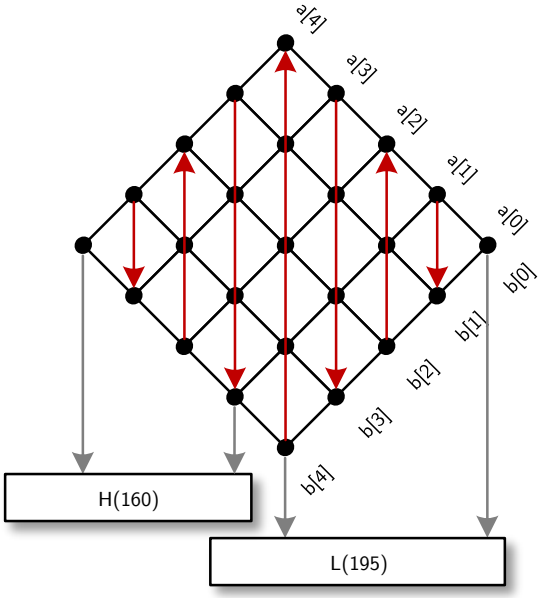



Figure 5.9: Modular multiplication in $\mathbb{F}_p$

Note that all intermediate values are stored in register $\mathbf{R_8}$, only in the very last step (the second reduction round) the result is written to the designated register. This fact allows the usage of *in-place* multiplication, i.e., the result can overwrite an operand. In contrast, this is not possible with *out-of-place* multiplication, the result and operand field registers must be different. The register mapping for the elliptic curve addition formulæ presented in Algorithm 5.2 takes this fact into account.

A smaller optimization has gone unmentioned until now. As one can notice in Figure 5.9, the multiplication sequence (denoted by red arrows) is reversed for every other column. This allows reusing one multiplication operand when switching to the next column, thus saving some memory cycles and power. When, for example, switching from column 1 to column 2, $a[0]$ is already stored in the operand register and does not have to be fetched again.

### 5.3.2 Avoiding a Third Reduction Round

The final result of the presented reduction mechanism is not guaranteed to be smaller than $2^{160}$, i.e., it might be 161 bits long. However, an occurrence of a 161-bit result is, with a chance of $2^{-93}$, very unlikely and also detectable. Hence, instead of performing yet another round of reduction to retrieve a 160-bit result, it is simply assumed that no 161-

bit results occur during signature generation. If, despite the diminishing odds, a 161-bit result is detected, an error flag is set in hardware and an error is returned at the end of the signing process. Computations are finished instead of stopped, an immediate abortion would give away the exact time of error when doing a timing attack. Note that in no case an erroneous result is returned.

The probability of $2^{-93}$ is valid for reduction after multiplication and is calculated as follows. The 195-bit intermediate reduction result obtained after the first reduction round can be split in two parts, a 160-bit lower part $\ell$ and a 35-bit upper part $h$. Computing $h + (h << 31)$ yields a 67-bit number. When adding this to $\ell$ the result can only be 161 bits long if the higher-order $160 - 67 = 93$ bit of $\ell$ are 1, only then a generated carry would propagate all the way to the front. The chance of this happening, assuming $\ell$ is a random bit string, is $2^{-93}$.

### 5.3.3 Implementation Results

All in all, modular multiplication in $\mathbb{F}_p$ takes 123 cycles. Due to fast-squaring mechanic squares can be computed in 83 cycles, i.e., exactly $10 \times 4 = 40$ cycles less (cf. Section 5.2.4). The runtime ratio of S/M is roughly 0.67, or 2/3. This ratio is important for selecting the fastest curve addition fomulæ.

## 5.4 Modular Arithmetic in $\mathbb{F}_n$

The Elliptic Curve Scalar Multiplication greatly benefits from the use of a pseudo-Mersenne prime, which makes reduction a lot faster. All other parts of the Elliptic Curve DSA however are performed in $\mathbb{F}_n$, with $n$ being the prime order of the curve. This modulus $n$ is not of any special form, thus a different reduction algorithm has to be implemented. The Montgomery multiplication algorithm presented in Section 2.2 and shown in Algorithm 2.2 is used for this purpose. A Montgomery multiplication of 2 operands $a, b$ is the efficient computation of $abR^{-1} \bmod N$ and is denoted by $\text{MonPro}(a, b)$. This section discusses its implementation aspects.

### 5.4.1 Avoiding 161-bit Integers

One of $n$'s properties makes implementing modular multiplication using the Montgomery method difficult: its bit length of 161. The implementation is geared towards handling of 160-bit integers, everything above that adds an area and time overhead. Also, 161 is not a multiple of the word size 32, which makes implementation more complex. Now a method is presented that allows working with 160-bit numbers.

An important choice is that of parameter $R$. According to Montgomery $R$ has to be chosen so that $R > N$, where $N$ is the used modulus. Typically a power of 2, ideally a multiple of the processors word size, is used. Then division by $R$ is a simple shift and modular reduction becomes a truncation.

Before presenting the options of $R$, please recall that the intermediate $m < R$ as $m = (T \bmod R)n' \bmod R$. Also note that 32-bit multiplications with a single-bit operand, i.e., one input word is either 0 or 1, have to be avoided. An attacker might be able to determine the value of this bit with SPA, a multiplication with 0 does not change any values in the accumulator and thus has a different power trace than a multiplication with 1. A possible workaround is widening one of the two multiplier inputs to 17 bit, then it is possible to perform a $32 \times 33$ multiplication. The previously single bit is incorporated as MSB of a 33-bit word.

There exist several possible choices of $R$, each has its own set of advantages and disadvantages.

**$R = 2^{161}$**
: While seemingly the smallest suitable option, 161 is not a multiple of 32. This makes shifting rather expensive, instead of using simple addressing dedicated shifters need to be included. The multiplier needs to be expanded to 17 bits in order to multiply the highest word of the 161-bit $m$ value. Additionally, the integrated product-scanning approach requires $R = 2^a$, where $a$ is a multiple of the word size, thus disqualifying this first option.

**$R = 2^{176}$ or $2^{192}$**
: Stepping up to multiples of 16 or 32, respectively, solves some of these problems. The integrated product scanning scheme is compatible with these two choices, making them a viable option. In case of 192-bits, the shifting can be done with simple addressing, thus eliminating the need for dedicated shifters. However, the need for a wider multiplier is only partially eliminated. While the computation of $m \times N$ can be carried out with the 16-bit multiplier, the multiplication of the two 161-bit operands

$T = a \times b$ still requires a $17 \times 17$ bit instance. Another downside is the additional storage space and time needed for computation of $m$ and its multiplication with $N$.

**$R = 2^{160}$**
Although violating the requirement of being greater than $N$, $R = 2^{160}$ is still a valid option under certain assumptions. In this case the bit length of the intermediate $m$ is also 160 bits, which has several advantages. Both computation of $m$ and its multiplication with $N$ is faster when compared to the previous case, also storage requirements are slightly lower. Multiplication logic can be easily reused for multiplying $m \times N$ due to the same number of partial products. Finally, shifting by 160 is a simple matter of addressing.

Considering all the above $R = 2^{160}$ was chosen. However, the problem remains that it does not satisfy $R > N$. It is now shown that an $R$ greater than both multiplication operands $a, b$ suffices.

An appropriate size of $R$ ensures that $t < 2N$, i.e., at most one subtraction is necessary for retrieving the final result. Montgomery argues that $0 \leq T + mN < RN + RN$, so $t = (T + mN)/R < 2N$ [56]. This statement still holds for $R < N$ if both multiplication operands $a, b$ are smaller than $R$. It now needs to be assured that this is true, i.e., both operands $a, b < 2^{160}$.

Due to the structure of $n$—the MSB is followed by many 0 bits—the probability of a random element in $\mathbb{F}_n$ being greater than $2^{160}$ is very low. In fact, this probability is only $2^{-79}$. It is assumed that all operands of Montgomery multiplications during the course of a signing operation are smaller than $R$. Or in other words, all operands and results are restricted to 160 bits.

Other than allowing $R = 2^{160}$, this restriction offers additional advantages. No added storage is needed for the MSB of potentially 161-bit long integers, all values fit into the 160-bit field registers $\mathbf{R}_{\boldsymbol{x}}$. Also, no 17-bit multiplier is required, the 16-bit instance suffices.

If a multiplication results in a 161-bit product, the outcome can not be used as input to another multiplication. Such an occurrence is detected and an error flag is set, after finishing the signing operation an error is returned instead of the result. Throughout the signing process only four Montgomery multiplication operands are not output of another Montgomery multiplication: the input message hash $e$, the ECSM result's $x$ coordinate $r$, the scalar $k$ and the private key $d$. All these are 160 bits long, due to either choice ($e$, $k$, and $d$) or being an element in $\mathbb{F}_p$ ($r$).

There are a total of 200 Montgomery multiplications computed for a single signing operation, this results in an overall probability of $2^{-73}$ of abortion due to a 161-bit result. This probability is deemed to be acceptable.

### 5.4.2 Integrated Product Scanning

Two ways of integrating the reduction steps into the multiplication were presented in Section 2.3.4. Standard multiplication uses the product-scanning approach, so the *Finely Integrated Product Scanning* (FIPS) method (Algorithm 2.3) is the obvious choice. This work uses *Coarsely Integrated Product Scanning* (CIPS), a derivation of the FIPS algorithm not covered by Koç et al. [46].

Recall that the reduction can proceed word-wise, i.e., $m[i] = T[i]n'_0 \bmod 2^w$, with $n'_0 = n' \bmod 2^w$. The word size used for reduction ($w$) does not need to be equal to the processor or RAM word size, which in this case is 32 bits. Here the parameter $w$ was

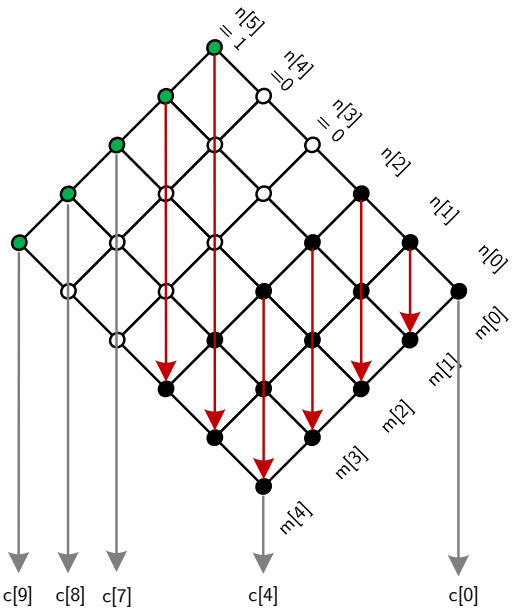



Figure 5.10: Multiplication with modulus $n$

chosen to be 16 bits, i.e., the width of the multiplier. This allows to compute each 16-bit part of $m$ in a single cycle. In an attempt to reduce both area and time requirements $w$ was set to 1. The outcome however was rather negative, this is discussed in Appendix C in greater detail.

The *Coarsely Integrated Product Scanning* (CIPS) algorithm is very similar to the FIPS approach, it however alternates between multiplication and reduction after each column by simply splitting the inner loop. This is needed due to the use of the reversed multiplication order in odd columns (see Figure 5.9). Pseudocode for the CIPS approach using $w = 16$ and a 32-bit RAM interface is shown in Algorithm 5.1. $\text{acc}_{15..0}$ denotes the 16 low-order bits of the accumulator, $\text{acc}_{31..16}$ the next-highest 16 bits.

Due to the differing word sizes—16 bit reduction and 32 bit memory—slight modifications are applied to the computation of $m$. A 32-bit memory word m[i] contains two 16-bit parts $\text{m[i]}_0$ and $\text{m[i]}_1$. In Line 9 of Algorithm 5.1, the lower part $\text{m[i]}_0$ is computed by multiplying the 16 lowest-order bit of the accumulator with the 16-bit constant $n'_0$. The result is stored in RAM, in the next cycle the saved word appears at the RAM output and is immediately written back to operand register A. During the next 2 cycles, $\text{m[i]}_0$ is multiplied with the lowest word of $n$, which is stored in operand register B. This process is repeated for the upper part $\text{m[i]}_1$. In RAM the intermediate m[i] is stored in the first register of the current result register set (Section 5.4.3), i.e., either in $\mathbf{R_4}$ or $\mathbf{R_6}$.

The multiplication $m \times N$ can be considerably sped up when taking the structure of the modulus into account. As depicted in Figure 5.10, two of $n$'s 32-bit words are equal to zero, i.e., they can be skipped in a multiplication (white dots). The highest word equals 1, which allows replacing a multiplication with a simple addition of the appropriate word of $m$ (green dots). This optimization reduces the cycle count for performing $m \times N$ to 65.

In contrast to modular multiplication in $\mathbb{F}_p$, there is no RAM register dedicated to storing intermediate values. Hence, *in-place* multiplication is not possible.

**Algorithm 5.1:** Coarsely Integrated Product Scanning (CIPS) with $w = 16$ and 32-bit RAM

```
1  s = 160/32 = 5;
2  for i=0 to s-1 do
3      for j=0 to i do
4          acc = acc + a[j]b[i-j];
5      end
6      for j=0 to i do
7          acc = acc + m[j]n[i-j];
8      end
9      m[i]_0 = acc_{15..0} n'_0 mod 2^16;
10     acc = acc + (m[i]_0 n[0]_0);
11     acc = acc + (m[i]_0 n[0]_1 << 16);
12     m[i]_1 = acc_{31..16} n'_0 mod 2^16;
13     acc = acc + (m[i]_1 n[0]_0 << 16);
14     acc = acc + (m[i]_1 n[0]_1 << 32);
15     acc >>= 32;
16     for i=s to 2s-1 do
17         for j=i-s+1 to s-1 do
18             acc = acc + a[j]b[i-j];
19         end
20         for j=i-s+1 to s-1 do
21             acc = acc + m[j]n[i-j];
22         end
23         acc >>= 32;
24     end
25 end
```

### 5.4.3 The Final Subtraction

Up until now, all explanations concern the calculation of $t$, which is of range $t < 2n$. The final result of the Montgomery multiplication is retrieved by subtracting the modulus if $t \geq n$, or by simply returning $t$ otherwise.

This final conditional subtraction can be avoided, as first shown by Walter [75, 76]. Instead of reducing the output to the desired range, the Montgomery multiplication is modified to accept inputs in the range of $a, b < 2N$, i.e., the product $t$ can be used as input for the next multiplication. The basic multiplication algorithm is unchanged, the lower bound for $R$ however is raised above $N$. A higher $R$ has, as already discussed before, several disadvantages, which is why a more traditional approach using subtractions was chosen.

A conditional subtraction must be avoided as it poses as a possible side channel. A very simple solution is to always execute the subtraction, it is needed anyway for performing the comparison with the modulus $N$. Both results $t$ and $t - N$ are stored in the RAM in neighboring registers, i.e, $t$ in $\mathbf{R}_{\boldsymbol{x}}$ and $t - N$ in $\mathbf{R}_{\boldsymbol{x+1}}$, respectively. Two such neighboring register pairs, dubbed *result register set*, are reserved in RAM: set 1 spans $\mathbf{R_4}$ and $\mathbf{R_5}$, set 2 $\mathbf{R_6}$ and $\mathbf{R_7}$. In-place multiplications are not possible, thus two register sets are needed for the case that a multiplication needs the outcome of the previous one. The

correct result is determined by the sign of the subtraction result, if negative ($t - N < 0$), the correct result is stored in $\mathbf{R_x}$, otherwise in $\mathbf{R_{x+1}}$. The sign is stored in a dedicated register (one per result set), which ensures that the next operation accesses the correct register. Note that this subtraction scheme might be exploited by so-called *safe-error* attacks (Section 5.9).

### 5.4.4 Switching domains

Prior to using a regular integer as an operand in the Montgomery multiplication, it has to be transformed into the Montgomery domain, which is denoted by $\bar{a} = aR \bmod N$. Also, the signature part $s$ needs to be transformed back. Both transformations are accomplished using Montgomery multiplications with special operands.

Three values need to be transformed into the Montgomery domain: $k$, $h$, and $r$. This is achieved by performing a Montgomery multiplication with $R^2 \bmod N$, i.e., $\mathrm{MonPro}(a, R^2) = aR^2R^{-1} = \bar{a}$. The 160-bit constant $R^2 \bmod n$ was precomputed and is stored in ROM register 16. The private key $d$ must be transformed during the (offline) key generation, the Montgomery representation is then written to ROM.

Transformation in the other direction is needed only once, namely for retrieving the final value of the signature part $s$. One can do this by multiplying with 1, i.e., $MonPro(\bar{a}, 1) = \bar{a}R^{-1} = aRR^{-1} = a$. Although one could optimize this multiplication by simply skipping the actual multiplication with 1, no such measure was implemented, as time savings would be minimal and the controller complexity would rise.

### 5.4.5 Additions to the datapath

The implemented Montgomery multiplication scheme requires some additions to the basic datapath, they are marked red in Figure 5.11. For the computation of a part of $m$, the current contents of the accumulator, more specifically either $\mathrm{acc}_{15..0}$ or $\mathrm{acc}_{31..16}$, need to be multiplied with the 16-bit constant $n'_0$, which is simply hardcoded as a possible input of the integer multiplier. The multiplication result is written to RAM. In the basic design, only the adder output is fed to RAM, now the multiplier output needs to be written. Instead of adding multiplexers, the 32 low-order bits of the first adder input (the accumulator) can be set to zero using AND gates, then the same output signal can be used for both cases. After having computed a part of $m$, it needs to be multiplied with the lowest-order word of the modulus $n$ (stored in operand register B). A simple trick is used here, after storing a word in RAM it appears at its output, so the computed part of $m$ can be copied to the operand register A without requiring multiplexers.

