KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK–ESAT
Kasteelpark Arenberg 10, 3001 Leuven-Heverlee

Secure and Efficient Coprocessor Design for Cryptographic Applications on FPGAs

Promotors:
Prof. Dr. ir. Bart Preneel
Prof. Dr. ir. Ingrid Verbauwhede

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Nele MENTENS

June 2007

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK–ESAT
Kasteelpark Arenberg 10, 3001 Leuven-Heverlee

Secure and Efficient Coprocessor Design for Cryptographic Applications on FPGAs

Jury:
Prof. Dr. ir. Etienne Aernoudt, voorzitter
Prof. Dr. ir. Bart Preneel, promotor
Prof. Dr. ir. Ingrid Verbauwhede, promotor
Prof. Dr. ir. Hugo De Man
Prof. Dr. ir. Joos Vandewalle
Prof. Dr. ir. Wim Dehaene
Prof. Dr. ir. Christof Paar, RUB, Bochum
Dr. ir. Jan Genoe, IMEC, Leuven

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Nele MENTENS

U.D.C. 519.72 June 2007

D/2007/7515/77

ISBN 978-90-5682-843-1

Acknowledgments

After 4 years of Ph.D. research, I would like to thank the people who supported me along the way.

First of all, great appreciation goes to my promoters, Prof. Bart Preneel and Prof. Ingrid Verbauwhede. They gave me the chance to be a Ph.D. researcher at COSIC and led my research in the right direction. Their complementary knowledge on cryptography and electronic system design was vital for this thesis to succeed.

My gratitude also goes to Prof. Hugo De Man and Prof. Joos Vandewalle, who were on my advisory committee and who carefully evaluated my work during the past 4 years. Furthermore, I give thanks to Prof. Wim Dehaene, Prof. Christof Paar and Jan Genoe for accepting to be on my examination committee and providing me with valuable suggestions and remarks to improve this manuscript. I also thank the chairman of the jury, Prof. Etienne Aernoudt.

Many thanks go to my colleagues at COSIC for making our research group a big happy family. Special thanks go to Lejla Batina, who has been my travelling companion and good friend since the beginning of my research at COSIC. I am grateful to Kazuo Sakiyama for the insights and discussions, but also for the diversions and laughs. I would also like to thank Frederik Vercauteren for his useful comments on this manuscript and, together with Jasper Scholten, for his help on mathematical problems. Without Péla Noë and Elvira Wouters, COSIC would be nothing but disorder, so many thanks to them as well for their administrative support. Besides my COSIC colleagues, I would also like to thank my ESAT colleague Marian Verhelst for her friendship and for the yoghurt breaks.

Furthermore, I would like to thank my colleagues at the Katholieke Hogeschool Limburg, who motivated me by always being interested in the progress of my research and who made it possible to practically arrange the combination of teaching in Diepenbeek and doing research in Leuven. Special thanks go to Jan Genoe, who encouraged me to start studying at the K. U. Leuven and supported me during both my master and doctoral studies.

Besides my colleagues, I appreciate the care and support of my good friends, who always managed to move my thoughts away from work when needed.

Last but not least, I would like to thank my family and in particular my parents

and my brother Roel for being there for me in an always warmhearted and happy home. Very special thanks and a lot of love go to my dearest Kamiel, who always motivated and supported me with great enthusiasm and with whom I hope to have a long and happy future.

Nele Mentens
June 2007

Abstract

Security in embedded systems requires the choice of a suitable implementation platform. For some systems, a general purpose microprocessor satisfies the requirements, but when high performance is the main criterium, cryptographic coprocessors in hardware are indispensable. When very high performance is required or when a high volume of coprocessors is needed, ASICs (Application Specific Integrated Circuits) are chosen as implementation platforms. In this case, the reconfigurability of FPGAs (Field Programmable Gate Arrays) is only used for prototyping. However, because of the efforts of FPGA manufacturing companies, the performance gap between ASICs and FPGAs becomes smaller and smaller. FPGAs have become heterogeneous systems with a variety of dedicated resources such as multiplier blocks, DSP slices, RAM blocks,... This explains the trend that FPGAs are more and more used as end products. Following this trend, the need for specific FPGA architectures can be justified. This work focuses on cryptographic coprocessor design, optimized for FPGAs. We implemented a coprocessor for public key cryptography, which provides RSA and elliptic curve cryptography. The most important operation in these algorithms is modular multiplication, for which we developed a high-speed architecture based on Montgomery's method. The control logic built on top of this data path provides a hierarchical solution, where all instructions in the hierarchy are accessible from outside the coprocessor. Our public key coprocessor shows a better area-speed trade-off than all known solutions. Furthermore, the area of the S-box in the symmetric key algorithm AES was evaluated. The conclusion of this evaluation was that all other known solutions were suboptimal.

Samenvatting

De veiligheid van ingebedde elektronische systemen is afhankelijk van het gekozen implementatie-platform. Voor sommige systemen is een microprocessor voor algemene toepassingen voldoende, maar wanneer hoge snelheid of een kleine oppervlakte het doel zijn, zijn cryptografische coprocessoren in hardware onontbeerlijk. Wanneer een zeer hoge performantie vereist is of wanneer er een groot aantal coprocessoren nodig zijn, worden ASIC's (Application Specific Integrated Circuits) gekozen als implementatie-platformen. In dit geval wordt de herconfigureerbaarheid van FPGA's (Field Programmable Gate Arrays) enkel gebruikt voor het maken van prototypes. Desalniettemin, dankzij de inspanningen van FPGA producenten, wordt het verschil in performantie tussen ASIC's en FPGA's steeds kleiner. Moderne FPGA's zijn heterogene systemen met een waaier aan speciale componenten zoals vermenigvuldigers, DSP blokken, RAM blokken,... Dit verklaart de trend dat FPGA's meer en meer als eindproduct gebruikt worden. Daarom is er nood aan specifieke FPGA architecturen voor de implementatie van cryptografische algoritmen. In dit doctoraatsproject wordt het ontwerp van coprocessoren, specifiek voor FPGA's, onderzocht. We hebben een coprocessor ontwikkeld voor publieke sleutel cryptografie, die zowel RSA als elliptische kromme cryptografie kan uitvoeren. De meest cruciale bewerking in deze algoritmen is modulaire vermenigvuldiging, waarvoor we een naar snelheid geoptimaliseerde architectuur geïmplementeerd hebben, gebaseerd op Montgomery's methode. De controlelogica bovenop dit datapad is hiërarchisch opgebouwd, waarbij alle bewerkingen in de hiërarchie uitvoerbaar zijn van buiten de coprocessor. Onze publieke sleutel coprocessor geeft een beter compromis tussen oppervlakte en snelheid dan alle gekende oplossingen. Verder werd de oppervlakte van de S-box in het symmetrische sleutel algoritme AES geëvalueerd met als besluit dat eerder voorgestelde oplossingen suboptimaal waren.

Contents

List of Abbreviations

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
CBC	Cipher Block Chaining
CCM	Counter with CBC-MAC
CDH	Computational Diffie-Hellman
CFB	Cipher FeedBack
CIOS	Coarsely Integrated Operand Scanning
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Semiconductor
CRT	Chinese Remainder Theorem
CPLD	Complex Programmable Logic Device
CSA	Carry-Save Adder
CTR	CounTeR
CWC	Carter-Wegman and CTR mode
DES	Data Encryption Standard
DLP	Discrete Logarithm Problem
DPA	Differential Power Analysis
DSA	Digital Signature Algorithm
DSP	Digital Signal Processing
DSRCP	Domain-Specific Reconfigurable Cryptographic Processor
DSS	Digital Signature Standard
EC	Elliptic Curve
ECB	Electronic CodeBook
ECC	Elliptic Curve Cryptography
ECCDH	Elliptic Curve Computational Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
FA	Full Adder
FF	Flip-Flop
FIOS	Finely Integrated Operand Scanning
FPGA	Field Programmable Gate Array

FSM	Finite State Machine
GCM	Galois CounterMode
GF	Galois Field
HECC	HyperElliptic Curve Cryptography
IAPM	Integrity Aware Parallelizable Mode
iff	if and only if
LSB	Least Significant Bit
LUT	Look-Up Table
MSB	Most Significant Bit
NAF	Non-Adjacent Form
NIST	National Institute of Standards and Technology
nMOS	negative-channel Metal-Oxide Semiconductor
OCB	Offset CodeBook
OFB	Output FeedBack
PKC	Public Key Cryptography
pMOS	positive-channel Metal-Oxide Semiconductor
RAM	Random Access Memory
RCA	Ripple Carry Adder
RSA	Rivest-Shamir-Adleman
SFF	Slice Flip-Flop
SOS	Separated Operand Scanning
SPA	Simple Power Analysis
TDEA	Triple Data Encryption Algorithm
TTP	Trusted Third Party
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
XECB	eXtended Electronic CodeBook

List of Notation

E_k	symmetric key encryption with the secret key k
D_k	symmetric key decryption with the secret key k
E_B	public key encryption with the public key B
D_b	public key decryption with the private key b
$H(m)$	hash value of the message m
S_a	signature generation with the private key a
V_A	signature verification with the public key A
$Mont_M(x, y)$	Montgomery multiplication of x and y with modulus M
$\text{lcm}(x, y)$	least common multiple of x and y
$\gcd(x, y)$	greatest common divisor of x and y
p	prime number
q	prime number
n	$p \cdot q$
$\lambda(n)$	Carmichael function of n; $\lambda(n) = lcm(p-1, q-1)$
$\varphi(n)$	Euler's totient function of n; $\varphi(n) = (p-1)(q-1)$
E	elliptic curve
$\#E$	number of points on the elliptic curve E
$\mathcal{O}$	point at infinity on an elliptic curve
P	point on an elliptic curve
k	scalar
kP	point multiplication of an elliptic curve point P with a scalar k
Q	kP

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

1.1 Introduction

In the ever expanding digital world, cryptography is becoming more and more important to provide services such as encryption, digital signatures and key establishment. By the use of encryption, data confidentiality can be achieved. Digital signatures ensure non-repudiation, data integrity and authentication of the origin of information. Key establishment is a support service for many types of cryptographic algorithms. While the most efficient way for encryption is symmetric key cryptography, digital signatures and key establishment in large scale open systems require public key cryptography [55]. Besides symmetric and public key algorithms, a third class of cryptographic algorithms can be categorized as hash functions, which map messages of a variable length to values of a fixed length. In this section, symmetric key and public key cryptography are introduced. The reason is that in later chapters efficient coprocessors implementing cryptographic algorithms in these categories are presented. This section also elaborates on the security of cryptosystems, because the implemented coprocessors also contain some security measures. Since this thesis does not include any work on hash functions, they are not covered in this introduction.

1.1.1 Symmetric Key Cryptography

The first notion of symmetric key cryptography dates from thousands of years ago. Julius Caesar encrypted his secret documents by replacing each character by the character that is located three positions further in the alphabet. Although it is obvious that this encryption technique is not free of flaws, it can be used as an example to explain the basics of symmetric key cryptography. In Caesar's scheme, encrypting a message means shifting each character over a certain number

of positions in the alphabet. The decryption operation shifts each character over the same number of positions back in the alphabet. The secret key in this scheme is the number of positions over which the characters are shifted. In symmetric key cryptography, we require that the encryption and decryption keys are equal or can be derived easily from each other. This is illustrated in Fig. 1.1, where Alice encrypts a plaintext m using an encryption function E and a key k, resulting in a ciphertext $c = E_k(m)$. Bob uses the same key for decrypting the ciphertext in order to recover Alice's original message $m = D_k(c)$. An eavesdropper, called Eve in Fig. 1.1, cannot recover the plaintext from the ciphertext without knowing the secret key k. She is allowed, however, to have full knowledge of the encryption and decryption schemes E and D. This is known as Kerckhoffs' principle: "A cryptosystem should be secure, even if an adversary knows everything about the system, except for the key" [55].

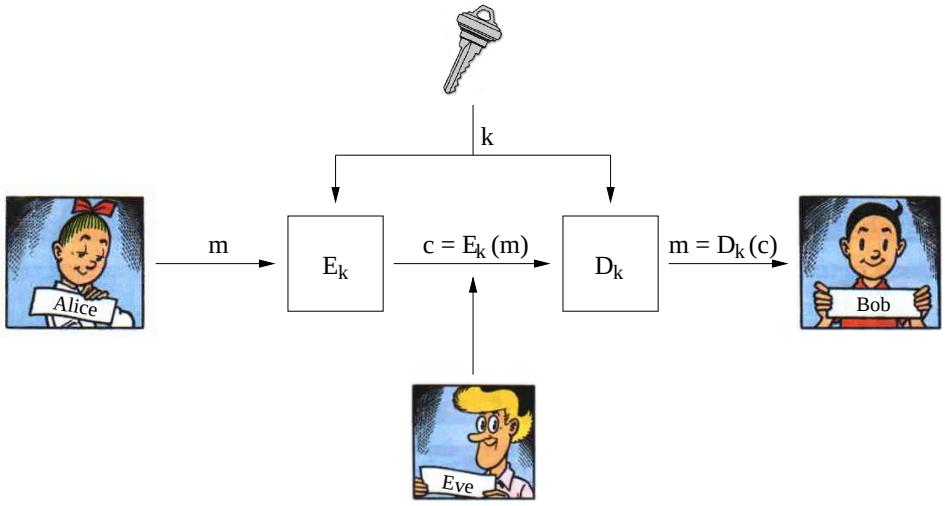

Figure 1.1: General model for symmetric key cryptography.

Whereas Caesar's cipher can easily be broken by a brute-force attack, i.e., trying all possible keys until a meaningful message is produced, some more secure and practical encryption schemes have been developed over the past decades. These schemes can be divided into block ciphers and stream ciphers. Whereas block ciphers operate on a "block" of data, stream ciphers evaluate one bit or one byte at a time. Stream ciphers also have an internal state, which is stored in a piece of memory. The difference between block ciphers and stream ciphers is shown in Fig. 1.2.

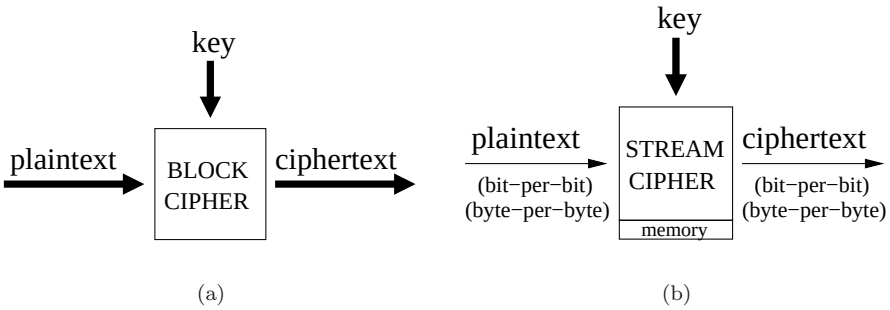

Figure 1.2: General model of a block cipher (a) and a stream cipher (b).

Block Ciphers

Two important principles in the construction of block ciphers are substitution and transposition. Substitution replaces characters or symbols by other characters or symbols, while transposition permutes the characters or symbols in a block of data. Substitution induces confusion in a cipher, i.e., it makes it hard to find a relationship between the key and the ciphertext on the one hand and the key and the plaintext on the other hand. Transposition causes diffusion, which makes sure there is no local relationship between the statistics of the symbols in the plaintext and the ciphertext.

The two most widely used block ciphers are DES and AES. The Data Encryption Standard (DES) was standardized in 1977 [59]. However, because of its 56-bit key, DES is considered to be insecure for practical applications. A 3-times cascaded version of DES, called Triple DES or TDEA, is believed to be practically secure [64].

Because the block length and the performance of Triple DES did not fulfil the requirements of future applications, an open competition for a new block cipher standard was launched by the National Institute of Standards and Technology (NIST). As a result, the Advanced Encryption Standard (AES) was announced in 2001 [62]. The AES cipher exists with a 128-, 192- and 256-bit key length.

In order for block ciphers to handle plaintexts that contain more bits than the block width, several modes of operation can be implemented. The most straightforward mode is the Electronic CodeBook (ECB) mode, in which the plaintext is divided into parts of which the number of bits is equal to the block width. Each block is fed through the block cipher using the same key. This mode of operation has several security flaws, because it does not hide data patterns. Better examples of block cipher modes are CBC, OFB, CFB and CTR modes [55]. These modes overcome the problems that arise in ECB mode. However, they only provide en-

cryption, while the importance of authenticated encryption has been acknowledged over the past years. Authenticated encryption can be achieved by modes such as CCM [63], CWC [45], GCM [65], IAPM [35], OCB [75] and XECB [25].

Stream Ciphers

Stream ciphers are used for applications where small area and/or high speed are important requirements. Examples of standardized stream ciphers are RC4, designed by Ron Rivest in 1987, A5/1 [4, 5] and E0 [11], which provide security for the Internet and wireless networks, GSM communication and the Bluetooth protocol, respectively. However, most standardized stream ciphers have been proven to be insecure over the past years. The stream cipher competition eSTREAM, organized by ECRYPT, tries to solve this problem [22].

A drawback of symmetric key cryptography is the need for a Trusted Third Party (TTP), where each user shares a different secret key with the TTP. It also requires the pre-agreement of a shared secret key, which can not always be achieved in an obvious way.

1.1.2 Public Key Cryptography

Diffie-Hellman Key Exchange

Before the invention of public key cryptography, the only way for users to agree on a secret key, was over a secured channel. This changed in 1976, when Diffie and Hellman introduced a method for secret key agreement over a public channel [21]. The simplest version of the Diffie-Hellman key exchange protocol uses a multiplicative group of integers modulo p and a generator g. Fig. 1.3 shows how the key exchange between Alice and Bob can be achieved. Both Alice and Bob have a public and a private key. The private key is an integer, which we denote by a for Alice and b for Bob. The respective public keys are equal to $A = g^a \mod p$ and $B = g^b \mod p$. After the exchange of the public keys, both Alice and Bob can compute $K = g^{ab} \mod p$, which is the shared secret key.

The security of Diffie-Hellman key exchange is based on the Computational Diffie-Hellman (CDH) assumption, which states that it is hard to compute $g^{ab} \mod p$ when p, g, $g^a \mod p$ and $g^b \mod p$ are given. The CDH assumption is related to the Discrete Logarithm Problem (DLP), which states that it is very hard to compute a when p, g and $A = g^a \mod p$ are given. However, this simple version of Diffie-Hellman key exchange does not provide authentication of the origin of information. Hence, it is vulnerable to a man-in-the-middle attack. This is illustrated in Fig. 1.4, where Eve impersonates Alice in order to agree on a shared key with Bob.

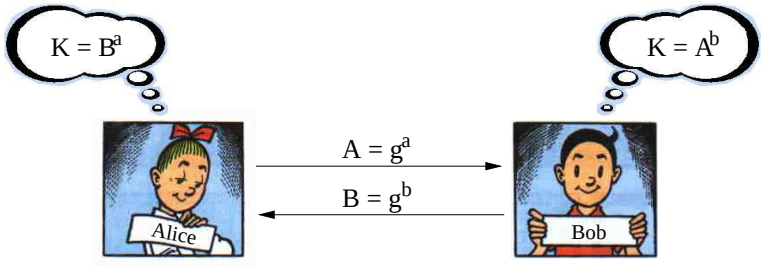

Figure 1.3: Diffie-Hellman key exchange.

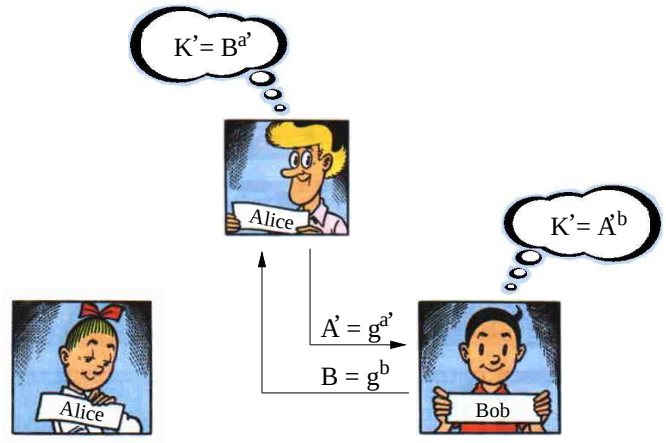

Figure 1.4: Man-in-the-middle attack on Diffie-Hellman key exchange.

RSA

Another breakthrough in public key cryptography was the invention of the RSA scheme by Rivest, Shamir and Adleman in 1978 [74]. Here, the private key of a user consists of two large primes p and q and an exponent d. The public key consists of a pair (n, e), where $n = p \cdot q$ (at least 1024 bits for security reasons) and e is such that $e = d^{-1} \mod \lambda(n)$. The corresponding p, q and d are kept secret. To encrypt a message m, the sender computes $c = m^e \mod n$ and decryption is described by $m = c^d \bmod n \equiv m^{1+k\lambda(n)} \equiv m \bmod n$. The previous equality follows from Fermat's theorem, given in Eq. (2.1), and the fact that $\lambda(n) = \mathrm{lcm}(p-1, q-1)$. The RSA function is the modular exponentiation with the public exponent e. The private exponent d is referred to as the trapdoor to invert the function.

Similar to the Diffie-Hellman protocol, the most important operation in RSA is modular exponentiation. The security of RSA, however, is not based on the discrete logarithm problem. The strength of RSA is based on the e-th root problem, which states that it is very hard to compute m when n, e and $c = m^e \mod n$ are given. Here, n needs to be hard to factor and m needs to be chosen uniformly at random in the interval $[0, n-1]$. In [12], Boneh gives an overview of attacks on the RSA cryptosystem. Because textbook RSA is insecure, the RSA algorithm requires a padding scheme in order to establish secure encryption or signing. Several standards contain padding schemes, such as the PKCS standard for RSA [76].

While the Diffie-Hellman protocol can only be used for key agreement, RSA can also provide public key encryption and digital signatures. The general model for public key encryption is shown in Fig. 1.5, where B and b are Bob's public and private key, respectively. Alice can use Bob's public key to encrypt a message. The only person who is able to decrypt the message is Bob. In a group of n users, only n key pairs are needed for public-key encryption.

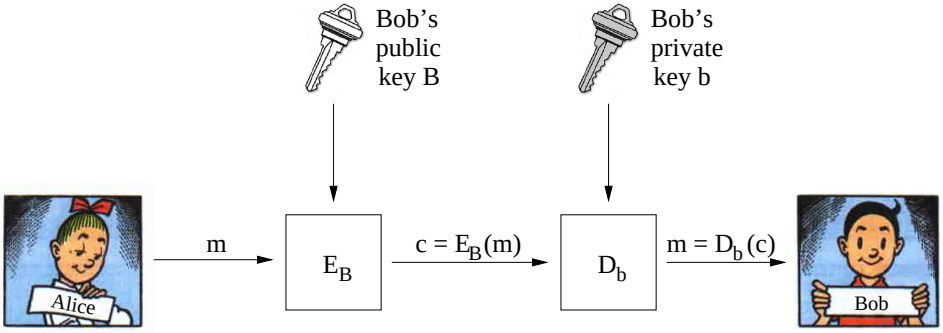

Figure 1.5: General model for public key encryption.

An example of a digital signature scheme is given in Fig. 1.6, where Alice sends a message to Bob. She signs the message using her private key and appends the signature to the message. Bob verifies the signature using Alice's public key and compares the result to the message. In practical applications, it would be too time-consuming to sign the complete message. That is why the hash value of the message, denoted by $H(m)$ in Fig. 1.6, is signed and appended to the message. Verification is done by hashing the message and comparing the hash value to the verified digital signature.

Digital Signature Algorithm

Another standardized algorithm for digital signatures is the Digital Signature Algorithm (DSA), which is described in the Digital Signature Standard (DSS) speci-

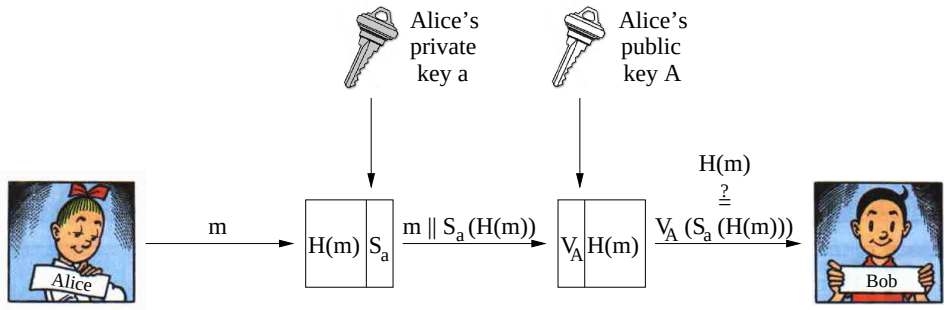

Figure 1.6: Example of a digital signature scheme, where S and V denote the signing of the message and the verification of the signature, respectively.

fication [60]. Although a new version of the standard, called FIPS 186-3, is on the way, we summarize the operations given in the established standard FIPS 186-2. The IEEE P1363 document on Standard Specifications for Public Key Cryptography [32] and Part 3 of the ISO 14888-3 standard on digital signatures [33] also describe algorithms for digital signatures. The DSA algorithm consists of three parts:

- Key generation:
  - The private key x is randomly generated, with $0 < x < q$ and q a 160-bit prime.
  - The public key consists of four parameters:
    1. q, which is also used to bound the private key;
    2. p, which is an L-bit prime, such that $512 \leq L \leq 1024$, L is divisible by 64 and q divides $p - 1$;
    3. g, which is calculated as $g = h^{\frac{p-1}{q}} \mod p > 1$ for a chosen h that satisfies $1 < h < p - 1$;
    4. y, which is equal to $y = g^x \mod p$. This public key parameter only belongs to one user, while the other parameters can be shared between a group of users.
- Signature generation:
  For every message m, a random value t is generated, with $0 < t < q$. The signature consists of two values:
  1. r, which is calculated as $r = (g^t \mod p) \mod q$;
  2. s, which is calculated as $s = (t^{-1}(H(m) + x \cdot r)) \mod q$, where $H(m)$ is a hash function applied to the message m.

- Signature verification:
  The signature is rejected if the conditions $0 < r < q$ or $0 < s < q$ are violated. If these conditions are satisfied, v is calculated in the following four steps:

  1. $w = s^{-1} \mod q$
  2. $u_1 = (H(m) \cdot w) \mod q$
  3. $u_2 = (r \cdot w) \mod q$
  4. $v = ((g^{u_1} \cdot y^{u_2}) \mod p) \mod q$

  The signature is valid if $v = r$.

The main operation in DSA is, similar to Diffie-Hellman and RSA, modular exponentiation.

Elliptic Curve Cryptography

More recent public key standards are based on Elliptic Curve Cryptography (ECC), introduced by Miller and Koblitz [57, 39]. They showed how a group structure defined on an elliptic curve can be used for cryptography. For cryptographic applications, elliptic curves are usually defined over binary extension fields, GF(2^n), or prime fields, GF(p). An introduction to these finite fields is given in Chapter 2. In this thesis, we only consider ECC over GF(p), because this allows the sharing of the data path with DSA and RSA, which is interesting when ECC as well as DSA and RSA are required in the same cryptographic implementation.

An elliptic curve over GF(p) is the set of solutions to the equation

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

with $a, b \in \mathrm{GF}(p)$. In this thesis, only non-singular curves are considered, i.e., $(4a^3 + 27b^2) \mod p \neq 0$. If (x, y) satisfies the above equation then the point $P = (x, y)$ is a point on the elliptic curve. The set of points on an elliptic curve together with the point at infinity, denoted by $\mathcal{O}$, can be seen as an additive Abelian group, with point addition as the group operation. An introduction to groups is given in Chapter 2. The addition of two points on the curve, $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, can be computed as follows (when $P_1, P_2 \neq \mathcal{O}$ and $P_1 \neq \pm P_2$):

$$\begin{aligned} P_3 &= P_1 + P_2 = (x_3, y_3) \\ \text{with } \lambda &= \frac{y_2 - y_1}{x_2 - x1} \mod p \\ x_3 &= (\lambda^2 - x_1 - x_2) \mod p \\ y_3 &= ((x_1 - x_3) \cdot \lambda - y_1) \mod p\,. \end{aligned} \tag{1.1}$$

A special case of point addition is point doubling, which can be computed as follows:

$$\begin{aligned} P_3 &= 2P_1 = (x_3, y_3) \\ \text{with } \lambda &= \frac{3 \cdot x_1^2 + a}{2 \cdot y_1} \mod p \\ x_3 &= (\lambda^2 - 2 \cdot x_1) \mod p \\ y_3 &= ((x_1 - x_3) \cdot \lambda - y_1) \mod p\,. \end{aligned} \tag{1.2}$$

To visualize these operations, Figs. 1.7 and 1.8 show the graphical representation of a point addition and a point doubling on an elliptic curve defined over $\mathbb{R}$.

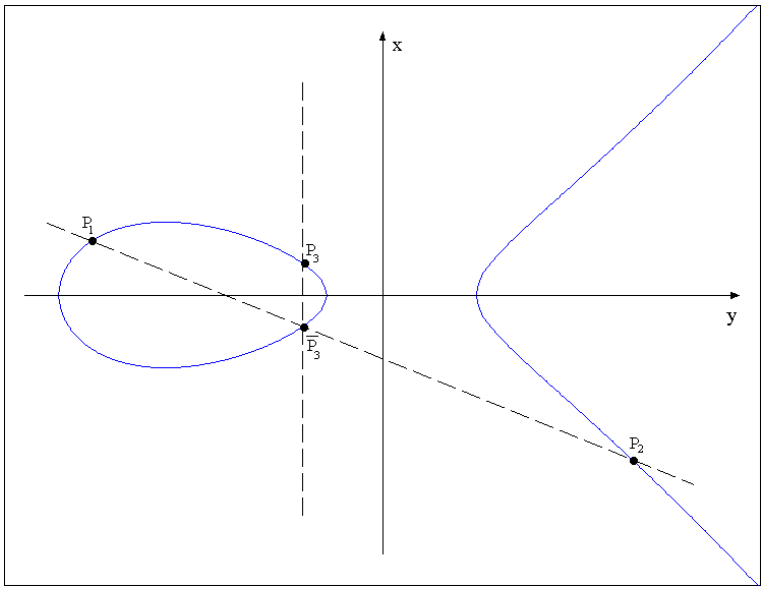

Figure 1.7: Graphical representation of the point addition of P_1 and P_2 on an elliptic curve over $\mathbb{R}$, resulting in the point P_3.

For elliptic curve based cryptosystems, the equivalent of modular exponentiation is point multiplication, which multiplies a point on an elliptic curve with a scalar, resulting again in a point on the curve. Point multiplication can be achieved by consecutive point additions and point doublings. When the point multiplication of a point P with a scalar k results in the point Q, this is denoted by $Q = kP$. The advantage of ECC over RSA, is that the security grows exponentially with the length of the parameters. This allows shorter parameters and signatures compared

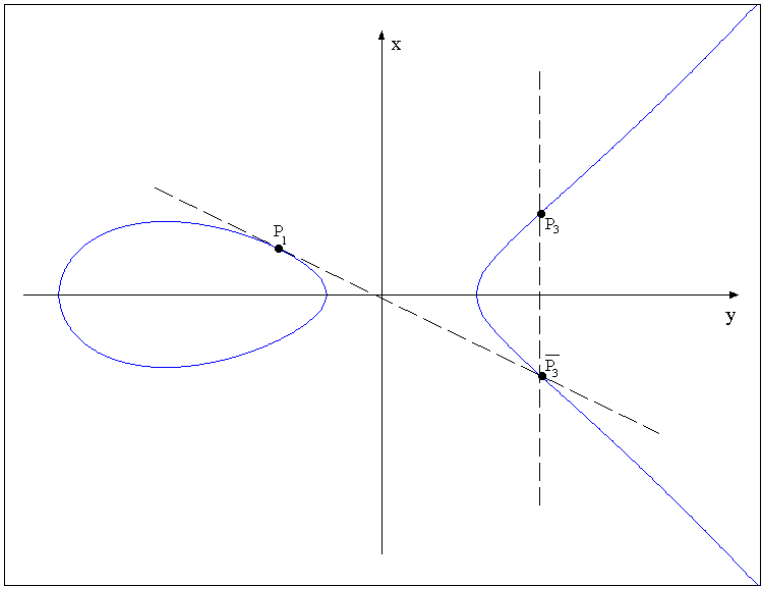

Figure 1.8: Graphical representation of the point doubling of P_1 on an elliptic curve over $\mathbb{R}$, resulting in the point P_3.

to RSA with the same level of security. The analogous versions of DSA, DLP and CDH for ECC, are the Elliptic Curve Digital Signature Algorithm (ECDSA), the Elliptic Curve Discrete Logarithm Problem (ECDLP) and the Elliptic Curve Computational Diffie-Hellman (ECCDH) assumption. Figure 1.9 shows the protocol for elliptic curve Diffie-Hellman key exchange, where k_A and k_B are the secret key of Alice and Bob, respectively. The elliptic curve parameters and the point P are publicly known. The security of this scheme is based on the ECCDH assumption, which states that it is hard to compute $k_A k_B P$ when P, $k_A P$ and $k_B P$ are given.