### 5.4.6 Implementation results

Using the described design, a single Montgomery multiplication can be carried out in 197 cycles, squarings amount to 157 cycles. This is a substantial increase when compared to multiplications in $\mathbb{F}_p$. This is explained by the fact that the modulus is not of any special form, and hence does not allow the use of the fast-reduction algorithm. The runtime ratio S/M is roughly 0.8.

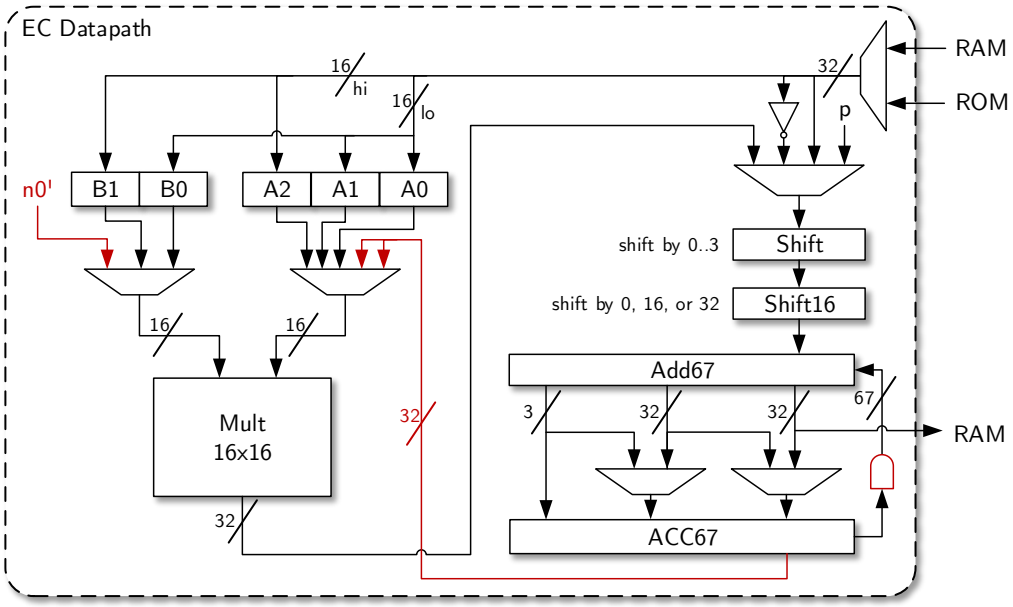



Figure 5.11: Additions to datapath required for Montgomery multiplication. Additions are marked red.

## 5.5 Tuning the Comb Method

The *Elliptic Curve Scalar Multiplication* (ECSM) is the most time consuming part of the Elliptic Curve DSA. For the signing operation, a random scalar $k$ is multiplied with a fixed-base point G, this is denoted by $Q = kG$. There exist several multiplication algorithms, some of which are mentioned in Section 2.4.4. Fixed-base comb methods offer a drastic speed up in exchange for higher memory requirements.

This thesis uses the comb method of Hedabou et al. [29], which offers several advantages over other methods. The used Zeroless Signed Digit (ZSD) recoding scheme [25] helps mitigating a possible side channel as no all-zero columns can occur. Also, the number of required precomputed points is slashed in half when compared to other methods, only $2^{w-1}$ points need to be stored.

### 5.5.1 Choosing the Comb Width

The choice of the comb width $w$ depends on the imposed requirements and constraints. Small widths already offer a relatively high speed up for small cost, e.g., for $w = 2$ computation time is halved and only two points need to be stored. Due to the exponential growth of precomputed points, a high $w$ is not very advisable. Also, the performance gain is minimal for each further bit added if $w$ is already high.

Table 5.1 compares both memory and time requirements for widths of up to 6. Given are the number of precomputed points, the size of the ROM, the number of necessary EC doubling-additions (DAs), and the computation time relative to the case $w = 1$. A width of 1 is equal to a standard, i.e., non-comb, left-to-right algorithm. In this case only the base point $G$ needs to be stored. Points are stored in affine, i.e., non-projective, coordinates, therefore $2 \times 160 = 320$ bits of ROM are needed per point. The number of doubling-additions is equal to the number of comb columns ($\ell$ in Figure 2.9) minus 1.

Table 5.1: Comparison of different comb widths

| $w$ | Points | ROM bit | DAs | Time rel. |
|---|---|---|---|---|
| 1 | 1 | 320 | 159 | 1.00 |
| 2 | 2 | 640 | 79 | 0.50 |
| 3 | 4 | 1 280 | 53 | 0.34 |
| 4 | 8 | 2 560 | 39 | 0.25 |
| 5 | 16 | 5 120 | 31 | 0.20 |
| 6 | 32 | 10 240 | 26 | 0.16 |

A width of 3 or 4 seems the most suitable, 2 does not offer enough performance gain, and 5 or 6 are too costly and offer little return. For this thesis $w = 4$ is chosen, it offers a good speed-up with acceptable cost.

## 5.6 Implementing the Doubling-Addition

Comb methods scan the scalar matrix from left to right, for each column, a point operation of form $2P + Q$ is performed. This section describes the efficient implementation of this operation dubbed *doubling-addition*. First, the used addition formulæ are discussed, they are then mapped to executable instruction code. Finally, a controller design that allows executing the stored program is shown.

There exists a broad range of efficient point addition and doubling formulæ (Section 5.6). Recall that the runtime of these algorithms is typically measured in required field multiplications (M) and squarings (S). Two doubling-addition schemes featuring a runtime of 11M + 7S were presented, namely Longa and Miri's scheme and Goundar's ZDAU operation. The former can be implemented using one less field register, also the overall instruction count of this method is lower than that of Goundar's algorithm. As all instructions are concluded by a reduction operation, the total cycle count of this method is slightly lower. Due to these advantages, Longa and Miri's formulæ are adapted for this work.

The chosen addition formulæ utilize standard Jacobian projective coordinates (Section 2.4.2), which represent the two-dimensional points by three coordinates. This allows to reduce the number of needed field inversions in $\mathbb{F}_p$ to one. The added point $Q$ is assumed to be stored in affine coordinates, so a mixed Jacobian-affine addition has to be performed.

### 5.6.1 Doubling-Addition Controller Design

The datapath was already presented in Section 5.2, it allows modular multiplication, squaring, addition, subtraction, and shifting. The doubling-addition controller, which implements the elliptic curve formulæ and utilizes the datapath's capabilities to perform its duties, follows now. The control logic is housed in a separate module (DA-Control in Figure 4.1) and is activated by the top-level controller whenever needed.

The elliptic curve doubling-addition is both a relatively lengthy and irregular process. Controllers based on finite-state machines (FSM) are, while simple and efficient in general, not suitable for these types of algorithms. Instead, the used controller architecture is based on a microcontroller-like design, i.e., instructions and accompanying operands are stored in a program memory and are fetched and executed one after another. An instruction counter keeps track of the current operation.

The main task in designing the controller is mapping the EC addition formulæ to instruction code. A two-operand code that allows instructions of form $r = a * b$, where $*$ stands for any of the supported operations, results in the most compact controller design. All code can be stored in a single table, one counter is sufficient for addressing the instruction. The major downside is the longer runtime. Two-operand code has a high number of instructions, as each operation is concluded by a reduction lowering the overall instruction count is desirable. The number of field registers required for a doubling-addition might also be higher.

The presented controller design thus allows the use of composite operations, e.g., $r = a \times b + 2c - 4d$. This reduces the operation count when compared to the two-operand approach and also saves memory cycles, as less intermediate values need to be stored and retrieved from the RAM. The multiply-accumulate unit embedded in the datapath can be

used to efficiently compute such composite instructions. Before explaining the controller design, the instruction code is discussed.

### Instruction code

The used mapping of Longa and Miri's addition formulæ is shown in Algorithm 5.2. $\mathbf{R_i}$, with $1 \leq i \leq 7$, denotes the RAM field registers. During doubling-additions $\mathbf{R_0}$ is reserved for the scalar $k$, the 224 bit-$\mathbf{R_8}$ is used for the unreduced operation results. As stated in Section 5.3.1, *in-place* multiplication is possible due to the use of $\mathbf{R_8}$ as dedicated temporary register. The mapping reflects this fact, e.g., in Lines 10 and 17.

The algorithm assumes that the Jacobian coordinates of the point $P = (X_1, Y_1, Z_1)$ are stored in $\mathbf{R_1}$ ($X_1$), $\mathbf{R_2}$ ($Y_1$), and $\mathbf{R_3}$ ($Z_1$), respectively. At the end of the doubling-addition the result $2P + Q = (X_4, Y_4, Z_4)$ is stored in the same registers, which allows to start the next iteration without further ado.

Prior to starting the actual computation, the four appropriate column digits $[s\ \alpha_2\ \alpha_1\ \alpha_0]$, with $s, \alpha_i \in \{-1, 1\}$, of the scalar matrix are fetched and stored in a dedicated register. Recall that the actual bit values $a_i \in \{0, 1\}$ are interpreted as $\alpha_i = (-1)^{1+a_i}$, i.e., zero bits are reinterpreted as $(-1)$. The (affine) coordinates $\mathrm{X}_{ROM}$ and $\mathrm{Y}_{ROM}$ of the corresponding comb point $[1\ \alpha_2\ \alpha_1\ \alpha_0]G$ are kept in the ROM field registers $(2(a_2\ a_1\ a_0))$ and $(2(a_2\ a_1\ a_0) + 1)$, respectively.

If the column's highest-order bit, i.e., the sign $s$, is negative, the precomputed point's $y$ coordinate has to be negated. To avoid conditional behavior a simple trick is used. First $\mathrm{Y}_{ROM}$ is copied to the RAM, then $2\mathrm{Y}_{ROM} \bmod p$ is computed and stored in a neighboring register. After this, depending on the sign of the column, either $y \equiv 2y - y \bmod p$ or $-y \equiv y - 2y \bmod p$ is computed (Line 3). Only a single bit of the operand addresses has to be conditionally flipped.

In order to avoid negative results after a subtraction, the a multiple of the modulus $p$ is added to the difference. These additions are not listed in Algorithm 5.2, but can be seen in Appendix B.

### Controller structure

For easy execution each *operation*—an *operation* is equal to a line in Algorithm 5.2—is split into multiple *steps*. Both operations and steps are stored in dedicated look-up tables, as can be seen in Figure 5.12. Steps are stored in the monolithic *StepLut* containing a total of 55 entries. The operation table *OpLut* contains 28 entries and stores the result register address *Res* and a pointer *OpOffset* pointing to the operations first step in the *StepLut*. The 3 bit counter *StepOffset* keeps track of the current step within an operation, it is added to the base address to retrieve the current *StepNr*.

Each entry of the *StepLut* contains a 3-bit opcode *Instruction*, two 3-bit arguments *OpA* and *OpB*, and two single-bit flags. Eight different instructions are supported: ADD, ADDP, SUB, MULT, SQUARE, SHIFT, COPY, and READK. READK is used as the first step of a doubling-addition and retrieves four column digits of the scalar $k$, all other instructions should be self-explanatory. The first argument OpA contains the index of the first operand field register. In the case of MULT, OpB points to the second operand register, for ADD, ADDP, SUB, and SHIFT it contains the number of shifts, which is directly fed to the first shifter in the datapath. The *LastStepFlag* determines if the step is the last within its operation. The *RomFlag* is set whenever OpA points to the ROM

**Algorithm 5.2:** Explicit register mapping of Longa's doubling-addition

**Input**: $P = (\mathbf{R_1}, \mathbf{R_2}, \mathbf{R_3}), Q = (\mathrm{X}_{ROM}, \mathrm{Y}_{ROM})$
**Output**: $2P + Q = (\mathbf{R_1}, \mathbf{R_2}, \mathbf{R_3})$

| | | |
|---|---|---|
| 1 | $\mathrm{R}_4 \leftarrow \mathrm{Y}_{ROM}$ | // $\mathrm{Y}_{ROM}$ |
| 2 | $\mathrm{R}_5 \leftarrow 2\mathrm{Y}_{ROM}$ | // $2\mathrm{Y}_{ROM}$ |
| 3 | $\mathrm{R}_4 \leftarrow \mathrm{R}_{4/5}$ - $\mathrm{R}_{5/4}$ | // $\pm\mathrm{Y}_{ROM}$ |
| 4 | $\mathrm{R}_5 \leftarrow {\mathrm{R}_3}^2$ | // $\mathrm{Z}^2$ |
| 5 | $\mathrm{R}_6 \leftarrow \mathrm{R}_3 \times \mathrm{R}_5$ | // $\mathrm{Z}^3$ |
| 6 | $\mathrm{R}_4 \leftarrow \mathrm{R}_4 \times \mathrm{R}_6$ - $\mathrm{R}_2$ | // $\alpha$ |
| 7 | $\mathrm{R}_6 \leftarrow \mathrm{X}_{ROM} \times \mathrm{R}_5$ - $\mathrm{R}_1$ | // $\beta$ |
| 8 | $\mathrm{R}_7 \leftarrow {\mathrm{R}_6}^2$ | // $\beta^2$ |
| 9 | $\mathrm{R}_3 \leftarrow \mathrm{R}_3 + \mathrm{R}_6$ | // Z + $\beta$ |
| 10 | $\mathrm{R}_6 \leftarrow \mathrm{R}_7 \times \mathrm{R}_6$ | // $\beta^3$ |
| 11 | $\mathrm{R}_3 \leftarrow {\mathrm{R}_3}^2$ - $\mathrm{R}_5$ - $\mathrm{R}_7$ | // Z' |
| 12 | $\mathrm{R}_5 \leftarrow \mathrm{R}_1 \times \mathrm{R}_7$ | // X'/4 |
| 13 | $\mathrm{R}_2 \leftarrow \mathrm{R}_2 \times \mathrm{R}_6$ | // Y'/4 |
| 14 | $\mathrm{R}_7 \leftarrow {\mathrm{R}_4}^2$ | // $\alpha^2$ |
| 15 | $\mathrm{R}_1 \leftarrow 4\mathrm{R}_7$ - $4\mathrm{R}_6$ - $8\mathrm{R}_5$ - $4\mathrm{R}_5$ | // $\theta$ |
| 16 | $\mathrm{R}_6 \leftarrow \mathrm{R}_4 + \mathrm{R}_1$ | // $\alpha + \theta$ |
| 17 | $\mathrm{R}_6 \leftarrow {\mathrm{R}_6}^2$ | // $(\alpha + \theta)^2$ |
| 18 | $\mathrm{R}_4 \leftarrow {\mathrm{R}_1}^2$ | // $\theta^2$ |
| 19 | $\mathrm{R}_2 \leftarrow 8\mathrm{R}_2$ | // Y' |
| 20 | $\mathrm{R}_6 \leftarrow \mathrm{R}_7 + \mathrm{R}_4$ - $\mathrm{R}_6$ - $2\mathrm{R}_2$ | // $\omega$ |
| 21 | $\mathrm{R}_5 \leftarrow 4\mathrm{R}_5$ | // X' |
| 22 | $\mathrm{R}_5 \leftarrow \mathrm{R}_5 \times \mathrm{R}_4$ | // X'$\theta^2$ |
| 23 | $\mathrm{R}_3 \leftarrow \mathrm{R}_3 \times \mathrm{R}_1$ | // $\mathrm{Z}_4$ |
| 24 | $\mathrm{R}_4 \leftarrow \mathrm{R}_1 \times \mathrm{R}_4$ | // $\theta^3$ |
| 25 | $\mathrm{R}_1 \leftarrow {\mathrm{R}_6}^2$ - $\mathrm{R}_4$ - $2\mathrm{R}_5$ | // $\mathrm{X}_4$ |
| 26 | $\mathrm{R}_4 \leftarrow \mathrm{R}_2 \times \mathrm{R}_4$ | // Y'$\theta^3$ |
| 27 | $\mathrm{R}_5 \leftarrow \mathrm{R}_5$ - $\mathrm{R}_1$ | // X'$\theta^2$ - $\mathrm{X}_4$ |
| 28 | $\mathrm{R}_2 \leftarrow \mathrm{R}_6 \times \mathrm{R}_5$ - $\mathrm{R}_4$ | // $\mathrm{Y}_4$ |

instead of the RAM. For the detailed code, i.e., the exact contents of both the operations and steps table, please refer to Appendix B.

Execution of a single operation is performed as follows. If the first instruction of an operation is either MULT or SQUARE, the columns 5 to 9 of the product are computed and stored in $\mathbf{R_8}$, as outlined Section 5.3.1, otherwise computation starts with column 0. The controller executes step after step until the LastStepFlag is set, then the 32-bit partial result is written to $\mathbf{R_8}$ and the stepOffset is reset to 0. This process is repeated for the columns 0 to 4. For multiplications and squarings, one column of the product-scanning scheme is processed at once. All other instructions as well as the addition required by the integrated reduction step also add to the accumulator. The computation is finally concluded by a reduction round, i.e., the upper part of $\mathbf{R_8}$ is added twice to the lower part and the result is written to the field register $\mathbf{R_{Res}}$.