Similar to DSA, the ECDSA algorithm can be used for generating and verifying digital signatures:

- Key generation:
  - The private key d is randomly generated in the interval $[1, n-1]$, where n is the order of the elliptic curve point P and P is a publicly known parameter.
  - The public key is the elliptic curve point Q, with $Q = dP$.
- Signature generation:
  For every message m, a random value k is generated in the interval $[1, n-1]$,

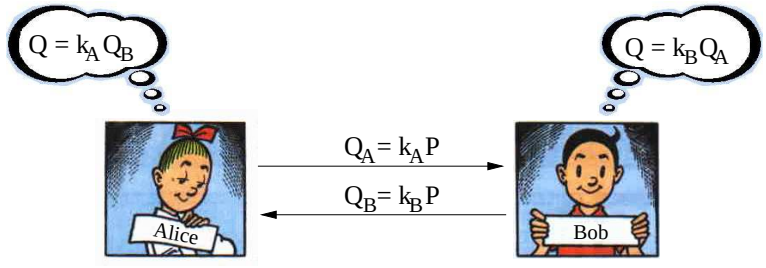

Figure 1.9: Elliptic curve Diffie-Hellman key exchange.

such that $\gcd(k, n) = 1$. The signature consists of two values:

1. r, which is calculated as $r = x \mod n$, with $kP = (x, y)$ and $r \neq 0$,
2. s, which is calculated as $s = (k^{-1}(H(m) + d \cdot r)) \mod n$, with $H(m)$ a cryptographic hash function applied to the message m.

- Signature verification:
  The signature is rejected if r and s are not in the interval $[1, n-1]$. If they are, x_v is calculated in the following four steps:

  1. $w = s^{-1} \mod n$
  2. $u_1 = (H(m) \cdot w) \mod n$
  3. $u_2 = (r \cdot w) \mod n$
  4. $(x_v, y_v) = u_1 P + u_2 Q$

  The signature is valid if $x_v = x$.

Although public key cryptography provides a broader range of services than symmetric key cryptography, the latter is much more efficient for encryption or data authentication. That is why, in most cases, authenticated encryption is achieved with symmetric key algorithms, while public key schemes are used for digital signatures and key establishment.

1.1.3 Security of Cryptosystems

Whereas the science of cryptography aims at the construction of new ciphers, cryptanalysis is the study of techniques to break these ciphers. These two research areas stimulate each other by surpassing each other step by step: once a new cipher is designed, cryptanalysts try to break it; once it is broken, cryptographers try to redesign it in order to overcome the flaws; etc.

Classical cryptanalysis focuses on weaknesses in the algorithm. The most straightforward weakness is a badly chosen key length. If the size of the key space is too small, the cipher can be broken by a brute-force attack. The two most frequently studied cryptanalytic techniques for symmetric key cryptography are linear cryptanalysis, which tries to find a linear approximation of the behavior of an algorithm [51, 52], and differential cryptanalysis, which exploits the relationship between differences in the input and subsequent differences in the output of a cipher [9]. For the cryptanalysis of public key cryptography, there exist several algorithms based on number theory.

More recently, a new class of cryptanalytic attacks has been introduced, called implementation attacks. In this case, the attacker does not focus on flaws in the algorithm, but tries to break the system by exploiting weaknesses in the implementation of the algorithm. Implementation attacks can be performed in an invasive or a non-invasive way. In the former case, the attacker has unlimited access to the cryptographic device. In the latter case, the attacker retrieves information without interfering with the normal functioning of the device.

An important class of attacks that can be categorized as non-invasive, are side channel attacks. Side channel attacks impose a new model on cryptosystems. An attacker is no longer limited to using plaintext and/or ciphertext information. Side channels such as power consumption, timing information, electromagnetic emanation, etc. can be used to extract sensitive information. This is illustrated in Fig. 1.10.

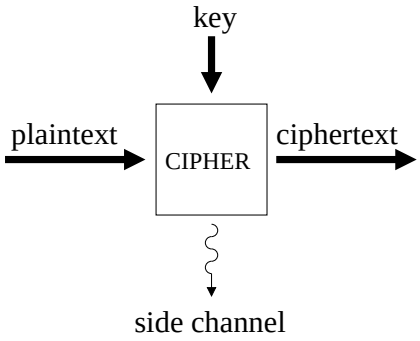

Figure 1.10: General model of a cryptosystem in the presence of a side channel.

The first official information on side channel attacks dates from 1956. In [99], Peter Wright describes how he helped the British secret services to break a rotor machine by listening to the clicking sound with a microphone. In the mid 1980s there was a lot of commotion about the electromagnetic emanation of video screens [94]. In 1996, Paul Kocher described how timing information can

be exploited as a side channel [42]. He also introduced the first attacks based on the power consumption of a cryptosystem [43]. In 2001, the first results on the analysis of the electromagnetic radiation of modern cryptographic devices were reported [72, 23]. However, measurements of electromagnetic fields have been performed since the 1950s for military purposes. This research has led to a never published set of standards for reducing the electromagnetic radiation of electronic devices. TEMPEST is the codeword that the American government used for these standards.

There are two main flows in recent research on side channel attacks. On the one hand, advanced analysis and processing techniques are developed to enhance side channel analysis attacks and in particular power analysis attacks. On the other hand, new countermeasures are implemented at all levels of design abstraction. Here, the trade-off between performance and side channel resistance is the key issue. The levels of design abstraction are depicted in Fig. 1.11. Practical examples show that the lower the level on which the countermeasure is implemented, the more effective it is. However, the degradation in area and speed also increases when we descend in the levels of design abstraction [43, 2, 18, 92].

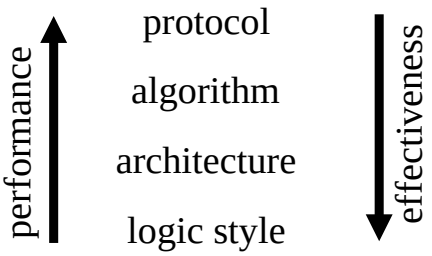

Figure 1.11: Behavior of the effectiveness of a countermeasure and the performance of the system for the adoption of countermeasures at different levels of design abstraction.

1.2 Motivation

Security in embedded systems requires the choice of a suitable implementation platform. For some systems a general purpose microprocessor suffices the requirements, but when high performance is the main criterium, cryptographic coprocessors in hardware are indispensable. Examples of high performance applications are ATMs and trusted computing platforms. When very high performance is required or when a high volume of coprocessors is needed, ASICs are chosen as implementation platforms. In this case, the reconfigurability of FPGAs is only

used for prototyping. However, thanks to the efforts of FPGA manufacturing companies, the performance gap between ASICs and FPGAs becomes smaller and smaller. The progress in the performance and capabilities of FPGAs is demonstrated in Table 1.1, which lines up the evolution of available resources in Xilinx products [100]. From the table, it can be derived that FPGAs have become heterogeneous systems with a variety of dedicated resources. This explains the trend that FPGAs are more and more used in end products such as routers and banking applications. Following this trend, the need for specific FPGA architectures can be justified. This work focuses on cryptographic coprocessor design, optimized for FPGAs.

In this thesis, two features of the Virtex-II Pro FPGA are exploited, i.e., the multiplier blocks and the RAM blocks. These blocks were added to Xilinx FPGAs in 2000 in the Virtex-II FPGA [106]. The Virtex-II Pro FPGA is an improved version of the Virtex-II FPGA. Figure 1.12 shows the floor plan of a Virtex-II Pro FPGA.

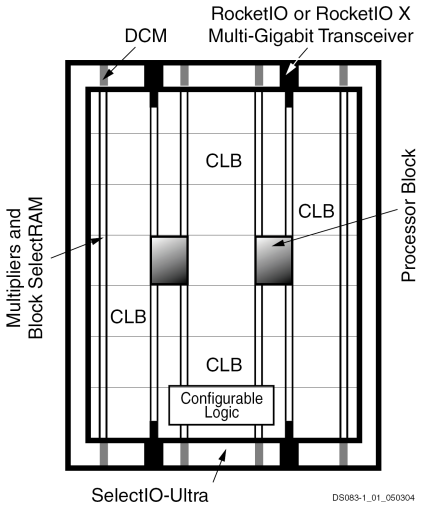

Figure 1.12: Floor plan of a Virtex-II Pro FPGA [105].

1.3 Contribution

In public key cryptography, the most important operations are finite field arithmetic operations. For RSA, DSA and Diffie-Hellman key exchange, the main operation is modular exponentiation, which can be decomposed into many mod-

Table 1.1: Evolution of the available resources on Xilinx products over the last decades, where CLBs denote Configurable Logic Blocks.

year	product	architecture	resources	technology (nm)
1991	XC4000 [108]	CPLD	64-8464 CLBs	250
1998	Virtex [107]	FPGA	384-6144 CLBs 32768-131072 Kb block RAM	220
1998	Spartan [102]	FPGA	100-784 CLBs	350
2002	Virtex-II Pro [105]	FPGA	5632-176384 CLBs 216-7992 Kb block RAM 4-20 multi-Gigabit transceivers 0-2 IBM PowerPC processors 12-444 multipliers 4-12 digital clock managers	130
2003	Spartan-3 [101]	FPGA	192-8320 CLBs 72-1872 Kb block RAM 4-104 multipliers 2-4 digital clock managers	90
2004	Virtex-4 [103]	FPGA	1536-22272 CLBs 864-9936 Kb block RAM 8-24 multi-Gigabit transceivers 1-2 IBM PowerPC processors 32-192 DSP slices 4-20 digital clock managers	90
2006	Virtex-5 [104]	FPGA	2400-25920 CLBs 1152-11664 Kb block RAM 8-24 multi-Gigabit transceivers 32-640 DSP slices 4-12 digital clock managers 2-6 PLL clock generators	65

ular squarings and modular multiplications. Although modular squaring can be performed more efficiently than modular multiplication, both operations are often sharing the same data path, i.e., a modular multiplier.

For elliptic curve based cryptosystems, point multiplication consists of subsequent point doublings and point additions/subtractions. These point operations consist of many finite field inversions, finite field multiplications and finite field additions/subtractions. Although ECC is standardized in two types of underlying fields, GF(2^n) and GF(p), we only take into account the latter, because this allows the sharing of the data path with DSA and RSA. By choosing an alternative representation, called projective representation, for the coordinates of the points, the time-hungry finite field inversions can be eliminated almost completely. This leaves modular multiplication to be the most critical operation in ECC.

The previous paragraphs show that the performance of the modular multiplier determines the overall performance of the public key coprocessor. An efficient way to perform modular multiplication is Montgomery multiplication [58]. That is why the first part of this thesis examines implementation trade-offs for Montgomery multipliers on FPGAs. By using the dedicated integer multipliers on a Virtex-II Pro FPGA in an optimal way, this work achieves a substantial speed-up compared to the state-of-the-art in Montgomery multipliers on FPGAs.

Although the performance of an embedded cryptosystem can already be improved by only moving the modular multiplication into the coprocessor, some systems require a programmable coprocessor that can also handle higher-level instructions such as modular exponentiation and point multiplication. In this case, hardwired control logic has to be added to the modular multiplier. Because of the complexity of some public key algorithms, the cost of this control logic is substantial. The situation gets worse when full programmability for ECC is required, i.e., when not only point multiplication, but also the underlying operations of which point multiplication consists need to be accessible to the user. This comes down to the implementation of a complex instruction hierarchy. To provide countermeasures against side channel attacks at the algorithmic level, the instruction hierarchy becomes even more complex. This work tries to cope with the complexity of the control logic, by dividing it into many levels, where each level can be addressed directly by the user of the coprocessor, but also by the higher-level functions. The public key algorithms are optimized for FPGA implementation starting from the assumption that there is practically no limitation in block RAM. Comparison to the state-of-the-art in public key coprocessors shows that this work outperforms all previously designed implementations on FPGAs.

In symmetric key coprocessors, the non-linear functions or S-boxes are the most critical building blocks. For FPGA design, the optimal choice is to implement these functions as look-up tables in the block RAM. However, the structure of the S-box may allow for very compact implementation without RAM, hence it is worth taking into account other implementation options. This work explores the

different implementation options and shows that there is room for improvement in previously reported designs. We point out an optimal implementation option for the AES S-box in a subset of all options.

1.4 Organization of the thesis

The outline of this thesis is as follows:

- **Chapter 1** introduces and motivates the work carried out within the framework of this thesis.
- **Chapter 2** provides background information on two basic notions on which the other chapters are based. The section on finite fields elaborates on both prime fields and binary extension fields, because these are used in Chapters 3 and 4 and Chapter 5, respectively. The section on power analysis attacks explains the principles on which the attacks are based and introduces some countermeasures. This section forms the basis for the countermeasures introduced in Chapter 4.
- **Chapter 3** explains Montgomery's method for modular multiplication. It discusses the parallelization of two algorithms for Montgomery multiplication and investigates the implementation options for high-speed Montgomery multipliers on FPGAs. The chapter ends with the comparison of our solution to previously designed Montgomery multipliers.
- **Chapter 4** starts with an overview of public key algorithms for digital signatures, including countermeasures at the algorithmic level. An instruction hierarchy for the efficient implementation of programmable public key coprocessors on FPGAs is presented and the implementation results are compared to the state-of-the-art in public key coprocessors on FPGAs.
- **Chapter 5** moves to symmetric key cryptography with the focus on AES. An evaluation of compact S-box implementations is presented based on the composite field approach and implementation results are compared to previous work.
- **Chapter 6** concludes the thesis and points out some open problems.

Chapter 2

Finite Field Arithmetic and Side Channel Analysis Attacks

In this chapter, we elaborate on some general notions on which the chapters in this thesis are based, i.e., finite field arithmetic and side channel analysis attacks.

Section 2.1 gives some background information on finite field arithmetic. This section forms the basis for Chapters 3, 4 and 5. Because Chapters 3 and 4 focus on Montgomery multiplication and public key cryptography over GF(p), respectively, Sect. 2.1.1 gives an introduction to prime field arithmetic. The area optimization of compact S-box implementations for AES, discussed in Chapter 5, uses a composite field approach. The composite fields that are used in this case are based on binary extension field arithmetic. That is why Sect. 2.1.2 elaborates on binary extension fields, denoted by GF(2^n). Section 2.1 starts with a general introduction to finite fields, which is applicable to both GF(p) and GF(2^n) arithmetic. The definitions and properties given in this section, are based on the book of Lidl and Niederreiter [48].

The background information given in Sect. 2.2 is on side channel analysis attacks and more specifically on power analysis attacks. Sect. 2.2.1 elaborates on the origin of data dependent power consumption differences, which form the basis of power analysis attacks. Sect. 2.2.2 gives an overview of some countermeasures against power analysis attacks described in literature. This forms the basis for Chapter 4, in which countermeasures at the algorithmic level are presented and implemented in our programmable public key coprocessor.

2.1 Finite Fields

Before giving the definition of a field, we elaborate on groups. A group is a set of elements (G) in combination with an operation, denoted by $\diamond$. $< G, \diamond >$ is a group iff (if and only if) the following requirements are satisfied:

- The result of the operation $\diamond$ applied on two elements in G is also an element in G:
$$\forall g_1, g_2 \in G : g_1 \diamond g_2 = g_3, \text{ with } g_3 \in G\,.$$
- The operation $\diamond$ is associative:
$$\forall g_1, g_2, g_3 \in G : g_1 \diamond (g_2 \diamond g_3) = (g_1 \diamond g_2) \diamond g_3\,.$$
- There exists an identity element, denoted by e:
$$\forall g_1 \in G : g_1 \diamond e = e \diamond g_1 = g_1\,.$$
- For each element $g_1 \in G$ there exists an inverse, denoted by g_2:
$$\forall g_1 \in G : g_1 \diamond g_2 = g_2 \diamond g_1 = e\,.$$

A group is Abelian iff the following property is satisfied:

- The operation $\diamond$ is commutative:
$$\forall g_1, g_2 \in G : g_1 \diamond g_2 = g_2 \diamond g_1\,.$$

When the operation $\diamond$ is written as $+$, the inverse of g_1 is denoted by $-g_1$ and 0 is used as the identity element. In this case, the group is said to be an additive group. When $\cdot$ is used as group operation, g_1^{-1} stands for the inverse of g_1 and the identity element is denoted by 1. The group is then said to be a multiplicative group.

While groups only provide one operation, public key cryptography is usually performed on elements of a field. A field is a set of elements (F) in combination with two operations ($+$, $\cdot$). $< F, +, \cdot >$ is a field iff the following requirements are satisfied:

- $< F, + >$ is an additive Abelian group.
- $< F \backslash \{0\}, \cdot >$ is a multiplicative Abelian group.
- The operation $\cdot$ is distributive over the operation $+$:
$$\forall f_1, f_2, f_3 \in F : f_1 \cdot (f_2 + f_3) = f_1 \cdot f_2 + f_1 \cdot f_3\,.$$

If a field has a finite number of elements, it is called a finite field or Galois field, denoted by GF. The number of elements in a finite field, also called the order of the field, is always a power of a prime. For cryptographic applications, two types of fields are mainly used:

- fields of which the elements are integers and the order is a prime, denoted by GF(p);
- fields of which the elements can be represented as n-bit vectors and the order is a power of two, denoted by GF(2^n).

These types of fields are discussed in the remainder of this section.

2.1.1 Arithmetic over GF(p)

In GF(p), an addition, subtraction or multiplication is performed modulo p, i.e., it can be computed as an integer addition, subtraction or multiplication followed by a reduction modulo p. For addition in GF(p), this comes down to an integer addition followed by a subtraction of p if the result of the addition is greater than or equal to p. A subtraction in GF(p) is computed by an integer subtraction followed by an addition of p if the result of the subtraction is less than zero. The following two examples of a modular addition and a modular subtraction are computed in GF(7):

$$\begin{aligned}(5+6) \mod 7 &= 11 \mod 7 = 4\\ (5-6) \mod 7 &= -1 \mod 7 = 6\,.\end{aligned}$$

For multiplication in GF(p), an integer multiplication is followed by a reduction step, which computes the remainder of the division of the product by p. The result is always in the range $[0, p-1]$. An example of a modular multiplication in GF(7) is computed as follows:

$$\begin{aligned}&(5\cdot 6) \mod 7 = 30 \mod 7\\ &\lfloor 30/7 \rfloor = 4\\ &30 - 4\cdot 7 = 2\\ &\Rightarrow (5\cdot 6) \mod 7 = 2\,.\end{aligned}$$

It is clear that addition in GF(p) is computationally much easier than multiplication in GF(p). That is why many efforts have been made to optimize multiplication algorithms in GF(p). Chapter 3 elaborates on an optimized algorithm for modular multiplication in GF(p).

Besides modular addition, subtraction and multiplication, public key algorithms also use modular inversions. An option for the computation of a modular inversion, is the use of Fermat's theorem, which states that

$$a^{p-1} \equiv 1 \mod p\,, \tag{2.1}$$

when $\gcd(a, p) = 1$. For the elements in GF$(p)\backslash\{0\}$ the requirement $\gcd(a, p) = 1$ is satisfied, so Fermat's theorem is applicable to the field GF(p). Using Fermat's theorem, an expression for the inverse of a field element can be derived:

$$a^{-1} \equiv a^{p-2} \mod p\,. \tag{2.2}$$

This shows that the inverse of an element in GF(p) can be computed by a modular exponentiation. Chapter 4 shows how this can be decomposed into modular squarings and multiplications.

Another way to invert an element in GF(p) is based on the Euclidean algorithm for the computation of the gcd of two integers. Algorithm 1 shows how the gcd of a and b can be computed by repeatedly reducing the largest of the two integers modulo the smallest of the two integers. By extending this algorithm to Alg. 2, not only the gcd, but also x and y can be determined such that $\gcd(a, b) = a \cdot x + b \cdot y$. When applying the extended Euclidean algorithm to a and p when p is a prime, the inverse of a can be computed as follows:

$$\begin{aligned} \gcd(a, p) &= a \cdot x + p \cdot y \\ 1 &= a \cdot x + p \cdot y \\ a^{-1} &= x \mod p\,. \end{aligned}$$

The advantage of the extended Euclidean algorithm over Fermat's theorem is that it is more efficient on most implementation platforms. For hardware implementation, the disadvantage of the extended Euclidean algorithm is the need for specialized hardware, while Fermat's theorem only needs a subtractor and a finite field multiplier. Because the public key algorithms presented in Chapter 1 do not contain many finite field inversions, we opt for Fermat's theorem in our public key coprocessor described in Chapter 4.

Algorithm 1 Euclidean algorithm for the computation of the gcd of two integers

Require: integers $a, b \geq 0$ with $a \geq b$
Ensure: $\gcd(a, b)$
1: **while** $b \neq 0$ **do**
2: $\quad c \leftarrow a \mod b$
3: $\quad a \leftarrow b$
4: $\quad b \leftarrow c$
5: **end while**
6: Return a

2.1.2 Arithmetic over GF(2^n)

The field GF(2^n) is an extension of GF(2), i.e., each element in GF(2^n) can be represented by n elements in GF(2). Although there are several options for the

Algorithm 2 Extended Euclidean algorithm for integers

Require: integers $a, b \geq 0$ with $a \geq b$
Ensure: $c = \gcd(a, b)$, x, y such that $\gcd(a, b) = a \cdot x + b \cdot y$

1: **if** $b = 0$ **then**
2: $c \leftarrow a$
3: $x \leftarrow 1$
4: $y \leftarrow 0$
5: Return c, x, y
6: **end if**
7: $x_2 \leftarrow 1$
8: $x_1 \leftarrow 0$
9: $y_2 \leftarrow 0$
10: $y_1 \leftarrow 1$
11: **while** $b > 0$ **do**
12: $r \leftarrow \lfloor \frac{a}{b} \rfloor$
13: $s \leftarrow a - r \cdot b$
14: $x \leftarrow x_2 - r \cdot x_1$
15: $y \leftarrow y_2 - r \cdot y_1$
16: $a \leftarrow b$
17: $b \leftarrow s$
18: $x_2 \leftarrow x_1$
19: $x_1 \leftarrow x$
20: $y_2 \leftarrow y_1$
21: $y_1 \leftarrow y$
22: **end while**
23: $c \leftarrow a$
24: $x \leftarrow x_2$
25: $y \leftarrow y_2$
26: Return c, x, y

representation of these elements, we stick to one representation throughout this thesis, i.e., the polynomial representation. In this case, the elements can be represented as polynomials of degree $n-1$ with coefficients in GF(2). Addition can be performed by adding the corresponding coefficients in GF(2). The following example illustrates addition in GF(2^7):

$$\begin{aligned}&(x^6+x^5+x+1)+(x^6+x^4+x^2+x)\\&=(1+1)x^6+x^5+x^4+x^2+(1+1)x+1\\&=x^5+x^4+x^2+1\,.\end{aligned}$$

Because addition does not result in a polynomial with a degree larger than $n-1$, the result does not need to be reduced. Multiplication, however, results in a polynomial with degree up to $2\cdot(n-1)$. Reduction is done modulo a polynomial of degree n. In order for an extension of GF(2) to be a field, this polynomial should be irreducible, which means that it should be impossible to write it as a product of polynomials with a degree less than n. An irreducible polynomial of degree n can be written as

$$P(x)=x^n+p_{n-1}x^{n-1}+\ldots+p_1x+p_0\,, \tag{2.3}$$

with $\forall i : p_i \in GF(2)$. A root α of the polynomial satisfies the following equations:

$$\begin{aligned}&\alpha^n+p_{n-1}\alpha^{n-1}+\ldots+p_1\alpha+p_0=0\\&\Rightarrow \alpha^n=p_{n-1}\alpha^{n-1}+\ldots+p_1\alpha+p_0\,.\end{aligned}$$

As a consequence, reduction modulo $P(x)$ can be done by replacing x^n with $p_{n-1}x^{n-1}+\ldots+p_1x+p_0$. The following example illustrates multiplication in GF(2^7) with $P(x)=x^7+x+1$:

$$\begin{aligned}&(x^6+x^5+x+1)\cdot(x^6+x^4+x^2+x)\\&=x^{12}+x^{11}+x^{10}+x^9+x^8+x^7+x^5+x^4+x^3+x\\&=(x^5+x^4+x^3+x^2+x+1)\cdot(x^7+x+1)+(x^6+x^5+x^4+x^3+x+1)\\&=x^6+x^5+x^4+x^3+x+1\,.\end{aligned}$$

Because the order of the field GF(2^n) is equal to 2^n, Fermat's theorem can be written as follows:

$$A(x)^{2^n-1}\equiv 1 \mod P(x)\,,$$

with $A(x)=a_{n-1}x^{n-1}+\ldots+a_1x+a_0$, $\forall a_i : a_i \in GF(2)$. From this, we can derive an equation for the inverse of $A(x)$:

$$A(x)^{-1}=A(x)^{2^n-2} \mod P(x)\,.$$

Equivalent to the Euclidean algorithm and the extended Euclidean algorithm for integers, Algs. 3 and 4 compute the gcd of two polynomials in $\mathbb{Z}_p[x]$, i.e.,

polynomials where the coefficients are integers modulo p. The latter algorithm also determines $s(x)$ and $t(x)$ such that the gcd of $a(x)$ and $b(x)$ is equal to $\gcd(a(x), b(x)) = a(x) \cdot s(x) + b(x) \cdot t(x)$. Another efficient method for the computation of the inverse of an element in $\mathrm{GF}(2^n)$ is based on composite field arithmetic. This approach is discussed in Chapter 5.

Algorithm 3 Euclidean algorithm for the computation of the gcd of two polynomials in $\mathbb{Z}_p[x]$

Require: polynomials $a(x), b(x) \in \mathbb{Z}_p[x]$
Ensure: $\gcd(a(x), b(x))$
1: **while** $b(x) \neq 0$ **do**
2: $c(x) \leftarrow a(x) \mod b(x)$
3: $a(x) \leftarrow b(x)$
4: $b(x) \leftarrow c(x)$
5: **end while**
6: Return a

2.2 Power Analysis Attacks

Although there are many types of side channels that can be exploited to attack a cryptosystem, this section only focuses on information leakage through the power consumption of a device. Section 2.2.1 elaborates on the power consumption of a CMOS gate, while Sect. 2.2.2 gives some examples of countermeasures against power analysis attacks.

2.2.1 The Origin of Data Dependent Differences in Power Consumption

To explain how the power consumption of a device depends on the processed data, we focus on the power consumption of a CMOS gate, which consists of three terms:

$$P_{tot}(t) = P_{sw}(t) + P_{lk}(t) + P_{sc}(t)\,. \tag{2.4}$$

Here, $P_{tot}(t)$ is the total instantaneous power consumption of the gate, $P_{sw}(t)$ is the power consumption caused by a switching event, $P_{lk}(t)$ is the power consumption due to leakage currents in the gate and $P_{sc}(t)$ is the short-circuit power consumption. The next paragraphs elaborate on these three terms, keeping in mind that each term in the total the power consumption is equal to

$$P_x(t) = V_{dd} \cdot i_x(t)\,,$$

Algorithm 4 Extended Euclidean algorithm for polynomials in $\mathbb{Z}_p[x]$

Require: polynomials $a(x), b(x) \in \mathbb{Z}_p[x]$
Ensure: $c(x) = \gcd(a(x), b(x))$, $s(x)$, $t(x)$ such that $\gcd(a(x), b(x)) = a(x) \cdot s(x) + b(x) \cdot t(x)$

1: **if** $b(x) = 0$ **then**
2: $c(x) \leftarrow a(x)$
3: $s(x) \leftarrow 1$
4: $t(x) \leftarrow 0$
5: Return $c(x)$, $s(x)$, $t(x)$
6: **end if**
7: $s_2(x) \leftarrow 1$
8: $s_1(x) \leftarrow 0$
9: $t_2(x) \leftarrow 0$
10: $t_1(x) \leftarrow 1$
11: **while** $b(x) \neq 0$ **do**
12: $u(x) \leftarrow a(x)$ div $b(x)$
13: $v(x) \leftarrow a(x) - u(x) \cdot b(x)$
14: $s(x) \leftarrow s_2(x) - u(x) \cdot s_1(x)$
15: $t(x) \leftarrow t_2(x) - u(x) \cdot t_1(x)$
16: $a(x) \leftarrow b(x)$
17: $b(x) \leftarrow v(x)$
18: $s_2(x) \leftarrow s_1(x)$
19: $s_1(x) \leftarrow s(x)$
20: $t_2(x) \leftarrow t_1(x)$
21: $t_1(x) \leftarrow t$
22: **end while**
23: $c(x) \leftarrow a(x)$
24: $s(x) \leftarrow s_2(x)$
25: $t(x) \leftarrow t_2(x)$
26: Return $c(x)$, $s(x)$, $t(x)$

where x denotes sw, lk or sc, V_{dd} is the supply voltage and $i_x(t)$ is the instantaneous current drawn from the power supply line as a consequence of a switching event, a leakage or a short-circuit phenomenon, respectively.

Switching Power

To illustrate the presence of switching power in a gate, we start from the behaviour of a CMOS invertor. There are four possible transitions to be evaluated:

1. When the input changes from a logical 1 to a logical 0, the output makes a transition from a logical 0 to a logical 1. This means that the output capacitor needs to be loaded, which causes a switching current to be extracted from the power supply line.

2. When the opposite transition occurs, the output of the invertor switches from a logical 1 to a logical 0. In the ideal case, no current is extracted from the power supply during this transition. The only current that flows is the current to discharge the output capacitor.

3. When a logical 0 is maintained at the input of the invertor, the output remains 1 and no switching current flows.

4. When the input level of the invertor stays at a logical 1, the output stays at a logical 0 and no switching current is extracted from the power supply line.

This is summarized in Fig. 2.1, which depicts the four possible transitions at the output of an invertor.

Leakage Power

Leakage power consumption in a gate occurs in every transistor. It consists of two parts: subthreshold leakage and gate leakage. This is illustrated for an nMOS transistor in Fig. 2.2. Subthreshold leakage is caused by a subthreshold current that increases when the threshold voltage drops. Gate leakage is caused by a current flowing into the gate of the transistor. This current increases with a decreasing gate oxide thickness [81]. A dropping threshold voltage and a decreasing gate oxide thickness occur whenever a transition to a new generation of CMOS technology is made. Whereas the gate leakage problem can be solved with high-κ metal gates, the subthreshold leakage of a transistor imposes a much bigger challenge on future generations of CMOS technology.

Short-circuit Power

Whenever a switching event occurs at the input of a CMOS invertor, the nMOS and pMOS transistors go from on to off or vice versa. However, because CMOS

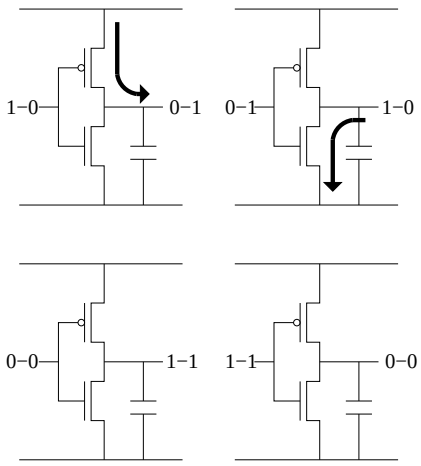

Figure 2.1: Switching current caused by the four possible transitions at the output of an invertor.

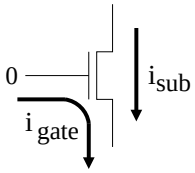

Figure 2.2: Illustration of subthreshold leakage current ($i_{sub}(t)$) and gate leakage current ($i_{gate}(t)$) in an nMOS transistor.

transistors are not perfect switches, there exists a range of input voltages for which both the nMOS and the pMOS transistor conduct a current. When both transistors are (partially) on, a current flows from the power supply line to the ground line. This current causes power dissipation in the invertor and is called short-circuit current.