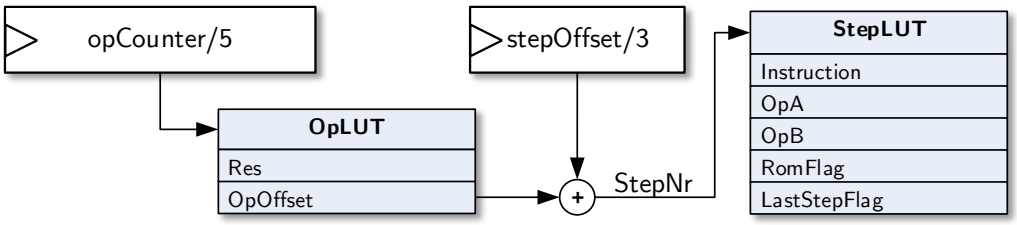



Figure 5.12: Doubling-addition controller

### Implementation results

With the described controller architecture, a doubling-addition can be performed in 2 251 cycles. The area requirements for the dedicated controller are 594 GEs, the two look-up tables take up 243 GEs.

## 5.7 Modular Multiplicative Inverse

During the course of an ECDSA signing operation, two modular multiplicative inversions have to be performed. The Jacobian representation of the point $kG$ must be converted to affine coordinates, this requires the inversion of the final points $Z$ coordinate in $\mathbb{F}_p$. Also, for computation of the signature value $s$, the inverse of the scalar $k^{-1} \bmod n$ is needed. Some widespread inversion algorithms are presented in Section 2.3.5. To achieve the primary goal of safety against all sorts of side-channel attacks a constant runtime and operation flow is vital. Inversion using Fermat's little theorem, i.e., computing $x^{-1} \equiv x^{n-2} \bmod n$, is the only presented algorithm that offers a constant runtime, which makes the algorithm choice obvious. Both inversion processes are steered by the top-level controller, which is discussed in Section 5.8.

When using Fermat's little theorem inversion is performed by means of a modular exponentiation $x^e \bmod n$. Modular exponentiation is very similar to the ECSM, in fact almost the same algorithms can be used. The additive notation of elliptic-curve operations is replaced with a multiplicative one, i.e., point doubling becomes modular squaring and point addition becomes a modular multiplication. Commonly used algorithms include binary left-to-right and right-to-left algorithms, the Montgomery powering ladder or sliding-window methods.

This thesis utilizes a left-to-right square-and-multiply algorithm for modular exponentiation. Both exponentiation operations have a fixed and publicly-known exponent, namely $n-2$ and $p-3$, respectively. No countermeasures for protecting the exponent against side-channel leakage have to be implemented, this allows the use of faster exponentiation algorithms. Recall that the binary left-to-right algorithm requires $|e|-1$ field squarings and $\mathrm{HW}(e)-1$ field multiplications, where $|e|$ denotes the bit length of the exponent $e$ and $\mathrm{HW}(e)$ its Hamming weight. A fixed exponent allows to optimize the exponentiation algorithm, the number of required field multiplications can be drastically lowered.

Sliding-windows algorithms require a costly precomputation phase, they then scan the exponent to determine the multiplication sequence. For a fixed exponent, it is possible to perform an offline search for an optimal multiplication sequence for the square-and-multiply algorithm. Precomputation of all possibilities of a fixed-size window, as required by sliding-window algorithms, is not necessary. Instead, the optimal set of precomputed powers can be determined individually for each exponent, this allows a maximum speed-up with minimal storage requirements. The number of needed multiplications can be drastically reduced, however, the number of squarings is fixed to $|e|-1$. This high number of squarings makes the fast-squaring mechanism discussed in Section 5.2.4 an absolute necessity. It is now shown how computation time is minimized for each exponentiation individually.

### $Z^{-1} \mod p$

The ECSM is carried out in Jacobian coordinates, the result's affine $x$ coordinate (the signature part $r$) is retrieved by computing $r = XZ^{-2} \bmod p$. Note that the square of the inverse is needed. A direct computation using Fermat's little theorem, i.e., $Z^{-2} \equiv Z^{p-3} \bmod p$, is faster than first computing the inverse and then squaring it.

The 160-bit exponent $e = p - 3$ (Equation 5.1) contains only three zero digits, thus its Hamming weight is 157. This constellation makes a traditional binary square-and-multiply approach extremely costly. However, there exists a much more efficient way.

$$e = p - 3 = \texttt{0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 7FFFFFFC} \tag{5.1}$$

The 128 high-order bits of the exponent are all equal to 1, thus it is necessary to compute $Z^{2^{128}-1}$. This is achieved by iteratively calculating $Z^{2^j-1}$ for $j = 2^i$ and $0 < i \leq 7$, starting with $Z^1$. The index $i$ is raised by using Equation 5.2, each incrementation takes $2^i$ squarings and only 1 multiplication. The computation of $Z^{2^{128}-1}$ thus takes 127S but only 7M. The values for $i = 2, 3, 4$ are later reused and stored in RAM registers.

$$(Z^{2^j-1})^{2^j} Z^{2^j-1} = Z^{2^{(j+j)}-2^j} Z^{2^j-1} = Z^{2^{(2j)}-1} = Z^{2^{2^{i+1}}-1} \tag{5.2}$$

There exists a simpler and less mathematical view on this problem. A squaring operation is equivalent to a binary left-shift of the exponent, multiplying two powers adds the two exponents. The fastest way to build the bit string $(1)^{128}$ using only shifts and additions is to iteratively compute $((1)^j << j) + (1)^j = (1)^{2j}$, again with $j = 2^i$ and $0 < i \leq 7$.

The single 0 bit at position 31 of the exponent is followed by another block of 29 1s. As $29 = 2^4 + 2^3 + 2^2 + 1$, this block can be computed using four multiplications with the stored powers of $Z$.

Summing up, an inversion in $\mathbb{F}_p$ takes 159S and only 11M. The total runtime is 14 550 cycles. For the detailed instruction sequence refer to the Appendix B

### $k^{-1} \mod n$

Finding an optimal multiplication sequence for inversions in $\mathbb{F}_n$ is harder. Apart from the long block of 0s, the 161 bit exponent $e$ (Equation 5.3) is not very regular, hence there is no obvious solution like for $\mathbb{F}_p$. The Hamming weight is 45, the now presented algorithm reduces the multiplication count to 26.

$$e = n - 2 = \texttt{0x1 00000000 00000000 0001F4C8 F927AED3 CA752255} \tag{5.3}$$

A modified sliding-window algorithm is used for inversion in $\mathbb{F}_n$. Instead of precomputing all possible values for a fixed-size window, only the powers $k^3, k^5$, and $k^9$ are precomputed and stored in RAM, this takes 3M and 3S. The actual square-and-multiply exponentiation is then started from $k^8$, which is a by-product of computing $k^9$. This simple trick saves 3S, the effective cost for precomputation is reduced to only 3M. The multiplication sequence, i.e., when to multiply with which precomputed power of $k$, was computed offline and is stored in a look-up table. The additional circuit area occupied by this LUT could be spared by determining the sequence online, e.g., by parsing the exponent. However, considering the differing window sizes, this is too complex and would possibly require an even larger area.

The inversion in $\mathbb{F}_n$ takes a total of 160S and 26M, or 30 252 cycles using the Montgomery multiplication scheme presented in Section 5.4.

The search for an optimal set of precomputed powers was done using a brute-force approach. Odd powers of up to $k^{31}$ were included in the search set $S$, i.e., the exponent's LSB is always set. The power set $P(S)$, i.e., the set of all possible subsets, was constructed, then each entry of $P(S)$ was analyzed in terms of precomputation cost and a possible multiplication sequence was determined. Note that, due to the lack of a fixed window size, there is no unique multiplication sequence. To simplify the search, only a single sequence was determined per set, similar to standard sliding-window algorithms the exponent was scanned from left to right and the powers were matched, higher powers were preferred if multiple options were possible. The fastest found option was $\{k^3, k^7, k^9\}$, which offers a very cheap precomputation and fits perfectly within the tight memory requirements.

Due to memory constraints, the maximum number of precomputed powers is 3. Multiplication in $\mathbb{F}_n$ is carried out using the Montgomery multiplication scheme presented in Section 5.4, thus field registers $\mathbf{R_{4..7}}$ are reserved for multiplication results (result register sets). The signature value $r$ is stored in $\mathbf{R_8}$, the scalar $k$ in $\mathbf{R_0}$. This leaves the 3 registers $\mathbf{R_{1..3}}$ at disposal.

## 5.8 The Top-Level Controller

The ECDSA controller is comprised of multiple sub-controllers, as seen in Figure 4.1. The top-level controller is in charge of overseeing the signature process and steering the sub-controllers, its structure is now discussed in greater detail.

The top-level controller is implemented following a hybrid approach, it utilizes both a finite-state machine and microcontroller-like programming. The absolute top-level of controlling is implemented using a FSM, as depicted in Figure 5.13. It features states dedicated to the comb method, the two inversions, and hash computations. All other operations needed for computing a signature are carried out in the state Prog. In this state, instructions are fetched from a program memory and then executed, which is very similar to the design of the doubling-addition controller.

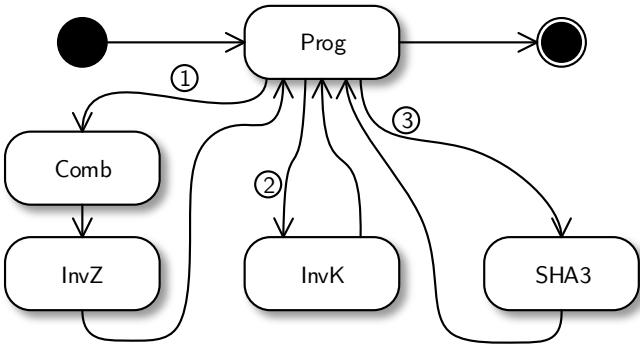



Figure 5.13: Top-level controller FSM

Apart from the FSM, the top-level controller houses several other parts, some of which are shown in Figure 5.14. The 4-bit *columnCounter* keeps track of the current word index, it is shared with the multiplication and doubling-addition controllers. The 8-bit *kCounter* is used for counting down the columns of the scalar matrix used in the comb method as well as for keeping track of the number of performed squarings for both field inversions. The 5-bit *opCounter* points to the current operation for both the doubling-addition and the top-level program. The *InvZLut* and *InvKLut* store the multiplication sequence for the field inversion, the *OpLut* stores the program executed during the Prog state.

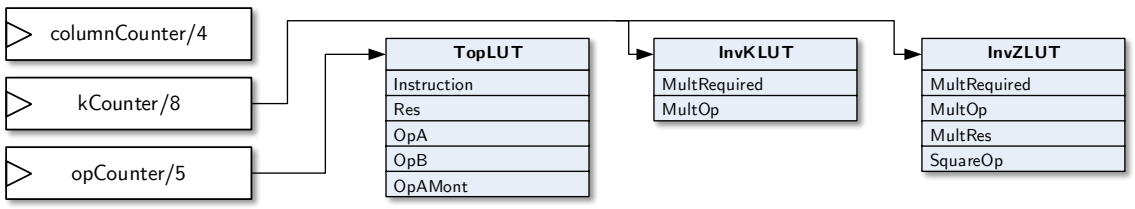



Figure 5.14: Top-level controller structure

The four dedicated states contain sub-FSMs for completing their task. The SHA state is the most simple one, the dedicated SHA controller is started and the end of the hashing process is awaited. For the Comb state, the doubling-addition controller is invoked for each column of the scalar matrix. The kCounter acts as column index, it is, starting from

39, being count down to 0. For the ECDSA, only the $x$ coordinate of the final point is required, so the computation of the $y$ coordinate (2 field squarings) is skipped in the last doubling-addition. The two inversion states need a more detailed explanation.

### 5.8.1 Field Inversions

Field inversions are performed by means of a modular exponentiation in either $\mathbb{F}_p$ or $\mathbb{F}_n$, the used algorithms are discussed in Section 5.7. They are implemented using two separate look-up tables and the 8-bit kCounter, which keeps track of the number of performed squarings. In the case of $Z^{-1} \bmod p$, the counter counts up starting from 0 until it reaches 159. For $k^{-1} \mod n$ the counter is set to 160 and then counts down to 0.

The two look-up tables InvZLUT and InvKLUT (Figure 5.14) are very similar to each other. The *MultRequired* flag determines if a multiplication is required for the current value of the kCounter, if set to 0 only a squaring is performed. The first operand of such a multiplication is always the outcome of the previous squaring operation, the second operands index is stored in *MultOp*.

The InvZLUT also contains the multiplications result register index and the operand for the squaring operations. Squarings always writes its outcome to $\mathbf{R_2}$, thus most of the *SquareOp* entries are 2.

All multiplications and squarings needed by the second inversion $k^{-1} \bmod n$ are carried out with the Montgomery multiplication scheme presented in Section 5.4. This scheme does not allow in-place multiplication and stores the result in two neighboring field registers, which are dubbed result register sets. Almost all performed multiplications and squarings use the previous result as an input. For this reason there exist two such result sets, they are used alternately. The result is written to one register set, while the operand is read from the other.

### 5.8.2 The TopLUT

While in state Prog the controller executes the program stored in the TopLUT by fetching one instruction after another. This look-up table contains 32 entries and is similar to the table used by the DA-Controller. No composite operations are needed, a two-operand code is sufficient. This allows the use of a single monolithic table.

Each table entry is comprised of a 3-bit instruction code, pointers to the result and operand registers, and a flag. Five instructions are supported: COPY, MULT, SQUARE, READK, and ADDN. The first three are self-explanatory, the two remaining need explanation. READK is equal to the doubling-addition instruction of the same name, the four digits of the highest-order column out of the scalar matrix are fetched and stored in a dedicated register.

ADDN performs a modular addition in $\mathbb{F}_n$, which is only needed for computing $s = k^{-1}h + k^{-1}rd \mod n$. The fast reduction mechanism used for calculations in $\mathbb{F}_p$ is not applicable in $\mathbb{F}_n$, so a different method has to be found. The sum is smaller than $2n$, so the same mechanism found in the final subtraction step of the Montgomery multiplication can be used. During computation of the sum also its difference to the modulus is determined. The sum $t$ is stored in $\mathbf{R_4}$, the difference $t - n$ in $\mathbf{R_5}$. Depending on the sign of the subtraction $t - n$ the correct result is chosen and used for the next operation. As also the case for the Montgomery multiplication, this opens up the possiblity of so-called safe-error attacks.

The *Res* field contains the index of the target register, in case of Montgomery multiplications it indicates which result register set to use, i.e., 0 for $\mathbf{R_{4/5}}$ and 1 for $\mathbf{R_{6/7}}$. *OpA* points to the first operand, *OpB* points to the second operand register and is only evaluated whenever a MULT instruction is executed. The *OpAMont* flag determines whether OpA points to a field register or to a Montgomery result register set. If the flag is set the correct result out of the result register set 0 or 1, depending on the LSB of OpA, is used.

For the exact content of the look-up table please refer to Appendix B.

### 5.8.3 The ECDSA Program

The execution mechanism was discussed in detail, however, no word was lost on what the program actually does. This shortcoming is now taken care of.

Algorithm 5.3 outlines the used program. Commented lines represent the top-level dedicated states, all other operations are performed during execution of the Prog state. In the first step, two 160-bit random numbers are retrieved from a random number generator and stored in the RAM. The random scalar $k$ is written to $\mathbf{R_0}$, the second integer $Z_r$ is used for Projective Coordinate Randomization and written to $\mathbf{R_3}$. Note that the implementation does not feature such an RNG, the random values are simply delivered by the simulation environment.

Due to some algorithm choices, the scalar $k$ must have certain properties, which need to be ensured while copying the random value to the RAM. The ZSD representation requires that the scalar is odd, hence the LSB of the lowest word is always set to 1. The doubling-addition formulæ (Section 5.6) compute $((P+Q)+P)$ instead of $(2P+Q)$, i.e., no doubling is performed. An elliptic curve point addition $P+Q$ requires that $P \neq \pm Q$. The first doubling-addition performed by the comb method is critical in this regard, the two highest-order columns of the scalar matrix must not point to the same precomputed point. This is ensured by performing a simple correction: the MSB of the scalar ($k_{159}$) is set to the value ($k_{39} \oplus k_{40}$) (Figure 5.15), where $\oplus$ denotes the binary XOR. Then the two columns are different in at least one row while being identical in another row. For the subsequent doubling-additions, the current point $P$ can never be equal to any stored point $Q$, no additional measures are necessary.

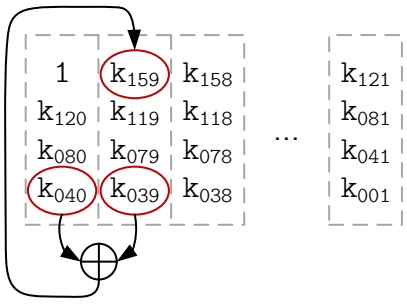



Figure 5.15: Correction of scalar $k$ to ensure $P \neq \pm Q$

After copying and correcting the random values, the powers $Z_r^2$ and $Z_r^3 \bmod p$ are computed. The four digits of the highest-order column out of the scalar matrix are fetched, the appropriate comb points affine coordinates $(x, y)$ are multiplied with the powers of $Z_r$, i.e., $X = xZ_r^2$ and $Y = yZ_r^3$. This technique is called *Randomized Projective Coordinates* (RPC) and helps thwarting differential Side-Channel Analysis (SCA) (Section 5.9).

During execution of the Comb state, 39 doubling-additions are performed, the following inversion is done as described in Sections 5.8.1. The signature value $r$ is computed by multiplying the final $X$ coordinate with the square of the inverse of Z, during this computation an inline check is performed. If the condition $0 < r < p$ is violated, an abortion flag is set, computation however continues. Then, the scalar $k$ is transformed to the Montgomery domain by computing $\bar{k} = \text{MonPro}(k, R^2 \bmod n)$. During the precomputation phase preceding the second inversion, the powers $k^3, k^7$, and $k^9$ are computed and stored in field registers 1 to 3.

Then, the second inversion $k^{-1} \mod n$ is carried out. The subsequent hashing operation computes an 800-bit state version of Keccak. The input-message hash $e$ as well as the signature part $r$ are then transformed to the Montgomery domain. Note that the non-transformed representation of $r$ is later used as a return value, thus it is also kept in RAM and not overwritten. Finally, the second signature part $s = k^{-1}h + k^{-1}dr \mod n$ is computed. During the addition, an inline check is performed to determine if $s \neq 0$. Note that the bracket in $s = k^{-1}(h + dr) \mod n$ is unfolded, two multiplications with $k^{-1}$ are performed. This prevents the multiplication $d \times r$, where a known value—$r$ is part of the signature—is multiplied with the private key $d$. Instead, the private key is multiplied with the random and not publicly-known $k^{-1}$, which makes (at least first-order) differential SCA targeting these operations harder (Section 5.9).

The exact program, i.e., the exact contents of the TopLUT, is listed in Appendix B.

**Algorithm 5.3:** ECDSA program

1 Retrieve random numbers from RNG, store in $\mathbf{R_0}$ (scalar $k$) and $\mathbf{R_3}$ (random $Z_r$)
2 Compute $Z_r^2$ and $Z_r^3 \mod p$
3 Retrieve the highest-order column of the scalar matrix
4 Perform projective coordinate randomization on the appropriate comb point
  `// Comb: Perform comb method ECSM`
  `// InvZ: Compute` $Z^{-2} \mod p$
5 Compute signature value $r = x = XZ^{-2} \mod p$
6 Verify that $r \neq 0$ and $r < p$
7 Transform $k$ to the Montgomery domain
8 Perform precomputations needed by the second inversion
  `// InvK: Compute` $k^{-1} \mod n$
  `// SHA: Hash the input message` $e = \texttt{SHA3}(m)$
9 Transform $e, r$ to the Montgomery domain
10 Compute $k^{-1}h$ and $(k^{-1}d)r \mod n$
11 Compute signature value $s = k^{-1}e + k^{-1}dr \mod n$
12 Convert $s$ back from Montgomery domain
13 Verify that $s \neq 0$
14 **return** $(r, s)$

## 5.9 Protection from Implementation Attacks

Implementation attacks (Section 3.2) target a specific implementation rather than the executed algorithm. Since the introduction of power analysis in 1999 by Kocher et al. [48], especially the field of side-channel analysis (SCA) spurred a lot of research. Myriads of attacks were proposed and countermeasures that help securing devices against those attacks were presented. The presented implementation should be secure against the state of the art of side-channel attacks. While absolute safety is not achievable and new attack techniques might show up any time, resistance against currently known attacks is possible. This section describes how security against implementation attacks (with focus on SCA) is achieved. Several countermeasures aimed at thwarting known attacks are discussed and remaining attack points are listed.

Please note that no effort was made to make the Keccak modules secure against SCA. Due to the highly regular structure of Keccak, SPA-based attacks are not very likely to succeed. However, DPA-based techniques might be successful. There exist several techniques aimed at making Keccak secure against first-order DPA, as shown by the Keccak team [11] and Bilgin et al. [12]. None of the proposed countermeasures were adopted in this work.

Following countermeasures were implemented.

- (Randomization of ECDSA)
- Constant runtime and operation flow
  - Constant runtime modular reduction
  - Constant runtime conditional negation of stored $y$ coordinate
  - Avoid negative results after subtractions by adding modulus beforehand
  - No all-zero columns in comb
  - Inversion using Fermat's little theorem
  - Always execute final subtraction in Montgomery multiplication
- Randomized Projective Coordinates [18]
- Reorder final computations to avoid multiplication of private key $d$ with known $r$

A constant runtime and operation flow mainly helps securing the implementation against SPA. The Randomized Projective Coordinate approach as well as the avoidance of the multiplication $d \times r$ is aimed at disabling DPA attacks. The countermeasures are now explained in greater detail.

### 5.9.1 Ensuring a Constant Runtime

An important part in securing the implementation against all sorts of SCA and especially SPA is ensuring an absolute constant runtime and operation flow. The control flow should have absolutely no dependency on the processed data. Reaching this goal requires caution in all steps of the implementation design, the necessary steps are now listed and further discussed.

**Modular reduction in $\mathbb{F}_p$.**
The fast reduction mechanism (Section 5.3) is used for all reduction processes in $\mathbb{F}_p$, i.e., not only for multiplications. Reduction after addition or similar operations can also be achieved by means of a conditional subtraction of the modulus. However, this is difficult to implement whenever the output range exceeds two times the modulus and it furthermore introduces a runtime dependency. A dummy operation, e.g., a subtraction of 0 performed if the result is in the exact range, is easily detectable.

**Negation of stored $y$ coordinate.**
The used comb method requires only half the memory when compared to other comb methods, but adds the need of a conditional negation of stored $y$ coordinates. A simple trick is used to perform this conditional operation in constant runtime without the need of dummy operations. As seen in the first few lines of Algorithm 5.2, the $y$ coordinate is first copied from ROM to RAM register $\mathbf{R_4}$, then $2y \bmod n$ is computed and stored in $\mathbf{R_5}$. Depending on the sign of the current comb column either $y = 2y - y \bmod n$ or $-y = y - 2y \bmod n$ is computed and stored in $\mathbf{R_4}$. This value always depends on both intermediates, thus thwarting safe-error attacks. The small differences in RAM address calculation should not be detectable by an SPA.

**Avoiding negative subtraction results.**
The doubling-addition formulæ require the execution of modular subtractions. The fast reduction mechanism does not apply to negative results, hence they have to be avoided. This is simply done by always adding a multiple of the modulus $p$, this ensures a positive result. This addition is done during the ACCP instruction and can be seen in the doubling-addition program code (Appendix B).

**No all-zero columns in comb method.**
In classical comb method all-zero columns might occur, i.e., all bits of a column out of the scalar matrix are 0. In this case no point addition is to be performed, which immediately reveals the column value and thus some scalar bits to an adversary. Use of the *zeroless* signed-digit recoding scheme obviously crosses this attack scenario.

**Inversion using Fermat's little theorem.**
There exist multiple algorithms that allow the computation of a modular multiplicative inverse. While inversion using Fermat's little theorem is not as fast as, e.g., the binary inversion or the Montgomery inversion algorithm, it does offer a constant runtime. Even the most basic modular exponentiation algorithms, e.g., the binary left-to-right Algorithm 2.4, feature a runtime and operation flow only dependent on the exponent, which is fixed in this case.

**Always execute subtraction in Montgomery multiplication.**
The intermediate result $t$ in the Montgomery multiplication scheme (Algorithm 2.2) is in range $t < 2N$, a final conditional subtraction of the modulus $N$ is required to retrieve the final result. The subtraction is always executed, both $t$ and $t - N$ are stored in RAM and

the correct value is then chosen. For a more thorough explanation of the final subtraction step refer to Section 5.4.3.

### 5.9.2 Countermeasures Against DPA

A constant operation flow is by no means a protection against Differential Power Analysis (DPA). Recall that DPA typically requires a large set of power traces. Also, an intermediate algorithm results of form $f(d, k)$ is needed, where $d$ is a known non-constant value, e.g., the algorithm in- or output, and $k$ is a part of the (fixed) secret key [53]. One possible way of protection is to avoid such computations, e.g., through randomization of the computed values.

The implemented anti-DPA countermeasures are now presented and discussed.

**In-built ECDSA randomization.**
The ECDSA signing operation requires that the cryptographic nonce $k$ is both random and secret. A violation of this rule can immediately reveal the secret key $d$ to an adversary. The randomization can thus not be counted as an additional countermeasure, but it makes DPA harder.

**Randomized Projective Coordinates.**
The use of Randomized Projective Coordinates (RPC) was first proposed in 1999 by Coron [18] and became an accepted and widely deployed countermeasure [37, 44, 62]. The projective coordinates of the ECSM base point, or in the case of comb methods the first used precomputed point, are randomized. When using Jacobian coordinates, the affine base point $(x, y)$ is transformed to the Jacobian point $(xZ_r^2, yZ_r^3, Z_r)$, where $Z_r$ denotes the randomly chosen $Z$ coordinate. The coordinates of all intermediate points computed during an ECSM are then also randomized, thus thwarting DPA attacks that try to determine the scalar $k$. In the ECDSA, the scalar is already randomized, but the cheap RPC adds another layer of security.

Adding RPC also protects against template-based SPA attacks. If not adding randomization, the computed values of the first doubling-addition depend on only six bits of the scalar. An adversary can record templates for all 64 different cases, assuming he has full access over a similar device where he can freely set the scalar $k$. For the actual attack, the adversary then records the power trace of the first doubling-addition and matches it against the precomputed templates. A single doubling-addition takes over 2k cycles, which might give a high chance for a successful attack. A randomization of the computed values immediately disables this attack scenario.

**Reordering final multiplications.**
The ECDSA algorithm requires the computation of $rd \mod n$ (Line 6 of Algorithm 2.10), i.e., the private key $d$ is multiplied with the publicly known signature part $r$. The multiplication is avoided by simply unfolding the bracket in $k^{-1}(e + rd)$. When performing $(k^{-1}d)r \mod n$ the private key $d$ is multiplied with the inverse of the random and secret scalar $k$. This multiplication is not susceptible to (at least first-order) DPA, the need for the additional modular multiplication $k^{-1}e \mod n$ is a relatively small price to pay.

### 5.9.3 Possible Attacks

Although a lot of effort was put into making the design secure against SCA, absolute safety is not possible and a few possible attack points remain.

One weakness might be the handling of the secret scalar $k$. Bit $k_{159}$ is not random, it is dependent on the value of 2 other bits. The ZSD recoding scheme requires that the scalar is odd, so the LSB $k_0$ is always set to 1. This is not ideal, as only a few leaked nonce bits over multiple signatures might already reveal the private key, as recently shown by Liu and Nguyen [51] and Mulder et al. [58]. A possible solution to this problem is to add a multiple of the (odd) group order $n$, i.e., $n$ if $k$ is even and $2n$ otherwise, to the scalar before starting the ECSM. Adding the group order to the scalar does not change the outcome of the ECSM. However, the conditional addition of either $n$ or $2n$ might be detectable by an SPA.

Another possible weakness is found in the final subtraction step of the Montgomery multiplication scheme. Both values $t$ and $t - N$ are computed and stored side-by-side in RAM. When inducing an error during the save operation of only one of these values, an adversary can determine the correct result by verifying the final output. When, for instance, an attacker is able to induce a glitch during writing of $t - N$ and observes that the final result is still correct, then $t$ must be the correct result. If the final signature is invalid then $t - N$ must have been correct. This is an example of a *safe-error* attack, which belongs to the group of fault attacks. The most obvious target of this attack is the multiplication with the private key $k^{-1} \times d$. For a greater key $d$ the probability that $t > N$ rises, i.e., it is more likely that $t - N$ is correct. An adversary can determine the probability that $t < N$ by performing the safe-error attack multiple times, it might then be possible to derive information on the up-most bits of the (Montgomery representation of the) private key.

The small chance of failure also poses as a side channel. On multiple occasions during signature generation computation errors might occur, these are detected and an error flag is set. For instance, both the fast reduction and the Montgomery reduction scheme have a non-zero chance failure. Also the checks if $0 < r < p$ and $s \neq 0$ might turn out negative. While the probability of errors is very low, an attacker might be able to provoke them and then use the gained information to deduce bits of the key or the nonce.

## 5.10 Summary

In this chapter, the details of the presented implementation were presented. The main datapath contains a 67-bit accumulator and a $16 \times 16$ bit multiplier that is used in a pipelined multiplication scheme. For modular reductions in $\mathbb{F}_p$, a fast reduction mechanism using only shifts and additions is used. The reduction process is integrated into multiplication, a slightly modified adder allows addition of the shifted intermediates. The prime $n$ is not of any special form, so the more general Montgomery multiplication scheme is used for reduction in $\mathbb{F}_n$. It is is implemented following the Coarsely Integrated Product Scanning (CIPS) approach.

The execution time of a point-scalar multiplication is drastically lowered through the use of a fixed-base comb method. The comb width was set to 4, so 8 curve points need to be precomputed and stored in ROM. For the point doubling-addition operation, formulæ by Longa and Miri are used, a dedicated controller supporting composite operations reduces the number of required modular reductions.

Prime-field inversions are computed by means of modular exponentiation, i.e., by utilizing Fermat's little theorem. The fixed moduli allow usage of highly optimized exponentiation algorithms, the number of required field multiplications can be drastically lowered.

The computation of $Z^{-2} \bmod p$ takes 159S + 11M, the inversion of the nonce $k$ in $\mathbb{F}_n$ 160S + 26M.

Countermeasures aimed at thwarting most common implementation attacks were implemented. First and foremost, signature generation has a constant and data independent runtime, which makes SPA-based attacks more difficult. Randomized Projective Coordinates counter attacks based on differential power analysis.

# Chapter 6

# SHA-3 Modules and Integration

The hash function is an integral part of the ECDSA algorithm. Instead of signing the whole message, the digest or hash of the input message is first computed, then this value is signed. Out of the myriads of existing hash functions, the Keccak algorithm, which won the SHA-3 competition and will be incorporated into the Secure Hash Standard (SHS), was chosen to be used in this work. Hashing is typically not nearly as time consuming as an elliptic curve scalar multiplication, nonetheless an efficient design is important. The signing process should not be slowed down significantly and the area footprint has to be small. This chapter presents a low-resource design for the Keccak algorithm and discusses how it is integrated into the existing modules.

The Keccak modules are based on the low-resource implementation of Pessl and Hutter [64], which is to the authors knowledge currently the smallest published full-state (1600 bits) ASIC implementation. In fact, a large portion of the VHDL code is reused, however, some adaptations are necessary. The RAM interface is widened up to 32 bits and the chosen rate and capacity differ. To better utilize the wider RAM interface, the used interleaving scheme is changed from factor-2 to factor-4. Finally, the Keccak parts need to be integrated into the other modules. The goal of the integration is the minimization of circuit area. Synergies between the modules need to be found, as many resources as possible need to be shared.

## 6.1 Basic Considerations

The basic design considerations are listed in [64], they are now summed up and discussed in the context of ECDSA. The width of the datapath is chosen to be (mostly) 16 bits, despite using a 32-bit RAM. A wider datapath would increase circuit area, while the computation time savings would be relatively low when compared to the time needed for, e.g., an ECSM. To keep both area and power requirements low, all operations are highly serialized.

The 800-bit state should be stored in a RAM. The memory requirement should not exceed these 800 bits, no additional storage for temporary results should be required. The RAM interface is 32-bit wide, byte-wise writing operations need to be supported.

The constants needed for the Keccak-$f$ transformation, i.e., the 32-bit round constants for the $\iota$ transformation and the 5-bit lane rotation offsets needed by $\rho$, are stored in a simple look-up table. The round constants can also be generated using a 7-bit LFSR,

for the sake of simplicity the LUT approach was chosen. Area-wise the two techniques are comparable.

There exist two basic approaches for designing a serialized KECCAK architecture. Lane-wise processing fetches and processes lane after lane, it is being used by most software implementations and the compact co-processor presented in [11]. This straight-forward approach has the downside that additional storage for intermediate results is needed, it is also relatively slow as each lane needs to be accessed multiple times throughout a single round. The hardware friendly slice-wise processing was first presented by Jungk and Apfelbeck [41]. In the case of slice-wise processing the $\pi$ operation becomes a simple rewiring, also the non-linear $\chi$ and the $\theta$ operation can be performed in single step. Jung and Apfelbeck's FPGA implementation processes 8 slices of the state in parallel, computation of a single round hence takes only 8 cycles. The round function is rescheduled. The initial round consists of $\pi \circ \rho \circ \theta$, which is followed by 23 rounds of $\pi \circ \rho \circ \theta \circ \iota \circ \chi$. Computation is concluded by the final round of $\iota \circ \chi$. This design requires the state to be stored in a 25 $8 \times 8$ distributed RAM, which is not an option here.

The now presented implementation utilizes both slice- and lane-wise processing. The rotation $\rho$ is performed on a lane-per-lane basis, all other transformations are executed slice-wise. The round function is rescheduled to allow computation of all 4 slice operations in a single cycle. The initial round consists of only $\theta$ and $\rho$, then 21 main rounds of $\rho \circ \theta \circ \iota \circ \chi \circ \pi$ are performed. The final round consists of $\iota \circ \chi \circ \pi$. This schedule differs slightly from the one used by Jungk and Apfelbeck. Special attention needs to be put upon to the memory layout, the combined processing approach requires that both slices and lanes can be accessed efficiently. This problem is solved by using a storage scheme that utilizes bit interleaving.

## 6.2 The KECCAK Architecture

The detailed architecture of the KECCAK modules is now presented. Special emphasis is put upon the two key ideas of the design: Combined slice- and lane-wise processing and the interleaved storage scheme. The structure of the KECCAK datapath is shown in Figure 6.1. It is comprised of four 32-bit registers $\mathbf{R_0}$ to $\mathbf{R_3}$ (not to be confused with the RAM field registers), four separate $\rho$ units and a slice unit. The interleave and deinterleave units are a basic rewiring. During lane processing each register stores a 32-bit lane, while slice processing the registers keep four slices with a total of 100 bits. The sponge unit, which is responsible for XORing the padded input message to the 640 low-order bits of the state, is not shown.

### 6.2.1 Interleaved storage

Combining both lane- and slice-wise processing when using a RAM for state storage is challenging, the state needs to be traversed in two different directions. The now presented scheme tries to solve this by merging multiple lanes into a single memory word.