Power analysis attacks are based on the correlation between the processed data and the power consumption of the device. For older CMOS technologies, the dominant term in Eq. (2.4) is $P_{sw}(t)$. This means that the data-power correlation, which is exploited in the attack, is based on the transitions of the gates in the device. For technologies with a minimal gate length of 90 nm and lower, $P_{lk}(t)$ becomes the dominant term in Eq. (2.4). This means that not only the transitions, but also the logic levels of the signals in a device should be taken into account.

2.2.2 Countermeasures Against Power Analysis Attacks

The first step in preventing side channel analysis attacks, in particular power analysis attacks, is implementing countermeasures against Simple Power Analysis (SPA). Using an SPA attack, an adversary tries to extract secret information by analyzing a single power trace of the implementation while it is executing an operation. The most important action in securing an implementation against SPA attacks, is making the power graphs look indistinguishable, even if different secret information is processed. This includes removing conditional branches.

To prevent first-order Differential Power Analysis (DPA) attacks [43, 44], which use many power consumption traces together with some statistical analysis, more advanced countermeasures are needed. As already mentioned in Chapter 1, countermeasures against side channel analysis attacks can be implemented at different levels of design abstraction. Figure 1.11 shows these levels: protocol, algorithm, architecture and logic style. Countermeasures at the protocol level lead to the smallest degradation in performance, but are also the least effective. Countermeasures at the logic style level are the most effective, but lead to the largest degradation in performance. For first-order DPA resistance, countermeasures at the algorithmic level meet the requirements.

Countermeasures at the algorithmic level are not strong enough to prevent more advanced attacks, such as higher-order DPA attacks [56] and template attacks [15]. That is why some systems require countermeasures at the logic level. An example of a countermeasure at the logic level is Sense-Amplifier Based Logic (SABL), introduced by Tiri *et al.* in [91]. This is a non-standard logic style which uses dual-rail differential logic in combination with precharge logic to make sure a switching event occurs on each signal in every clock cycle. Tiri and Verbauwhede also developed another logic style for securing cryptographic implementations: Wave Dynamic Differential Logic (WDDL) [93]. The advantage of WDDL over SABL is that it can can be built upon a standard cell library. Another example of a countermeasure at the logic level, is Masked Dual-rail Precharge Logic (MDPL) [70]. It was proposed by Popp *et al.*, who claim that MDPL has no routing constraints as opposed to SABL and WDDL.

In this thesis, we restrict ourselves to countermeasures against power analysis attacks at the algorithmic level. The algorithms used to prevent first-order DPA attacks are given in Chapter 4.

Chapter 3

Efficient Montgomery Multiplication in Prime Fields on FPGAs

In public key cryptography, modular arithmetic is of crucial importance for the performance of the system. To boost the performance, modular operations are often implemented in a hardware coprocessor. Two types of fields are used for public key algorithms: GF(p) and GF(2^n). Whereas GF(2^n) is mainly used for elliptic curve cryptography, GF(p) can be employed in many public key algorithms. That is why we focus on the implementation of GF(p) operations. Amongst these operations, modular multiplication is the most important one. It is often implemented using Montgomery's algorithm. This chapter starts with introducing Montgomery multiplication over GF(p) in Sect. 3.1. Next, Sect. 3.2 elaborates on two established implementation options when only one w-bit integer multiplier and adder are available. Because FPGAs provide many more dedicated multiplier blocks, these algorithms are extended to parallel versions. In Sects. 3.3 and 3.4, the implementation of these parallelized algorithms is discussed and implementation results are given. A short overview of previously designed Montgomery multipliers over GF(p) on FPGAs is given in Sect. 3.5 together with a comparison of their implementation results to our fastest solution. Finally, Sect. 3.6 concludes this chapter.

3.1 Montgomery Multiplication

Montgomery multiplication is an efficient method to perform modular multiplication. It was introduced by Montgomery in 1985 [58]. The Montgomery algorithm

computes $(X \cdot Y \cdot R^{-1}) \mod M$, where X and Y are the operands, M is the modulus and R is a power of two. Whereas computing $(X \cdot Y) \mod M$ requires a trial division, Montgomery multiplication only needs a division by a power of two, which comes for free in hardware.

Before performing a modular multiplication using the Montgomery algorithm, the operands need to be transformed into Montgomery representation. The Montgomery representation of an integer X, denoted by X_{Mont}, can be computed by performing a Montgomery multiplication on X and R^2, denoted by $Mont_M(X, R^2)$, resulting in $X_{Mont} = Mont_M(X, R^2) = (X \cdot R^2 \cdot R^{-1}) \mod M = (X \cdot R) \mod M$. After computing the Montgomery multiplication of two operands in Montgomery representation, the result is also in Montgomery representation and can be converted back by multiplication with R^{-1}, which comes down to Montgomery multiplication with 1. This can be illustrated as follows:

- Conversion of X into Montgomery representation:

$$\begin{aligned} X_{Mont} &= Mont_M(X, R^2) \\ &= (X \cdot R^2 \cdot R^{-1}) \mod M \\ &= (X \cdot R) \mod M\,. \end{aligned}$$

- Conversion of Y into Montgomery representation:

$$\begin{aligned} Y_{Mont} &= Mont_M(Y, R^2) \\ &= (Y \cdot R^2 \cdot R^{-1}) \mod M \\ &= (Y \cdot R) \mod M\,. \end{aligned}$$

- Computation of the result in Montgomery representation:

$$\begin{aligned} T_{Mont} &= Mont_M(X_{Mont}, Y_{Mont}) \\ &= (X_{Mont} \cdot Y_{Mont} \cdot R^{-1}) \mod M \\ &= (X \cdot R \cdot Y \cdot R \cdot R^{-1}) \mod M \\ &= (X \cdot Y \cdot R) \mod M\,. \end{aligned}$$

- Conversion of T_{Mont} into T:

$$\begin{aligned} T &= Mont_M(T_{Mont}, 1) \\ &= (X \cdot Y \cdot R \cdot 1 \cdot R^{-1}) \mod M \\ &= (X \cdot Y) \mod M\,. \end{aligned}$$

This means that four Montgomery multiplications are needed for one modular multiplication. That is why the use of Montgomery multiplication is only interesting when many consecutive modular multiplications need to be performed. In this

case, the Montgomery representation can be maintained for intermediate results. Conversion is only needed before the first and after the last modular multiplication. Note that a single modular multiplication can also be performed with only two Montgomery multiplications, which makes the Montgomery algorithm even more interesting for practical implementations. In this case, only one operand is converted into Montgomery representation and the result does not need to be converted back into normal representation:

- Conversion of X into Montgomery representation:

$$\begin{aligned} X_{Mont} &= Mont_M(X, R^2) \\ &= (X \cdot R^2 \cdot R^{-1}) \mod M \\ &= (X \cdot R) \mod M\,. \end{aligned} \tag{3.1}$$

- Computation of the result:

$$\begin{aligned} T &= Mont_M(X_{Mont}, Y) \\ &= (X_{Mont} \cdot Y \cdot R^{-1}) \mod M \\ &= (X \cdot R \cdot Y \cdot R^{-1}) \mod M \\ &= (X \cdot Y) \mod M\,. \end{aligned} \tag{3.2}$$

Algorithm 5 shows the method for Montgomery multiplication [58]. Here, the operands, X and Y, and the modulus, M, are represented as n-digit words, where every digit consists of b bits. As can be seen from Alg. 5, the only division that is performed, is a division by 2^b. Because the algorithm ensures that the b LSBs of T are equal to zero, this division comes down to a right-shift operation over b bits. In [58], it is proven that, if the inputs X and Y are smaller than M, the result is also bounded by M, while the intermediate result T has a bound of $2 \cdot M$. Because of this bound on T, only the LSB of the digit T_n can be equal to one. The input M' can be computed by using the observation that $-M' \cdot M \equiv 1 \mod 2^i$ for all $i \leq b$. In this chapter, we assume that M' is pre-computed.

Because of the bound of $2 \cdot M$ on the intermediate result T, a final subtraction needs to be performed in case $T \geq M$. This conditional subtraction is sensitive to timing attacks [42] and should therefore be avoided. In [96] and [95], Walter presents an algorithm that performs one extra iteration, but makes the conditional final subtraction unnecessary. The improved algorithm is shown in Alg. 6. It is time-constant and avoids the implementation of a subtractor.

Walter proves that, if the inputs X and Y are smaller than $2 \cdot M$, the output is also bounded to $2 \cdot M$, while the intermediate result T has a bound of $4 \cdot M$. He also shows that, after converting from Montgomery to normal representation, the result is smaller than M again. Note that only the LSB of the digits X_n and Y_n can be equal to one in order to satisfy $0 \leq X, Y < 2 \cdot M$; the rest of the bits are

Algorithm 5 Montgomery multiplication

Require: $M = (M_{n-1} \dots M_0)_{2^b}$, $X = (X_{n-1} \dots X_0)_{2^b}$, $Y = (Y_{n-1} \dots Y_0)_{2^b}$ with $0 \leq X, Y < M$, $R = 2^{n \cdot b}$, $\gcd(M, 2^b) = 1$ and $M' = -M^{-1} \mod 2^b$
Ensure: $(X \cdot Y \cdot R^{-1}) \mod M$
1: $T = (T_n \dots T_0)_{2^b} \leftarrow 0$
2: **for** i from 0 to $n - 1$ **do**
3: $\quad U_i \leftarrow ((T_0 + X_0 \cdot Y_i) \cdot M') \mod 2^b$
4: $\quad T \leftarrow (T + X \cdot Y_i + M \cdot U_i)/2^b$
5: **end for**
6: **if** $T \geq M$ **then**
7: $\quad T \leftarrow T - M$
8: **end if**
9: Return T

zero. For digit T_n the two LSBs can be equal to one, while the rest of the bits are zero, because $T < 4 \cdot M$ is always ensured. However, in the implementations that are discussed in Sects. 3.3 and 3.4, we always consider the complete length of X, Y and T, i.e., $(n + 1) \cdot b$, for ease of notation.

Algorithm 6 Improved Montgomery multiplication

Require: $M = (M_{n-1} \dots M_0)_{2^b}$, $X = (X_n \dots X_0)_{2^b}$, $Y = (Y_n \dots Y_0)_{2^b}$ with $0 \leq X, Y < 2 \cdot M$, $R = 2^{(n+1) \cdot b}$, $\gcd(M, 2^b) = 1$ and $M' = -M^{-1} \mod 2^b$
Ensure: $(X \cdot Y \cdot R^{-1}) \mod M$
1: $T = (T_n \dots T_0)_{2^b} \leftarrow 0$
2: **for** i from 0 to n **do**
3: $\quad U_i \leftarrow ((T_0 + X_0 \cdot Y_i) \cdot M') \mod 2^b$
4: $\quad T \leftarrow (T + X \cdot Y_i + M \cdot U_i)/2^b$
5: **end for**
6: Return T

The algorithms in this section consist of two parts: a multiplication of X with Y_i and a b-bit reduction of the product. Because the b LSBs need to be reduced, the reduction digit is computed based on these bits, i.e., $X_0 \cdot Y_i$. The algorithms presented in the next section differ in the way they interleave the multiplication and the reduction parts of the Montgomery algorithm.

3.2 Parallelization of Algorithms for a w-bit Data Path

An overview of different algorithms for Montgomery multiplication using a single w-bit integer multiplier is given by Koç *et al.* in [41]. In this section, we highlight two of the described methods: Separated Operand Scanning (SOS) and Coarsely Integrated Operand Scanning (CIOS). Whereas the SOS method performs the multiplication of X and Y and the reduction of the result separately, the CIOS method interleaves these two operations. In [41], another method for Montgomery multiplication, called Finely Integrated Operand Scanning (FIOS), is also presented. Similar to the CIOS method, the FIOS method also interleaves the multiplication and the reduction steps. However, the CIOS algorithm multiplies larger parts of X and Y before performing a reduction operation in comparison to the FIOS algorithm. To be more suitable for implementation on an FPGA, these algorithms are extended to parallel versions. When fully parallelizing the CIOS and the FIOS methods, the same parallel algorithm is obtained. That is why only the CIOS algorithm is evaluated next to the SOS method. Note that the algorithms presented in this section are based on Montgomery's algorithm as given in Alg. 5, while the architectures in Sects. 3.3 and 3.4 follow Alg. 6.

3.2.1 The SOS Method

The SOS method for Montgomery multiplication using a single w-bit data path is given in Alg. 7. A parallelized version of the SOS method comes down to Alg. 6 with b equal to the full number of bits of the operands. Whereas Alg. 7 uses a w-bit M' to compute the reduction digits U sequentially, the parallelized version of the algorithm computes a full length reduction digit U based on a full length M'. With full length we mean the size of X and Y in the improved Montgomery algorithm (Alg. 6), i.e., with at least one additional bit in comparison to Alg. 5. In this case, the parallelized SOS algorithm consists of only one execution of the loop in Alg. 6, corresponding to the following steps:

- a full length integer multiplication: $XY = X \cdot Y$;
- a full length integer multiplication modulo 2^b: $U = (XY \cdot M') \mod 2^b$;
- another full length integer multiplication: $MU = M \cdot U$;
- a full length integer addition: $T = XY + MU$;
- a right-shift over the full operand length: $T = T/2^b$.

These steps are summarized in Fig. 3.1.

Algorithm 7 Separated Operand Scanning (SOS) method for Montgomery multiplication, where ADD propagates the carry throughout T and RED denotes the conditional final reduction step [41].

Require: $M = (M_{n-1} \dots M_0)_{2^w}$, $X = (X_{n-1} \dots X_0)_{2^w}$, $Y = (Y_{n-1} \dots Y_0)_{2^w}$ with $0 \leq X, Y < M$, $R = 2^{n \cdot w}$ with $\gcd(M, 2^w) = 1$ and $M' = -M^{-1} \mod 2^w$

Ensure: $(X \cdot Y \cdot R^{-1}) \mod M$

1: $T = (T_{2n-1} \dots T_0)_{2^w} \leftarrow 0$
2: **for** i from 0 to $n-1$ **do**
3: $\quad C \leftarrow 0$
4: $\quad$ **for** j from 0 to $n-1$ **do**
5: $\quad\quad (C, S) \leftarrow T_{i+j} + X_j \cdot Y_i + C$
6: $\quad\quad T_{i+j} \leftarrow S$
7: $\quad$ **end for**
8: $\quad T_{i+n} \leftarrow C$
9: **end for**
10: **for** i from 0 to $n-1$ **do**
11: $\quad C \leftarrow 0$
12: $\quad U \leftarrow (T_i \cdot M') \mod 2^w$
13: $\quad$ **for** j from 0 to $n-1$ **do**
14: $\quad\quad (C, S) \leftarrow T_{i+j} + U \cdot M_i + C$
15: $\quad\quad T_{i+j} \leftarrow S$
16: $\quad$ **end for**
17: $\quad \mathrm{ADD}(T_{i+n}, C)$
18: **end for**
19: **for** i from 0 to n **do**
20: $\quad T_i \leftarrow T_{i+n}$
21: **end for**
22: $\mathrm{RED}(T)$
23: Return T

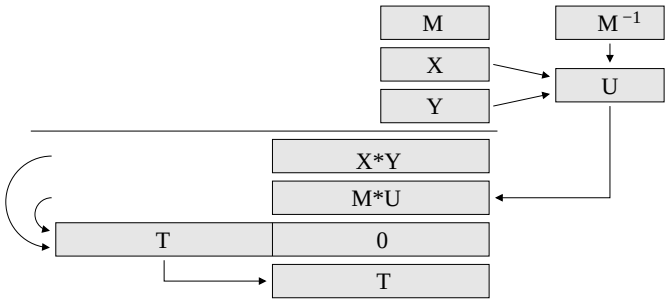

Figure 3.1: Schematic representation of the execution of the parallelized SOS algorithm.

3.2.2 The CIOS Method

The CIOS method is given in Alg. 8. It computes Alg. 6 with b equal to w, the size of the data path. The parallelized versions of the operations inside the loop are executed as follows:

- an integer multiplication of a full length operand with a b-bit operand: $XY_i = X \cdot Y_i$;
- an integer addition of two b-bit operands ignoring the overflow bit: $H_i = (T_0 + (XY_i)_0) \mod 2^b$, where $(XY_i)_0$ is the least significant b-bit digit of XY_i;
- an integer multiplication of two w-bit operands modulo 2^b: $U_i = (H_i \cdot M') \mod 2^b$;
- an integer multiplication of a full length operand with a b-bit operand: $MU_i = M \cdot U_i$;
- a 3-input full length integer addition: $T = T + XY_i + MU_i$;
- a right-shift over b bits: $T = T/2^b$.

These steps are summarized in Fig. 3.2.

Algorithm 8 Coarsely Integrated Operand Scanning (CIOS) method for Montgomery multiplication, where RED denotes the conditional final reduction step [41].

Require: $M = (M_{n-1} \dots M_0)_{2^w}$, $X = (X_{n-1} \dots X_0)_{2^w}$, $Y = (Y_{n-1} \dots Y_0)_{2^w}$ with $0 \leq X, Y < M$, $R = 2^{n \cdot w}$ with $\gcd(M, 2^w) = 1$ and $M' = -M^{-1} \mod 2^w$

Ensure: $(X \cdot Y \cdot R^{-1}) \mod M$

1: $T = (T_{n+1} \dots T_0)_{2^w} \leftarrow 0$
2: **for** i from 0 to $n-1$ **do**
3: $C \leftarrow 0$
4: **for** j from 0 to $n-1$ **do**
5: $(C, S) \leftarrow T_j + X_j \cdot Y_i + C$
6: $T_j \leftarrow S$
7: **end for**
8: $(C, S) \leftarrow T_n + C$
9: $T_n \leftarrow S$
10: $T_{n+1} \leftarrow C$
11: $C \leftarrow 0$
12: $U \leftarrow (T_0 \cdot M') \mod 2^w$
13: **for** j from 0 to $n-1$ **do**
14: $(C, S) \leftarrow T_j + U \cdot M_j + C$
15: $T_j \leftarrow S$
16: **end for**
17: $(C, S) \leftarrow T_n + C$
18: $T_n \leftarrow S$
19: $T_{n+1} \leftarrow T_{n+1} + C$
20: **for** j from 0 to n **do**
21: $T_j \leftarrow T_{j+1}$
22: **end for**
23: **end for**
24: RED(T)
25: Return T

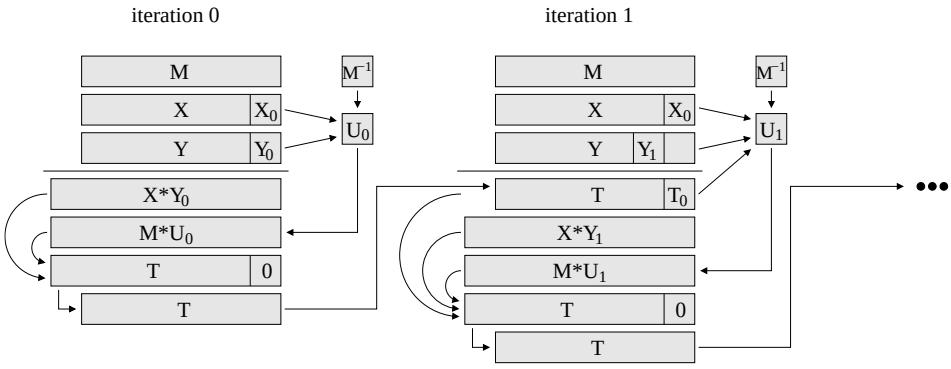

Figure 3.2: Schematic representation of the execution of the parallelized CIOS algorithm.

3.3 Implementation According to the Parallelized SOS Method

From Sect. 3.2.1, we can derive that the hardware requirements for the implementation of a Montgomery multiplier according to the parallelized SOS method are:

1. a full length integer multiplier;
2. a full length integer multiplier modulo 2^b;
3. another full length integer multiplier;
4. a double length integer adder;
5. a right-shifter over b positions.

The first three resources can be shared, because the modulus of the second operation is a power of two: an operation modulo a power of two simply ignores the bits that would make the result bigger than the modulus minus 1. The three multiplications can thus be performed on the same multiplier. Although the least significant half of the result of the addition will be zeros, it needs to be computed anyway in order to know the carry-bit going into the most significant half of the result. That is why the adder needs to be of double size. The right-shifter does not lead to extra hardware: it simply throws away all bits that are smaller than 2^b.

Our implementation target is a Xilinx Virtex XC2VP30-7FF1152 FPGA, which contains 136 dedicated multiplier blocks and 13696 slices [105]. We use the dedicated multiplier blocks to implement the full length integer multiplier. For the implementation of the adder, we employ the fast addition logic that can be configured in the slices of the FPGA. The exact implementation details of this logic are proprietary information of the FPGA vendor. For the storage of the intermediate values and for data transfer to and from the outside of the Montgomery multiplier, we use the block RAM on the FPGA. We foresee an extra option to configure the data path as two parallel data paths of half the size. The need for this configurability will become clear in Chapter 4. The resulting data path is depicted in Fig. 3.3 for a 1024-bit Montgomery multiplication. For RSA, a 1024-bit key offers a reasonable level of security for the coming few years. For ECC, the security level of a 160-bit key exceeds that of a 1024-bit RSA key [24]. However, our design approach can be applied to any key length. The details of the multiplier, the adder and the RAM bank are discussed in this section. The last part of the section gives the implementation results for the parallelized SOS Montgomery multiplier.

3.3.1 The Multiplier

The dedicated multiplier blocks that are available on recently designed FPGAs, have a width of 18x18 [105]. However, because we want to store the result of the integer multiplication in the RAM bank, the available options for the block RAM width should be taken into account. The block RAM width that gives the best approximation to the 18-bit size of the multipliers, is 16 bits. That is why we employ the dedicated multiplier blocks as 16x16 multipliers. Since we implement the improved Montgomery algorithm (Alg. 6) i.o. the original method (Alg. 5), at least one additional overflow bit is needed to avoid the final reduction step. However, because we use 16x16 multipliers, there will be 16 additional bits, resulting in Alg. 6 with $b = 16$ and $n = 64$. Because the dedicated multipliers are "free", this does not lead to a substantial area overhead compared to an ASIC-oriented data path with only one overflow bit. Applying this technique to the mode in which the multiplier is configured as two parallel halves, leads to a multiplier with an operand width of 1056 bits (because the parallel mode requires two 512+16 multiplications in parallel). The multiplier is depicted in Fig. 3.4. It contains a total of 66 16x16 multipliers. Besides dedicated multiplier blocks, our integer multiplier also contains 66 Carry-Save Adders (CSAs) and 2 Ripple Carry Adders (RCAs). A detailed CSA is shown in Fig. 3.5. The RCAs are implemented using the fast addition logic in the configurable part of the FPGA.

As can be seen from Fig. 3.4, one of the operands is provided in parallel (X), while the other one is inserted in a sequential manner (Y). This is also visible in Fig. 3.3. The result of the multiplication is provided in a digit-serial manner, where the digit size is equal to the size of the multiplier blocks, i.e., 16. The *big* signal decides on the configuration option. In the parallel option, t_{left} and t_{right}

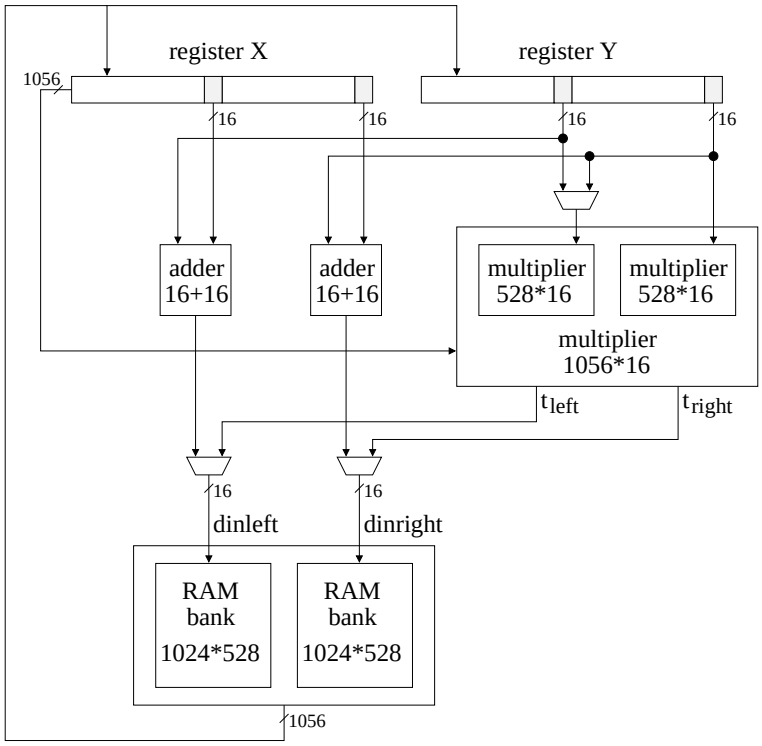

Figure 3.3: Details of the architecture of the 1024-bit SOS-based Montgomery multiplier.

are the digit-serial output signals, while y_{left} and y_{right} are the digit-serial inputs. When the multiplier is used for one big multiplication, t_{right} is the digit-serial output signal, while y_{right} is the digit-serial input.

The first digit of the result is ready after two clock cycles. In the first cycle, the digits of x are multiplied to a digit of y and the results are stored in two 16-bit parts: the least significant and the most significant part of the 32-bit result. In Fig. 3.4, R denotes a 16-bit register. In the next cycle, the most significant part of each result is added to the least significant part of the next result, i.e., the result of the 16-bit multiplication on the left-hand side in Fig. 3.4. Because the carry-chain of a regular addition would be too long, 16-bit CSAs are used for the addition of two 16-bit words. This solution also introduces a carry chain in between the CSAs: a CSA receives a carry-bit input from the lower level and gives a carry-bit output to the higher level. To compute a digit of t in every clock cycle, the sum and carry outputs of the least significant CSA need to be added up. The

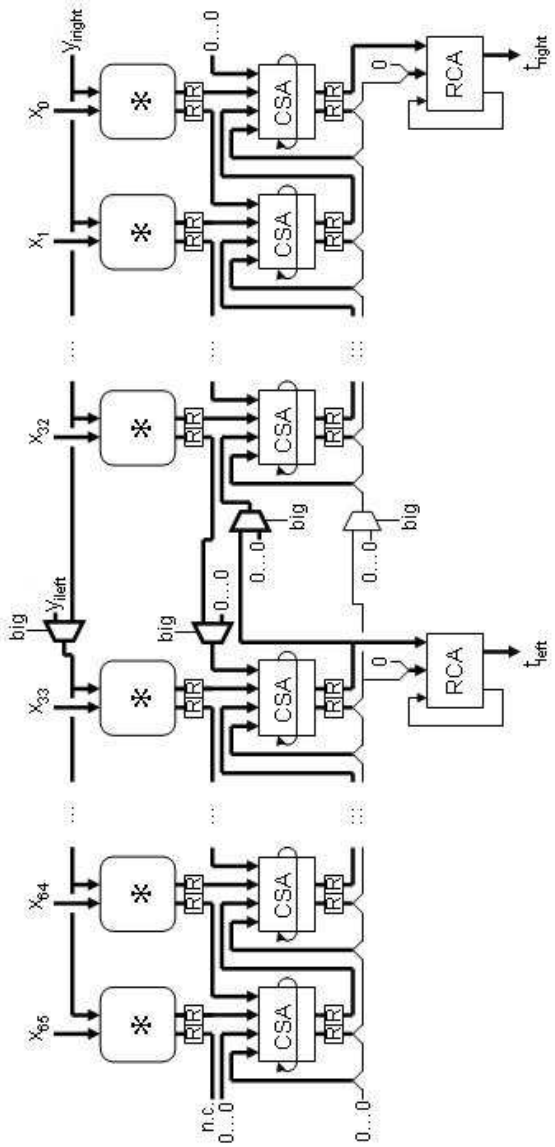

Figure 3.4: Details of the integer multiplier inside the SOS-based Montgomery multiplier. Here, CSA and RCA denote Carry-Save Adder and Ripple Carry Adder, respectively.

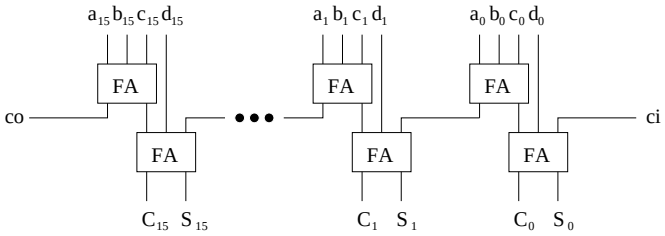

Figure 3.5: Details of a CSA inside the SOS-based Montgomery multiplier, with $(a_{15} \dots a_0)$, $(b_{15} \dots b_0)$, $(c_{15} \dots c_0)$ and $(d_{15} \dots d_0)$ the 16-bit input digits, $(S_{15} \dots S_0)$ the 16-bit sum output digit, $(C_{15} \dots C_0)$ the 16-bit carry output digit, *ci* the carry input bit from the previous level and *co* the carry output bit to the next level.

difference in weight of the 16-bit sum and carry at the output of a CSA is a factor 2. This means that, to compute the 16-bit addition, 16 bits of the sum are added to the 15 least significant bits of the carry, concatenated with an LSB zero. The addition of the outputs of the next CSA needs the MSB of the carry output of this CSA. The detailed connection between the CSAs in Fig. 3.4 is given in Fig. 3.6. Because the multiplication of the digits of x with the next digit of y needs to be added to the previous result, which has a weight that is a factor 2^{16} lower, the previous result is shifted over 16 positions to the right, entering into the CSA on the right-hand side. For the carry output, only 15 bits are going the CSA on the right-hand side, they are merged with the MSB of the carry output of that CSA. Because of the shift of the previous result, the overflow carry-bit at the output of a CSA, needs to be led back to the input of that same CSA through a register. The final computation of a digit of t is done using an RCA, of which the carry output is also led back to the carry-input of the RCA through a register. These feedback registers are omitted in Fig. 3.4.

3.3.2 The Adder

Because the interface to the RAM bank is 16-bit digit-serial, the adder is implemented as a 16-bit digit-serial adder, where the output carry-bit is led back into the input carry-bit through a register. To perform two additions in parallel, we do not split the adder into two parts, like we do for the multiplier, but we repeat it in order to have two 16-bit digit-serial adders in parallel. The logic inside the 16-bit adder is implemented in the configurable part of the FPGA and is proprietary information of the FPGA vendor [105].

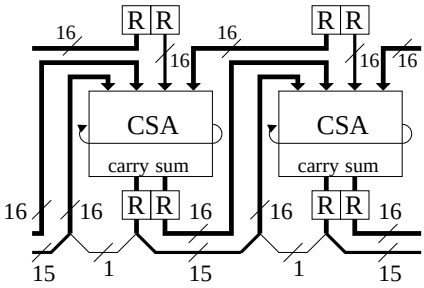

Figure 3.6: Detailed view of the dataflow in two adjacent CSAs in Fig. 3.4.

3.3.3 The RAM Bank

The RAM blocks, described in [105], are organized in a bank, which is depicted in more detail in Fig. 3.7. The RAM bank is used to store the intermediate values in between the steps in Sect. 3.2.1. It consists of 66 dedicated RAM blocks with a width of 16 bits and a depth of 1024. The results of the full length integer multiplications are split into two parts and stored in two consecutive addresses. If the multiplier or the adder are configured as one arithmetic block, the RAM bank is also used as a whole. All RAM blocks receive the same reset (rst), clock (clk), address ($address$) and enable (en), but can be written separately ($we(0) \ldots we(65)$). If the parallel option is chosen for either the multiplier or the adder, the RAM bank is also split into two parts. However, the reset, clock, address and enable value stay the same for the two parts.

To limit the number of multiplexers, the data path only allows two multiplications or two additions in parallel. An addition in parallel with a multiplication is not possible. The PKC algorithms, explained in Chapter 4, are rewritten for this specific architecture.

3.3.4 Implementation Results

In this section, we give the implementation results for our SOS-based Montgomery multiplier using a 1056-bit integer multiplier, which can be configured as two parallel halves, and two 16-bit digit-serial adders. Although the data path is optimized for 1024-bit Montgomery multiplications, it can also handle smaller operands. Performing Montgomery multiplications of bigger sizes requires more control logic and is therefore discussed in Chapter 4.

When we denote the number of 16-bit digits in the modulus by n, the number of cycles for the integer multiplication $XY = X \cdot Y$ is equal to $2 \cdot n + 4$. The first 2 cycles are needed for the pipelining header in the multiplier, i.e., for the first

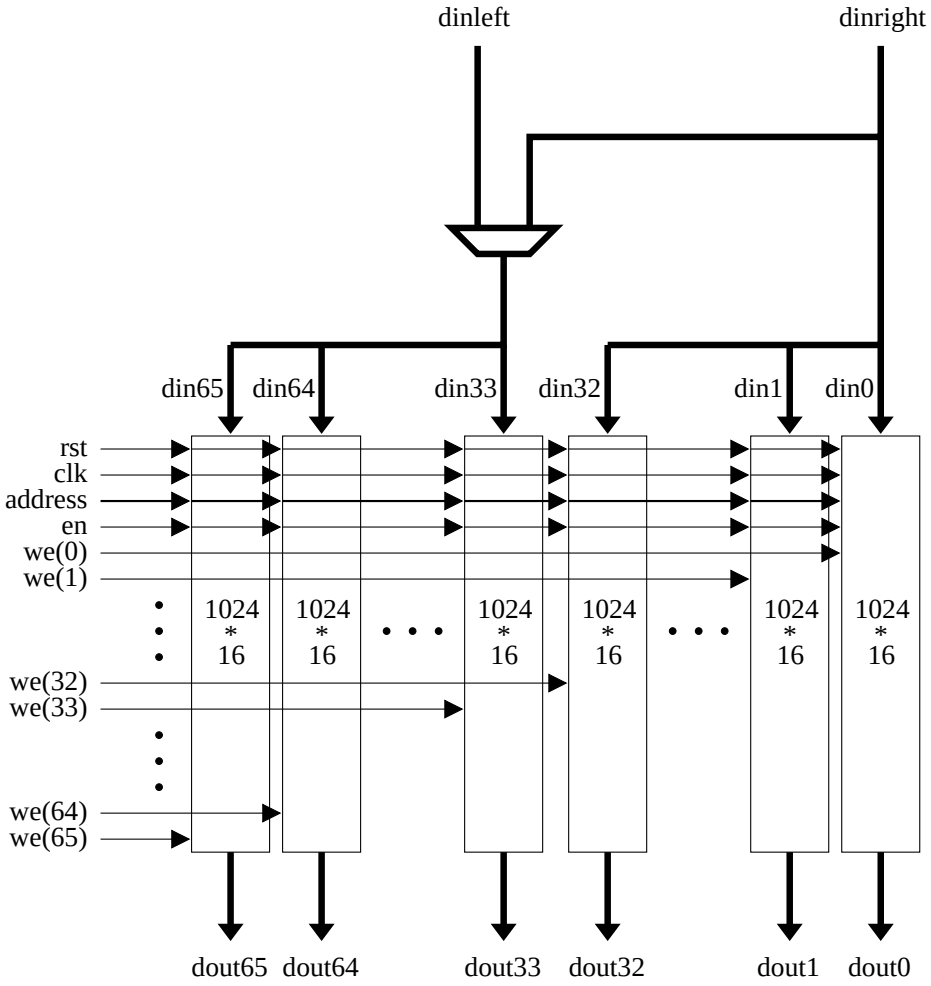

Figure 3.7: Details of the RAM bank inside the SOS-based Montgomery multiplier, where the thick lines denote 16-bit signals. All other values are single bits, except for the address, which is 10 bits wide.

bits to be stored in the registers denoted by R in Fig. 3.4. In the next $2 \cdot n + 2$ cycles, the result of the multiplication of X and Y is computed. The computation of $U = (XY \cdot M') \mod 2^{(n+1)\cdot 16}$ requires $n + 3$ cycles and the multiplication $UM = U \cdot M$ is performed in $2 \cdot n + 3$ cycles (we need one cycle less than for the computation of XY, because M is one b-bit digit smaller than U, X and Y). The final result $T = XY + UM$ is computed in $2 \cdot n + 2$ cycles.

When taking into account the communication of the multiplier with the RAM, but not the communication of the RAM with the user of the coprocessor, the number of cycles for a Montgomery multiplication of $n \cdot 16$ bits consists of:

- 2 cycles for reading X and Y from the RAM and loading them into the input registers;
- $2 \cdot n + 4$ cycles for the full length multiplication with XY as a result;
- 1 cycle to write the last 16 bits of XY into the RAM;
- 2 cycles for reading XY and M' from the RAM and loading them into the input registers;
- $n + 3$ cycles for the half length multiplication with U as a result;
- 1 cycle to write the last 16 bits of U into the RAM;
- 2 cycles for reading M and U from the RAM and loading them into the input registers;
- $2 \cdot n + 3$ cycles for the full length multiplication with MU as a result;
- 1 cycle to write the last 16 bits of MU into the RAM;
- 2 cycles for reading the least significant halves of XY and MU from the RAM and loading them into the input registers;
- $n + 1$ cycles for the first part of the addition with the least significant half of T as a result;
- 1 cycle to write the last 16 bits of the least significant half of T into the RAM;
- 2 cycles for reading the most significant halves of XY and MU from the RAM and loading them into the input registers;
- $n+1$ cycles for the second part of the addition with the most significant half of T as a result;
- 1 cycle to write the last 16 bits of the most significant half of T into the RAM.

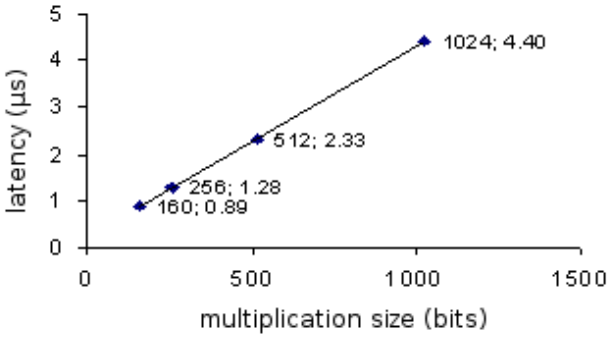

Figure 3.8: Execution time of one Montgomery multiplication as a function of the multiplication size.

This leads to a total of $7 \cdot n + 27$ cycles for a $n \cdot 16$-bit Montgomery multiplication. The parallelized SOS architecture was implemented on a Xilinx Virtex XC2VP30-7FF1152 FPGA [105]. This led to the implementation results given in Table 3.1 and depicted in Fig. 3.8. The figure shows that the latency of a Montgomery multiplication is linear w.r.t. the operand size.

Table 3.1: Implementation results of our Montgomery multiplier according to the parallelized SOS method, where t_x is the latency of an x-bit Montgomery multiplication. The size of one RAM block is 1024 by 16.

resources	8192 slices 66 MULTs 66 RAM blocks
maximum frequency	108 MHz
t_{160}	0.89 μs
t_{256}	1.28 μs
t_{512}	2.33 μs
t_{1024}	4.40 μs

3.4 Implementation According to the Parallelized CIOS Method

From Sect. 3.2.2, we can derive that the hardware requirements for the implementation of a Montgomery multiplier according to the parallelized CIOS method are:

1. an integer multiplier with one full length input and one b-bit input;
2. a b-bit integer adder;
3. a b-bit integer multiplier where the most significant half of the result is ignored;
4. another integer multiplier with one full length input and one b-bit input;
5. a 3-input full length integer adder;
6. a right-shifter over b positions.

The first, the third and the fourth resources can be mapped onto one integer multiplier with a full length and a b-bit input. However, for reasons that will become clear in the remainder of this section, it is more optimal to provide two of these integer multipliers in parallel. Reducing the area can still be done afterwards, but in a different way.

In this section, we present our architecture according to the CIOS method using a step-by-step approach. Throughout this section, we adopt the notation used in Alg. 6. The last part of the section gives the implementation results for the CIOS-based Montgomery multiplication architectures.

3.4.1 Improved Baseline Implementation with Carry-save Representation

A baseline implementation of Alg. 6 according to the parallelized CIOS method computes a new full length value of T in each clock cycle, resulting in a cycle count of $n+1$, where n is the number of b-bit digits in the operands, as stated in Alg. 6. However, the critical path in this implementation is very long, as can be seen in Fig. 3.9. Especially the $(n+1) \cdot b$-by-b-bit and the $n \cdot b$-by-b-bit multiplications (denoted by I and III), even if performed in parallel, and the long integer addition in combination with a b-bit right-shift operation (denoted by IV) in Step 4 of Alg. 6 determine the critical path. As follows from the bounds on Alg. 6, the output of IV is bounded to $(n+1) \cdot b$ bits. This makes IV a 3-input adder with inputs of $(n+1) \cdot b$, $(n+1) \cdot b$ and $(n+2) \cdot b$ bits, followed by a b-bit right-shifter. The result of the addition is $(n+2) \cdot b$ bits of which the b LSBs are 0. After shifting over b positions, the output of IV is $(n+1) \cdot b$ bits. Note that operation I in

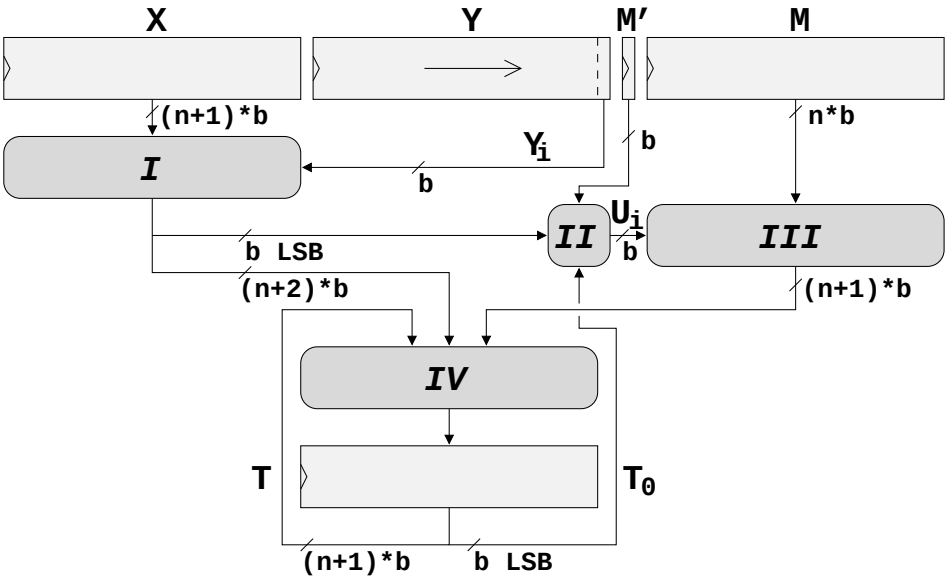

Figure 3.9: Architecture of the baseline Montgomery multiplier, where I and III denote $(n+1) \cdot b$-by-b-bit and $n \cdot b$-by-b-bit multipliers, respectively, II denotes a b-bit addition followed by a b-bit multiplication modulo 2^b and IV denotes a 3-input adder in combination with a b-bit right-shift operation. Here b LSB denotes the b least significant bits of a signal.

Fig. 3.9 performs the partial multiplication of X and Y, while operations II and III prepare the reduction part to be added to the partial product of X and Y using operation IV. As can be seen from the figure, the multiplication of X and Y and the reduction of the result are interleaved or coarsely integrated.

The large delay in building blocks I, III and IV is caused by the carry chains inside the multiplications and the integer addition. An improvement on the architecture is depicted in Fig. 3.10, where the $(n+1) \cdot b$-by-b-bit and $n \cdot b$-by-b-bit multiplications are carried out by multipliers of b-by-b bits, without combining the results, i.e., the result of the multiplications is written in a special carry-save form, where the weight of the carry is a factor 2^b bigger than the weight of the sum. This is shown in Fig. 3.11 for the improved $n \cdot b$-by-b-bit multiplication, denoted by III'. The improved $(n+1) \cdot b$-by-b-bit multiplication, denoted by I', just uses one additional b-by-b-bit multiplier. The computation of U_i slightly changes to cope with the carry-save form, i.e., the input T_0 to II' consists of two parts that need to be added to the b LSBs of the result of I'. This b-bit addition is followed by a b-bit multiplication modulo 2^b with M'.

A carry-save form can be maintained throughout the whole implementation by

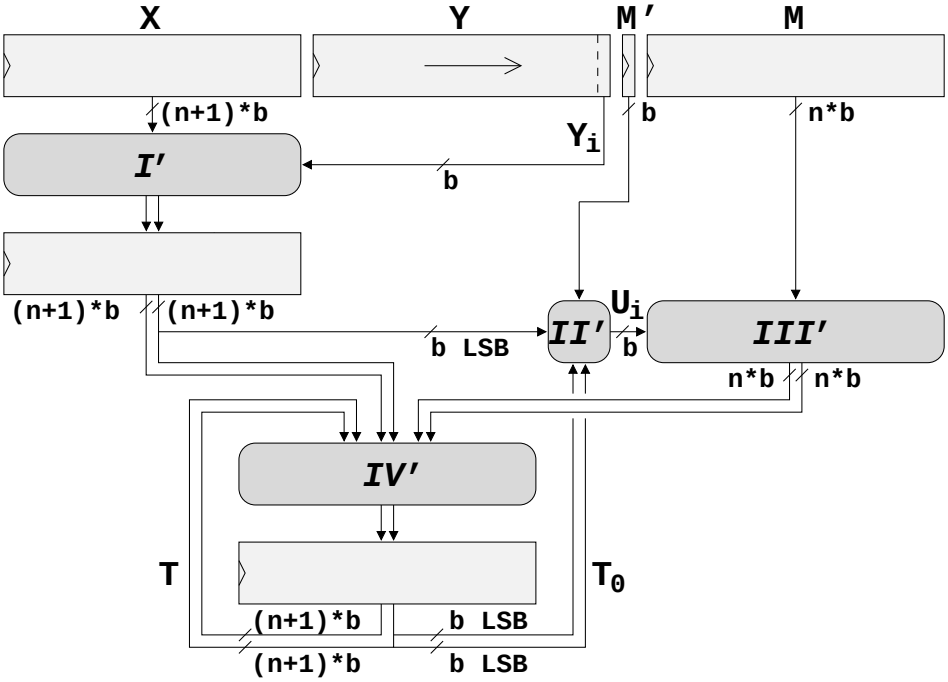

Figure 3.10: Architecture of the improved baseline Montgomery multiplier. The adder for converting the carry-save representation into the final result at the end of the computation is omitted in this figure. Here b LSB denotes the b least significant bits of a signal.

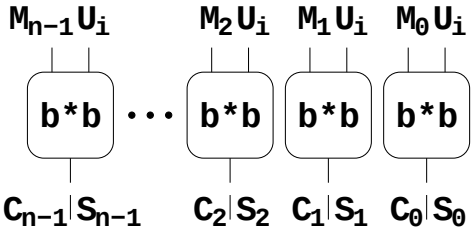

Figure 3.11: Schematics of the $n \cdot b$-by-b-bit multiplier, indicated with III' in Fig. 3.10. The outputs of the b-by-b-bit multipliers are $2 \cdot b$ bits, of which the least significant b bits are part of the sum and the most significant b bits are part of the carry.

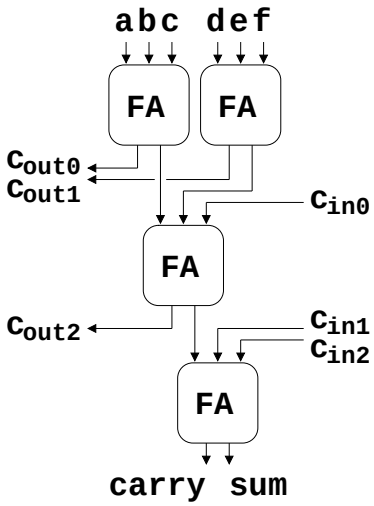

Figure 3.12: Details of a one-bit 6-2 carry-save adder.

replacing the long integer addition with 6-2 carry-save logic. Figure 3.12 depicts a one-bit 6-2 carry save adder, while Fig. 3.13 shows the complete 6-2 carry-save adder. Here, $S_{I'}$ and $S_{III'}$ are the sum outputs of blocks I' and III', respectively. The carry outputs of these blocks are denoted by $C_{I'}$ and $C_{III'}$. The sum and carry coming from the feedback register at the output of block IV' are denoted by $S_{IV'prev}$ and $C_{IV'prev}$, respectively. S_{int} and C_{int} are the internal sum and carry before the shift operation. The weight of the carry output of the 6-2 carry-save adder is a factor 2 bigger than the weight of the sum output.

The right-shift over b positions in Step 4 of Alg. 6 is performed by shifting the sum over b positions and the carry over $b-1$ positions. After this shift, the weights of the sum and carry become equal. The b sum bits and $b-1$ carry bits that are shifted out, sum up to 0, but the overflow carry output of this addition can be different from 0. That is why this addition must be performed and the overflow carry bit must be taken into account. A register is inserted in IV' to store the overflow bit. The alignment of the inputs and outputs in block IV' is summarized in Fig. 3.14. The weight of the carry outputs of I' and III' is b bits shifted compared to the corresponding sum outputs. Because the length of X is $(n+1)\cdot b$ bits, while the length of M is only $n\cdot b$ bits, the sum and carry results of I' are b bits longer than the sum and carry results of III'. Adding the outputs of I' and III' to the previous output of IV' results in a new carry and sum output of IV'. Signal c is the overflow carry bit as a result of the right-shift of the least significant part of the output of IV', which needs to be added to the inputs of IV' in the next iteration. The subscripts *prev* and *next* denote the output and input

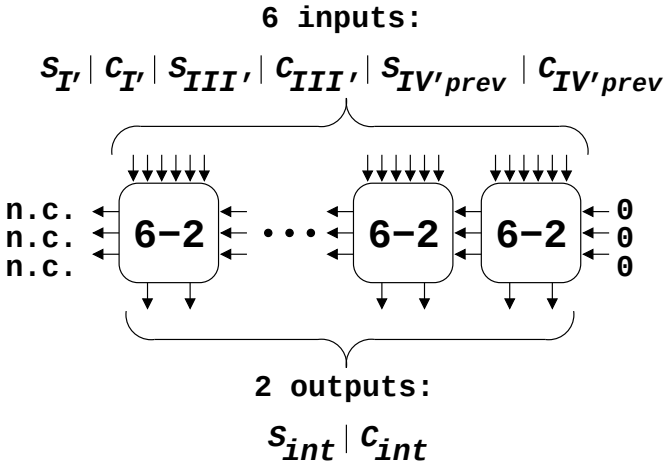

Figure 3.13: Interconnection of the one-bit 6-2 carry-save adders in the carry-save logic.

of a register, respectively.

A register is also inserted to store the result of $X \cdot Y_i$. This shortens the critical path without changing anything to the feedback loop. The resulting architecture is shown in Fig. 3.10. It contains $2 \cdot n + 2$ b-by-b-bit multipliers. An adder is inserted to transform the carry-save output into the final result. This adder is not shown on the figure.

The number of cycles to complete one Montgomery multiplication is equal to $1 + (n + 1) + \alpha$ clock cycles, where the first cycle is needed to compute $X \cdot Y_0$. After this first computation, $n+1$ carry-save values are computed using 1 cycle per computation. The number of cycles for the final addition is denoted by α. This value depends on the width of the adder, which should be chosen to be balanced with the path from the input registers to the output registers of the carry-save adder. Performance results for this architecture are given in Sect. 3.4.4.

3.4.2 Two-stage Pipelined Implementation

The implementation in Fig. 3.10 requires $2 \cdot n + 2$ b-by-b-bit multipliers. For big operand sizes and/or small FPGAs, it is not feasible to occupy that many multipliers. That is why we present a version which is downsized to half of the operand size, i.e., approximately half of the multiplier blocks. The architecture is depicted in Fig. 3.15; it looks similar to the one presented in Fig. 3.10. The only difference is that the $(n + 1) \cdot b$-by-b-bit and $n \cdot b$-by-b-bit multiplications (in carry-save representation) are performed in two cycles instead of one. This can

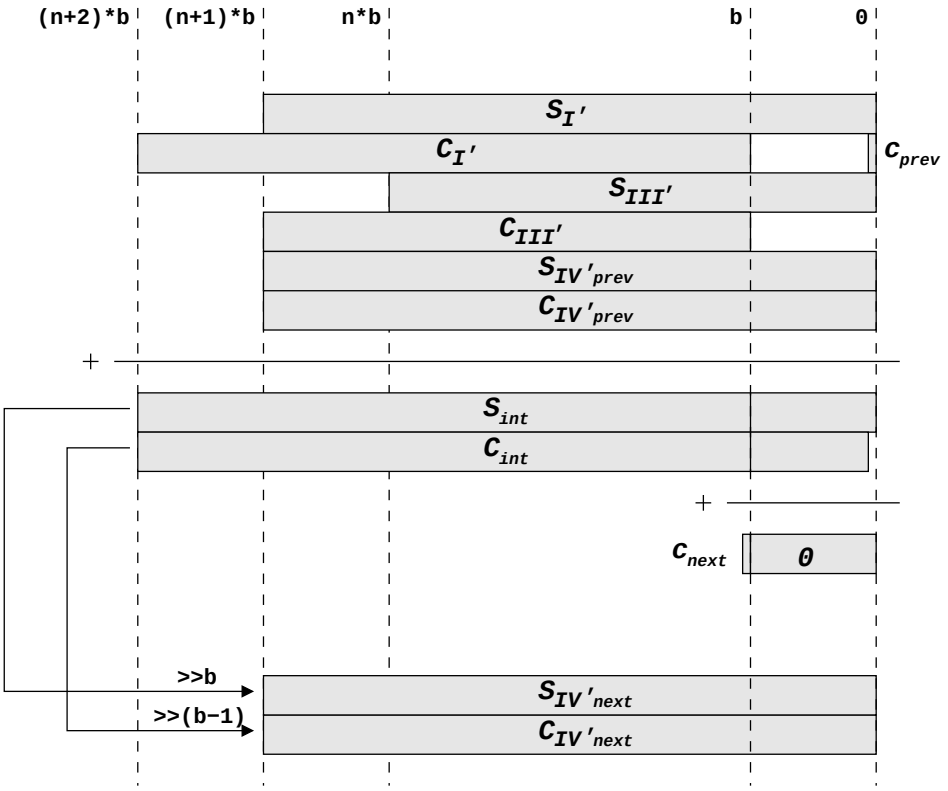

Figure 3.14: Alignment of the inputs and outputs of IV' inside the CIOS-based Montgomery multiplier.

be done with an $(\lceil \frac{n}{2} \rceil + 1) \cdot b$-by-$b$-bit and an $\lceil \frac{n}{2} \rceil \cdot b$-by-$b$-bit multiplier. Another b-by-b-bit multiplier is used for the computation of U_i, resulting in a total number of $n + 2$ multipliers.

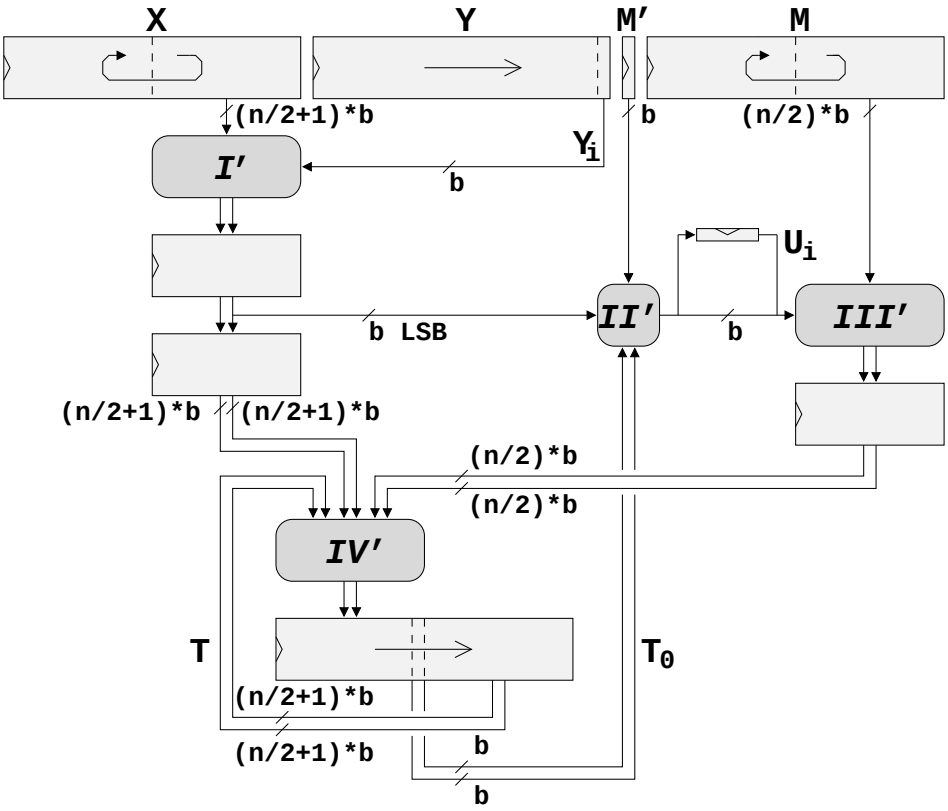

Figure 3.15: Architecture of the two-stage pipelined Montgomery multiplier, with I', II', III' and IV' half as wide as in Fig. 3.10. The adder for converting the carry-save representation into the final result at the end of the computation is omitted in this figure. Here b LSB denotes the b least significant bits of a signal.

Although the multiplication of X and Y_i in Step 4 of Alg. 6 is performed in two cycles, the first part can already be used for the computation of U_i before the second part is finished. Once U_i is computed, the first part of the multiplication of M and U_i can be performed. Because the second part of this multiplication also needs U_i, the value is stored in a register for two cycles. By inserting two pipelined registers to save the $X \cdot Y_i$ results and a register to save the $M \cdot U_i$ results, the addition in Step 4 can be performed using carry-save logic that is half the size of

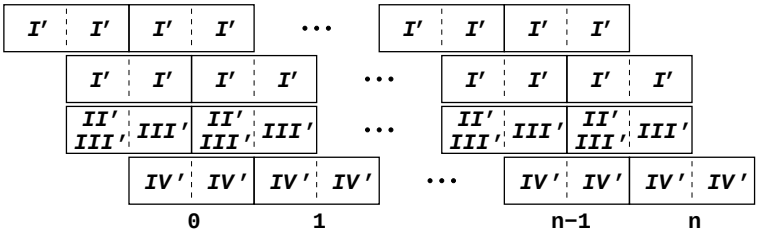

Figure 3.16: Pipelining schedule of the two-stage pipelined Montgomery multiplier with a horizontal time axis. The second line of I' operations denotes the storage of the result of I' in a pipelining register. After the last step in the pipelining schedule, an addition needs to be performed to convert the carry-save representation into the final result. This addition is omitted in this schedule.

the operands. The pipelining schedule of this computation is shown in Fig. 3.16.

While pipelining registers are usually inserted to increase the throughput of a system, we use pipelining inside the feedback loop, to compensate for the speed degradation caused by the downsizing of the architecture. As a result, it is not possible to pipeline consecutive Montgomery multiplications in order to increase the throughput.

As can be derived from the pipelining schedule in Fig. 3.16, the number of cycles for one Montgomery multiplication using this architecture is equal to $2 + 2 \cdot (n + 1) + \alpha$, with α the number of cycles for the final addition. The only difference with the architecture in Fig. 3.10, is the feedback loop, which needs 2 cycles instead of 1 per iteration. Performance results for this architecture are given in Sect. 3.4.4.

3.4.3 Four-stage Pipelined Implementation

To downsize the Montgomery multiplier even more, a four-stage pipelined version can be implemented. In this case, the number of multiplier blocks is equal to $\lceil \frac{n}{2} \rceil + 2$. The architecture of this Montgomery multiplier is depicted in Fig. 3.17.

The pipelining schedule for this implementation is shown in Fig. 3.18. The feedback loop is performed in 4 cycles per iteration, which leads to $4+4 \cdot (n+1)+\alpha$ cycles for one Montgomery multiplication, where α is the cycle budget for the final addition. Performance results for this architecture are given in Sect. 3.4.4.

3.4.4 Implementation Results

The architectures described in the previous sections can be implemented with any digit size. Unlike the parallel SOS architecture in Sect. 3.3, this architecture does not need block RAM to complete a Montgomery multiplication. However, because

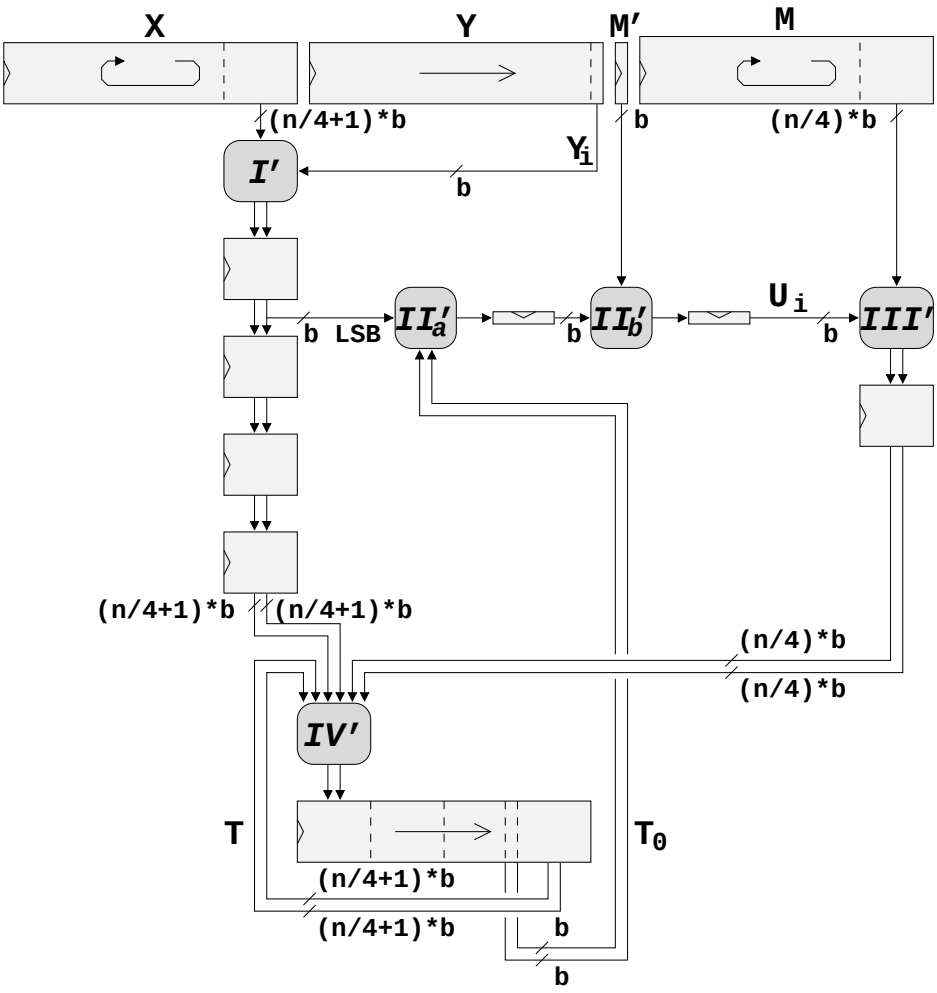

Figure 3.17: Architecture of the four-stage pipelined Montgomery multiplier, where I', II', III' and IV' are a quarter as wide as in Fig. 3.10. The adder for converting the carry-save representation into the final result at the end of the computation is omitted in this figure. Here b LSB denotes the b least significant bits of a signal.