The 800-bit state is not stored lane-after-lane in RAM, but interleaved. Four consecutive 32-bit lanes are bit-wise interleaved and form a virtual 128-bit word, which is then stored in RAM. An $n$ bit memory word then contains information on 4 lanes but only $n/4$ slices, which helps drastically speeding up the loading and storing operations during the slice-processing phase. An example with $n = 8$ is seen in Figure 6.2. The number of lanes, 25, is not a multiple of the interleaving factor 4, so a single lane is not stored

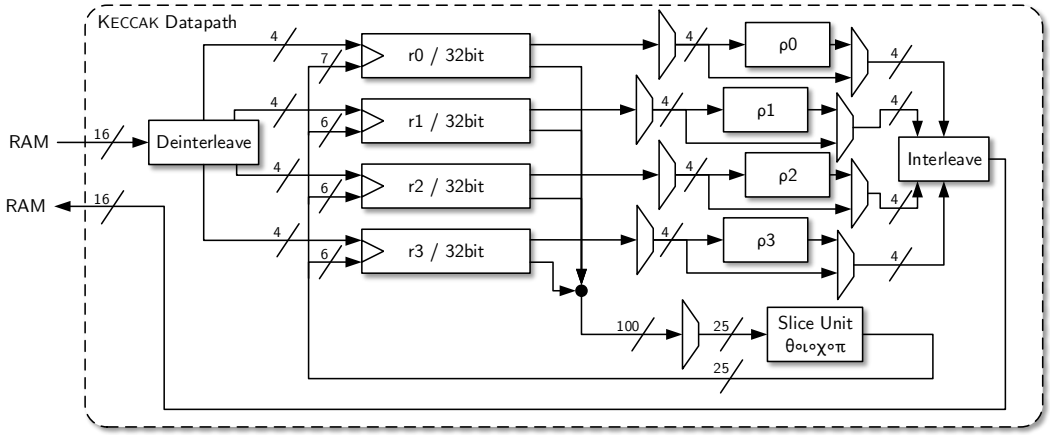



Figure 6.1: KECCAK datapath

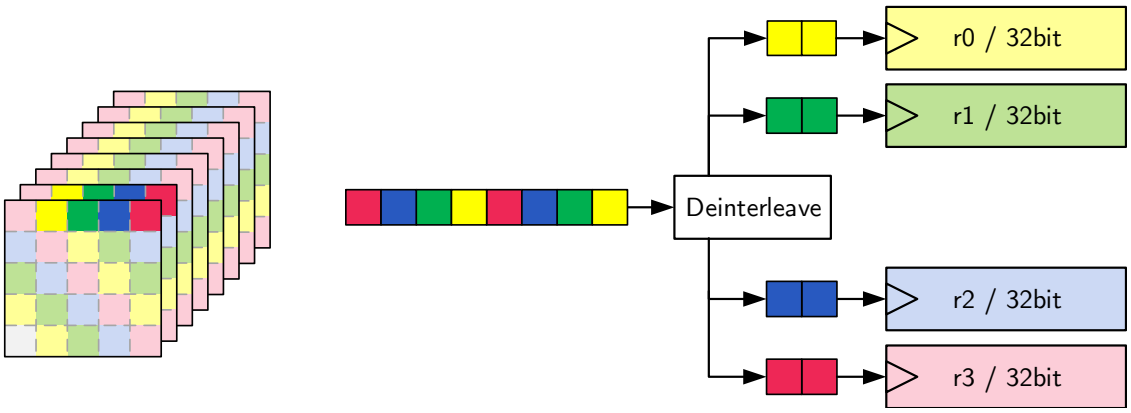



Figure 6.2: KECCAK core interleaving scheme

interleaved. The lane in the origin, i.e., at position $[x = 0, y = 0]$, is the only one with a rotation offset of 0, hence it can be skipped by the $\rho$ transformation. This makes this lane the ideal candidate for non-interleaved storage. For easy addressing it is kept at the end of the state address space, i.e., at RAM address 24.

### 6.2.2 Combined Processing

Each (rescheduled) round of Keccak-$f$ consists of a slice- and a lane processing phase.

During lane processing four consecutive lanes at a time are fetched into the registers (6.4, middle) and then rotated using four separate $\rho$ units. Each $\rho$ unit (Figure 6.3) consists of a 4-bit barrel shifter and a rotation register that serves as a basic delay element. The 4-bit rotation offset is split in-two, the high-order bits are handled by proper register addressing, while the two low-order bits are fed to the barrel shifter. The 32-bit RAM interface would also allow the use of 8 bits wide $\rho$ units, computation time would be decreased by roughly 1 kCycle. While this is a considerable speed-up in context of hashing, it is almost negligible when compared to the time needed for, e.g., an ECSM or field inversion. For this reason a width of 4 was chosen, which saves circuit area.

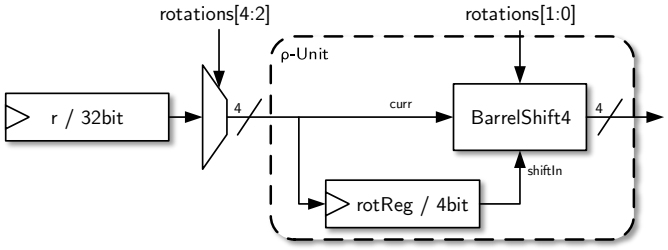



Figure 6.3: Keccak $\rho$ unit

For slice processing, four slices are fetched, then they are separately fed to the slice unit (in ascending order). The slice unit (Figure 6.4) first computes the $\pi$ operation, which is just a rewiring of the input. The non-linear $\chi$ operation is performed as shown in Figure 2.12a. A single bit of the round constant is added ($\iota$), afterwards the five column parities are computed and added to the shifted column parity of the previous slice. The result is finally added (XORed) to the slice. For the first and final round of computation some operations need to be skipped, which is why multiplexers were introduced.

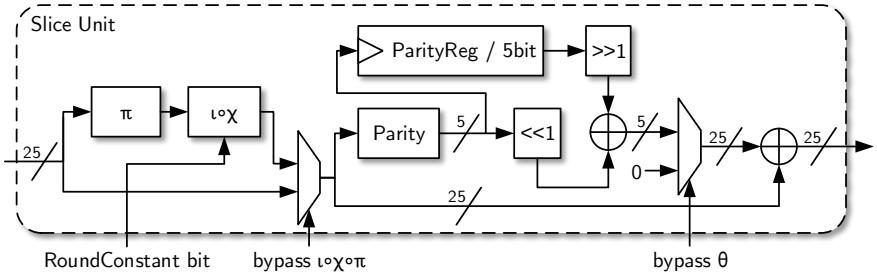


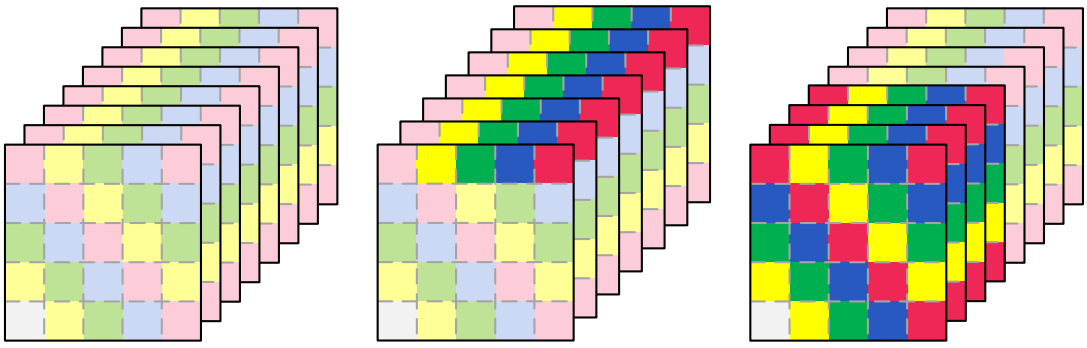

Figure 6.4: Keccak slice unit

## 6.3 Permutation Computation

Due to the use of an 800-bit state the KECCAK-$f$ permutation consists of 22 rounds of five transformations. As stated in Section 6.1, the round function is rescheduled around the lane-based $\rho$ function to allow computation of all four slice-based operations at once. To recap, the first round consists of only $\theta$ and $\rho$, the 21 main rounds of $\rho \circ \theta \circ \iota \circ \chi \circ \pi$ and the last round of $\iota \circ \chi \circ \pi$. Each modified round then consists of a slice-processing phase, during which $\pi, \chi, \iota$ and $\theta$ are computed, followed by a lane-processing phase of only $\rho$.

Slice processing starts of with a precomputation. The $\theta$ operation of the very first slice 0 requires the column parity of the last slice 31, thus this last slice is fetched, its parity is computed and stored in the parity register of the slice unit. Then, four slices at a time are fetched (see Figure 6.5c), they are fed to the slice unit and the result is stored back in the register. The final register content is interleaved and stored back to RAM. Loading or storing four slices is done in $\lceil 25/4 \rceil = 7$ cycles, actual slice computation takes 4. This is repeated 8 times to cover all 32 slices. Please note that all slice loading and storing operations are carried out on 16-bit words, i.e., effectively only half of the memory bus width is used. A 32-bit memory word contains information on $32/4 = 8$ slices, however, the internal registers can only store 4. To fully utilize the 32-bit bus, the internal memory thus needs to be increased to $8 \times 25 = 200$ bit, the computation time savings are however not worth the additional circuit area.

During lane processing, four lanes at a time are loaded into the internal registers (Figure 6.5b). The lane-loading operation is the only part of KECCAK that fully utilizes the 32-bit RAM interface, it takes only 4 cycles to fill the internal registers. All other loading or storing operations are performed on either 16 or 8-bit words, which is mainly done to save circuit area. Rotation starts by initializing the rotation registers, then the lanes are rotated by their respective rotation offset, using the separate $\rho$ units, and immediately stored back to RAM. The rotation and storing stage takes 8 cycles to complete. This process is repeated 6 times to handle all 24 lanes with a rotation offset other than 0.



(a) Whole state (b) Lane processing (c) Slice processing

Figure 6.5: KECCAK loaded parts of the state

Until now all explanations dealt with the KECCAK-$f$ permutation function, the handling of the sponge construction is now discussed. XORing the state with the padded input and retrieving the final hash works similar to the lane-processing phase of the KECCAK-$f$ permutation. The core registers are filled with four lanes, the dedicated sponge unit is then used to XOR 8 bits of the input at a time with the stored lanes. The result is finally written back to RAM. Other than by lane processing the lane at the origin can not

be skipped. Additional logic is required to handle this single non-interleaved lane during sponge computations.

**Runtime.**
The total runtime of hashing a single 640-bit message block is roughly 5.5 kcycles, which is 2 kcycles below the cycle count of the fastest 800-bit state version presented in [64]. This reduction can be contributed to the use of a 32-bit memory bus and the extension of the interleaving scheme to factor 4, which allows better bus width utilization during the slice processing phase. The throughput rises from 38.6 to 115.6 kBit/s(@1 MHz), which is mainly due to the massively higher rate $r$. The implementation in [64] adheres to the suggestion $c = 2n$ [10], so the rate $r$ is only $800 - 512 = 288$. For this thesis $c = n$, with $n$ only 160 bits, was chosen, so $r = 800 - 160 = 640$.

## 6.4 The Keccak Controller

The Keccak controller is implemented using a finite-state machine approach. The detailed state machine, including all states and possible transitions, is shown in Figure 6.6. Most states should be self-explanatory, some require additional explanation. The SpongeInit, SliceInit, and RhoInit states are used to feed the first address of the respective loading sequence to the RAM. For a RAM reading process the address needs to be given the cycle before the respective value appears at the output, i.e., there is a delay of one cycle. This delay is the reason for the init states. During the load states the address of the *next* to be read word is presented to the RAM. The Init, PermutationInit, and RoundInit states initialize certain variables, e.g., loop counters and flags, to the required value. The SliceParity state corresponds to the precomputation of the parity of the last slice 31, which needs to be executed at the beginning of the slice-processing phase.

## 6.5 Integration

Until now the Keccak modules were viewed as completely separate entities. Adding an entirely independent Keccak instance to the ECDSA implementation would be a waste of precious resources, hence the hashing modules have to be integrated into the existing design. As many resources as possible need to be shared to keep the area footprint of the added Keccak modules low.

One major shared part is obviously the RAM, the Keccak modules use the same RAM used by other parts of the ECDSA algorithm. The 800-bit state is stored in the lower-order parts of the RAM, the ECDSA field registers $\mathbf{R_0}$ to $\mathbf{R_4}$ are hence overwritten during the hashing process. The time of invoking the hashing algorithm in Algorithm 5.3 was chosen so that only 2 values need to be saved from being overwritten. The signature part $r$ is kept in $\mathbf{R_8}$, the inverse of the scalar $k$ is stored in $\mathbf{R_6}$. After hashing is finished, the Keccak controller dumps the 160-bit message digest in $\mathbf{R_7}$.

The Keccak algorithm is based on logic operations like XOR and rotations. In contrast, all other parts of the ECDSA rely on integer arithmetic, e.g., addition and multiplication. Also, the used Keccak architecture is highly specialized and far from a multipurpose microcontroller-like architecture. These facts make it hard to find similarities in the datapaths, hardly any resource sharing can be done.

It is however possible to merge the internal registers. The Keccak datapath includes a total of 144 register bits, needed by the main registers ($4 \times 32$) and the rotation registers

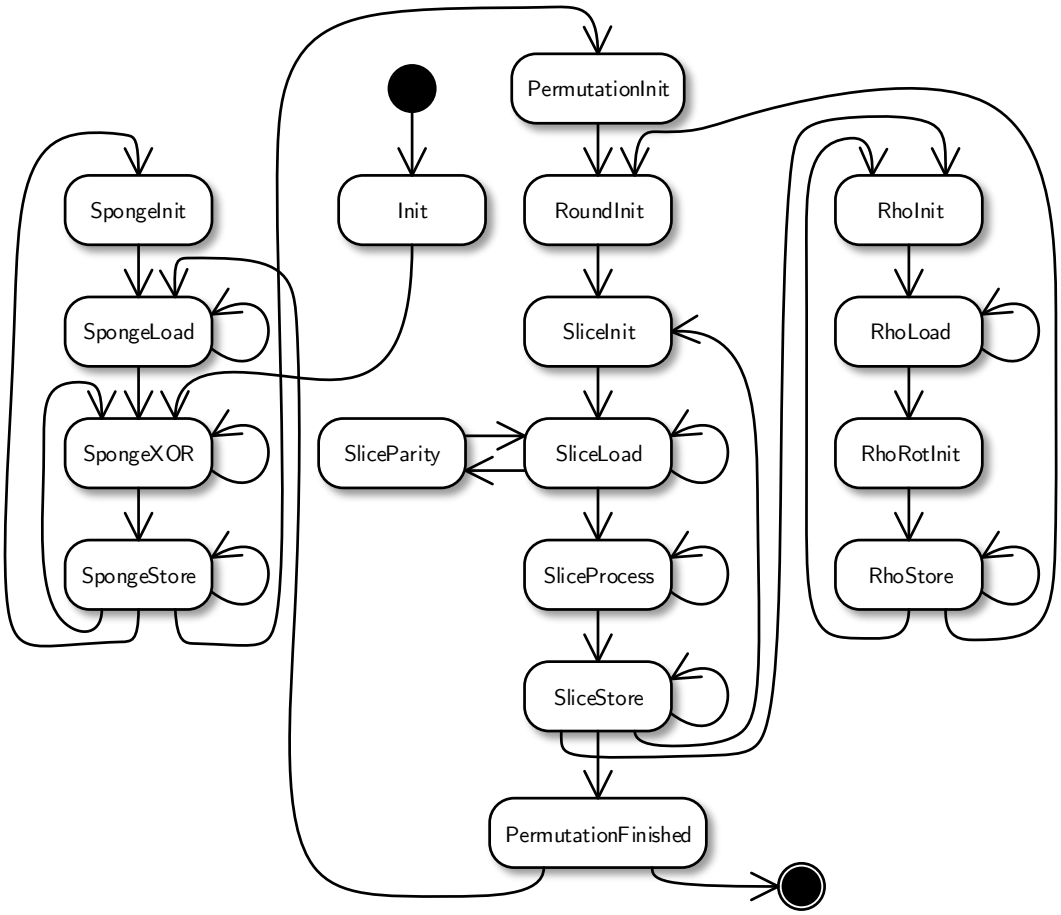



Figure 6.6: Keccak controller FSM

$(4 \times 4)$. The general ECDSA datapath (Figure 5.3) contains a 67-bit accumulator and $5 \times 16$ bit multiplication operand registers, which adds up to a total of 147 register bits. These numbers match up nicely. In fact, the ECDSA datapath was designed with the Keccak memory requirements in mind. The registers are hence merged into a single shared register file, as previously seen in Figure 4.1.

The structure of the common register file is illustrated in Figure 6.7. Due to the combined processing approach and the interleaved storage scheme, the load and store logic of the Keccak part is more complex, which is why the Keccak register file poses as the starting point. The accumulator and multiplication operand registers were then fitted.

The write-enable logic of Keccak's main registers allows enabling the write operation for each register nibble individually. If not enabled the nibble value will be kept and not be overwritten by its input at the clock transition. The enable logic generates a single 8 bit, i.e., a bit per nibble, write-enable signal that is fed to all 4 registers. Thus, a total of 16 bits are switched per bit of the enable signal. Due to this design, the ECDSA registers need to be spread over the four main registers to allow individual writing. As seen in Figure 6.7 the 64 low-order bit of the accumulator are spread over the lower half of the 4 registers, the up-most 3 bits are stored in a separate register. The 4 operand registers

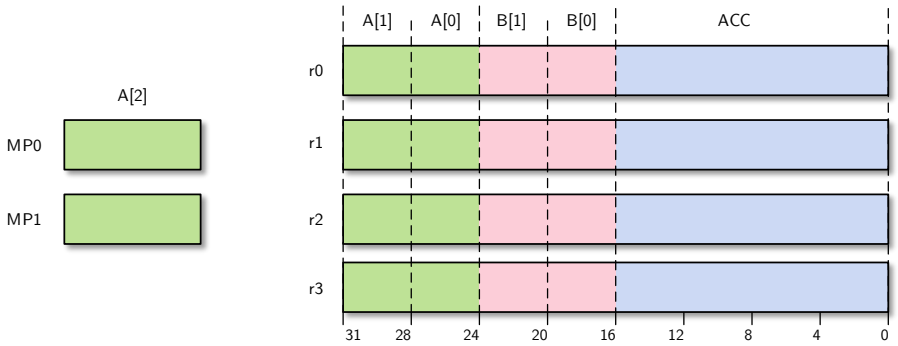



Figure 6.7: Common register file organisation

A[0], A[1], B[0], and B[1] are spread over the top half. A[2] is stored in the two 8 bit multi-purpose registers MP0 and MP1.

They are dubbed *multi-purpose* as they serve multiple functions during hashing. During the lane-processing phase they serve as $4 \times 4$ bit rotation registers. In the slice-processing phase the 5-bit slice parity is kept in MP0. The RAM allows byte-wise writing operations, however, only 4 bits of the lane[0,0] are used during an iteration of the slice-processing phase. The other nibble of the 8-bit memory word is kept in MP1.

The controllers are kept separate, no effort was made to integrate the KECCAKinto the top-level controller. The top-level controller can invoke the hashing operation by simply setting a start flag, hereby enabling the sub-controller. The KECCAK controller contains 3 counters, e.g., a loop counter and a round counter, which are merged with the top-level counters seen in Figure 5.14 to save additional area.

## 6.6 Summary

In this chapter, the hashing-specific modules of the hardware implementation were discussed. They are largely based on an existing design, some changes were made to adapt to, e.g., the wider bus. A slightly modified state-interleaving scheme (factor 4 instead of 2) is used to achieve a higher bus-width utilization. Also, the KECCAK components are tightly integrated into the existing design to minimize the area impact of hashing.

# Chapter 7

# Results and Discussion

In this chapter, the implementation results are presented. Detailed listings of area, power, and time requirements are given, they are then compared to related work. Finally, some points of discussion are raised and an outlook for possible future work is given.

## 7.1 Design Flow and Tools

After designing the implementation using pen and paper, it must somehow be transformed into gates, wires, and transistors. Various tools, programs, and description languages are required for this multi-step process, this section presents the most important parts of the used tool flow.

The first step was the generation of a cycle-accurate high-level model using the JAVA programming language. The final model was used for retrieving intermediate values and generating test vectors, which are required for testing the actual hardware model. The design was then implemented using the VHDL hardware description language.

The Synopsys Design Compiler 2013.03 mapped the HDL description to gates out of the *FSC0L_D* standard-cell library by Faraday. This library is based on the low-leakage 130 nm process technology by UMC. After place-and-route (PAR) (using Cadence RTL-to-GDSII), the power consumption is determined with the Cadence Encounter Power System v8.10. The power simulation is performed on the gate level, which might lack the accuracy of a transistor-level simulation, but is much faster in exchange. As a storage block a single-port SRAM hard macro provided by Faraday is used.

## 7.2 Implementation Results

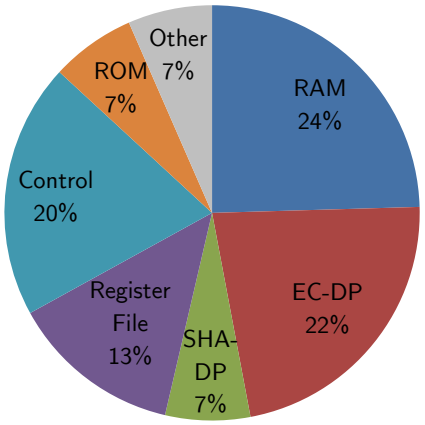
This section provides detailed results for area, power, and time requirements.

### 7.2.1 Area Requirements

The circuit area is given as reported by the RTL compiler. It is expressed in terms of $\mu m^2$ and *gate equivalents* (GEs), 1 GE is equivalent to the size of a 2-input NAND gate ($5.12\,\mu m^2$ in the used process). The scaling to GEs allows to compare designs using different process technologies, the physical size is obviously strongly dependent on the used technology node and can not be used for a fair comparison. Please beware that GE is not an exact measurement. Synthesizing the same design with varying tool flow, used parameters and process technology, might yield very different GE results.

| Component | Area | |
|---|---|---|
| | [GEs] | [µm²] |
| EC-DP | 2 800 | 14 334 |
| Multiplier | 1 616 | 8 273 |
| SHA-DP | 823 | 4 211 |
| Register File | 1 662 | 8 509 |
| Control | 2 473 | 12 663 |
| Top | 756 | 3 871 |
| Mult | 667 | 3 414 |
| DA | 594 | 3 039 |
| Keccak | 457 | 2 340 |
| Other | 816 | 4 178 |
| ROM | 820 | 4 198 |
| **Core Total** | **9 393** | **48 093** |
| RAM | 3 055 | 15 642 |
| **Total** | **12 448** | **63 735** |

(a) Area of chip components




(b) Area distribution

Figure 7.1: Area of chip components

The implementation requires a total area of 12 448 GEs (or 63 735 µm²). In Figure 7.1, a detailed analysis of the circuit size is given, Figure 7.1b illustrates the area distribution. With 3 kGEs, the RAM macro takes up roughly 1/4 of the total area. The ECDSA datapath, with its 2.8 kGEs the next biggest block, is dominated by the 16-bit integer multiplier. After adding the Keccak datapath and the shared register file, the complete datapath amounts to 42 % of the total area. The controller, which is comprised of 4 sub-controllers, takes up exactly 20 %. Interestingly, only 820 GEs are required for the ROM that is used to store the pre-computed elliptic curve points (2 560 bits). This is a very small price to pay when considering the massive speed-up provided by the fixed-base comb method (factor 4 times faster).

If no RAM macro is available, the memory must be realized using standard cells. A latch-based version of the RAM takes up 8.6 kGEs, thus raising the total area to roughly 18 kGEs. Bear in mind that the presented design revolves around usage of a RAM macro (including its limitations), so these values should not serve as a reference.

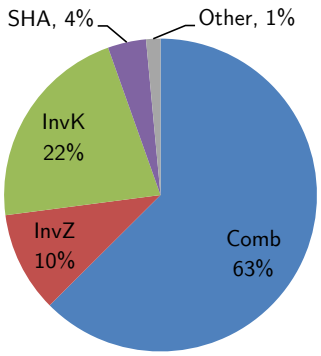
Due to the tight integration, it is difficult to accurately estimate the area impact of the Keccak algorithm. The hashing-specific modules (controller, datapath, LUT) add up to 1.4 kGEs. After addition of the required glue logic and considering the more complex register file, the hardware cost of hashing can be roughly estimated with 2 to 2.5 kGEs. In [64], a Keccak core area of 3 kGEs was reported, the main registers take up 1.2 kGEs. Hence, it was possible to save some area by sharing resources.

### 7.2.2 Timing Details

Signing a 160-bit message takes exactly 139 930 cycles, which translates to a time of 140 ms at an operating frequency of 1 MHz. The runtime of the hashing operation depends on the message length, all other parts of ECDSA computation feature a constant execution time.

| Operation | Cycles |
|---|---|
| Comb | 87 513 |
| InvZ | 14 550 |
| InvK | 30 252 |
| SHA | 5 537 |
| Other | 2 078 |
| Total | 139 930 |

(a) Runtime of algorithm parts




(b) Time distribution

Figure 7.2: Runtime analysis

As illustrated in Figure 7.2, the runtime is dominated by the ECSM with comb methods (taking up roughly 2/3 of the execution time). The comb method slashes the runtime by a factor of four, without it the ECSM would require approximately 350 kCycles. A tenth of the time is devoted to the computation of $Z^{-2} \bmod p$ (InvZ), the inversion of $k$ (InvK) is twice as expensive. This is due to the fact that the Montgomery multiplication scheme is more expensive than the fast reduction, also the number of required field multiplications and squarings is higher.

With a runtime of less than 5 %, hashing the 160-bit message is relatively cheap. The message-block size $r$ is set to 640 bits, each additional block adds 5.5 kCycles to the execution time.

The place-and-route (PAR) tool reported a maximum clock frequency of 55 MHz (after synthesizing the design using 1 MHz). However, this frequency is dependent on several register-transfer level compiler settings. Also, the synthesizer can adapt to higher frequencies by changing the design. The targeted application does not require high frequencies, so no detailed tests were conducted.

### 7.2.3 Power Consumption

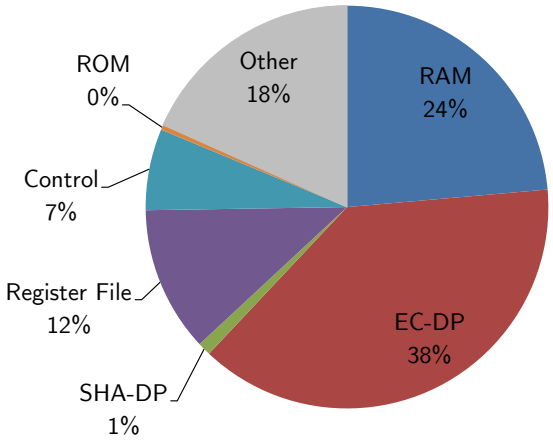
The mean power consumption was determined for typical process conditions. The implementation has a power consumption of 42.7 µW at a clock frequency of 1 MHz. Due to the usage of a low-leakage process technology, the static power consumption is in the nanowatt region. This makes a separation into static and dynamic power consumption pointless.

Figure 7.3 gives details of the power consumption of different chip components, the distribution only barely reflects the area requirements. The EC datapath and the register file combined drain more than half of the total power. The consumption of ROM and Keccak datapath is almost negligible, but bear in mind that this stems from the low activity of these parts. Hashing takes up less than 5 % of the total runtime, due do the use of operand isolation the hashing specific parts drain almost no power during the other steps of the signing process. Also, the ROM has a much lower activity than the RAM.

Interestingly, the *Other* part, which consists of mostly glue logic, consumes almost 20 % of the total power. The main culprit is the RAM multiplexer, which selects the RAM input from either the Keccak or the EC datapath output. This single part has a very high switching activity, which explains the relatively large current drain.

| Component | Power [µW/MHz] |
|---|---|
| EC-DP | 16.3 |
| Multiplier | 7.8 |
| SHA-DP | 0.5 |
| Register File | 4.9 |
| Control | 2.8 |
| Other | 8.1 |
| ROM | 0.2 |
| **Core Total** | **32.7** |
| RAM[1] | 10.0 |
| **Total** | **42.7** |

(a) Power consumption of chip components




(b) Power distribution

Figure 7.3: Area of chip components

For battery-powered devices the limiting factor is the energy consumption, power is only a secondary concern. Energy consumption is computed by multiplying the power drain with the runtime, this yields a consumption of 6 µJ.

## 7.3 Comparison

In this section, the outcomes of this thesis are compared to other reported low-resource implementations. Table 7.1 gives a detailed comparison.

In [44], Kern presented an ECDSA design also using the `secp160r1` curve. His implementation uses a 350 nm process and implements the SHA-1 hashing algorithm. It is both larger (by 5.8 kGEs) and considerably slower (factor 3.6) than the here presented work. Recently, Roy et al. [67] showed a point-multiplication coprocessor supporting both the `secp160r1` and a 192-bit curve. It requires 26 kGEs (in a 32 nm process technology) and takes 250 kCycles for 160-bit point multiplication. This design too is slower, roughly by a factor of 2.

192-bit prime field implementations have been shown by Wenger [81], Hutter et al. [33] and Fürbass et al. [21]. They are larger and significantly slower, which can partly be contributed to the larger prime-field size. Wenger's design is the smallest of the three, however, it is 10 times slower than the presented work.

Binary-field point-multiplication devices were presented by Hein et al. [30], Bock [13], and Lee et al. [49]. Area-wise they are comparable to the outcome of this thesis, although binary-field implementations are considered to be cheaper [81]. Also they include neither hashing nor signing capabilities.

Power consumption is extremely difficult to compare over different process technologies, so values are only given for designs using a 130 nm process. The 192-bit prime curve processor by Wenger—it uses the same 130 nm process technology used here—requires 39.54 µW/MHz and 55 µJ of energy. In comparison, the power consumption of our design is

[1]Please note that the specific RAM macro required for this work was not available for synthesis. The power consumption was estimated to be 10 µW/MHz.

Table 7.1: Comparison of prime and binary field ECC implementations

| | Techn. [nm] | Area [GEs] | Time [Cycles] | Power[a] [µW/MHz] | Field | Features[b] |
|---|---|---|---|---|---|---|
| **This work**[c] | **130** | **12 448** | **139 930** | **42.70** | $\mathbb{F}_{p160}$ | **ECDSA, SHA-3** |
| Roy13 [67] | 32 | 26 000 | 250 000 | - | $\mathbb{F}_{p160}$ | ECC, dual field |
| Kern10 [44] | 350 | 18 247 | 511 864 | - | $\mathbb{F}_{p160}$ | ECDSA, SHA-1 |
| Wenger11 [80, 81][c] | 130 | 14 644 | 1 394 000 | 39.54 | $\mathbb{F}_{p192}$ | ECDSA, SHA-1 |
| Hutter10 [33] | 350 | 19 115 | 859 188 | - | $\mathbb{F}_{p192}$ | ECDSA, SHA-1 |
| Fürbass07 [21] | 350 | 23 656 | 500 000 | - | $\mathbb{F}_{p192}$ | ECC |
| Hein08 [30] | 180 | 11 904 | 296 000 | - | $\mathbb{F}_{2^{163}}$ | ECC |
| Bock08 [13] ($d$=4) | 220 | 12 876 | 80 000 | - | $\mathbb{F}_{2^{163}}$ | ECC, DH |
| Lee08 [49] ($d$=1) | 130 | 12 506 | 302 457 | 32.42 | $\mathbb{F}_{2^{163}}$ | ECC, Schnorr |

[a] Power values of designs using other process technologies are omitted
[b] ECC refers to plain point-scalar multiplication.
[c] Uses a RAM macro for storage.

slightly higher (42.70 µW/MHz), but due to the lower cycle count the energy consumption is considerably smaller (6 µJ).

In Figure 7.4 the cycle count of the discussed implementations is plotted over the area requirements. As one could expect, faster implementations generally have higher area requirements. The presented implementation does not adhere to this trend, as it is both faster and smaller than previous work.

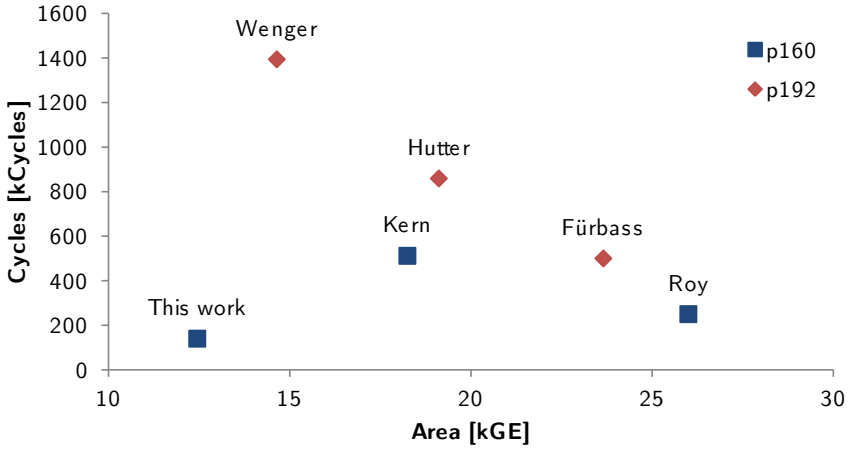



Figure 7.4: Time over area comparison of prime-field ECC and ECDSA implementations

## 7.4 Discussion and Future Work

In the following, some points for discussion and possible future work are presented.

**Extension to larger fields.**
The chosen 160-bit curve allows a small and thus cheap implementation, but does not offer a very high security level. For higher security demands the design can relatively easy be extended to larger, e.g., 192 or 224-bit, prime-field curves. The datapath would be mostly unchanged, the small adaptations in the controller would also not increase the circuit size. However, the memory requirements would rise, in the case of 192-bit fields by 20 %. This would translate to an estimated area increase of 1 to 2 kGEs.

Execution time would take a bigger hit. The product-scanning multiplication algorithm has quadratic complexity, increasing the field size by a factor of 1.2 (to 192 bits) thus results in a runtime increase of 44 %. The total runtime of signature generation is expected to rise between 40 and 50 %, to somewhere around 200 kCycles. These numbers for area and time requirements would still be competitive and an improvement over related work.

**Integration.**
The implementation is designed as a coprocessor, its sole purpose is signature computation. For usage in a real-life RFID tag additional logic is needed, e.g., for handling the cryptographic protocol or for the RFID front end. On the other side, for devices already featuring a RAM only the ECDSA core, with its area of 9.4 kGEs, needs to be added.

**SCA strengthening.**
A lot was written on the topic of security against SCA and other implementation attacks. Yet, no real-life tests have been conducted, as they would require the production of a real silicon chip. This would allow a more thorough evaluation of the design, especially the claimed security against side-channel analysis could be tested. It would be interesting to investigate power and EM attacks on the implemented comb method to evaluate if it is susceptible to these attacks and if the selection of the comb point stored in the ROM can be identified out of the traces. Also, second-order DPA attacks might target the multiplication with the private key $d$. Finally, susceptibility to fault attacks, in particular of the final subtraction in the Montgomery multiplication, could be tested.