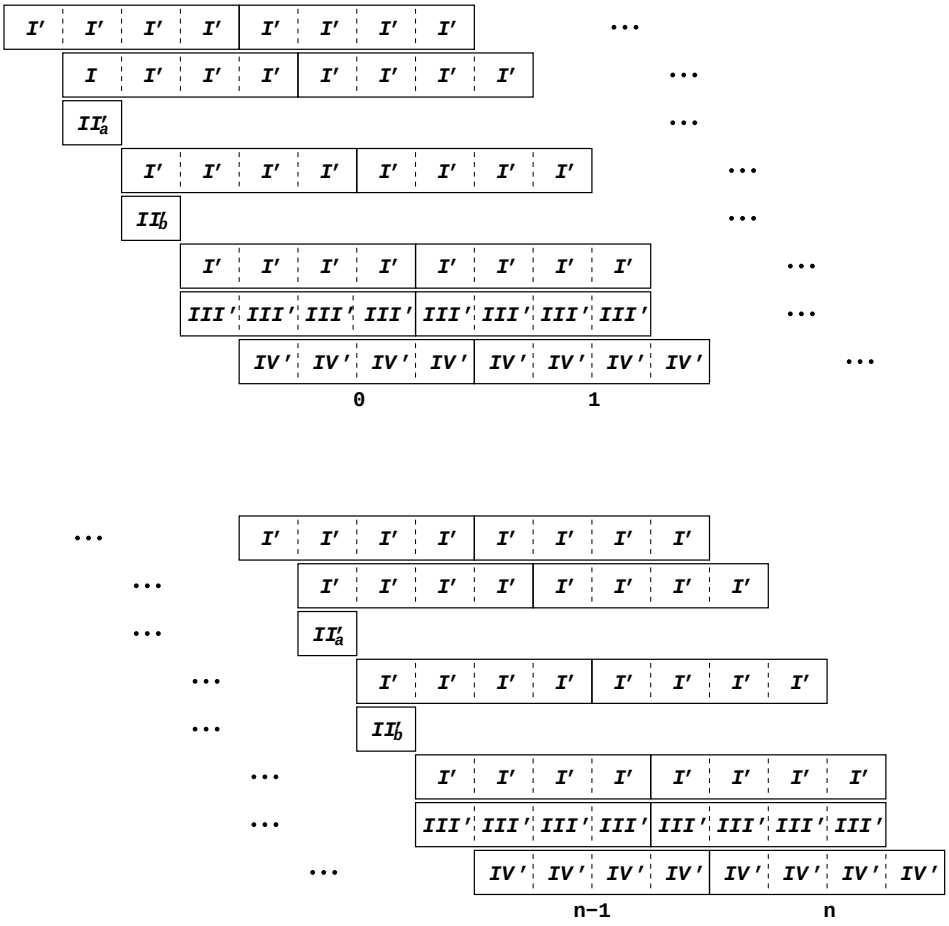

Figure 3.18: Pipelining schedule of the four-stage pipelined Montgomery multiplier. The second, third and fourth lines of I' operations denote the storage of the result of I' in pipelining registers. After the last step in the pipelining schedule, an addition needs to be performed to convert the carry-save representation into the final result. This addition is omitted in this schedule.

the implementation needs to be compatible with the block RAM on the FPGA in order to extend it to a PKC coprocessor, we choose $b = 16$ for the same reason as in Sect. 3.3.

The architectures are implemented on a Xilinx Virtex XC2VP30-7FF1152 FPGA [105]. The results for the improved baseline implementation (i.e., one pipelining stage) and the two- and four-stage pipelined implementations are given in Table 3.2 for several operand sizes and data path widths. The input and output registers, however, need to be at least as wide as the operation performed on the Montgomery multiplier. The adder size in the table is denoted by β, for which $\alpha = \lceil \frac{(n+1)\cdot b}{\beta} \rceil$, where α is the number of cycles needed for the final addition, as mentioned in the previous sections. Although the architecture can be implemented for any operand size, the results do not go further than the four-stage pipelined 2048-bit Montgomery multiplication. The reason is that bigger implementations would exceed the available area of 13696 slices on the XC2VP30 FPGA. The flexibility of the Montgomery multiplier w.r.t. bigger operand sizes is discussed in the last paragraph of Sect. 4.2.2.

For every operand size, the three presented architectures are inserted in the table. It is obvious that the critical path decreases when the number of pipelining stages increases. However, the critical path cannot be divided into two or four equal parts, because registers can only be positioned in between multiplier and adder blocks. This results in an imbalance in the paths between the registers, resulting in an increase of the multiplication time as a function of the downsizing factor, which is equal to the number of pipelining stages. This is shown in Fig. 3.19.

As can be seen from the table, the architectures with the same number of pipelining stages have critical paths of more or less equal lengths. The only reason for the critical path to be longer for larger operand sizes, is the routing overhead on the FPGA. The critical path as a function of the operand size and the downsizing factor is given in Fig. 3.20.

When comparing the architectures with the same operand sizes, we can see that a reduction of the number of multiplier blocks, together with the insertion of pipelining stages, does not give rise to a drastic reduction of the number of slices. This means that the majority of the occupied slices contain registers.

Because the CIOS-based architecture clearly outperforms the implementation based on the parallelized SOS method, comparison to previously designed architectures in terms of speed and area is only performed with the CIOS-based architecture.

Table 3.2: Implementation results for different numbers of pipelining stages and operand sizes x, where t_x denotes the latency for an x-bit Montgomery multiplication. The parameters b and n are according to Alg. 6.

max. operand size x	160 ($b = 16$) ($n = 10$)			256 ($b = 16$) ($n = 16$)			512 ($b = 16$) ($n = 32$)			1024 ($b = 16$) ($n = 64$)			2048 ($b = 16$) ($n = 128$)
# stages = downsizing factor	1	2	4	1	2	4	1	2	4	1	2	4	4
# MULTs	22	12	8	34	18	10	66	34	18	130	66	34	66
# slices	1169	1276	1119	1927	1953	1642	3969	3736	3012	7244	7239	5781	11504
max. freq. (MHz)	74	109	185	73	109	185	64	107	178	64	93	158	130
min. period (ns)	13.5	9.1	5.4	13.6	9.1	5.4	15.5	9.3	5.6	15.5	10.7	6.3	7.7
adder size β	88	88	44	136	68	34	132	88	66	260	104	80	86
# cycles	14	26	52	20	40	80	38	74	145	69	142	277	544
t_{160} (μs)	0.19	0.24	0.28	0.19	0.24	0.28	0.22	0.24	0.29	0.22	0.28	0.33	0.40
t_{256} (μs)				0.27	0.36	0.43	0.31	0.37	0.45	0.31	0.42	0.50	0.61
t_{512} (μs)							0.59	0.69	0.81	0.59	0.79	0.91	1.11
t_{1024} (μs)										1.07	1.52	1.75	2.13
t_{2048} (μs)													4.19

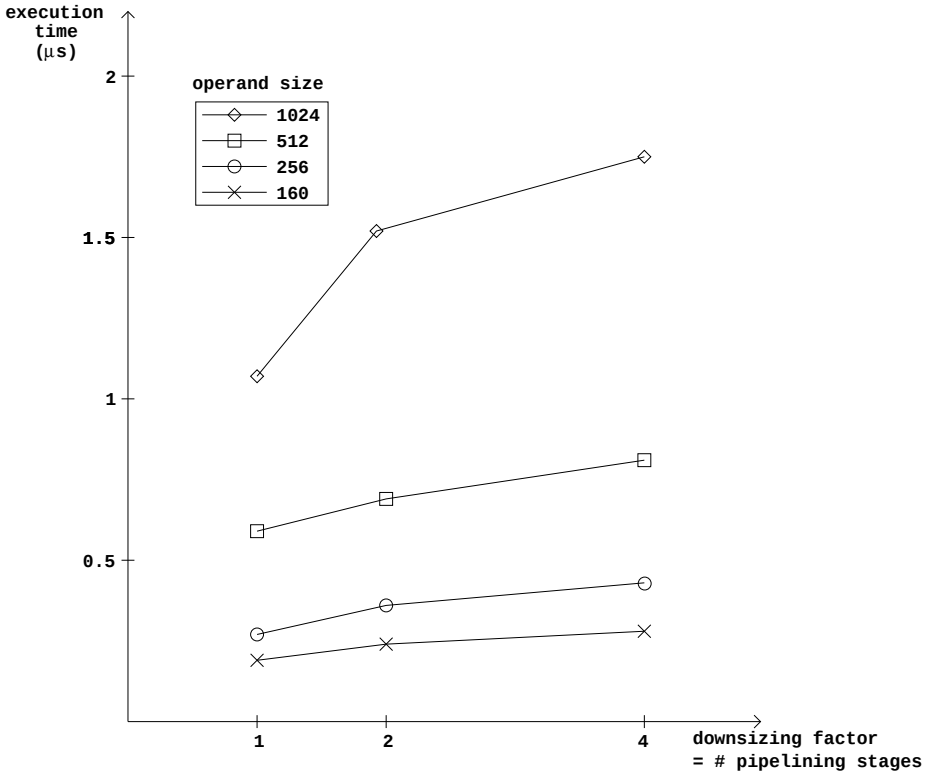

Figure 3.19: Execution time of one Montgomery multiplication as a function of the number of pipelining stages.

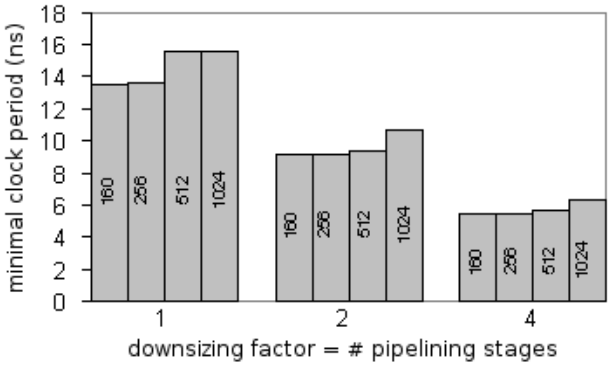

Figure 3.20: Minimal clock period as a function of the operand size and the downsizing factor for our CIOS-based Montgomery multiplier.

3.5 Comparison to Previously Designed Montgomery Multipliers

3.5.1 Previous Work

Because of the importance of modular multiplication in public key cryptosystems, there exists a substantial amount of previous work on the implementation of Montgomery multipliers. In this section, we highlight the most important articles on Montgomery multipliers over GF(p) on FPGAs. However, because many implementations of Montgomery multipliers are extended to public key coprocessors, some of this previous work is repeated in Sect. 4.3.

In [90], Tenca and Koç introduced a scalable radix-2 Montgomery multiplier. Orlando and Paar implemented a higher radix Montgomery multiplier over GF(p) using Booth recoding and the pre-computation of frequently used values [67]. A scalable systolic array was implemented by Batina *et al.* in [6].

Many recently designed hardware implementations of Montgomery multipliers use carry-save adders in order to speed up the computation. This approach is used by Bunimov *et al.* in [13]. They use 3-2 carry-save logic in combination with a look-up table and a bit-per-bit evaluation of one of the operands. A practical FPGA implementation of this design is given in [3]. Manochehri and Pourmozafari introduce pipelining inside the CSA logic in [50].

In [53], McIvor *et al.* give a comparison of the algorithms presented in [41]. Their hardware architectures are using the dedicated multipliers and the efficient carry-lookahead logic embedded in Virtex-II-Pro FPGAs. The implementation results are given for 128-bit and 256-bit Montgomery multipliers. In [38], Kelley and Harris use processing elements consisting of two $w \cdot v$-bit multipliers, two

3-2 carry-save adders and one $w + v$ carry-propagate adder to obtain a scalable Montgomery multiplier.

3.5.2 Comparison of Implementation Results

Table 3.3 compares our CIOS-based architecture to the fastest of the implementations mentioned above. Note that these results are based on our Montgomery multiplier extended with an interface to the user (e.g. a microprocessor) and to a RAM bank of the same width as the operand size, similar to the RAM bank in Fig. 3.7. Moreover, the Montgomery multiplier can be configured as two independent parallel halves, leading to one more dedicated multiplier block for the computation of $X \cdot Y_i$ and another one for the computation of the second U_i. Therefore the area is slightly bigger than the area reported in Table 3.2. The maximum frequency slightly decreases because of the area increase, which makes the latency of the operations somewhat larger. As can be seen from the table, the latency of a Montgomery multiplication performed on our Montgomery multiplier is smaller than on previously designed Montgomery multipliers on FPGA with a comparable number of dedicated multiplier blocks and slices. Especially the comparison to the Montgomery multiplier presented in [38] is interesting, because this is the fastest published Montgomery multiplier on FPGA. The table shows that our solution outperforms this implementation in terms of speed. The area comparison is harder to evaluate, because the area reported in [38] does not seem to include control logic nor an interface to the user. We assume that the reported RAM denotes the distributed RAM in the FPGA. Finally, the table also shows the implementation results for the CIOS algorithm reported in the original paper of Koç *et al.* [41]. It is clear that moving the parallelized algorithm to hardware results in a substantial speed-up.

3.6 Conclusions

This section presented two architectures for Montgomery multiplication, optimized for implementation on an FPGA. The architectures are based on the Separated Operand Scanning (SOS) and the Coarsely Integrated Operand Scanning (CIOS) methods. Because these algorithms are targeting a single w-bit data path, we parallelized them in order to be more suitable for FPGA implementation. The implementation results show that the CIOS-based architecture outperforms the SOS-based architecture. That is why we chose our CIOS-based solution to compare to the performance of previously designed Mongomery multipliers on FPGAs. The results show that our solution leads to the fastest published Montgomery multiplier on FPGA.

In order to facilitate the design procedure, a better design method could have been chosen by starting from a higher-level language such as GEZEL [83]. Since

Table 3.3: Comparison of our implementation to previously designed FPGA implementations of Montgomery multipliers, where our-x-y denotes our x-bit Montgomery multiplier with a downsizing factor of y and SSFs stands for the number of slice flip-flops.

reference	max. freq. (MHz)	area	FPGA	t_{Mont} (μs)			throughput ($Mbps$)		
				160	256	1024	160	256	1024
our-160-1	72	24 MULTs + 1403 slices = 24 MULTs + 2178 LUTs + 1344 SFFs	XC2VP30	0.20			800		
our-160-4	183	10 MULTs + 1350 slices = 10 MULTs + 2350 LUTs + 1552 SFFs	XC2VP30	0.28			571		
our-256-1	67	36 MULTs + 2216 slices = 36 MULTs + 3473 LUTs + 2016 SFFs	XC2VP30	0.21	0.29		762	883	
our-256-4	182	12 MULTs + 1858 slices = 12 MULTs + 3261 LUTs + 2272 SFFs	XC2VP30	0.28	0.44		571	582	
our-1024-2	87	68 MULTs + 7944 slices = 68 MULTs + 12845 LUTs + 8448 SFFs	XC2VP30	0.30	0.46	1.62	533	557	632
our-1024-4	152	36 MULTs + 6650 slices = 36 MULTs + 11725 LUTs + 8032 SFFs	XC2VP30	0.34	0.53	1.82	471	483	563
[53]	76	64 MULTs + 4663 slices	XC2VP125		1.22			210	
[54]	76	11617 slices	XC2V3000			13.11			78
[38]	135	32 MULTs + 2593 LUTs + 5k RAM	XC2V2000		0.39	2.4		671	427
[38]	135	8 MULTs + 695 LUTs + 5k RAM	XC2V2000		0.68	8.3		385	123
[41]	60		Pentium-60			799			1.3

GEZEL foresees library blocks for arithmetic operations, our data path, with "black box" multipliers and adders, is particularly suitable to be modeled using GEZEL. Building a GEZEL model as the first step in the design procedure, would have facilitated the design iterations afterwards.

Chapter 4

Secure and Efficient Implementation of Public Key Coprocessors on FPGAs

This chapter starts with introducing the most popular algorithms for public key cryptography in Sect. 4.1. Besides an overview of implementation options for public key standards, in particular digital signature standards (Sect. 4.1.1), this first section also elaborates on side channel security at the algorithmic level (Sect. 4.1.2). A hierarchy of instructions is constructed in Sect. 4.2 for the implementation of a programmable coprocessor that can handle public key algorithms, but also the underlying algorithms. Section 4.3 compares our design with previously designed public key coprocessors. Finally, Sect. 4.4 concludes this chapter.

4.1 Implementation Options for Public Key Algorithms

The first part of this section elaborates on the implementation of algorithms for digital signatures, while the second part discusses countermeasures for side channel attacks at the algorithmic level.

4.1.1 Implementation of Algorithms for Digital Signatures

DSA and RSA

The most important operation in the Digital Signature Algorithm (DSA) and RSA is modular exponentiation. A straightforward way to compute a modular expo-

nentiation is the binary method a.k.a. the square-and-multiply method. In this algorithm, the exponentiation is computed by consecutive squarings and multiplications based on a bit-per-bit evaluation of the exponent. The left and the right side of Alg. 9 show the left-to-right and the right-to-left variants of this method, respectively, which differ in the order the exponent bits are evaluated.

Whereas the right-to-left algorithm allows the parallelization of the squaring and the multiplication, these operations need to be performed consecutively in the left-to-right algorithm. Moreover, the right-to-left algorithm needs two registers for the storage of intermediate results, while the left-to-right algorithm only needs one. When hardware resources are limited, the left-to-right algorithm is chosen and a single multiplier is implemented to perform the squaring as well as the multiplication. In this case, a less time-consuming algorithm is the k-ary method, which evaluates k bits at a time, as shown in Alg. 10. The speed-up is realized by the pre-computation of some powers of c, as given in Alg. 11. The total number of modular multiplications for one modular exponentiation is equal to $(2^k - 2) + (k + 1) \cdot b$. The pre-computation phase consists of $(2^k - 2)$ multiplications and the evaluation phase consists of $k \cdot b$ squarings and b multiplications, which comes down to $(k + 1) \cdot b$ multiplications when squaring is performed on the multiplier.

RSA encryption involves the exponentiation of the message m to the power e, where e is the public exponent. This exponent is usually chosen to have a small size and a low Hamming weight, which speeds up the exponentiation algorithm substantially. RSA decryption, however, consists of an exponentiation to the power d, where d is the private exponent. In order for the RSA system to be secure, this value cannot be chosen to have a small size and the exponentiation used for decryption is therefore much more time-consuming than the exponentiation used for encryption.

In 1982, Quisquater and Couvreur found a way to speed up RSA decryption by using the Chinese Remainder Theorem (CRT) [71]. This requires 3 parameters to be stored together with the private key:

1. $d_p = d \mod (p - 1)$;
2. $d_q = d \mod (q - 1)$;
3. $q_{inv} = (1/q) \mod p$, with $p > q$.

The CRT algorithm is given in Alg. 12. The first two steps are the most time-consuming, because they consist of many consecutive multiplications. The exponents d_p and d_q are half the size of d. This approximately leads to a reduction of the number of multiplications by half, when the hardware allows to perform the exponentiations in parallel. For our Montgomery multiplier presented in Chapter 3, the multiplication time is linear w.r.t. the size, resulting in another reduction of the computation time by a factor 2 because the moduli are half the size of n. Note that c needs to be reduced modulo p and q before performing the modular

Algorithm 9 Left-to-right binary modular exponentiation (left) and right-to-left binary modular exponentiation (right).

Require: n, c,
$d = (d_{b-1}d_{b-2}\dots d_1d_0)_2$
Ensure: $c^d \mod n$
1: $a \leftarrow 1$
2: **for** i from $b-1$ down to 0 **do**
3: $a \leftarrow a^2 \mod n$
4: **if** $d_i = 1$ **then**
5: $a \leftarrow (a \cdot c) \mod n$
6: **end if**
7: **end for**
8: Return a

Require: n, c,
$d = (d_{b-1}d_{b-2}\dots d_1d_0)_2$
Ensure: $c^d \mod n$
1: $a \leftarrow 1$, $s \leftarrow c$
2: **for** i from 0 to $b-1$ **do**
3: **if** $d_i = 1$ **then**
4: $a \leftarrow (a \cdot s) \mod n$
5: **end if**
6: $s \leftarrow s^2 \mod n$
7: **end for**
8: Return a

Algorithm 10 k-ary algorithm for modular exponentiation

Require: n, $c_i = c^i$ $(i = 0 \dots 2^b - 1)$,
$d = (d_{b-1}d_{b-2}\dots d_1d_0)_{2^k}$
Ensure: $c^d \mod n$
1: $a \leftarrow 1$
2: **for** i from $b-1$ down to 0 **do**
3: $a \leftarrow a^{2^k} \mod n$
4: $a \leftarrow (a \cdot c_{d_i}) \mod n$
5: **end for**
6: Return a

Algorithm 11 Pre-computation for k-ary modular exponentiation

Require: n, k, c
Ensure: $c_i = c^i \mod n$, with $i = 0 \dots 2^k - 1$
1: $c_0 \leftarrow 1$
2: **for** i from 1 to $2^k - 1$ **do**
3: $c_i \leftarrow (c_{i-1} \cdot c) \mod n$
4: Return c_i
5: **end for**

exponentiations in this case. In total, a speed-up of up to 4 can be observed, compared to modular exponentiation to the power d without the use of CRT.

Algorithm 12 RSA decryption using the Chinese Remainder Theorem

Require: p, q, d_p, d_q, q_{inv}, c,
Ensure: c^d
1: $m1 \leftarrow c^{d_p} \mod p$
2: $m2 \leftarrow c^{d_q} \mod q$
3: $h \leftarrow (q_{inv} \cdot (m1 - m2)) \mod p$
4: $m \leftarrow m2 + h \cdot q$
5: Return m

ECDSA

The most important operation in the Elliptic Curve Digital Signature Algorithm (ECDSA), as described in Chapter 1, is elliptic curve point multiplication, which consists of consecutive point additions and point doublings. Before elaborating on point multiplication, we discuss the implementation options for point addition and point doubling.

As can be seen from Eqs. (1.1) and (1.2), point addition consists of 5 multiplications, 2 addition/subtractions and 1 inversion in GF(p). Point doubling consists of 3 multiplications, 8 addition/subtractions and 1 inversion in GF(p), i.e., when the multiplications with 2 and 3 are performed as 1 and 2 additions, respectively. Because finite field inversion is a time and/or area consuming operation, a conversion to another coordinate system is often employed. In the above equations, affine coordinates are used. To avoid a finite field inversion in each point addition and doubling, a conversion to projective coordinates can be made. Although there exist many types of projective coordinates, we only focus on modified Jacobian coordinates, which are shown to be very efficient by Cohen *et al.* in [16]. In this case, a point is converted to a projective representation using four coordinates: (X, Y, Z, aZ^4), with $Z \neq 0$ and a the curve parameter. The conversion is computed as follows

$$\begin{aligned} X &= x \cdot Z^2 \,, \\ Y &= y \cdot Z^3 \,, \\ aZ^4 &= a \cdot Z^4 \,. \end{aligned} \tag{4.1}$$

and the curve equation is transformed into

$$Y^2 = X^3 + aXZ^4 + bZ^6 \,.$$

Conversion back to affine coordinates is done as follows:

$$x = \frac{X}{Z^2}\,, \qquad y = \frac{Y}{Z^3}\,. \tag{4.2}$$

The algorithms for point addition and point doubling in modified Jacobian coordinates are given in Alg. 13. In order for Alg. 13 to be performed on a data path with one modular multiplier and one modular adder, which can only be used separately, the algorithms are rewritten into Alg. 14. From these steps, we can derive that a point addition consists of 14 multiplications and 7 additions/subtractions in GF(p). A point doubling consists of 8 multiplications and 14 additions/subtractions. These algorithms contain many more finite field multiplications and additions/subtractions than their affine versions (Eqs. (1.1) and (1.2)). However, because they do not contain finite field inversions, their performance is, in most implementations, better than the performance of the affine algorithms. The modular additions/subtractions in Alg. 14 can be performed using Alg. 15. Note that this algorithm maintains the Montgomery representation, i.e., when the inputs are in Montgomery representation, the output is also in Montgomery representation. The algorithm is time-constant and thus resistant against timing attacks [42].

Algorithm 13 Algorithms for point addition and point doubling in modified Jacobian coordinates [16]. All operations are performed in the field GF(p).

Require: $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$,
$P_2 = (X_2, Y_2, Z_2, aZ_2^4)$
Ensure: $P_1 + P_2$
1: $U_1 = X_1 \cdot Z_2^2$
2: $U_2 = X_2 \cdot Z_1^2$
3: $S_1 = Y_1 \cdot Z_2^3$
4: $S_2 = Y_2 \cdot Z_1^3$
5: $H = U_2 - U_1$
6: $r = S_2 - S_1$
7: $X_3 = -H^3 - 2 \cdot U_1 \cdot H^2 + r^2$
8: $Y_3 = -S_1 \cdot H^3 + r \cdot (U_1 \cdot H^2 - X_3)$
9: $Z_3 = Z_1 \cdot Z_2 \cdot H$
10: $aZ_3^4 = a \cdot Z_3^4$
11: Return $P_3 = (X_3, Y_3, Z_3, aZ_3^4)$

Require: $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$
Ensure: $2P_1$
1: $S = 4 \cdot X_1 \cdot Y_1^2$
2: $U = 8 \cdot Y_1^4$
3: $M = 3 \cdot X_1^2 + aZ_1^4$
4: $T = -2 \cdot S + M^2$
5: $X_3 = T$
6: $Y_3 = M \cdot (S - T) - U$
7: $Z_3 = 2 \cdot Y_1 \cdot Z_1$
8: $aZ_3^4 = 2 \cdot U \cdot aZ_1^4$
9: Return $P_3 = (X_3, Y_3, Z_3, aZ_3^4)$

Point multiplication of an elliptic curve point P with a scalar k, resulting in the elliptic curve point $Q = kP$, can be computed by consecutive point additions and point doublings. The most straightforward algorithms for point multiplication are, analogous to the binary algorithms for modular exponentiation, the left-to-

Algorithm 14 Multiplication and addition steps for point addition (left) and point doubling (right) in modified Jacobian coordinates.

Require: $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$,
$P_2 = (X_2, Y_2, Z_2, aZ_2^4)$
Ensure: $P_1 + P_2$
1: $T_1 \leftarrow Z_2^2$
2: $T_2 \leftarrow X_1 \cdot T_1$
3: $T_3 \leftarrow Z_1^2$
4: $T_4 \leftarrow X_2 \cdot T_3$
5: $T_1 \leftarrow Z_2 \cdot T_1$
6: $T_4 \leftarrow T_4 - T_2$
7: $T_1 \leftarrow Y_1 \cdot T_1$
8: $T_5 \leftarrow Z_1 \cdot T_3$
9: $T_5 \leftarrow Y_2 \cdot T_5$
10: $T_3 \leftarrow Z_1 \cdot Z_2$
11: $T_5 \leftarrow T_1 - T_5$
12: $Z_3 \leftarrow T_3 \cdot T_4$
13: $T_3 \leftarrow T_4^2$
14: $T_2 \leftarrow T_2 \cdot T_3$
15: $T_3 \leftarrow T_4 \cdot T_3$
16: $T_4 \leftarrow 2 \cdot T_2$
17: $T_6 \leftarrow T_5^2$
18: $T_4 \leftarrow T_3 + T_4$
19: $T_1 \leftarrow T_1 \cdot T_3$
20: $X_3 \leftarrow T_6 - T_4$
21: $T_3 \leftarrow Z_3^2$
22: $T_4 \leftarrow T_2 - X_3$
23: $T_4 \leftarrow T_5 \cdot T_4$
24: $T_3 \leftarrow T_3^2$
25: $Y_3 \leftarrow T_4 - T_1$
26: $aZ_3^4 \leftarrow a \cdot T_3$
27: Return $P_3 = (X_3, Y_3, Z_3, aZ_3^4)$

Require: $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$
Ensure: $2P_1$
1: $T_1 \leftarrow Y_1^2$
2: $T_2 \leftarrow 2 \cdot X_1$
3: $T_3 \leftarrow T_1^2$
4: $T_2 \leftarrow 2 \cdot T_2$
5: $T_1 \leftarrow T_1 \cdot T_2$
6: $T_3 \leftarrow 2 \cdot T_3$
7: $T_2 \leftarrow X_1^2$
8: $T_3 \leftarrow 2 \cdot T_3$
9: $T_4 \leftarrow Y_1 \cdot Z_1$
10: $T_3 \leftarrow 2 \cdot T_3$
11: $T_5 \leftarrow T_3 \cdot aZ_1^4$
12: $T_6 \leftarrow 2 \cdot T_2$
13: $T_2 \leftarrow T_6 + T_2$
14: $T_2 \leftarrow T_2 + aZ_1^4$
15: $T_6 \leftarrow T_2^2$
16: $Z_3 \leftarrow 2 \cdot T_4$
17: $T_4 \leftarrow 2 \cdot T_1$
18: $X_3 \leftarrow T_6 - T_4$
19: $T_1 \leftarrow T_1 - X_3$
20: $T_2 \leftarrow T_1 \cdot T_2$
21: $aZ_3^4 \leftarrow 2 \cdot T_5$
22: $Y_3 \leftarrow T_2 - T_3$
23: Return $P_3 = (X_3, Y_3, Z_3, aZ_3^4)$

Algorithm 15 Algorithm for modular addition/subtraction

Require: M, X, Y, $0 \leq X, Y < M$	**Require:** M, X, Y, $0 \leq X, Y < M$
Ensure: $(X + Y) \mod M$	**Ensure:** $(X - Y) \mod M$
1: $T_1 \leftarrow X + Y$	1: $T_1 \leftarrow X - Y$
2: $T_2 \leftarrow T_1 - M$	2: $T_2 \leftarrow T_1 + M$
3: **if** $T_1 \geq 0$ **then**	3: **if** $T_2 \geq 0$ **then**
4: $T \leftarrow T_1$	4: $T \leftarrow T_2$
5: **else**	5: **else**
6: $T \leftarrow T_2$	6: $T \leftarrow T_1$
7: **end if**	7: **end if**
8: Return T	8: Return T

right and the right-to-left binary algorithm, given on the left and the right side of Alg. 16, respectively.

Algorithm 16 Left-to-right binary point multiplication (left) and right-to-left binary point multiplication (right).

Require: P, $\mathcal{O}$, $k = (k_{l-1}k_{l-2}\ldots k_1k_0)_2$	**Require:** P, $\mathcal{O}$, $k = (k_{l-1}k_{l-2}\ldots k_1k_0)_2$
Ensure: kP	**Ensure:** kP
1: $Q \leftarrow \mathcal{O}$	1: $Q \leftarrow \mathcal{O}, S \leftarrow P$
2: **for** i from $l-1$ to 0 **do**	2: **for** i from 0 to $l-1$ **do**
3: $Q \leftarrow 2Q$	3: **if** $k_i = 1$ **then**
4: **if** $k_i = 1$ **then**	4: $Q \leftarrow Q + S$
5: $Q \leftarrow Q + P$	5: **end if**
6: **end if**	6: $S \leftarrow 2S$
7: **end for**	7: **end for**
8: Return Q	8: Return Q

Similar to modular exponentiation, the left-to-right algorithm has to be performed sequentially, while the right-to-left algorithm can perform point doubling and point addition in parallel. Because the Montgomery multipliers presented in Chapter 3 can be configured to perform two Montgomery multiplications in parallel, we opt for the right-to-left algorithm in our public key coprocessor. Because we only foresee the possibility to perform two Montgomery multiplications or two modular additions/subtractions in parallel, Alg. 14 is rewritten in order to have modular multiplications and modular additions/subtractions in the same steps for point addition and point doubling. Moreover, the results of the corresponding steps need to be stored in the same memory location in the RAM, because we are targeting a RAM bank as depicted in Fig. 3.7, which uses the same address value

for all its internal blocks. This results in Alg. 17, which consists of 19 modular multiplications and 9 modular additions/subtractions. Note that the increase in the number of intermediate registers, compared to Alg. 14, is not of great importance, because the RAM bank has a depth of 1024, which is more than sufficient for the implementation of our public key coprocessor.

For the sake of completeness, we elaborate on the case when only one finite field arithmetic unit is available with practically no restriction on the number of intermediate registers. In this case, the left-to-right algorithm can be sped up by using the k-ary algorithm for elliptic curve point multiplication. This algorithm is analogous to Alg. 10 for which the pre-computation phase is given in Alg. 11. However, for elliptic curves, it is more efficient to represent the key in a Non-Adjacent Form (NAF), as introduced by Reitwiesner in [73]. NAF is a signed-digit representation in which non-zero values cannot be adjacent. After conversion to NAF (Alg. 18), a point multiplication can be performed by evaluating every second key value. The left and the right side of Alg. 19 show this algorithm in a left-to-right and a right-to-left variant, respectively. Because the non-zero values can be 1 or -1, this algorithm does not only consist of point doublings and additions, but also of the inverse operation of point addition, i.e., point subtraction. In GF(p), a point subtraction can be calculated by using the observation that $-P(x, y) = Q(x, -y)$. A NAF-based algorithm for modular exponentiation would require a modular inversion as the inverse operation of modular multiplication. Because modular inversion is a very costly operation, conversion of the exponent to NAF is not used for DSA and RSA.

In order to improve point multiplication based on a NAF-recoded key, the conversion to NAF can be optimized by imposing more than one zero in between two non-zero key values. This representation is called wNAF, where $w - 1$ is the number of zeros in between each pair of non-zero values [10, 87]. In general, the average fraction of non-zero bits after wNAF recoding is equal to $1/(w + 1)$. This value corresponds to the average number of modular multiplications. The number of modular squarings is always equal to the number of bits in the exponent. Therefore, when multiplication and squaring are both performed on the modular multiplier, wNAF recoding leads to a speed-up factor of

$$\frac{1 + \frac{1}{2}}{1 + \frac{1}{w+1}}.$$

However, as can be seen from Alg. 19, point multiplication using a NAF-recoded scalar is not time-constant and therefore susceptible to SPA attacks.

As can be seen in Alg. 18, the conversion to NAF is done from LSB to MSB, i.e., from right to left. For point multiplication there are two possibilities for the evaluation of the key values: from left to right (on the left side of Alg. 19) and from right to left (on the right side of Alg. 19). Because the right-to-left algorithm needs two memory locations for the storage of Q and S, while the left-to-right algorithm

Algorithm 17 Multiplication and addition steps for point addition (left) and point doubling (right) in modified Jacobian coordinates, where modular multiplications and modular additions/subtractions are performed in corresponding steps.

Require: $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$, $P_2 = (X_2, Y_2, Z_2, aZ_2^4)$
Ensure: $P_1 + P_2$

1: $T_1 \leftarrow Z_2^2$
2: $T_2 \leftarrow X_1 \cdot T_1$
3: $T_3 \leftarrow Z_1^2$
4: $T_4 \leftarrow X_2 \cdot T_3$
5: $T_1 \leftarrow Z_2 \cdot T_1$
6: $T_4 \leftarrow T_4 - T_2$
7: $T_7 \leftarrow Y_1 \cdot T_1$
8: $T_5 \leftarrow Z_1 \cdot T_3$
9: $T_6 \leftarrow Y_2 \cdot T_5$
10: $T_3 \leftarrow Z_1 \cdot Z_2$
11: $T_5 \leftarrow T_1 - T_6$
12: $T_9 \leftarrow T_3 \cdot T_4$
13: $T_{16} \leftarrow T_4^2$
14: $T_{10} \leftarrow T_1 + T_2$ (dummy)
15: $T_{10} \leftarrow T_3 + T_4$ (dummy)
16: $T_2 \leftarrow T_2 \cdot T_{16}$
17: $T_{11} \leftarrow T_4 \cdot T_{16}$
18: $T_4 \leftarrow 2 \cdot T_2$
19: $T_{13} \leftarrow T_5^2$
20: $T_4 \leftarrow T_{11} + T_4$
21: $T_7 \leftarrow T_7 \cdot T_{11}$
22: $T_{14} \leftarrow T_{13} - T_4$
23: $T_{15} \leftarrow T_9^2$
24: $T_3 \leftarrow T_2 - T_{14}$
25: $T_4 \leftarrow T_5 \cdot T_3$
26: $T_{15} \leftarrow T_{15}^2$
27: $T_1 \leftarrow T_4 - T_7$
28: $T_2 \leftarrow a \cdot T_{15}$
29: Return $P_3 = (T_{14}, T_3, T_9, T_2)$

Require: $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$
Ensure: $2P_1$

1: $T_1 \leftarrow Y_1^2$
2: $T_2 \leftarrow 2 \cdot X_1$
3: $T_3 \leftarrow T_1^2$
4: $T_4 \leftarrow 2 \cdot T_2$
5: $T_1 \leftarrow T_1 \cdot T_4$
6: $T_4 \leftarrow 2 \cdot T_3$
7: $T_7 \leftarrow X_1^2$
8: $T_5 \leftarrow 2 \cdot T_4$
9: $T_6 \leftarrow Y_1 \cdot Z_1$
10: $T_3 \leftarrow 2 \cdot T_5$
11: $T_8 \leftarrow T_1 + T_2$ (dummy)
12: $T_9 \leftarrow T_3 \cdot aZ_1^4$
13: $T_{16} \leftarrow 2 \cdot T_7$
14: $T_{10} \leftarrow T_6 + T_7$
15: $T_{10} \leftarrow T_{10} + aZ_1^4$
16: $T_2 \leftarrow T_{10}^2$
17: $Z_{11} \leftarrow 2 \cdot T_6$
18: $T_4 \leftarrow 2 \cdot T_1$
19: $T_{13} \leftarrow T_1^2$ (dummy)
20: $T_4 \leftarrow T_2 - T_4$
21: $T_7 \leftarrow T_8 \cdot T_9$ (dummy)
22: $T_{14} \leftarrow T_1 - T_4$
23: $T_{15} \leftarrow T_{14} \cdot T_{10}$
24: $T_3 \leftarrow 2 \cdot T_9$
25: $T_4 \leftarrow T_5 \cdot T_4$ (dummy)
26: $T_{15} \leftarrow T_3^2$ (dummy)
27: $T_1 \leftarrow T_{15} - T_3$
28: $T_2 \leftarrow T_1 \cdot T_2$ (dummy)
29: Return $P_3 = (T_4, T_1, Z_3, T_3)$

Algorithm 18 Conversion to NAF [73]

Require: $k = (k_{l-1}k_{l-2}\dots k_1k_0)_2$
Ensure: $k = (k_l l_{l-1}\dots k_1k_0)_{NAF}$ with $k_i \in \{-1, 0, 1\}$
1: $c_0 \leftarrow 0$
2: **for** i from 0 to l **do**
3: $\quad c_{i+1} \leftarrow \lfloor (k_i + k_{i+1} + c_i)/2 \rfloor$
4: $\quad s_i \leftarrow k_i + c_i - 2 \cdot c_{i+1}$
5: **end for**
6: Return $(s_l s_{l-1}\dots s_0)$

Algorithm 19 Left-to-right point multiplication (left) and right-to-left point multiplication (right) with a NAF-recoded scalar.

Require: P, $\mathcal{O}$, $k = (k_l k_{l-1}\dots k_1k_0)_{NAF}$
Ensure: kP
1: $Q \leftarrow \mathcal{O}$
2: **for** i from l to 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $i \mod 2 = 0$ **then**
5: $\quad\quad$ **if** $k_i = 1$ **then**
6: $\quad\quad\quad Q \leftarrow Q + P$
7: $\quad\quad$ **else if** $k_i = -1$ **then**
8: $\quad\quad\quad Q \leftarrow Q - P$
9: $\quad\quad$ **end if**
10: $\quad$ **end if**
11: **end for**
12: Return Q

Require: P, $\mathcal{O}$, $k = (k_l k_{l-1}\dots k_1k_0)_{NAF}$
Ensure: kP
1: $Q \leftarrow \mathcal{O}, S \leftarrow P$
2: **for** i from 0 to l **do**
3: $\quad$ **if** $i \mod 2 = 0$ **then**
4: $\quad\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad\quad Q \leftarrow Q + S$
6: $\quad\quad$ **else if** $k_i = -1$ **then**
7: $\quad\quad\quad Q \leftarrow Q - S$
8: $\quad\quad$ **end if**
9: $\quad$ **end if**
10: $\quad S \leftarrow 2S$
11: **end for**
12: Return Q

only needs to store Q, the latter is preferred for memory constrained devices. However, since the conversion to NAF can only be done from right to left, on-the-fly conversion, which would be the most memory-efficient, is impossible. Recently, a more memory-efficient signed digit representation was introduced by Okeya *et al.*: the Mutual Opposite Form (MOF) [66]. The advantage over NAF is that key recoding can be done on-the-fly, even for left-to-right point multiplication.

4.1.2 Side Channel Security

Because FPGA implementations have been repeatedly shown to be susceptible to side channel attacks (see [68, 88, 98]), we have implemented several countermeasures against power analysis attacks. However, as already mentioned in Chapters 1 and 2, side channel resistance can be aimed for at many levels in the design ab-

straction hierarchy. We only focus on the algorithmic level, which leads to a variety of countermeasures, explained in the next paragraphs.

Countermeasures for Simple Power Analysis Resistance

The first step in preventing side channel attacks, in particular power analysis attacks, is implementing countermeasures against Simple Power Analysis (SPA). Using an SPA attack, an adversary tries to extract secret information by analyzing a single power trace of the implementation while it is executing an operation. Starting from the observations made in Sect. 2.2, we assume that the most power is consumed when the value of a signal changes, i.e., we assume that the Hamming distance of the subsequent values of a signal determines the power consumption. The most important action in securing an implementation against SPA attacks, is making the power graphs look indistinguishable, even if different secret information is processed. Therefore, we made sure that all operations are time-constant and that the secret key cannot be derived by observing a single power trace. This includes removing conditional branches.

For RSA, we implemented the k-ary method to speed up modular exponentiation (Alg. 10). When multiplication is not skipped if the evaluated key digit consists of only zeros, this method has no conditional branches. To make the power graphs of the squarings and the multiplications look similar, they are performed using the same multiplier.

For ECC, we inserted dummy operations in the right-to-left point multiplication algorithm (Alg. 16 (right)). This results in a time-constant and power-similar version of the algorithm, which is given in Alg. 20. Just like for RSA, finite field squarings and multiplications are executed on the same multiplier.

Algorithm 20 SPA-resistant right-to-left binary point multiplication.

Require: P, $\mathcal{O}$,
$k = (k_{l-1}k_{l-2}\dots k_1k_0)_2$
Ensure: kP
1: $Q \leftarrow \mathcal{O}, S \leftarrow P$
2: **for** i from 0 to $l-1$ **do**
3: **if** $k_i = 1$ **then**
4: $Q \leftarrow Q + S$
5: **else**
6: $T \leftarrow Q + S$
7: **end if**
8: $S \leftarrow 2S$
9: **end for**
10: Return Q

Key Blinding

To prevent first-order Differential Power Analysis (DPA) attacks [43, 44], it is not sufficient to make the operations time-constant and the power traces indistinguishable. DPA uses many power consumption traces together with some statistical analysis to recover values that are processed in the device. To make it impossible to extract secret values, the values are "blinded". For RSA, the blinded exponent d' is obtained using

$$d' = d + \varphi \cdot r\,, \tag{4.3}$$

where d is the original exponent, r is a random number and φ is Euler's totient function of the modulus n. More details on exponent blinding are given in [42], where it is proven that $g^d = g^{d'} \bmod n$. This property makes it possible to choose a different r for each exponentiation, changing the exponent without changing the intended result. Because DPA is based on power consumption data for many executions of the algorithm using the same key, exponent blinding succeeds in protecting the cryptosystem against first-order DPA attacks.

For ECC, resistance against DPA attacks can be achieved by blinding the scalar used for point multiplication. The blinded scalar k' can be computed as follows:

$$k' = k + \#E \cdot r\,. \tag{4.4}$$

Here, k is the original scalar, $\#E$ is the number of points on the curve and r is a random number (of typically 20 bits [17]). Because kP and $k'P$ always result in the same point on the elliptic curve, this method is effective against first-order DPA attacks when the random number is changed for every execution of the point multiplication.

Point Randomization

In addition to key blinding, we have implemented another countermeasure against DPA attacks on elliptic curve point multiplication: point randomization. This countermeasure exploits the fact that the Z-coordinate can be chosen randomly when using projective coordinates [17]. In our case, this comes down to choosing a different Z-coordinate for each point multiplication during the conversion of the input point P to modified Jacobian coordinates.

Note that the presented countermeasures only prevent first-order DPA attacks. Recently, more advanced attacks have been developed, such as higher-order DPA attacks [56], DPA using advanced stochastic models [84], template attacks [1], collision-based attacks [46, 85] and power analysis attacks based on glitches [49].

4.2 Mastering the Complexity of the Control Logic

Table 4.1 lists the required instructions for our programmable public key coprocessor. Because some algorithms make use of other algorithms, we categorized the instructions in levels, which represent the instruction hierarchy. To provide a flexible solution, we made all the levels in the hierarchy accessible. In this way, additional new cryptographic algorithms can be executed by the user, which makes our public key coprocessor fully programmable. Table 4.1 also shows that some operations can be performed with or without countermeasures, allowing the user to decide on the trade-off between security and performance. Besides the vertical hierarchy in the table, there is also a horizontal classification. This distinguishes between operations of different sizes and between operations that configure the data path either as a whole or as two parallel halves.

Because the Montgomery multipliers discussed in Chapter 3 provide the possibility to use the data path as two parallel halves, a VLIW (Very Long Instruction Word) control architecture would be another option for the implementation of the hierarchy of instructions shown in Table 4.1. However, because the VLIW approach is based on instruction level parallelism, balancing the operations for SPA resistance would be much more complex than in our approach, where parallel operations are built in in a single instruction. Another problem that rises in VLIW control architectures is data contamination in register files. Overcoming this problem makes the implementation of the control logic in a VLIW coprocessor much more complex.

A superscalar approach decides on the parallelism of the instructions by dynamically checking data dependencies [34]. This approach has the same disadvantages as the VLIW approach. Moreover, it needs some extra memory to store a sequence of instructions in order to decide which operations can be executed independently.

Figure 4.1 shows the general architecture of our coprocessor, where the thick and the thin lines are used to represent the data flow and the control flow, respectively. The logic controlling the data path (DP) and data memory (RAM) is divided into two parts, which both consist of Finite State Machines (FSMs):

- The lowest-level FSMs (LL): these FSMs are directly controlling the data path and the data memory. For the SOS- and CIOS-based coprocessors, the LL FSMs execute the operations on level 6 and 5, respectively.

- The higher-level FSMs (HL): to limit the number of multiplexers, these FSMs are not connected to the data path or the data memory. The HL FSMs control the LL FSMs and take care of the user commands. This keeps complexity under control, because the LL functions are reused for different HL functions. For the SOS- and CIOS-based coprocessors, the HL FSMs execute the operations on levels 1-5 and 1-4, respectively.

Table 4.1: Instructions of the public key coprocessor categorized by level.

Level	Instruction	Operation
1	`CRT`	modular exponentiation using CRT
	`CRT_w`	modular exponentiation using CRT with key blinding
	`pmul_w`	ECC point multiplication with key blinding and projective coordinate randomization
2	`expmod_w`	modular exponentiation with key blinding
	`pmul`	ECC point multiplication with projective coordinate randomization
3	`expmod`	modular exponentiation
	`Mexpmod_w`	modular exponentiation with key blinding in Montgomery representation
	`Minvmod`	modular inversion in Montgomery representation
	`Mpaddpdouble`	ECC point addition and doubling in Montgomery representation
SOS 4	`mulmod`	modular multiplication
	`Mexpmod`	modular exponentiation in Montgomery representation
SOS 5	`Mont`	Montgomery multiplication
	`addsubmod`	modular addition/subtraction
SOS 6	`mul`	integer multiplication
	`addsub`	integer addition/subtraction
CIOS 4	`Mexpmod`	modular exponentiation in Montgomery representation
	`mulmod`	modular multiplication
	`addsubmod`	modular addition/subtraction
	`mul`	integer multiplication
CIOS 5	`Mont`	Montgomery multiplication
	`addsub`	integer addition/subtraction

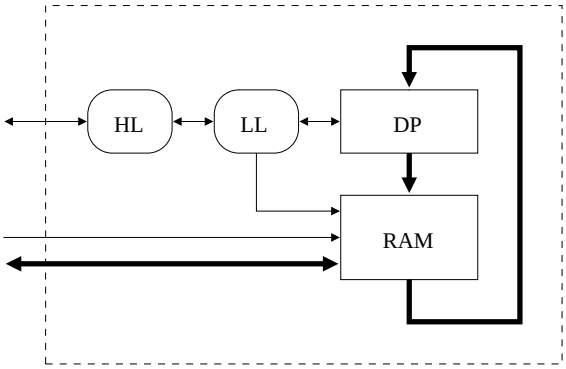

Figure 4.1: Architecture of the public key coprocessor (DP = data path, RAM = data memory, LL = lowest-level FSMs, HL = higher-level FSMs, thick lines = data, thin lines = control).

The data memory is not only accessible by the LL FSMs, but also by the user. Before sending a command to the coprocessor, the user writes the input operands and parameters to predefined addresses in the data memory. After the coprocessor has executed the command, the user reads the result from a predefined address in the data memory.

The data path for the programmable coprocessor contains one of the Montgomery multipliers presented in Chapter 3, i.e., either the SOS-based or the CIOS-based Montgomery multiplier. The architectures of these Montgomery multipliers are constructed in a different way: the SOS-based architecture consists of an integer multiplier and an integer adder/subtractor, while the CIOS-based solution is a specific architecture for Montgomery multiplication. Because of this difference, the instruction hierarchy of the programmable public key coprocessor, that is built on top of the Montgomery multiplier, is not the same for the SOS-based and the CIOS-based data paths. This can also be seen in Table 4.1. In the next subsections, we give more details on the lowest levels in the hierarchy of the SOS- and CIOS-based public key coprocessors.

4.2.1 Instruction Hierarchy for the SOS-based Coprocessor

In this section we clarify the relations between the 6 instruction levels in our SOS-based coprocessor. The one but last and the last paragraph elaborate on the horizontal classification of the operations and the flexibility w.r.t. operations that have a bigger size than the data path width, respectively.

Level 6

As explained in Chapter 3, the data path of our SOS-based Montgomery multiplier consists of an integer multiplier and an integer adder. In addition, the data path of the public key coprocessor needs shift registers for the DSA and RSA key and the ECC key. In Fig. 4.2, the data path, data memory and lowest-level FSMs are depicted in more detail. The two shift registers for the key are denoted by << and >>. Because our coprocessor evaluates an RSA key from MSB to LSB and an ECC key from LSB to MSB, a left-shift (<<) and a right-shift (>>) register are foreseen in the data path. The reason for this difference in key evaluation comes from the fact that ECC uses smaller operand lengths than RSA. This makes it possible to perform an elliptic curve point doubling and addition in parallel without increasing the size of the data path. Point multiplication algorithms evaluating the key from right to left, such as the algorithm on the right side of Alg. 16, inherently allow this kind of parallelism. For RSA, the bigger operand size made us choose for a more memory-efficient left-to-right algorithm, in particular the k-ary method given in Alg. 10. The *LL* FSMs perform an integer multiplication (`mul`) and an integer addition/subtraction (`addsub`). In Table 4.1, the operations that load and shift the key registers are omitted. However, in our coprocessor, these operations are included and taken into account when presenting the implementation results in Sect. 4.3.

Level 5

On the level above the lowest level, the parallelized SOS method, explained in Sect. 3.2.1, performs a Montgomery multiplication (`Mont`) using the level 6 operations `mul` and `addsub`. As explained in Sect. 3.1, this can also be used for conversion to Montgomery representation and conversion back to normal representation. With Alg. 15, the integer adder/subtractor (`addsub`) on level 6 can be used to perform a modular addition/subtraction (`addsubmod`).

Level 4

On level 4, Montgomery multiplication is used to perform a modular multiplication (`mulmod`) following the steps in Eqs. (3.1) and (3.2). Modular exponentiation in Montgomery representation (`Mexpmod`) is also a level 4 operation. It performs Alg. 10 preceded by Alg. 11 (using level 5, `Mont`), without conversion to Montgomery representation before or conversion back to normal representation after the modular exponentiation. This means that the algorithm assumes the input to be in Montgomery representation and also ensures that the result of the modular exponentiation is in Montgomery representation. The modular exponentiation is uses the k-ary algorithm (Algs. 10 and 11). As explained in Sect. 4.1, the number of modular multiplications to be performed is equal to $(2^k - 2) + (k + 1) \cdot b$. Based

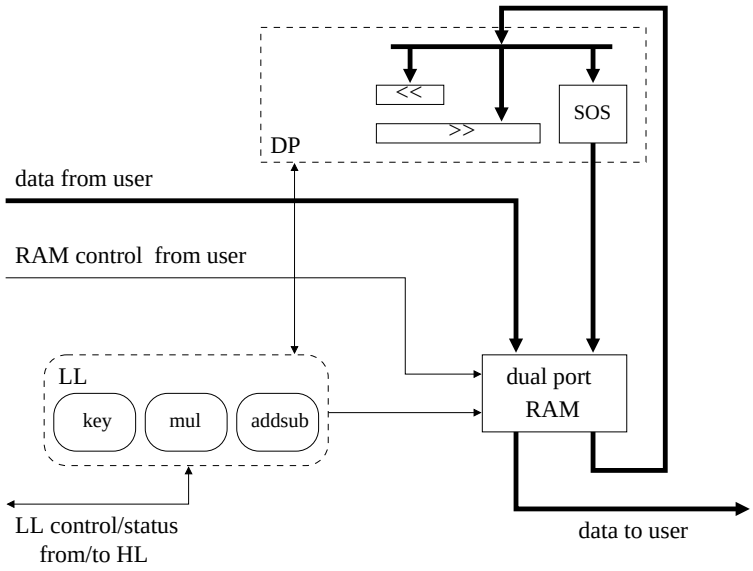

Figure 4.2: Data path (*DP*), RAM and lowest-level FSMs (*LL*) of our SOS-based public key coprocessor.

on Table 4.2, the parameter k can be chosen for both 512-bit and 1024-bit modular exponentiation, where the former can be performed twice in parallel. For 1024-bit modular exponentiation, $k = 6$ leads to the smallest number of multiplications. For 512-bit modular exponentiation k is chosen to be 5. The number of temporary registers needed to store the pre-computed values is not taken into account in this decision, because our RAM bank is sufficiently deep.

Level 3

Modular exponentiation (`expmod`) consists of modular exponentiation in Montgomery representation (level 4, `Mexpmod`), preceded by conversion to Montgomery representation (level 5, `Mont`) and followed by conversion back to normal representation (level 5, `Mont`). Modular exponentiation with key blinding in Montgomery representation (`Mexpmod_w`) can be computed as follows:

- compute Eq. (4.3) using the level 6 operations `mul` and `addsub`;
- compute a modular exponentiation in Montgomery representation with the blinded key as exponent (level 4, `Mexpmod`).

Table 4.2: Number of multiplications as a function of k for 1024-bit and 512-bit exponentiation using the k-ary algorithm (Algs. 10 and 11).

k	512		1024	
	b	$2^k - 2 + (k+1) \cdot b$	b	$2^k - 2 + (k+1) \cdot b$
4	128	654	256	1294
5	103	648	205	1260
6	86	664	171	1259
7	74	718	147	1302

Modular inversion in Montgomery representation (`Minvmod`) is also a level 3 operation. It is performed according to Eq. (2.2) using the following steps:

- compute $p - 2$ using an integer subtraction (level 6, `addsub`);
- compute a modular exponentiation in Montgomery representation with $p-2$ as the exponent (level 4, `Mexpmod`).

Another level 3 operation is `Mpaddpdouble`, which performs an elliptic curve point addition and doubling in parallel, following Alg. 17. Therefore, the level 5 operations `Mont` and `addsubmod` are used.

Level 2

Level 2 contains modular exponentiation with key blinding (`expmod_w`). This operation consists of a conversion to Montgomery representation (level 5, `Mont`), followed by a modular exponentiation with key blinding in Montgomery representation (level 3, `Mexpmod_w`) and a conversion back to normal representation (level 5, `Mont`). Another level 2 operation is elliptic curve point multiplication with randomized projective coordinates (`pmul`). This operation follows the next steps:

- convert the coordinates of the input point, x and y, and the curve parameter, a, to Montgomery representation (level 5, `Mont`);
- convert the affine coordinates of the input point in Montgomery representation to modified Jacobian projective coordinates following Eq. (4.1) (level 5, `Mont`);
- compute the point multiplication following the right side of Alg. 16, using consecutive point additions and doublings in parallel (level 3, `Mpaddpdouble`);

- convert the modified Jacobian projective coordinates of the resulting elliptic curve point to affine coordinates following Eq. (4.2) (level 5, `Mont`; level 3, `Minvmod`);
- convert the affine coordinates of the resulting elliptic curve point from Montgomery to normal representation (level 5, `Mont`).

Level 1

On level 1, modular exponentiation using the CRT algorithm (Alg. 12) is performed (`CRT`). It consists of the following steps:

- reduce c modulo p and modulo q (level 5, `Mont`), where the result is in Montgomery representation;
- compute the first three steps in Alg. 12 (level 5, `Mont` and `addsubmod`; level 4, `Mexpmod`), where the modular exponentiations are performed in parallel;
- convert h from Montgomery to normal representation (level 5, `Mont`);
- compute Step 4 in Alg. 12 (level 6, `mul` and `addsub`).

The first step needs some clarification. Reducing c modulo p or q, where p and q are assumed to be half the size of c, can be done with the parallelized SOS method for Montgomery multiplication, because the algorithm can handle an intermediate result $X \cdot Y$ up to double the length of the modulus M. When $X = c$, $Y = 1$ and $M = p$ or q, this results in a product with the same bounds. However, the results will be the Montgomery product of c and 1, i.e., $(c \cdot R^{-1}) \mod p$ and $(c \cdot R^{-1}) \mod q$. To put c in Montgomery representation, i.e., $c \cdot R$, two extra Montgomery multiplications with R^2 need to be performed. Modular exponentiation with key blinding using the CRT algorithm (`CRT_w`) is performed in a similar way. The only difference with `CRT` is that the modular exponentiations in Alg. 12 are executed by the level 3 operation `Mexpmod_w` instead of the level 4 operation `Mexpmod`. Finally, elliptic curve point multiplication with key blinding and projective point randomization (`pmul_w`) is also a level 1 operation. It is performed according to the following steps:

- compute Eq. (4.4) using the level 6 operations `mul` and `addsub`;
- compute a point multiplication with the blinded key as exponent (level 5, `pmul`).

Horizontal Classification

The horizontal classification for our SOS-based public key coprocessor divides the operations in three main groups:

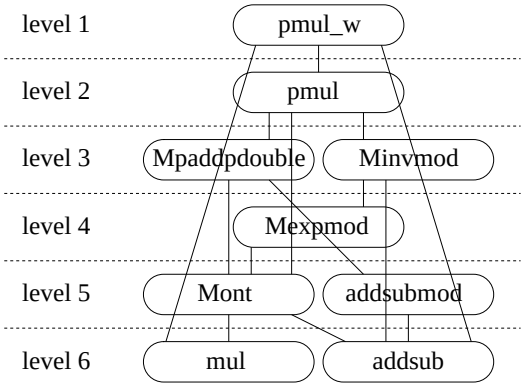

Figure 4.3: Hierarchy for the instructions in the elliptic curve point multiplication group in the SOS-based public key coprocessor.

- operations that are used to establish an elliptic curve point multiplication;
- operations that are used to establish a modular exponentiation;
- operations that are used to establish a modular exponentiation using CRT.

To illustrate the vertical hierarchy in these three groups, Figs. 4.3, 4.4 and 4.5 depict the relations between the operations in the hierarchy. For the point multiplication group, the data path is configured as two parallel halves. In this way, point addition and point doubling can be performed in parallel. Because the RAM bank, depicted in Fig. 3.7, uses the same address for the left and the right half, the results of the corresponding steps in the point addition and point doubling algorithms need to be stored at the same address in the RAM. Alg. 17 satisfies this requirement. For the modular exponentiation group, the data path is configured to perform full length operations. For the CRT group, the data path is configured as two parallel halves in the first two steps of Alg. 12. For the third step, this configuration is maintained, but only half of the data path is used. The fourth step uses the data path in full length configuration to be able to combine the half length intermediate results into the final result of the exponentiation.

Note that `CRT` and `CRT_w` could have also been put at level 2, because the highest-level instruction they rely on is at level 3, while there are no other instructions built upon `CRT` or `CRT_w`. For the same reason, `expmod_w` could have been put on level 1 and `expmod` could have been a level 2 or a level 1 operation.

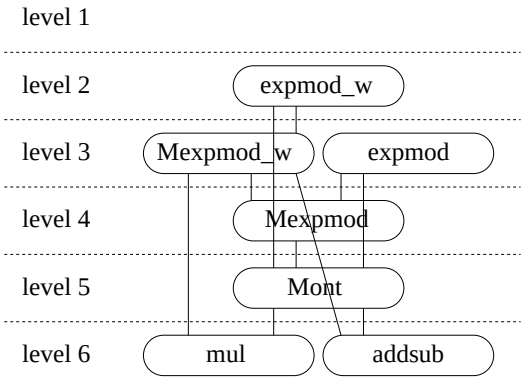

Figure 4.4: Hierarchy for the instructions in the modular exponentiation group in the SOS-based public key coprocessor.

Flexibility w.r.t. the Size of the Operations

To see how the data path is capable of performing operations with a size that is bigger than its width, we need to evaluate both the integer addition and the integer multiplication.

If the size of the input registers in Fig. 3.3 is smaller than the operands to be added, the least significant part of the operands is loaded, added up and stored in the RAM, while the last carry output is stored in the carry feedback register. This carry serves as the input carry for the next, more significant part. In this way, the integer adder has a practically unlimited input size capacity and the latency is linear in the size of the operands.

Without changing anything to the data path, it is also possible to perform an integer multiplication with a size that is bigger than the data path width. This can be done using the approach of Karatsuba [37], which is given in Alg. 21 and depicted in Fig. 4.6 for operand sizes that are double the size of the data path.

4.2.2 Instruction Hierarchy for the CIOS-based Coprocessor

Because the data path of the CIOS-based coprocessor performs a Montgomery multiplication without using a separate integer multiplier and adder/subtractor, the lowest levels of the hierarchy are different from the SOS-based architecture. To be able to perform an integer addition/subtraction, the adder at the output of the CIOS-based Montgomery multiplier, which is used to convert from the carry-save form into the normal form, is made accessible from the input registers. An integer

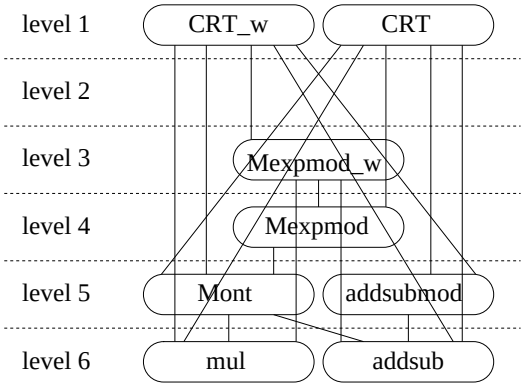

Figure 4.5: Hierarchy for the instructions in the CRT group in the SOS-based public key coprocessor.

Algorithm 21 Karatsuba multiplication

Require: $A = A_1 \cdot 2^{n/2} + A_0$, $B = B_1 \cdot 2^{n/2} + B_0$
with $0 \leq A, B < 2^n$ and $0 \leq A_1, A_0, B_1, B_0 < 2^{n/2}$
Ensure: $A \cdot B$
1: $T_{00} \leftarrow A_0 \cdot B_0$
2: $T_{01} \leftarrow A_1 \cdot B_0$
3: $T_{10} \leftarrow A_0 \cdot B_1$
4: $T_{11} \leftarrow A_1 \cdot B_1$
5: $T_0 \leftarrow T_{00} + T_{01} \cdot 2^{n/2}$
6: $T_1 \leftarrow T_{10} + T_{11} \cdot 2^{n/2}$
7: $T \leftarrow T_0 + T_1 \cdot 2^{n/2}$
8: Return T

multiplication can be performed using the following observation:

$$((a \cdot R) \cdot b \cdot R^{-1}) \mod 0 = a \cdot b\,. \tag{4.5}$$

If $a \cdot R$ is performed as an integer multiplication, with R a power of 2, it comes down to a simple left-shift operation over $\log_2 R$ positions. If R is chosen to be of the same size as a, the number of bits of the product can be up to the double of the number of bits of a. Using a Montgomery multiplier of double size of a, an integer multiplication with b, which is of the same size as a, can be performed when the modulus is chosen to be zero and the number of iterations in Alg. 6 is cut by half (to make sure the Montgomery multiplication is completed with a parameter R that is of the size of a). In order to perform an integer multiplication of the full

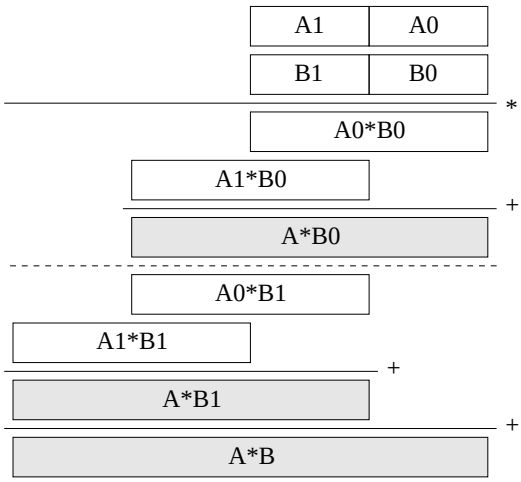

Figure 4.6: Schematic representation of the Karatsuba algorithm for integer multiplication of A and B, where the least significant halves of A and B are denoted by A_0 and B_0, respectively, and the most significant halves are denoted by A_1 and B_1. The final step adds the two previous intermediate results, indicated with grey boxes.

length of the Montgomery multiplier, the Karatsuba method can be used, as given in Alg. 21.