Also, additional countermeasures could be implemented. Currently the LSB of the nonce $k$ is always set to 1, as required by the ZSD recoding scheme. There exists a better solution, before performing the ECSM one could add the odd group order $n$ to an even $k$, which does not change the final result. To retain a constant runtime this addition must always be executed. Thus, for an even $k$ one has to add the odd $n$, and for an odd $k$ the even $2n$. This conditional behavior might be detectable in an SPA, but it is still a better option than fixing the bit altogether.

The Keccak computation is currently not secured, an adversary might be able to determine the message or the hash. However, many cryptographic protocols involving ECDSA do not rely on message secrecy. Yet, an SCA resistant hashing operation might be required for random number generation.

**Experiment with comb size.**
For this work the comb width $w$ was fixed to 4, the reasons are discussed in Section 5.5. It might be interesting to experiment with other sizes and to perform a more detailed evaluation.

**Implement the Karatsuba multiplication.**
The next step in speeding up the implementation would be exchanging the school-book

multiplication (Section 5.2.2) with a more sophisticated algorithm. When using the Karatsuba multiplication algorithm, 32-bit multiplications can be computed in only 3 cycles (25 % faster than currently implemented). While the comb method only affects the ECSM, speeding up the multiplication also reduces time required for both inversions. Additional adders and registers for storing intermediates would be required, in total the area would probably rise somewhere between 1 and 2 kGEs.

## 7.5 Summary

In this chapter, the detailed implementation results were presented. It was shown that the hardware design requires 12.4 kGEs (or 63 735 $\mu m^2$) in a 130 nm low-leakage process. Signature generation takes a constant 140 kCycles. This makes this design smaller and considerably faster than related work. The power consumption is roughly 42.7 $\mu$W at a clock frequency of 1 MHz, which makes the design suitable for the use on passively-powered RFID tags.

# Chapter 8

# Conclusions

In this thesis, a low-resource hardware design of the ECDSA based on a 160-bit curve was presented. After stating the requirements and the most important design decisions, the design was presented and the results were thoroughly discussed.

Several new techniques were implemented. Application of an SCA-resistant comb method (with a width $w = 4$) dramatically decreased the runtime of the point-scalar multiplication. With only 800 GEs, the area impact of this measure is acceptable. New point-addition formulæ based on co-$Z$ notation further increased performance. The implementation features a 32-bit datapath, but with an integrated 16-bit multiplier. To achieve high multiplier utilization, a pipelined multiplication algorithm is executed. 32-bit multiplications take four cycles, during which the next multiplication operands are fetched. Especially prime-field inversions benefit from a dedicated fast-squaring operation. A single-port RAM macro is used as main storage element, which is much smaller than dual-ported macros or synthesized storage blocks.

The Keccak hashing algorithm was first evaluated in the context of ECDSA hardware implementations. The used Keccak architecture is based on a previous implementation, but it is strongly integrated into the existing design. Sharing of RAM and register file decreases the area impact of hashing.

Several countermeasures aimed at thwarting common implementation attacks were added. Most prominently, signature generation has a constant and data-independent runtime. Constant-time prime-field inversions are computed with the help of Fermat's little theorem, i.e., by means of a modular exponentiation. Also, the implemented comb method is highly regular. To counter DPA-based attacks, Randomized Projective Coordinates (RPC) were applied and the final multiplications were rearranged.

The implementation results have shown that a 160-bit ECDSA can be implemented using just 12.4 kGEs. The RAM takes up 3 kGEs, the computation core requires the other 9.4 kGEs. Signature generation is performed in a constant runtime of only 140 kCycles, where the point-scalar multiplication takes up the biggest part. These figures make the presented design both smaller and considerably faster than previous work. The power consumption is 42.7µW at a clock frequency of 1 MHz. The short computation allows operation at very low clock frequencies while still attaining (relatively) fast tag-response times. This would drastically decrease power consumption and would enable long reading ranges.

# Appendix A

# Abbreviations

| | |
|---|---|
| **AES** | Advanced Encryption Standard |
| **AMBA** | Advanced Microcontroller Bus Architecture |
| **APB** | Advanced Peripheral Bus |
| **ASIC** | Application-Specific Integrated Circuit |
| **CIPS** | Coarsely Integrated Product Scanning |
| **DA** | Doubling-Addition |
| **DES** | Data Encryption Standard |
| **DPA** | Differential Power Analysis |
| **DSA** | Digital Signature Algorithm |
| **EC** | Elliptic Curve |
| **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 |
| **ECSM** | Elliptic Curve Scalar Multiplication |
| **FIPS** | Finely Integrated Product Scanning |
| **FSM** | Finite State Machine |
| **GE** | Gate Equivalent |
| **HDL** | Hardware Description Language |
| **HMAC** | Keyed-Hash Message Authentication Code |
| **HW** | Hammin Weight |
| **LFSR** | Linear-Feedback Shift Register |
| **LSB** | Least Significant Bit |
| **LUT** | Lookup Table |
| **MAC** | Multiply-Accumulate |
| **MSB** | Most Significant Bit |
| **NFC** | Near Field Communication |
| **NIST** | National Institute of Standards and Technology |
| **PAR** | Place And Route |
| **RAM** | Random-Access Memory |
| **RFID** | Radio-Frequency Identification |
| **RNG** | Random Number Generator |
| **ROM** | Read-Only Memory |
| **RPA** | Refined Power-Analysis Attack |

| | |
|---|---|
| **RPC** | Randomized Projective Coordinates |
| **RTL** | Register-Transfer Level |
| **SCA** | Side-Channel Analysis |
| **SECG** | Standards for Efficient Cryptography Group |
| **SHA** | Secure Hash Algorithm |
| **SPA** | Simple Power Analysis |
| **SRAM** | Static Random-Access Memory |
| **XOR** | Exclusive Or |
| **ZPA** | Zero-Value Point Attack |
| **ZSD** | Zeroless Signed Digit |

# Appendix B

# Program Code

In this chapter, the detailed program code is listed. In Table B.1, the code used for the elliptic curve doubling-addition is shown. For a detailed explanation of the fields and their usage see Section 5.6. The OpTable and StepTable are merged, the three leftmost columns belong to the OpTable, while the others belong to the StepTable. The columns OpNr and StepNr contain the respective entry index, which is obviously not stored in the table.

In Table B.2, the contents of the TopLut, which houses the program executed by the top-level controller, are listed. The controller is thoroughly discussed in Section 5.8, a condensed and more readable version of the program is given in Algorithm 5.3 The unnumbered lines in Table B.2 refer to the dedicated states, the second column gives a short description of the respective operation.

In Algorithm B.1, the detailed program for the computation of $Z^{-2} \bmod p$ (see Section 5.7) is shown. It requires a total of 159S + 11M. The inversion of the scalar $k$ is not shown, as it is very lengthy and can be easily reconstructed by an interested reader.

Table B.1: Combined OpTable and StepTable

| **OPNr** | **Res** | **StepOffset** | **StepNr** | **Instr.** | **OpA** | **OpB** | **RomFlag** | **lastOp** |
|---|---|---|---|---|---|---|---|---|
| 0 | - | 0 | 0 | readK | - | - | 0 | 1 |
| 1 | 4 | 1 | 1 | copy | y | - | 1 | 1 |
| 2 | 5 | 2 | 2 | shift | 4 | 1 | 0 | 1 |
| 3 | 4 | 3 | 3 | accP | - | 0 | 0 | 0 |
| | | | 4 | acc | 4 | 0 | 0 | 0 |
| | | | 5 | sub | 4 | 0 | 0 | 1 |
| 4 | 5 | 6 | 6 | square | 3 | - | 0 | 1 |
| 5 | 6 | 7 | 7 | mult | 3 | 5 | 0 | 1 |
| 6 | 4 | 8 | 8 | mult | 4 | 6 | 0 | 0 |
| | | | 9 | accP | - | 0 | 0 | 0 |
| | | | 10 | sub | 2 | 0 | 0 | 1 |
| 7 | 6 | 11 | 11 | mult | 5 | x | 1 | 0 |
| | | | 12 | accP | - | 0 | 0 | 0 |
| | | | 13 | sub | 1 | 0 | 0 | 1 |
| 8 | 7 | 14 | 14 | square | 6 | - | 0 | 1 |
| 9 | 3 | 15 | 15 | acc | 3 | 0 | 0 | 0 |
| | | | 16 | acc | 6 | 0 | 0 | 1 |
| 10 | 6 | 17 | 17 | mult | 7 | 6 | 0 | 1 |
| 11 | 3 | 18 | 18 | square | 3 | - | 0 | 0 |
| | | | 19 | accP | - | 1 | 0 | 0 |
| | | | 20 | sub | 5 | 0 | 0 | 0 |
| | | | 21 | sub | 7 | 0 | 0 | 1 |
| 12 | 5 | 22 | 22 | mult | 1 | 7 | 0 | 1 |
| 13 | 2 | 23 | 23 | mult | 2 | 6 | 0 | 1 |
| 14 | 7 | 24 | 24 | square | 4 | - | 0 | 1 |
| 15 | 1 | 25 | 25 | accP | - | 4 | 0 | 0 |
| | | | 26 | acc | 7 | 2 | 0 | 0 |
| | | | 27 | sub | 6 | 2 | 0 | 0 |
| | | | 28 | sub | 5 | 3 | 0 | 0 |
| | | | 29 | sub | 5 | 2 | 0 | 1 |
| 16 | 6 | 30 | 30 | acc | 4 | 0 | 0 | 0 |
| | | | 31 | acc | 1 | 0 | 0 | 1 |
| 17 | 6 | 32 | 32 | square | 6 | - | 0 | 1 |
| 18 | 4 | 33 | 33 | square | 1 | - | 0 | 1 |
| 19 | 2 | 34 | 34 | shift | 2 | 3 | 0 | 1 |
| 20 | 6 | 35 | 35 | accP | - | 2 | 0 | 0 |
| | | | 36 | acc | 7 | 0 | 0 | 0 |
| | | | 37 | acc | 4 | 0 | 0 | 0 |
| | | | 38 | sub | 6 | 0 | 0 | 0 |
| | | | 39 | sub | 2 | 1 | 0 | 1 |
| 21 | 5 | 40 | 40 | shift | 5 | 2 | 0 | 1 |
| 22 | 5 | 41 | 41 | mult | 5 | 4 | 0 | 1 |
| 23 | 3 | 42 | 42 | mult | 3 | 1 | 0 | 1 |
| 24 | 4 | 43 | 43 | mult | 1 | 4 | 0 | 1 |
| 25 | 1 | 44 | 44 | square | 6 | - | 0 | 0 |
| | | | 45 | accP | - | 2 | 0 | 0 |
| | | | 46 | sub | 4 | 0 | 0 | 0 |
| | | | 47 | sub | 5 | 1 | 0 | 1 |
| 26 | 4 | 48 | 48 | mult | 2 | 4 | 0 | 1 |
| 27 | 5 | 49 | 49 | accP | - | 0 | 0 | 0 |
| | | | 50 | acc | 5 | 0 | 0 | 0 |
| | | | 51 | sub | 1 | 0 | 0 | 1 |
| 28 | 2 | 52 | 52 | mult | 6 | 5 | 0 | 0 |
| | | | 53 | accP | - | 0 | 0 | 0 |
| | | | 54 | sub | 4 | 0 | 0 | 1 |

Table B.2: TopLut

| OPNr | Description | Instr. | res | opA | opAMont | opB |
|---|---|---|---|---|---|---|
| 0 | copy randK | copy | 0 | | | |
| 1 | copy randZ | copy | 3 | | | |
| 2 | square randZ | square | 1 | 3 | | |
| 3 | mult randZ | mult | 2 | 3 | | 1 |
| 4 | read column | readK | | | | |
| 5 | mult $xz^2$ | mult | 1 | 1 | | |
| 6 | mult $yz^3$ | mult | 2 | 2 | | |
| | Comb | | | | | |
| | InvZ | | | | | |
| 7 | mult $XZ^{-2}$ | mult | 1 | 1 | | 2 |
| 8 | copy $x$ | copy | 8 | 1 | | |
| 9 | toMont $k$ | mult | 0 | 0 | | |
| 10 | copy $\bar{k}$ | copy | 0 | 0 | 1 | |
| 11 | square $k^2$ | square | 1 | 0 | | |
| 12 | mult $k^3$ | mult | 0 | 1 | 1 | 0 |
| 13 | copy $k^3$ | copy | 1 | 0 | 1 | |
| 14 | mult $k^5$ | mult | 0 | 1 | 1 | 1 |
| 15 | copy $k^5$ | copy | 2 | 0 | 1 | |
| 16 | mult $k^8$ | mult | 1 | 1 | | 2 |
| 17 | mult $k^9$ | mult | 0 | 1 | 1 | 0 |
| 18 | copy $k^9$ | copy | 3 | 0 | 1 | |
| | InvK | | | | | |
| 19 | copy $k^{-1}$ | copy | 6 | 1 | 1 | |
| | SHA-3 | | | | | |
| 20 | toMont $h$ | mult | 0 | 7 | | |
| 21 | copy $k^{-1}$ | copy | 0 | 6 | | |
| 22 | mult $k^{-1}e$ | mult | 1 | 0 | 1 | 0 |
| 23 | copy $k^{-1}e$ | copy | 1 | 1 | 1 | |
| 24 | copy $r$ | copy | 2 | 8 | 0 | |
| 25 | toMont $r$ | mult | 1 | 2 | | |
| 26 | copy $\bar{r}$ | copy | 2 | 1 | 1 | |
| 27 | mult $k^{-1}d$ | mult | 0 | 0 | | |
| 28 | mult $k^{-1}dr$ | mult | 1 | 0 | 1 | 2 |
| 29 | addMont | acc | 0 | 1 | 1 | 1 |
| 30 | fromMont $s$ | mult | 1 | 0 | 1 | |
| 31 | copy $s$ | copy | 0 | 1 | 1 | |

**Algorithm B.1:** Computation of $Z^{-2}$ mod

**Input**: $R_3 = Z$
**Output**: $R_2 = Z^{-2} \bmod p$

1 $R_2 \leftarrow {R_3}^2$;
2 $R_7 \leftarrow R_3 \times R_2$ ; // $R_7 = Z^{2^2-1}$
3 $R_2 \leftarrow {R_7}^2$;
4 $R_2 \leftarrow {R_2}^2$;
5 $R_4 \leftarrow R_2 \times R_7$; // $R_4 = Z^{2^4-1}$
6 $R_2 \leftarrow {R_4}^2$;
7 **Repeat** $R_2 \leftarrow {R_2}^2$ **times** *3*;
8 $R_5 \leftarrow R_4 \times R_2$; // $R_5 = Z^{2^8-1}$
9 $R_2 \leftarrow {R_5}^2$;
10 **Repeat** $R_2 \leftarrow {R_2}^2$ **times** *7*;
11 $R_6 \leftarrow R_5 \times R_2$; // $R_6 = Z^{2^{16}-1}$
12 $R_2 \leftarrow {R_6}^2$;
13 **Repeat** $R_2 \leftarrow {R_2}^2$ **times** *15*;
14 $R_7 \leftarrow R_6 \times R_2$; // $R_7 = Z^{2^{32}-1}$
15 $R_2 \leftarrow {R_7}^2$;
16 **Repeat** $R_2 \leftarrow {R_2}^2$ **times** *31*;
17 $R_7 \leftarrow R_6 \times R_2$; // $R_7 = Z^{2^{64}-1}$
18 $R_2 \leftarrow {R_7}^2$;
19 **Repeat** $R_2 \leftarrow {R_2}^2$ **times** *63*;
20 $R_7 \leftarrow R_7 \times R_2$; // $R_7 = Z^{2^{128}-1}$
21 $R_2 \leftarrow {R_2}^2$;
22 **Repeat** $R_2 \leftarrow {R_2}^2$ **times** *16*;
23 $R_2 \leftarrow R_6 \times R_2$;
24 **Repeat** $R_2 \leftarrow {R_2}^2$ **times** *8*;
25 $R_2 \leftarrow R_5 \times R_2$;
26 **Repeat** $R_2 \leftarrow {R_2}^2$ **times** *4*;
27 $R_2 \leftarrow R_4 \times R_2$;
28 $R_2 \leftarrow {R_3}^2$;
29 $R_2 \leftarrow R_3 \times R_2$;
30 $R_2 \leftarrow {R_2}^2$;
31 $R_2 \leftarrow {R_3}^2$;

# Appendix C

# A Take at a Faster and Smaller Montgomery Multiplication

An attempt was made to speed up and simultaneously simplify the Montgomery multiplication scheme. When setting the word size $w$ to 1, i.e., to radix 2, some operations can be simplified. As computations with a modulus of $2^0 = 1$ become simple bit operations, retrieving a single bit of $m$ is achieved with a simple bit test. Multiplication with $m$ becomes a conditional addition.

Note that there already exist several proposals for dedicated Montgomery multipliers utilizing Radix 2. The *Multiple Word Radix-2 Montgomery Multiplication algorithm* (MWR2MM) was proposed by Tenca and Koç [74] and later optimized by Huang et al. [32]. A dedicated unit for performing Montgomery multiplications is however expensive in terms of area, therefore these algorithms are not considered here. Instead, the 16-bit integer multiplier is reused.