Although the operations on the three highest levels (1, 2, and 3) in Table 4.1 are the same for the SOS-based and the CIOS-based coprocessor, the sequence of the invoked sub-operations is not exactly the same. The reason for this is that the reduction operation that is needed for modular exponentiation using CRT, cannot be performed in the same way as for the SOS-based coprocessor. If we would use the CIOS-based Montgomery multiplier with $X = c$, $Y = 1$ and p or q the modulus of half the size of c, we would need a Montgomery multiplier of the size of c with a modulus of half the size. This would reduce c modulo p or q, but the R that would appear in the reduced values $(c \cdot R^{-1}) \mod p$ and $(c \cdot R^{-1}) \mod q$, would be of the size of c instead of of the size of p and q. However, we need the Montgomery reduced value of c w.r.t. a parameter R that is of the size of p and q in order to be able to perform two modular exponentiations of this size in parallel. To solve this problem, we perform the parallelized SOS method on our CIOS-based data path, i.e., we use integer multiplications and additions to perform the reduction needed for modular exponentiation using CRT. The reduction step follows the SOS method, explained in Sect. 4.2.1, where the procedure for an integer multiplication on a CIOS-based Montgomery multiplier is based on Eq. (4.5) and the Karatsuba

method (Alg. 21).

We use these observations to discuss the instruction hierarchy of our CIOS-based coprocessor in the following paragraphs. As shown in Table 4.1, the two lowest levels (4 and 5) are different from the SOS hierarchy. However, because of the reduction problem discussed above, some level 1 FSMs also differ from the SOS-based coprocessor. That is why we elaborate on level 5, 4 and 1 in the following paragraphs. The last paragraph discusses the flexibility of the CIOS architecture w.r.t. operations for which the operand size is bigger than the data path width.

Level 5

At the lowest level, a Montgomery multiplication and an integer addition/subtraction are performed (`Mont` and `addsub`, respectively). The architecture is shown in Fig. 4.7. Similar to the SOS-based architecture, key registers are needed for the evaluation of the keys in elliptic curve point multiplication and modular exponentiation. The level 5 FSM controlling these registers is not discussed here.

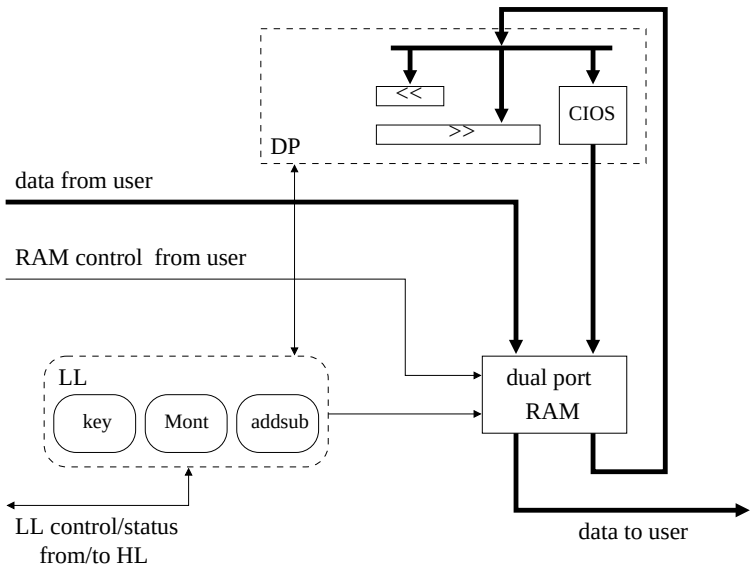

Figure 4.7: Data path, RAM and lowest-level FSMs of our CIOS-based public key coprocessor.

Level 4

The level above the lowest level performs a modular exponentiation in Montgomery representation (`Mexpmod`) based on `Mont` at level 5. A modular multiplication (`mulmod`) is performed using Eqs. (3.1) and (3.2) (level 5, `Mont`). Following Alg. 15, a modular addition/subtraction (`addsubmod`) is computed (using level 5, `addsub`). Finally, integer multiplication is also a level 4 operation. It is computed according to Eq. (4.5) and the Karatsuba method (Alg. 21) (level 5, `Mont` and `addsub`).

Level 1

Because only `CRT` and `CRT_w` are different from level 1 in Sect. 4.2.1, we only discuss these operations. A modular exponentiation using CRT is performed according to the following steps:

- reduce c modulo p and modulo q with the SOS method on the CIOS data path, as explained above (level 4, `mul`, level 5, `addsub`), where the result is in Montgomery representation;
- compute the first three steps in Alg. 12 (level 5, `Mont` and `addsubmod`; level 4, `Mexpmod`), where the modular exponentiations are performed in parallel;
- convert h from Montgomery to normal representation (level 5, `Mont`);
- compute Step 4 in Alg. 12 (level 4, `mul`; level 5, `addsub`).

Modular exponentiation using CRT with key blinding follows the same steps. The only difference is that `Mexpmod_w` is used for the parallel modular exponentiations in Montgomery representation instead of `Mexpmod`.

Horizontal Classification

Because of the difference in the vertical hierarchy between the SOS- and the CIOS-based coprocessors, the three main groups in the horizontal classification, i.e., point multiplication, modular exponentiation and modular exponentiation using CRT, also consist of a different hierarchy tree. This is depicted in Fig. 4.8, Fig. 4.9 and Fig. 4.10, respectively.

Note that `CRT` and `CRT_w` could have also been put on level 2, because the highest-level instruction they rely on is at level 3, while there are no other instructions built upon `CRT` or `CRT_w`. For the same reason, `expmod_w` could have been put on level 1 and `expmod` could have been a level 2 or a level 1 operation.

Flexibility w.r.t. the Size of the Operations

Although the data path for the CIOS-based Montgomery multiplier can be downsized by any arbitrary factor, where the resolution is determined by the 16-bit

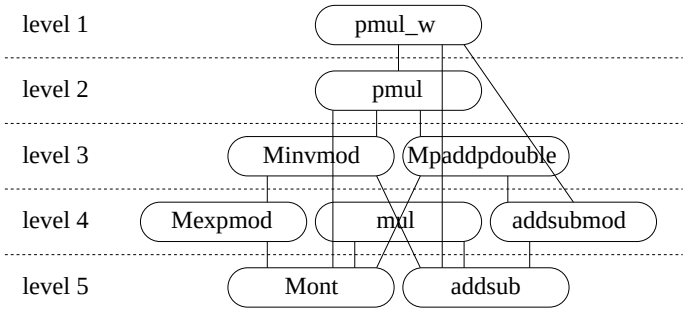

Figure 4.8: Hierarchy for the instructions in the elliptic curve point multiplication group in the CIOS-based public key coprocessor.

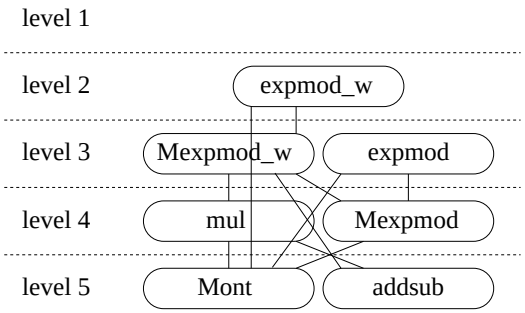

Figure 4.9: Hierarchy for the instructions in the modular exponentiation group in the CIOS-based public key coprocessor.

dedicated integer multipliers on the FPGA, the shift register that stores the result of the feedback loop, denoted by T in Alg. 6, needs to be of the full size of the operation. When settling for a high communication overhead with the RAM, the input shift registers could be made smaller than the operand size, but this would not take away the restriction on the size of T, which makes the CIOS-based Montgomery multiplier less flexible than the SOS-based Montgomery multiplier.

4.2.3 Comparison of the SOS- and CIOS-based Coprocessors

To make a fair comparison between our SOS-based and CIOS-based coprocessors, we present the implementation results for both architectures on the same FPGA (XC2VP30-7FF1152) with a comparable number of dedicated multiplier blocks.

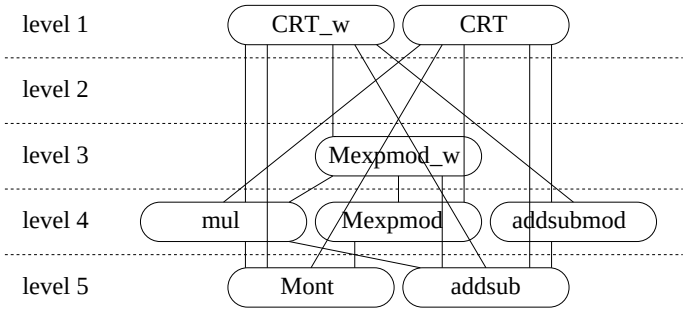

Figure 4.10: Hierarchy for instructions in the CRT group in the CIOS-based public key coprocessor.

This means that we use the CIOS-based coprocessor with a Montgomery multiplier that is downsized by a factor 2, as explained in Sect. 3.4.2. Table 4.3 gives the maximum frequency and the used resources on the FPGA. Both the SOS and the CIOS coprocessors can handle 160-, 256- and 1024-bit operations, the latter for instructions related to modular exponentiation both with and without the use of CRT. Tables 4.4 and 4.5 show that the CIOS-based coprocessor outperforms the SOS-based solution for all higher-level operations. However, the latency of some lower-level operations is smaller for the SOS-based architecture. Note that for the computation of a 2048-bit modular exponentiation using CRT the data path cannot be used as two parallel halves. Therefore, the speed-up compared to a modular exponentiation without the use of CRT is only a factor two.

Table 4.3: Comparison of the maximum operating frequency and area of our SOS-based and CIOS-based coprocessor.

	SOS	CIOS
max. freq. (MHz)	108	87
area	11198 slices 66 MULTs 66 RAM blocks	10392 slices 68 MULTs 66 RAM blocks

Note that the CIOS-based architecture is not optimal for integer multiplication. However, because it outperforms the SOS-based architecture for Montgomery multiplication, the overall performance is in favour of the CIOS-based coprocessor. That is why the next sections only focus on our CIOS-based solution.

Table 4.4: Number of cycles and latency of integer addition/subtraction, integer multiplication, modular addition/subtraction, Montgomery multiplication and modular multiplication on our SOS-based and CIOS-based coprocessor.

		SOS		CIOS	
		# cycles	latency (μs)	# cycles	latency (μs)
`addsub`	160	14	0.13	5	0.06
	256	20	0.19	6	0.07
	512	36	0.33	8	0.09
	1024	68	0.63	13	0.15
`mul`	160	23	0.21	58	0.67
	256	35	0.32	81	0.93
	512	67	0.62	139	1.60
	1024	131	1.21	260	2.99
`addsubmod`	160	28	0.26	10	0.11
	256	40	0.37	12	0.14
	512	72	0.67	16	0.18
	1024	136	1.26	26	0.30
`Mont`	160	97	0.90	30	0.34
	256	139	1.29	43	0.49
	512	251	2.32	77	0.89
	1024	475	4.40	146	1.68
`mulmod`	160	194	1.80	60	0.69
	256	278	2.57	86	0.99
	512	502	4.65	154	1.77
	1024	950	8.80	292	3.36

4.2.4 Performance of Protocols on Top of Our CIOS Coprocessor

In this section, we make a list of instructions (from Table 4.1) to be performed for Diffie-Hellman key exchange, elliptic curve Diffie-Hellman key exchange, the generation and verification of a DSA signature and the generation and verification of an ECDSA signature. Based on these lists of instructions, we discuss the performance of the above protocols on our CIOS-based coprocessor. Here, we use the fact that a modular reduction can be performed using a modular multiplication with one. Because the algorithms contain a hash function, which is not included in our coprocessor, we use the results of the SHA-1 implementation presented in [47]. We denote the hash instruction by `hash`. Modular inversion can be performed

Table 4.5: Number of cycles and latency of point multiplication with randomized projective coordinates (without and with key blinding), modular exponentiation (without and with key blinding) and modular exponentiation using CRT (without and with key blinding) on our SOS-based and CIOS-based coprocessor.

		SOS		CIOS	
		# cycles	latency (ms)	# cycles	latency (ms)
`pmul`	160	352189	3.26	110855	1.27
	256	805945	7.46	248459	2.86
`pmul_w`	160	394132	3.65	124124	1.43
	256	866026	8.02	267052	3.07
`expmod`	1024	598975	5.55	184106	2.12
`expmod_w`	1024	608950	5.64	187454	2.15
`CRT`	1024	164223	1.52	50680	0.58
	2048	1199033	11.10	369070	4.24
`CRT_w`	1024	170465	1.58	52896	0.61
	2048	1219393	11.29	375772	4.32

using Fermat's theorem, which requires an integer subtraction and a modular exponentiation. The last step in ECDSA signature verification is a point addition. This is only provided in Montgomery representation. Therefore, all four coordinates need to be converted before the point addition. Moreover, the operation gives the resulting point in projective coordinates, which means that we need to convert back to affine coordinates afterwards. This comes down to two Montgomery multiplications (to compute Z^2 and Z^3), two modular inversions (to compute Z^{-2} and Z^{-3}) and two Montgomery multiplications (to multiply the X- and Y-coordinate with Z^{-2} and Z^{-3}, respectively). Finally, the two affine coordinates of the resulting point are converted from Montgomery to normal representation using two Montgomery multiplications.

Based on the above observations and the algorithms presented in Chapter 1, the sequence of instructions for the Diffie-Hellman, elliptic curve Diffie-Hellman, DSA and ECDSA protocols can be written as follows:

- 1024-bit Diffie-Hellman key exchange:
  `expmod`, `expmod`;
- 160-bit elliptic curve Diffie-Hellman key exchange:
  `pmul`, `pmul`;
- 1024-bit DSA signature generation:

`expmod`, `mulmod`, `addsub`, `expmod`, `mulmod`, `hash`, `addsubmod`;

- 1024-bit DSA signature verification:
  `addsub`, `expmod`, `hash`, `mulmod`, `mulmod`, `expmod`, `expmod`, `mulmod`, `mulmod`;
- 160-bit ECDSA signature generation:
  `pmul`, `mulmod`, `addsub`, `expmod`, `mulmod`, `hash`, `addsubmod`;
- 160-bit ECDSA signature verification:
  `addsub`, `expmod`, `hash`, `mulmod`, `mulmod`, `pmul`, `pmul`, `Mont`, `Mont`, `Mont`, `Mont`, `Mpadddouble`, `Mont`, `Mont`, `addsub`, `expmod`, `expmod`, `Mont`, `Mont`.

Because the RAM bank in our coprocessor is interfaced through a 16-bit signal, we need $\lceil n/16 \rceil$ cycles to write/read an n-bit value to/from the RAM bank. The inputs and parameters for a certain instruction need to be at fixed addresses in the RAM bank before the operation starts and the outputs also appear at fixed addresses. That is why the results of a certain instruction inside one of the protocols above need to be read from a certain output address and written back to a certain input address before the next instruction starts. This causes some communication overhead, which is taken into account in Table 4.6. The table gives the latencies for the above protocols on our 1024-bit two-stage pipelined CIOS coprocessor, assuming that the non-pipelined hash function in [47] has a throughput of 3.1 *Gbps*.

Table 4.6: Latency of public key protocols on our 1024-bit two-stage pipelined CIOS coprocessor.

protocol	latency (ms)
Diffie-Hellman	4.24
elliptic curve Diffie-Hellman	2.55
DSA signature generation	4.26
DSA signature verification	6.39
ECDSA signature generation	1.33
ECDSA signature verification	2.64

4.3 Comparison to Previous Work

4.3.1 Previous Work

Detailed surveys on finite fields multipliers and coprocessors for public key cryptography are given in [7] and [29].

Goodman and Chandrakasan proposed a Domain-Specific Reconfigurable Cryptographic Processor (DSRCP) in [27]. The instruction set definition of the DSRCP was dictated by the IEEE 1363 Public Key Cryptography Standard document [31]. The DSRCP performs a variety of algorithms ranging from modular integer arithmetic to elliptic curve arithmetic. The various complex modular arithmetic operations are implemented using microcode, while simple operations are implemented directly in hardware. Multiplication is performed using Montgomery multiplication. The DSRCP processor is fabricated in a 0.25 μm CMOS technology. The goal of this coprocessor was to achieve an energy-efficient implementation.

Orlando and Paar implemented a higher radix Montgomery multiplier using Booth recoding and the pre-computation of frequently used values. This resulted in the first ECC processor over GF(p) [67]. Bednara *et al.* gave a comparison of several methods for hardware implementation of elliptic curve cryptosystems in [8], focusing on FPGA-based implementations. A scalable systolic array and an RSA and ECC architecture were implemented by Batina *et al.* in [6]. The fastest published ECC implementation over GF(p) on an FPGA is the one of Sakiyama *et al.* [80]. They implemented a dual-field architecture supporting GF(p) as well GF(2^n) arithmetic.

An example of the utilization of CRT is presented by Grosschädl in [28]. In this paper the multiplier architecture of an RSA chip is presented. The multiplier data path is reconfigurable to execute either one 1024-bit or two 512-bit modular exponentiations in parallel. The implementation of RSA decryption in [54] is also using CRT. The fastest published RSA decryption using CRT on an FPGA is the one implemented by Tang *et al.* in [89]. Up to now, the fastest reported RSA implementation on an FPGA, that does not apply CRT, is presented by Kelley and Harris in [38].

4.3.2 Comparison of Implementation Results

Table 4.7 compares the implementation results of our CIOS-based coprocessor to the fastest previously designed implementations. Unlike the work mentioned above, we implemented many countermeasures against side channel attacks. However, because these countermeasures only affect the complexity of the control logic in a minor way, they do not give rise to a substantial area increase. The results are based on the implementation of the instruction hierarchy together with the CIOS-based Montgomery multiplier on a Xilinx Virtex XC2VP30-7FF1152 FPGA [105]. The RAM bank is not included in the resources. For the latency of a 1024-bit modular exponentiation, both with and without the use of CRT, the fastest published implementation is [89]. The table shows that our coprocessor shows similar results in terms of speed, but outperforms the design in [89] in terms of area. The reason is that the architecture in [89] is fully parallel, while our solution is two or four times downsized. Even though the FPGA used in [89] is a Virtex-II, while we use the newer Virtex-II Pro, the performance of the architectures can be com-

pared based on the number of cycles for one 1024-bit RSA decryption both with and without the use of CRT. This can be done by comparing the frequency and latency of our two times downsized architecture (87 MHz and 2.12 ms, respectively) to the frequency and latency of the fully parallel architecture in [89] (90 MHz and 2.33 MHz, respectively). Because the frequency of our implementation is slightly lower while the latency is slightly higher, the comparison of the number of cycles is slightly in favour of our implementation. The fastest published implementation of 160-bit and 256-bit ECC over GF(p) is described in [80]. The latencies for a 160- and a 256-bit elliptic curve point multiplication are comparable to the latencies achieved by our coprocessor. The authors of [80] achieved this high speed by optimizing the point addition and point doubling algorithms for more than one parallel modular multipliers. That is why their reported area is bigger than ours. A summary of the most important comparison results is given in Figs. 4.11, 4.12 and 4.13.

4.4 Conclusions

This chapter describes an instruction hierarchy for programmable public key coprocessors. Because of the difference in the architectures for the SOS-based and the CIOS-based Montgomery multipliers, the hierarchy of instructions also differs. We elaborate on this difference and present two hierarchic architectures that reduce the complexity of the algorithms. Moreover, to obtain a fully programmable coprocessor, all instructions are made accessible from the outside of the coprocessor. The implementation results of both the SOS-based and the CIOS-based coprocessors are given. The overall performance of the CIOS-based solution is much better than that of the SOS-based solution. The SOS-based architecture only outperforms the CIOS-based coprocessor for some low-level functions. That is why we compared our CIOS-based public key coprocessor to previously designed coprocessors. From this comparison, we conclude that our solution implements the fastest published public key coprocessor on an FPGA.

Table 4.7: Comparison of our coprocessor to previously designed FPGA implementations of public key coprocessors, where our-x-y denotes our x-bit implementation with a downsizing factor of y, SSFs stands for the number of slice flip-flops and (w) denotes the implementation with countermeasures against side channel attacks.

reference	max. freq. (MHz)	area	FPGA	t_{exp} (ms)	t_{CRT} (ms)	t_{ECC} (ms)		
				1024	1024	160	192	256
our-160-1	72	24 MULTs + 2171 slices = 24 MULTs + 3401 LUTs + 2624 SFFs	XC2VP30			1.00 1.12 (w)		
our-160-4	183	10 MULTs + 2118 slices = 10 MULTs + 3714 LUTs + 2832 SFFs	XC2VP30			1.09 1.22 (w)		
our-256-1	67	36 MULTs + 3529 slices = 36 MULTs + 5555 LUTs + 3296 SSFs	XC2VP30			1.08 1.21 (w)		2.27 2.35 (w)
our-256-4	182	12 MULTs + 3171 slices = 12 MULTs + 5581 LUTs + 3552 SFFs	XC2VP30			1.10 1.23 (w)		2.65 2.85 (w)
our-1024-2	87	68 MULTs + 10384 slices = 68 MULTs + 16845 LUTs + 9728 SFFs	XC2VP30	2.12 2.15 (w)	0.58 0.61 (w)	1.27 1.43 (w)		2.86 3.07 (w)
our-1024-4	152	36 MULTs + 9090 slices = 36 MULTs + 16105 LUTs + 9312 SFFs	XC2VP30	2.33 2.36 (w)	0.64 0.67 (w)	1.32 1.47 (w)		3.17 3.41 (w)
[67]	40	11416 LUTs + 5735 SFFs + 4kbit bRAM	XCV1000E				3	
[80]	100	8954 slices + 6 bRAMs	XC2VP30			1.04		
[80]	100	10847 slices + 9 bRAMs	XC2VP30					2.70
[3]	69	4608 slices	XVC2000E	22.7	6.1			
[54]	101	24767 slices	XC2V6000		2.63			
[38]	135	32 MULTs + 2593 LUTs + 5k RAM	XC2V2000	5.0				
[38]	135	8 MULTs + 695 LUTs + 5k RAM	XC2V2000	17				
[89]	90	62 MULTs + 14334 slices	XC2V4000	2.33	0.66			

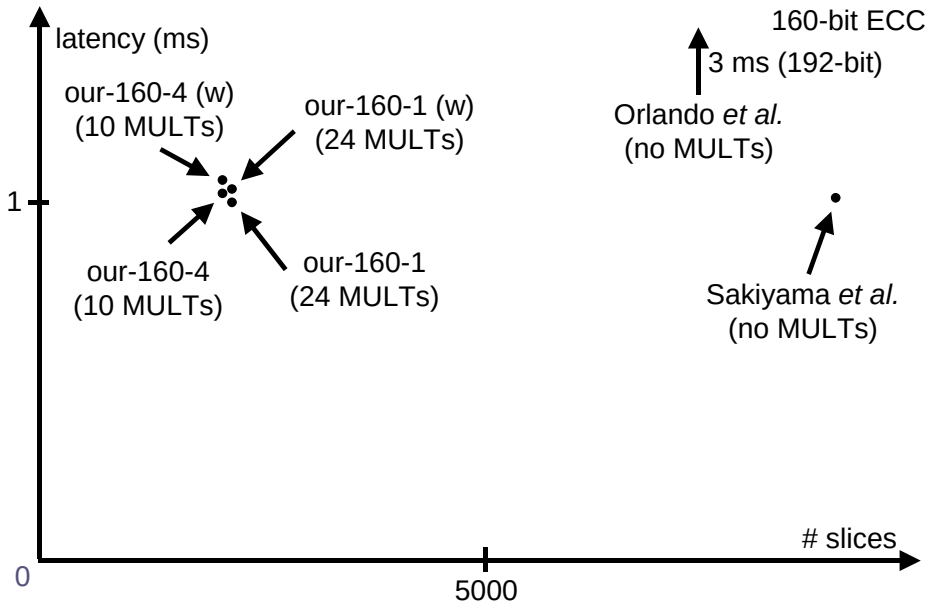

Figure 4.11: Comparison of the latency and area for a 160-bit ECC point multiplication, where (w) denotes the implementation with countermeasures against side channel attacks.

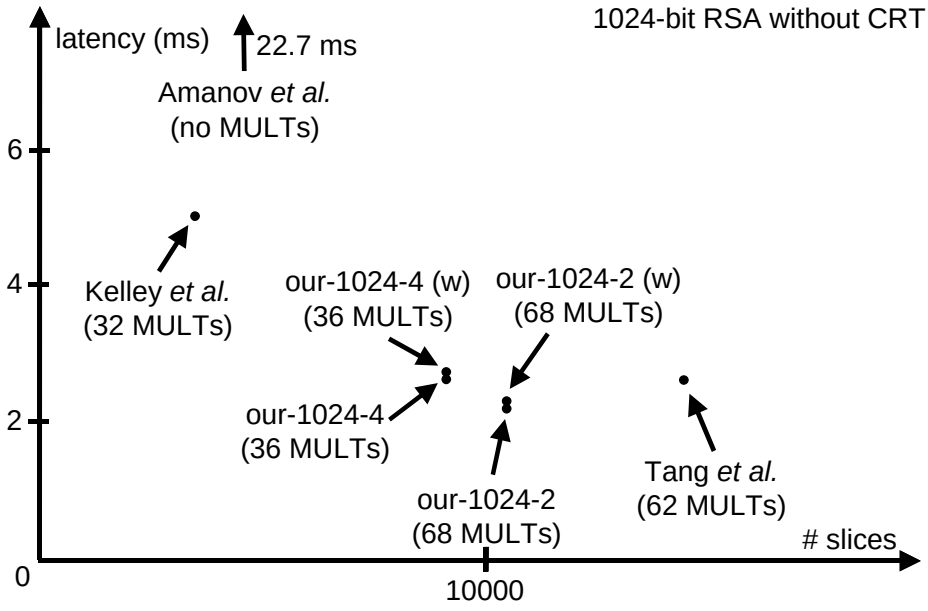

Figure 4.12: Comparison of the latency and area for a 1024-bit modular exponentiation without the use of CRT, where (w) denotes the implementation with countermeasures against side channel attacks.

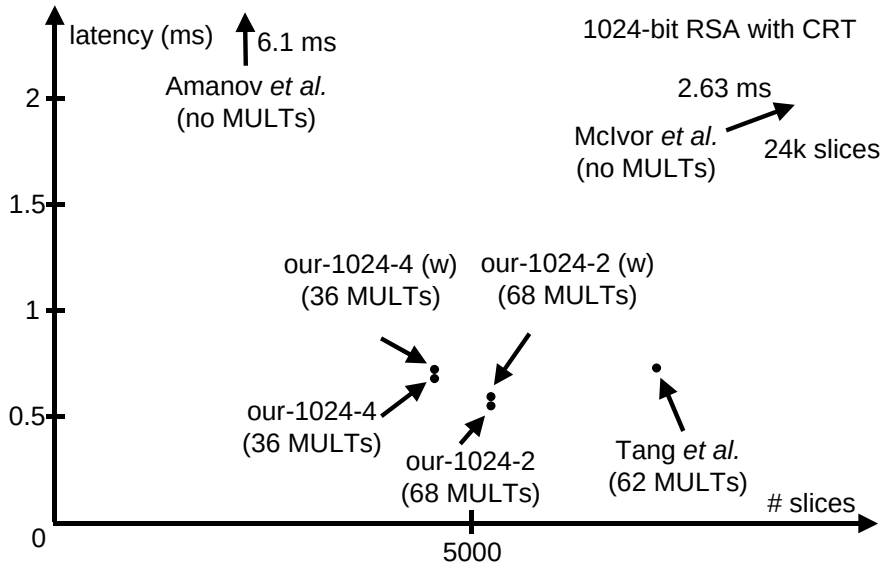

Figure 4.13: Comparison of the latency and area for a 1024-bit modular exponentiation with the use of CRT, where (w) denotes the implementation with countermeasures against side channel attacks.

Chapter 5

A Systematic Evaluation of Compact Implementations for the AES S-box

This chapter gives an overview of implementation options for the AES S-box; the most compact solution uses composite field arithmetic. We elaborate on previous implementations of the AES S-box using composite fields. Our contribution is the observation that the parameters in the most compact solution can be chosen more optimally. To support this observation, we explore a subset of all implementation options, resulting in an area reduction of 5% compared to previously reported implementations. Because the S-box forms a large part of the overall implementation of AES, this area reduction is still relevant. Although the general goal of this thesis is the design of cryptographic coprocessors on FPGAs, we perform our evaluation using a CMOS standard cell library. The reason is that the resolution of the area on an ASIC implementation is larger than that of an FPGA. Since the S-box area is so small, it would not be accurate to compare designs based on FPGA implementations.

5.1 AES

After an open competition ending in 2000, the National Institute for Standard and Technology (NIST) selected the Rijndael block cipher as the new Advanced Encryption Algorithm (AES) [62]. The AES algorithm, designed by Joan Daemen and Vincent Rijmen, has an SPN (Substitution Permutation Network) structure. Its use is mandatory for the encryption of sensitive but unclassified US government information; in 2003 the US government has announced that it can also be used for

encrypting secret and top secret information (for the last category key lengths of at least 192 bits need to be used). AES is currently replacing the Data Encryption Standard [61] as the worldwide standard block cipher.

Rijndael has a variable block and key length which can be 128, 192 or 256 bits; the AES standard includes only block lengths of 128 bits. The number of rounds of the AES algorithm is equal to 10, 12 or 14 when the key length is 128, 192 or 256 bits, respectively. For encryption, each round, except for the last one, consists of four operations: SubBytes, ShiftRows, MixColums and AddRoundKey. The last round contains only three operations: SubBytes, ShiftRows and AddRoundKey. Before the first round, the AddRoundKey operation is also performed. This is depicted in Fig. 5.1, where all lines represent 128-bit values, except for the input key, which can be 128, 192 or 256 bits. The 128-bit values in between the four round operations represent the state of the block cipher, which can be depicted as a square matrix with bytes as entries. In the following sections, the four operations inside the rounds are described, together with the key expansion routine, which derives round keys from the input key. The description gives a short overview of the AES encryption algorithm, but for more details and for a description of the decryption algorithm, the reader is encouraged to consult the standard [62] or the references [19] and [20].

5.1.1 Round Operations

SubBytes

In the SubBytes operation, each byte in the state is transformed independently using a substitution table, called S-box. The S-box consists of two operations:

1. a multiplicative inverse in the finite field GF(2^8), where the byte consisting of all zero-bits is mapped to itself;

2. an affine transformation, which can be represented as follows:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix},$$

where $B = [b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7]^T$ and $C = [c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7]^T$ are the input and output bytes of the affine transformation.

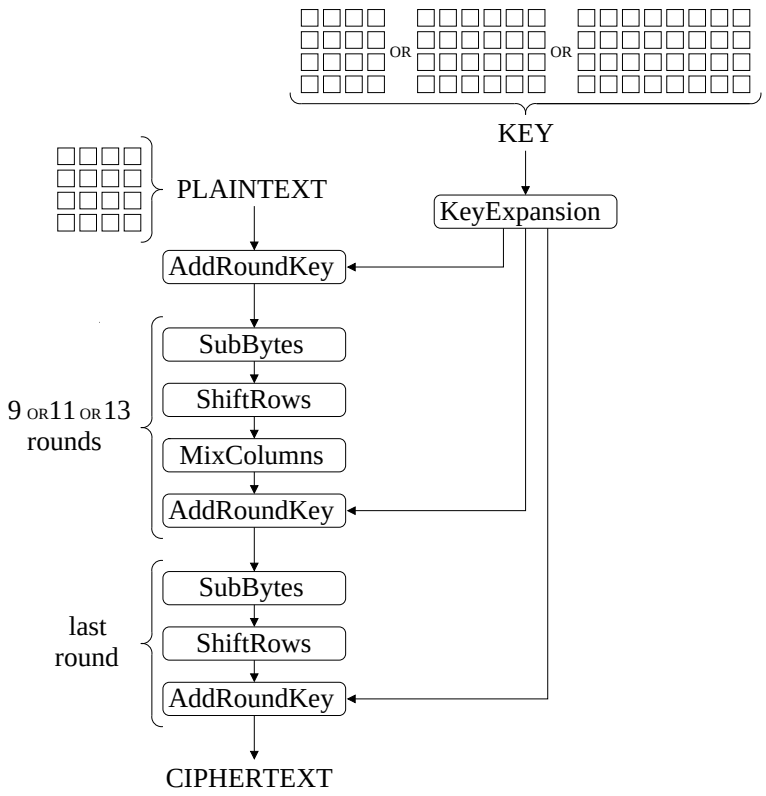

Figure 5.1: Schematic representation of the AES encryption algorithm.

The field GF(2^8) is defined with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. An element of the field GF(2^8) is represented by a polynomial of degree 7 (cf. Chapter 2): $A(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$. The inverse of $A(x)$, called $B(x)$, satisfies the equation $A(x) \cdot B(x) \equiv 1 \mod m(x)$.