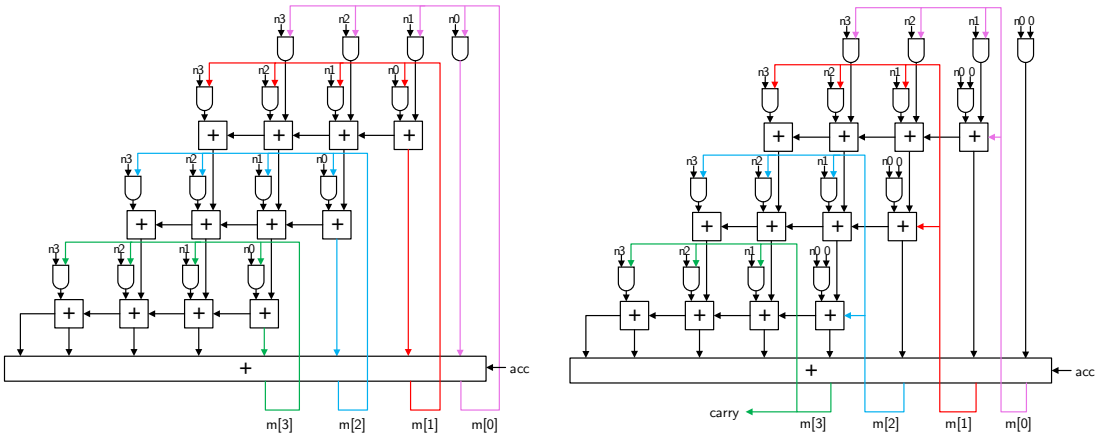
Algorithm C.1 shows how 16 bits of $m$ can be calculated and multiplied with $n$. Bit after bit is tested, if it is 1 the low-order bits of the modulus ($n_0$) are added at the current bit position. The next iteration of the loop tests the updated accumulator. When finished, the 16 lower-order bits of the accumulator are zero.

**Algorithm C.1:** Computing a 16-bit Part of $m$ in Radix 2

```
1 for i = 0 to 15 do
2     m[i] = acc[i];
3     acc = acc + (m[i]×n0 << i);
4 end
```

It is possible to perform all of Algorithm C.1 in a single cycle, i.e., compute 16 bits of $m$ and simultaneously multiply it with $n$. As can be seen seen in Figure C.1, this is achieved by slightly modifying the 16-bit multiplier. The first operand is set to $n_0$, the output of the accumulation adder is fed back to the second multiplier input. A straightforward implementation results in a purely combinational hardware loop (Figure C.1a). The architecture depicted in Figure C.1b avoids such a loop. It is based on the observation that the low-order bits of the final accumulator value are known to be zero, thus there is no need to compute them. The first addition of each row is skipped by simply setting one operand to zero, which breaks the loop. The now-missing carry bit is compensated by feeding the adder output to the carry input of the next adder row. This is correct due to

the aforementioned observation of the result being zero. This scheme can not be applied to the last row, instead the carry bit is stored in a register and added to the LSB for the next multiplication.




(a) Combinational loop

(b) Loop corrected

Figure C.1: Multiplier in Montgomery configuration

On theory this approach is smaller and faster than the used method with$w = 16$. No cycles need to be spent on the computation of $m$, hence saving 10 cycles per multiplication.

However, in practice these savings come with a hefty cost. The drastically lengthened critical path forces the RTL Compiler to construct a faster multiplier, which results in an almost doubled gate count. Combined with a higher multiplexing effort, this scheme adds roughly 500 GEs to the area requirements in comparison to the used method. The more complex multiplier also has a negative impact on power consumption. Considering the relatively small time savings, the described multiplication technique is not worth the additional logic.

# Bibliography


[1] T. Akishita and T. Takagi. Zero-Value Point Attacks on Elliptic Curve Cryptosystem. In C. Boyd and W. Mao, editors, *Information Security*, volume 2851 of *Lecture Notes in Computer Science*, pages 218–233. Springer Berlin Heidelberg, 2003.

[2] American National Standards Institute (ANSI). AMERICAN NATIONAL STANDARD X9.62-2005. Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA), 2005.

[3] ARM. AMBA APB Protocol Version: 2.0 - Specification, 2010. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0024-/`.

[4] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. Cryptology ePrint Archive (`http://eprint.iacr.org/`), Report 2004/100, 2004.

[5] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in Cryptology—CRYPTO '86*, pages 311–323, London, UK, 1986. Springer-Verlag.

[6] D. Bernstein and T. Lange. Explicit-formulas database. www.hyperelliptic.org/EFD.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Sponge Functions Corner. `http://sponge.noekeon.org/`.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions. Submission to NIST (Round 3), 2011.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference. Submission to NIST (Round 3), 2011.

[10] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission. Submission to NIST (Round 3), 2011.

[11] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. V. Keer. Keccak Implementation Overview, V3.2, 2012.

[12] B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In *CARDIS 2013*, Lecture Notes in Computer Science. Springer-Verlag, 2013.

[13] H. Bock, M. Braun, M. Dichtl, E. Hess, J. Heyszl, W. Kargl, H. Koroschetz, B. Meyer, and H. Seuschek. A Milestone Towards RFID Products Offering Asymmetric Authentication Based on Elliptic Curve Cryptography. Invited talk at RFIDsec 2008, July 2008.

[14] M. K. Brown, D. R. Hankerson, J. C. L. Hernández, and A. J. Menezes. Software Implementation of the NIST Elliptic Curves Over Prime Fields. In D. Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographers' Track at the RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2001.

[15] Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0. Available online at `http://www.secg.org/`, September 2000.

[16] Certicom Research. Standards for Efficient Cryptography (SECG), SEC 1: Elliptic Curve Cryptography, Version 1.0. Available online at `http://www.secg.org/`, September 2000.

[17] Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0. Available online at `http://www.secg.org/`, January 2010.

[18] J.-S. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES'99, First International Workshop, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.

[19] S. R. Dussé and B. Kaliski. A Cryptographic Library for the Motorola DSP56000. In *Advances in Cryptology - EUROCRYPT '90, Workshop on the Theory and Application of of Cryptographic Techniques, Aarhus, Denmark, May 21-24, 1990, Proceedings*, volume 473 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 1990.

[20] M. Feldhofer and J. Wolkerstorfer. *RFID Security: Techniques, Protocols and System-On-Chip Design*, chapter Hardware Implementation of Symmetric Algorithms for RFID Security, pages 373–415. Springer, 2008.

[21] F. Fürbass and J. Wolkerstorfer. ECC Processor with Low Die Size for RFID Applications. In *Proceedings of 2007 IEEE International Symposium on Circuits and Systems*. IEEE, IEEE, May 2007.

[22] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[23] D. Genkin, A. Shamir, and E. Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013. `http://eprint.iacr.org/`.

[24] L. Goubin. A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. In Y. Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January*

*6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2003.

[25] R. R. Goundar, M. Joye, A. Miyaji, M. Rivain, and A. Venelli. Scalar multiplication on weierstraß elliptic curves from co-z arithmetic. *Journal of cryptographic engineering*, 1(2):161–176, 2011.

[26] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, 6th International Workshop, Cambridge, MA, USA, August 11-13, 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2004.

[27] G. Hachez and J.-J. Quisquater. Montgomery Exponentiation with no Final Subtractions: Improved Results. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000, Second International Workshop Worcester, MA, USA, August 1718, 2000 Proceedings*, volume 1965, pages 91–100. Springer Berlin / Heidelberg, August 2000.

[28] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2004.

[29] M. Hedabou, P. Pinel, and L. Bénéteau. Countermeasures for Preventing Comb Method Against SCA Attacks. In *Proceedings of the First International Conference on Information Security Practice and Experience*, ISPEC'05, pages 85–96, Berlin, Heidelberg, 2005. Springer-Verlag.

[30] D. Hein. Elliptic Curve Cryptography ASIC for Radio Frequency Authentication. Master thesis, Technical University of Graz, April 2008.

[31] N. Howgrave-Graham and N. P. Smart. Lattice Attacks on Digital Signature Schemes. *Designs, Codes and Cryptography*, 23(3):283–290, August 2001. ISSN 0925-1022.

[32] M. Huang, K. Gaj, S. Kwon, and T. El-Ghazawi. An Optimized Hardware Architecture for the Montgomery Multiplication Algorithm. In R. Cramer, editor, *Public Key Cryptography - PKC 2008*, volume 4939 of *Lecture Notes in Computer Science*, pages 214–228. Springer Berlin Heidelberg, 2008.

[33] M. Hutter, M. Feldhofer, and T. Plos. An ECDSA Processor for RFID Authentication. In S. B. O. Yalcin, editor, *Workshop on RFID Security – RFIDsec 2010, 6th Workshop, Istanbul, Turkey, June 7-9, 2010, Proceedings*, volume 6370 of *Lecture Notes in Computer Science*, pages 189–202. Springer, 2010.

[34] M. Hutter, M. Joye, and Y. Sierra. Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-Z Coordinate Representation. In A. Nitaj and D. Pointcheval, editors, *Progress in Cryptology - AFRICACRYPT 2011 Fourth International Conference on Cryptology in Africa, Dakar, Senegal, July 5-7, 2011. Proceedings*, volume 6737 of *Lecture Notes in Computer Science*, pages 170–187. Springer, 2011.

[35] M. Hutter and E. Wenger. Fast Multi-Precision Multiplication for Public-Key Cryptography on Embedded Microprocessors. In B. P. und Tsuyoshi Takagi, editor, *Cryptographic Hardware and Embedded Systems – CHES 2011, 13th International Workshop, Nara, Japan, September 28 - October 1, 2011, Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer, 2011.

[36] International Organisation for Standardization (ISO). Information Technology - Security Techniques - Entity authentication mechanisms - Part 3: Entity authentication using a public key algorithm, 1993.

[37] T. Izu and T. Takagi. A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks. In *5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002 Paris, France, February 1214, 2002 Proceedings*, volume 2274, pages 280–296. Springer-Verlag, 2002.

[38] M. Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 135–147, 2007.

[39] M. Joye and C. Tymen. Protections against Differential Analysis for Elliptic Curve Cryptography. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 2001.

[40] M. Joye and S.-M. Yen. The Montgomery Powering Ladder. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2003.

[41] B. Jungk and J. Apfelbeck. Area-Efficient FPGA Implementations of the SHA-3 Finalists. In *Reconfigurable Computing and FPGAs–ReConFig 2011, International Conference, November 30-December 2, Cancun, Mexico, 2011*, pages 235–241, 2011.

[42] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics-Doklady*, pages 595–596, 1963.

[43] Keccak Design Team. The Keccak sponge function family. `http://keccak.noekeon.org/`.

[44] T. Kern and M. Feldhofer. Low-Resource ECDSA Implementation for Passive RFID Tags. In *17th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2010), December 12-15th, 2010, Athens, Greece, Proceedings*, pages 1236–1239. IEEE, 2010.

[45] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[46] Ç. K. Koç, T. Acar, and B. S. K. Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[47] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer, 1996.

[48] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[49] Y. K. Lee, K. Sakiyama, L. Batina, and I. Verbauwhede. Elliptic-Curve-Based Security Processor for RFID. *IEEE Transactions on Computers*, 57(11):1514–1527, November 2008.

[50] C. H. Lim and P. J. Lee. More Flexible Exponentiation with Precomputation. In Y. Desmedt, editor, *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107, 1994.

[51] M. Liu and P. Nguyen. Solving BDD by Enumeration: An Update. In E. Dawson, editor, *Topics in Cryptology - CT-RSA 2013*, volume 7779 of *Lecture Notes in Computer Science*, pages 293–309. Springer Berlin Heidelberg, 2013.

[52] P. Longa and A. Miri. New composite operations and precomputation scheme for elliptic curve cryptosystems over prime fields. In *in Public Key Cryptography (PKC08), LNCS*, pages 229–247. Springer, 2008.

[53] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN 978-0-387-30857-9.

[54] N. Meloni. New Point Addition Formulae for ECC Applications. In C. Carlet and B. Sunar, editors, *Arithmetic of Finite Fields*, volume 4547 of *Lecture Notes in Computer Science*, pages 189–201. Springer Berlin Heidelberg, 2007.

[55] V. S. Miller. Use of Elliptic Curves in Cryptography. In H. C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1986.

[56] P. L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44:519–521, 1985.

[57] P. L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, January 1987. ISSN 0025-5718.

[58] E. Mulder, M. Hutter, M. Marson, and P. Pearson. Using Bleichenbacher's Solution to the Hidden Number Problem to Attack Nonce Leaks in 384-Bit ECDSA. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 435–452. Springer Berlin Heidelberg, 2013.

[59] National Institute of Standards and Technology (NIST). SHA-3 Standardization. `http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3_standardization.html`. Last retrieved January 2014.

[60] National Institute of Standards and Technology (NIST). FIPS-180-3: Secure Hash Standard, October 2008. Available online at `http://www.itl.nist.gov/fipspubs/`.

[61] National Institute of Standards and Technology (NIST). FIPS-186-3: Digital Signature Standard (DSS), 2009. Available online at `http://www.itl.nist.gov/fipspubs/`.

[62] K. Okeya, K. Miyazaki, and K. Sakurai. A Fast Scalar Multiplication Method with Randomized Projective Coordinates on a Montgomery-Form Elliptic Curve Secure against Side Channel Attacks. In K. Kim, editor, *Information Security and Cryptology ICISC 2001*, volume 2288 of *Lecture Notes in Computer Science*, pages 428–439. Springer Berlin Heidelberg, 2002.

[63] S. Örs, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of a Montgomery modular multiplier in a systolic array. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8 pp.–, 2003.

[64] P. Pessl and M. Hutter. Pushing the limits of sha-3 hardware implementations to fit on rfid. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 126–141. Springer Berlin Heidelberg, 2013.

[65] J.-J. Quisquater and D. Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In I. Attali and T. P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[66] D. C. Ranasinghe and P. H. Cole. *Networked RFID Systems and Lightweight Cryptography*. Springer Berlin Heidelberg, 2008.

[67] S. S. Roy, B. Yang, V. Rozic, N. Mentens, J. Fan, and I. Verbauwhede. Designing tiny ecc processor. Available online at `https://www.cosic.esat.kuleuven.be/ecc2013/files/sujoy.pdf`, 2013. Presentation at the 17th Workshop on Elliptic Curve Cryptography (ECC 2013),.

[68] M.-J. O. Saarinen and D. Engels. A do-it-all-cipher for rfid: Design requirements (extended abstract). Cryptology ePrint Archive: Report 2012/317, June 2012.

[69] S. Sarma. Towards the 5 Cent Tag. White paper, MIT Auto-ID Center, 2001.

[70] S. E. Sarma, S. A. Weis, and D. W. Engels. Radio Frequency Identification: Risks and Challenges. *CryptoBytes (RSA Laboratories)*, 6(1):325, Spring 2003.

[71] S. E. Sarma, S. A. Weis, and D. W. Engels. RFID Systems and Security and Privacy Implications. In B. S. K. andÇetin Kaya Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 454–470. Springer, August 2003.

[72] M. Stevens. New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis. In T. Johansson and P. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 245–261. Springer Berlin Heidelberg, 2013.

[73] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate. Cryptology ePrint Archive, Report 2009/111, 2009. `http://eprint.iacr.org/`.

[74] A. F. Tenca and C. K. Koç. A Scalable Architecture for Montgomery Multiplication. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 1999.

[75] C. D. Walter. Montgomery Exponentiation needs no Final Subtractions. *Electronics Letters*, 35:1831–1832, 1999.

[76] C. D. Walter. Montgomery's Multiplication Technique: How to Make it Smaller and Faster. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES99, First International Workshop, Worcester, MA, USA, August 1213, 1999 Proceedings.*, volume 1717, pages 80–93. Springer, 1999.

[77] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In V. Shoup, editor, *Advances in Cryptology - CRYPTO 2005, 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[78] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In R. Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[79] S. A. Weis, S. E. Sarma, R. L. Rivest, and D. W. Engels. Security and Privacy Aspects of Low-Cost Radio Frequency Identification Systems. In D. Hutter, G. Müller, W. Stephan, and M. Ullmann, editors, *Security in Pervasive Computing, 1st Annual Conference on Security in Pervasive Computing, Boppard, Germany, March 12-14, 2003, Revised Papers*, volume 2802 of *Lecture Notes in Computer Science*, pages 201–212. Springer, March 2003.

[80] E. Wenger, M. Feldhofer, and N. Felber. Low-Resource Hardware Design of an Elliptic Curve Processor for Contactless Devices. In Y. Chung and M. Yung, editors, *WISA*, volume 6513, pages 92–106. Springer, 2010.

[81] E. Wenger and M. Hutter. Exploring the Design Space of Prime Field vs. Binary Field ECC-Hardware Implementations. In P. Laud, editor, *16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers*, volume 7161 of *Lecture Notes in Computer Science*, pages 256–271. Springer Berlin Heidelberg, 2011.

[82] S.-M. Yen and M. Joye. Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. In *IEEE Transactions on Computers*, volume 49 of *IEEE Transactions on Computers*, pages 967–970. IEEE Computer Society, 2000.

[83] S.-M. Yen, S. Kim, S. Lim, and S. Moon. A Countermeasure Against One Physical Cryptanalysis May Benefit Another Attack. In *Proceedings of the 4th International Conference Seoul on Information Security and Cryptology*, ICISC '01, pages 414–427, London, UK, UK, 2002. Springer-Verlag.

[84] S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon. RSA Speedup with Residue Number System Immune against Hardware Fault Cryptanalysis. In K. Kim, editor, *Information Security and Cryptology - ICISC 2001, 4th International Conference Seoul, Korea, December 6-7, 2001, Proceedings*, volume 2288 of *Lecture Notes in Computer Science*, pages 397–413. Springer, 2002.