ShiftRows

In the ShiftRows transformation, a cyclic shift over 1, 2 and 3 positions is performed on the last three rows in the state. This is illustrated in Fig. 5.2.

MixColums

In the MixColumns operation, each column in the state is transformed independently. The columns are represented as polynomials of degree 3 with as coefficients

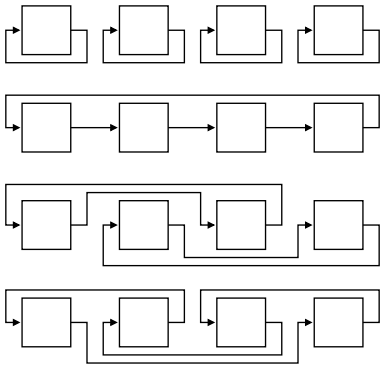

Figure 5.2: Schematic representation of the ShiftRows transformation.

the bytes in the column. Each byte is viewed as a polynomial over GF(2^8). The MixColumns transformation performs a multiplication modulo $x^4 + 1$ of each column with the fixed polynomial $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$. The coefficients of this polynomial are in hexadecimal little-endian notation. The multiplication modulo $x^4 + 1$ with $a(x)$ comes down to replacing the bytes in each column in the following way:

$$
\begin{aligned}
B'_0 &= (\{02\} \cdot B_0) \oplus (\{03\} \cdot B_1) \oplus B_2 \oplus B_3\,, \\
B'_1 &= B_0 \oplus (\{02\} \cdot B_1) \oplus (\{03\} \cdot B_2) \oplus B_3\,, \\
B'_2 &= B_0 \oplus B_1 \oplus (\{02\} \cdot B_2) \oplus (\{03\} \cdot B_3)\,, \\
B'_3 &= (\{03\} \cdot B_3) \oplus B_1 \oplus B_2 \oplus (\{02\} \cdot B_3)\,.
\end{aligned}
$$

Here, the $\oplus$ denotes an addition over GF(2^8), which is the same as a bit-wise XOR operation. The multiplications are performed over GF(2^8) with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ as the modulus.

AddRoundKey

The round keys are computed in words of 32 bits and added to the state in a column-by-column way. Before the first round, a 128-bit round key, i.e., four 32-bit words, is added to the state. Next, a round key is added to the state at the end of every round. This means that 44, 52 or 60 words of 32 bits need to be derived from the input key, when the key size is 128, 192 or 256, respectively. This is done in the key expansion routine, which is described in the next section.

5.1.2 Key Expansion

The 32-bit words used for the round keys are computed as follows:

- The first 4, 6 or 8 words are equal to the words of the input key, starting from the LSB side.
- The next words are computed as the XOR of the previous word with the word four positions earlier.
- Exceptions are the words with positions that are multiples of 4, 6 or 8, for a key size of 128, 192 or 256, respectively. These words are computed as the XOR of the Round Function applied to the previous word with the word four positions earlier. The Round Function is explained below.
- There is another exception when the key size is 256 bits. The words that have positions that are multiples of 4, but not of 8, are formed by the XOR of the Modified Round Function applied to the previous word with the word four positions earlier. The Modified Round Function is explained below.

The Round Function consists of three operations, performed on a 32-bit word:

1. a byte-wise cyclic rotation of the word over 1 position to the left;
2. a substitution of each byte using the S-box described in Sect. 5.1.1;
3. a bit-wise XOR of the word with a constant value: This value depends on the position of the word and is equal to $[x^{i-1}\{00\}, \{00\}, \{00\}]$, where $x = \{02\}$ and x^i is computed over GF(2^8) modulo $x^8 + x^4 + x^3 + x + 1$. The value i equals the position of the word divided by 4, 6 or 8, when the key length is 128, 192 or 256, respectively.

The Modified Round Function only consists of an S-box substitution.

5.2 Implementation Options for the AES S-box

Design challenges for AES mainly lie in exploring the options for the S-box design. The reason is two-fold. Firstly, the S-box appears 16 times in every round to operate on the state and is implemented 40, 48 or 56 times in the Key Expansion when the key length is 128, 192 or 256, respectively. Therefore, an improvement in the performance of one S-box is reflected many times in the overall AES implementation. Secondly, all other operations can be implemented in a more or less straightforward manner, while the S-box leaves room for several implementation options. The implementation options for the AES S-box can be divided into three categories:

1. S-box implementation using Look-Up Tables (LUTs) in logic;
2. S-box implementation in RAM;
3. S-box implementation using composite field arithmetic.

The first option translates the 8-bit substitution table into a LUT in combinatorial logic. Implementing the S-box in a LUT gives the fastest results, but takes a lot of area. A more suitable solution, especially for FPGA implementations, is the use of block RAM. Because a lot of RAM is available on the FPGA anyway, this saves some area in the configurable part of the FPGA. However, the architecture has to be redesigned in order to be able to address the RAM and to cope with the one-cycle delay for looking up values in the RAM. Another solution, which is more compact than the LUT approach and can also be implemented in combinatorial logic, takes advantage of the arithmetic structure of the S-box by using composite field arithmetic.

In the next two subsections, we explain how finite field inversion can be computed in composite fields and we compare previously designed S-box implementations.

5.2.1 Composite Field Inversion

In this subsection, we introduce composite field arithmetic for field extensions of degree 2, because these are applicable to the finite field inversion in the AES S-box.

Let us view $\mathrm{GF}((2^n)^2)$ as a field extension of degree 2 over $\mathrm{GF}(2^n)$. The field $\mathrm{GF}((2^n)^2)$ is generated as an extension field of $\mathrm{GF}(2^n)$ using an irreducible polynomial $f(x) = x^2 + \alpha x + \beta$, with $\alpha, \beta \in \mathrm{GF}(2^n)$. The extension field can be viewed as a two-dimensional vector space over $\mathrm{GF}(2^n)$. Hence, an arbitrary element $\Delta \in \mathrm{GF}((2^n)^2)$ can be written as $\Delta = \delta_1 x + \delta_0$, where $\delta_1, \delta_0 \in \mathrm{GF}(2^n)$.

We want to calculate the inverse of Δ, i.e., Δ^{-1}, such that $\Delta \cdot \Delta^{-1} \equiv 1 \mod f(x)$. The inverse can be written as

$$\Delta^{-1} = (\Delta^r)^{-1} \cdot \Delta^{r-1}\,, \tag{5.1}$$

where r is an arbitrary integer. To choose r, we use the observation of Paar in [69], which states that an element of an extension field $\mathrm{GF}((2^n)^m)$, raised to the power $\frac{2^{nm}-1}{2^n-1}$, results in an element of the subfield $\mathrm{GF}(2^n)$. This is equivalent to computing the norm of the element. For field extensions of degree 2, this yields

$$r = \frac{2^{2n}-1}{2^n-1}\,.$$

The computation of the inverse can be done in four steps [30]:

1. exponentiation in $\mathrm{GF}((2^n)^2)$: Δ^{r-1};

2. multiplication in GF($(2^n)^2$), with the result in GF(2^n): $\Delta^r = \Delta^{r-1} \cdot \Delta$;
3. inversion in GF(2^n): $(\Delta^r)^{-1}$;
4. multiplication in GF($(2^n)^2$): $(\Delta^r)^{-1} \cdot \Delta^{r-1}$.

The result of the first exponentiation can be computed by using the observation that

$$r - 1 = \frac{2^{2n} - 1}{2^n - 1} - 1 = \frac{(2^n + 1)(2^n - 1)}{2^n - 1} - 1 = 2^n \,.$$

This means that

$$\Delta^{r-1} = \Delta^{2^n} = (\delta_1 x + \delta_0)^{2^n} = \delta_1 x^{2^n} + \delta_0 \,.$$

Because $f(x^{2^n}) = 0$, x^{2^n} must be equal to the second root, besides x, of $f(x)$. It can be verified that this second root is $x + \alpha$, which leads us to the following solution for the first exponentiation:

$$\Delta^{r-1} = \delta_1 x^{2^n} + \delta_0 = \delta_1(x + \alpha) + \delta_0 = \delta_1 x + (\delta_1 \alpha + \delta_0) \,.$$

After multiplication with Δ, a polynomial in GF(2^n) is obtained:

$$\Delta^r = \delta_1^2 \beta + \delta_1 \delta_0 \alpha + \delta_0^2 \,.$$

Hence, the inversion can be done in GF(2^n). The inverse is multiplied with Δ^{r-1} resulting in the following polynomial:

$$\Delta^{-1} = \delta_1(\delta_1^2 \beta + \delta_1 \delta_0 \alpha + \delta_0^2)^{-1} x + (\delta_0 + \delta_1 \alpha)(\delta_1^2 \beta + \delta_1 \delta_0 \alpha + \delta_0^2)^{-1} \,. \tag{5.2}$$

This equation consists of operations that can be performed in the subfield GF(2^n).

When we apply the theory of field extensions of degree 2 to the inversion in the AES S-box, there are two implementation options. We can either perform the subfield operations in Eq. (5.2) directly in GF(2^4) or we can use the equation recursively. This comes down to using the tower field GF($(((2^2)^2)^2$). These two options were implemented in [97] and [82] and are compared to each other in the next section. The transformation from GF(2^8) to GF($(2^4)^2$) or GF($(((2^2)^2)^2$) is also explained below.

5.2.2 Comparison of Previously Designed S-box Implementations

S-box Implementation in GF($(2^4)^2$)

In [78], Rudra *et al.* elaborate on the use of the composite field GF($(2^4)^2$) for the AES S-box inversion, but no hardware implementation is presented. Wolkerstorfer

et al. apply the idea in [97] to make a compact implementation of the AES S-box. To go from GF(2^4) to the extension field GF($(2^4)^2$) they use an irreducible polynomial of the form $x^2 + \alpha x + \beta$, with $\alpha = 1$ and $\beta \in$ GF(2^4). In this case, Eq. (5.2) becomes

$$\Delta^{-1} = \delta_1(\delta_1^2\beta + \delta_1\delta_0 + \delta_0^2)^{-1}x + (\delta_0 + \delta_1)(\delta_1^2\beta + \delta_1\delta_0 + \delta_0^2)^{-1}\,.$$

This can be constructed as in Fig. 5.3, where the building blocks are operations over GF(2^4) modulo $x^4 + x + 1$. They are implemented as 4-bit look-up tables.

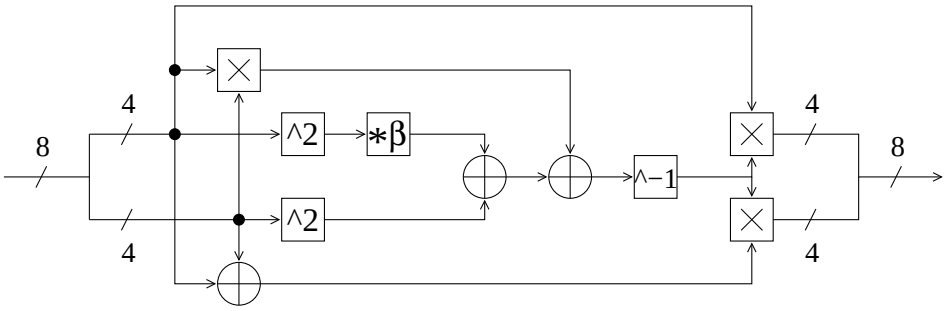

Figure 5.3: Architecture of the inversion by Wolkerstorfer *et al.* [97].

The transformation to go from GF(2^8) to GF($(2^4)^2$) can be computed with a transformation matrix. In [69], Paar explains how this matrix can be created. Because the matrix maps 8 base elements in GF(2^8) to base elements in GF($(2^4)^2$), it depends on the choice of irreducible polynomials to create the subfield and the extension field. Moreover, for each set of irreducible polynomials, there are 8 possible transformation matrices. After performing the inversion using the GF($(2^4)^2$) representation we need to go back to the GF(2^8) representation using the inverse of the transformation matrix. This matrix can be combined with the affine transformation matrix at the end of the S-box. The transformation matrices together with the composite field inversion form the global structure of the S-box implementation as depicted in Fig. 5.4.

In [97], the transformation from GF(2^8) to GF($(2^4)^2$) is performed using the

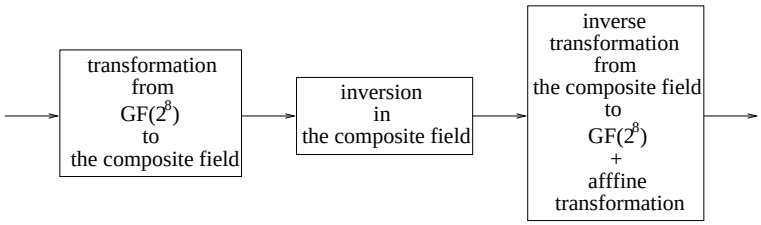

Figure 5.4: Structure of the AES S-box implementation using composite fields.

following matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

S-box Implementation in $\mathrm{GF}(((2^2)^2)^2)$

Equation (5.2) can be used recursively to find the inverse in $\mathrm{GF}(((2^2)^2)^2)$. This field is an extension of degree 2 over $\mathrm{GF}((2^2)^2)$ constructed using the irreducible polynomial $P(x) = x^2 + p_1 x + p_0$, where $p_1, p_0 \in \mathrm{GF}((2^2)^2)$. $\mathrm{GF}((2^2)^2)$ is a field extension of degree 2 over $\mathrm{GF}(2^2)$ using the irreducible polynomial $Q(y) = y^2 + q_1 y + q_0$ and $q_1, q_0 \in \mathrm{GF}(2^2)$. $\mathrm{GF}(2^2)$ is a field extension of degree 2 over $\mathrm{GF}(2)$ using the irreducible polynomial $R(z) = z^2 + z + 1$.

In [82], Satoh *et al.* made the following choices for the coefficients of the irreducible polynomials:

$$\begin{aligned} p_1 &= 1 = \{0001\}_2\,, \\ p_0 &= \lambda = \omega y = (z+1)y = \{1100\}_2\,, \\ q_1 &= 1 = \{01\}_2\,, \\ q_0 &= \phi = z = \{10\}_2\,. \end{aligned}$$

Equation (5.2) is implemented as

$$\begin{aligned}
&\Delta = \delta_1 x + \delta_0 \in \mathrm{GF}(((2^2)^2)^2): \\
&\qquad \Delta^{-1} = (\delta_1 x + (\delta_1 + \delta_0)) \cdot (\lambda \delta_1^2 + (\delta_1 + \delta_0)\delta_0)^{-1}, \\
&\delta = d_1 y + d_0 \in \mathrm{GF}((2^2)^2): \\
&\qquad \delta^{-1} = (d_1 y + (d_1 + d_0)) \cdot (\phi d_1^2 + (d_1 + d_0)d_0)^{-1}.
\end{aligned} \tag{5.3}$$

Inversion in $\mathrm{GF}(2^2)$ requires only one addition:

$$d = t_1 z + t_0 \in \mathrm{GF}(2^2) : d^{-1} = t_1 z + (t_1 + t_0). \tag{5.4}$$

These operations are implemented as depicted in Fig. 5.5. Unlike the inversion in $\mathrm{GF}((2^4)^2)$, the building blocks are not implemented as 4-bit look-up tables, but as operations in $\mathrm{GF}((2^2)^2)$, which can be computed as follows (with $a_1, a_0, b_1, b_0 \in \mathrm{GF}(2^2)$):

- $(a_1 y + a_0) \cdot (b_1 y + b_0) = (a_1 b_1 + a_1 b_0 + a_0 b_1)y + (a_1 b_1 \phi + a_0 b_0)$;
- $(a_1 y + a_0)^2 = a_1 y + (a_1 \phi + a_0)$;
- $(a_1 y + a_0) \cdot \lambda = (a_1 y + a_0)\omega y = (a_1 + a_0)\omega y + a_1 \omega \phi$.

These equations consist of operations in $\mathrm{GF}(2^2)$ that can be computed as follows (with $g_1, g_0, h_1, h_0 \in \mathrm{GF}(2)$):

- $(g_1 z + g_0) \cdot (h_1 z + h_0) = (g_1 h_1 + g_0 h_1 + g_1 h_0)z + (g_1 h_1 + g_0 h_0)$;
- $(g_1 z + g_0)^2 = a_1 z + (a_1 + a_0)$;
- $(g_1 z + g_0) \cdot \phi = (g_1 z + g_0) \cdot z = (g_1 + g_0)z + g_1$;
- $(g_1 z + g_0) \cdot \omega = (g_1 z + g_0) \cdot (z + 1) = g_0 z + (g_1 + g_0)$.

The transformation matrix used in [82] to go from a representation in $\mathrm{GF}(2^8)$ to a representation in $\mathrm{GF}(((2^2)^2)^2)$ is

$$\begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{bmatrix}.$$

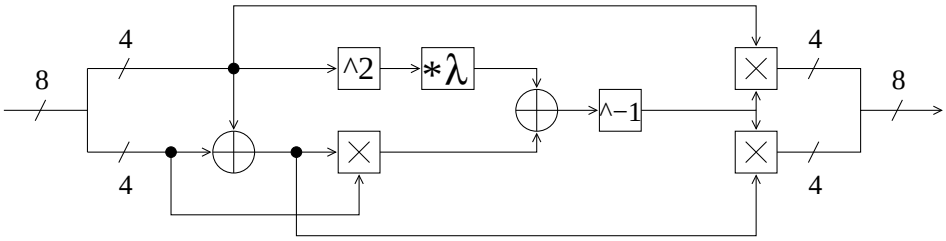

Figure 5.5: Schematic of the S-box by Satoh *et al.* [82], where the building blocks are operations in GF($(2^2)^2$), which are decomposed into operations in GF(2^2).

Implementation Results of the Composite Field Approaches

In [82], the GF($(2^4)^2$) and GF($((2^2)^2)^2$) S-boxes are compared to each other and to the LUT approach in a 0.11 μm CMOS technology. The implementations consist only of combinatorial logic. The results are repeated in Table 5.1. As can be seen from the table, the LUT S-box has the smallest latency, but takes a lot more gates than the composite field S-boxes. Because the GF($(2^4)^2$) S-box uses LUTs for the computations in GF(2^4), it takes a little bit more area than the GF($((2^2)^2)^2$) S-box. The latencies are comparable for both composite field implementations.

Table 5.1: Area and latency comparison of a LUT-based S-box to the composite field S-boxes implemented in [97] and [82]. The composite field implementations are S-boxes merged with inverse S-boxes.

implementation	area (# gates)	latency (ns)
[82]	294	3.69
[97]	362	3.75
LUT	696	2.71

5.3 Further Reduction of the S-box Area

In both the GF($(2^4)^2$) and GF($((2^2)^2)^2$) implementations, the area and latency of the S-box depend on some implementation choices:

- The irreducible polynomials determine the equations for the inversions in the subfields and the extension fields.

- The transformation matrix determines the area and latency of the transformation and the inverse transformation, where the latter is combined with the affine transformation.

Because the S-box in $\mathrm{GF}(((2^2)^2)^2)$ leads to the smallest implementation, we start from this implementation to reduce the area of the S-box.

W.r.t. the irreducible polynomials, we evaluate the options for $P(x) = x^2 + p_1x + p_0$, with $p_1, p_0 \in \mathrm{GF}((2^2)^2)$, and $Q(y) = y^2 + q_1y + q_0$, with $q_1, q_0 \in \mathrm{GF}(2^2)$. The lowest-level polynomial should always be equal to $R(z) = z^2 + z + 1$ in order to be irreducible. We stick to the choice of Satoh *et al.* to make $p_1 = q_1 = 1$ and $q_0 = \phi = z$. Based on Eq. (5.3) and the fact that the transformation matrix depends on $P(x)$ and $Q(y)$, we conclude that the hardware complexity of the circuit depends on the choice of $p_0 = \lambda$. That is why we explore all values of λ to determine the most compact solution for the S-box. The structure of the S-box at the highest level stays the same (see Fig. 5.5). The building blocks to be evaluated are:

- the constant multiplication with λ;
- the transformation matrix together with the inverse transformation matrix in combination with the affine transformation.

There are 8 choices for λ. For every λ, Table 5.2 gives the number of 2-input XORs for the constant multiplication with λ and the total number of '1' entries in de matrices for every option of the transformation matrix. Out of 8 possible transformation matrices for every λ, the one that gives the least total number of '1' entries is given in the last column.

From Table 5.2 we conclude that the solution of Satoh *et al.*, with $\lambda = (z+1)y$, uses the most area efficient constant multiplication. The transformation matrix they chose gives a total number of '1' entries equal to 61. Their implementation can be made more efficient by choosing the most compact transformation matrix which leads to a total number of '1' entries equal to 59. But the most optimal solution seems to be to change the implementation even more by taking $\lambda = zy$. The constant multiplication requires only 1 XOR more than Satoh's constant multiplication, but the total number of '1' entries in the matrices is reduced by 5. On the other hand, the design with a maximized gate count seems to be using 5 XOR gates for the constant multiplication and 71 '1' entries in the matrices. However, because of optimizations done by the synthesis tool, the area complexity cannot be predicted in this manner. That is why we implemented and synthesized the options using a UMC 0.18 μm CMOS standard cell library (1.8V/3.3V). From the implementation results we observed that the "best case" and the "worst case" in Table 5.2 correspond to the the most and the least compact implementations after synthesis, respectively. Figure 5.6 shows all options and points out the two extremes together with Satoh's solution.

Table 5.2: Comparison of the hardware complexity when using different polynomials $x^2 + x + \lambda$ for the generation of $\mathrm{GF}(((2^2)^2)^2)$, where # '1' denotes the number of '1' entries in the transformation matrix and the inverse transformation matrix in combination with the affine transformation.

λ	$\{\lambda\}_2$	# XORs in $*\lambda$	# '1' in matrices	min. # '1'
$(z+1)y+z$	$\{1110\}_2$	4	57, 58, 59, 60, 62, 63, 63, 67	57
zy	$\{1000\}_2$	4	54, 57, 59, 59, 61, 62, 63, 66	54
$zy+(z+1)$	$\{1011\}_2$	5	59, 63, 63, 64, 65, 65, 67, 71	59
$zy+z$	$\{1010\}_2$	4	56, 59, 59, 61, 61, 66, 70, 71	56
$(z+1)y$	$\{1100\}_2$	3	59, 61, 62, 63, 63, 64, 66, 69	59
$(z+1)y+1$	$\{1101\}_2$	4	60, 61, 62, 62, 63, 64, 65, 68	60
$zy+1$	$\{1001\}_2$	5	55, 59, 59, 61, 62, 63, 65, 67	55
$(z+1)y+(z+1)$	$\{1111\}_2$	3	59, 59, 61, 62, 64, 64, 66, 72	59

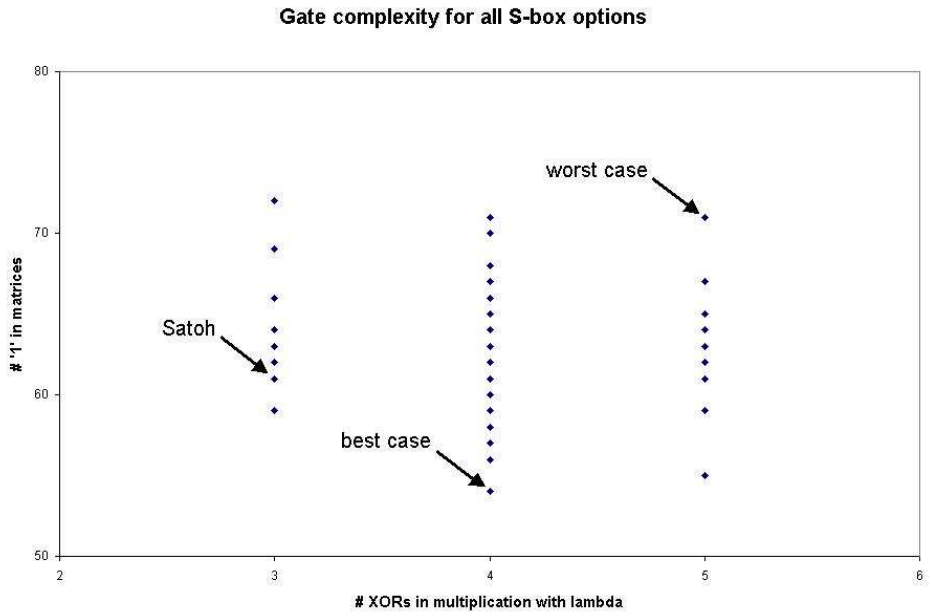

Figure 5.6: Graphical representation of the values in Table 5.2. The arrows point at Satoh's S-box, the S-box with minimized gate count ("best case") and the S-box with maximized gate count ("worst case") after synthesis.

We now elaborate on the "best case" by taking a look at the constant multiplication with λ and the transformation matrices. Figure 5.7 gives the gate-level implementation of both Satoh's (top) and our (bottom) constant multiplication. As can be seen, our constant multiplication requires one extra XOR gate compared to Satoh's implementation.

The optimal transformation has the following form:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix},$$

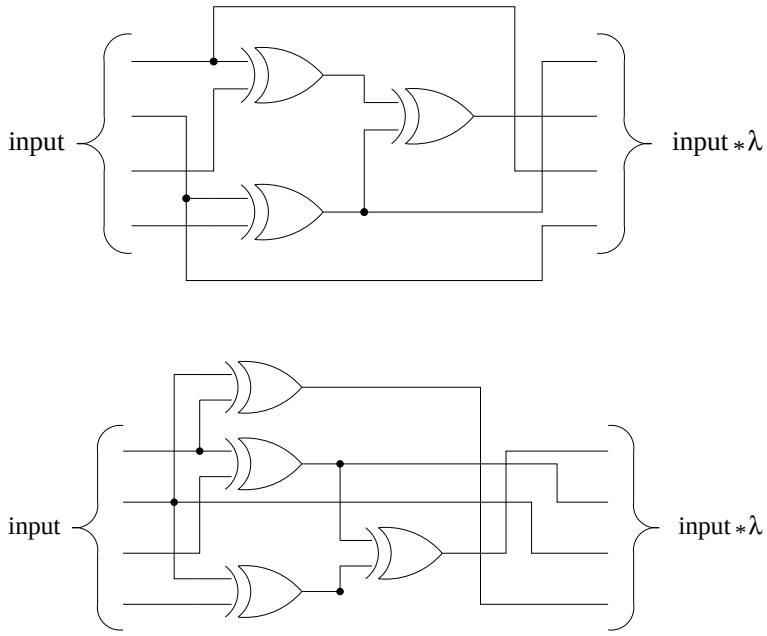

Figure 5.7: Gate-level implementation of Satoh's (top) and our (bottom) constant multiplication with λ.

where $a_i, b_i \in \mathrm{GF}(2)$ are the coefficients of $a \in \mathrm{GF}(2^8)$, $b \in \mathrm{GF}(((2^2)^2)^2)$ respectively.
This results in the following combination of the inverse transformation with the affine transformation

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix},$$

where $c_i, d_i \in \mathrm{GF}(2)$ are the coefficients of $c \in \mathrm{GF}(((2^2)^2)^2)$, $d \in \mathrm{GF}(2^8)$ respectively.

The total number of '1' entries in the 8×8 matrices is equal to 54. The addition of the column vector in the affine transformation is fixed and hence does

not have to be considered for optimization. The number of '1' entries in the matrices in Satoh's implementation is equal to 61. Implementing the matrices in a straightforward way, i.e., without searching for common terms in the rows, the number of XORs would be equal to the number of '1' entries minus the number of rows in the matrices. This would lead to an XOR gate count of 38 and 45 for our and Satoh's S-box respectively, which results in a reduction of 7 XOR gates.

We can summarize this section by stating that, in a straightforward implementation, our S-box would require 6 XOR gates less than Satoh's S-box (7 less for the implementation of the matrices and 1 extra for the multiplication with λ). The implementation results of both approaches after synthesis are compared below. To show the upper bound of the gate complexity, we also implemented the "worst case" S-box and included it in the comparison.

We implemented both our new optimized S-box and Satoh's S-box using a UMC 0.18 μm CMOS standard cell library (1.8V/3.3V). To show that the area is sensitive to the choice of the polynomials and the transformation matrix we also implemented the "worst case" S-box (with maximized gate count). All three implementations are synthesized with a rather slow target delay of 10 ns. Table 5.3 gives the number of gates (in equivalent number of 2-input NAND gates) for all designs. Satoh *et al.* implemented their S-box using a 0.11 μm CMOS standard cell library, which resulted in 294 gates with a delay of 3.69 ns. This corresponds to 286 gates in our 0.18 μm CMOS technology with a maximum delay of 10 ns. The table shows our new S-box has a 5% area reduction compared to Satoh's S-box. This is equivalent to the expected reduction of 6 XOR gates. The table also shows that a bad choice for the polynomials and the transformation matrix can lead to an area enlargement of 9%.

Table 5.3: Area comparison of our S-box with minimized/maximized gate count and Satoh's S-box (in equivalent number of 2-input NAND gates).

	"best case"	Satoh	"worst case"
number of gates	272	286	297

5.4 Conclusions and Follow-up Work

In this chapter, we gave an overview of AES S-box implementations. We focused on the composite field approach, because this gives the most compact solutions. After evaluating only a subset of the implementation options, we observe that some parameters in the smallest previously designed S-box are not chosen in an optimal way. We compare this design to the "best case" and the "worst case" in

our subset of solutions. The most compact solution gives an area reduction of 5% compared to the smallest previously designed implementation.

We only evaluated a subset of implementation options to show that it is worth choosing some parameters in a reasoned way, but this inspired David Canright to expand our evaluation, resulting in an S-box area decrease of 15% compared to our solution [14].

Chapter 6

Conclusions and Open Problems

6.1 Conclusions

This thesis dealt with implementation issues for cryptographic coprocessors on FPGAs. Since FPGAs are more and more used as end products, the design of coprocessors for cryptographic applications needs to be fine-tuned for these implementation platforms.

The first contribution of the thesis consists in the design of an important part of the data path of a cryptographic coprocessor, i.e., the modular multiplier. Although public key cryptography can be built upon GF(p) as well as GF(2^n) arithmetic, we only focus on the former. In this case, Montgomery multiplication is a very efficient way of computing a modular multiplication. We elaborate on two algorithms for Montgomery multiplication, for which the target platform is a single w-bit data path, i.e., the algorithms were developed for being used on a microprocessor. These algorithms are denoted by SOS and CIOS. Because FPGAs have much more resources than microprocessors, we propose the parallelization of these algorithms. When optimizing the architecture for the implementation of the parallelized SOS and CIOS algorithms, we take into account that recently designed FPGAs contain dedicated multiplier blocks. Because these multiplier blocks are located outside the reconfigurable part of the FPGA, they do not contribute to the number of reconfigurable slices used on the FPGA. Hence, the architecture is optimized to use these multiplier blocks for all the multiplications to be performed according to the parallelized SOS and CIOS algorithms. Moreover, in both our SOS- and CIOS-based architectures we reduce the critical path by maintaining an adapted carry-save form. For the parallelized SOS architecture, which consists of an integer multiplier and adder, carry-save adders are used inside the integer

multiplier. For the parallelized CIOS architecture, a carry-save form is maintained throughout the whole architecture. Only at the very end of a Montgomery multiplication, the final result is obtained by combining the carry and the sum. When comparing our CIOS-based architecture to published Montgomery multipliers on FPGAs, we conclude that our solution provides the fastest results.

The second contribution of the thesis is the implementation of an instruction hierarchy for public key coprocessors including countermeasures for power analysis attacks at the algorithmic level. Because we want the coprocessor to be fully programmable, the lower-level operations are not only accessible by the higher-level operations, but also by the user. We introduce two instruction hierarchies to be built on top of the SOS-based and the CIOS-based data path. Because the SOS-based Montgomery multiplier consists of an integer multiplier and adder, the lowest-level FSMs in the instruction hierarchy compute an integer multiplication and addition/subtraction. At the next level, Montgomery multiplication and modular addition/subtraction are performed. The CIOS-based Montgomery multiplier, however, is dedicated for Montgomery multiplication. It does contain an integer adder, which can be extended to an adder/subtractor, but it does not contain an integer multiplier. However, an integer multiplication can be performed on the combination of the Montgomery multiplier and the adder/subtractor. That is why the lowest level of the CIOS-based coprocessor consists of Montgomery multiplication and integer addition/subtraction, while the level above the lowest level contains the FSMs for integer multiplication and modular addition/subtraction. The highest-level instructions are similar for the SOS-based and the CIOS-based coprocessor. The only difference is the reduction step that is needed for modular exponentiation using CRT. For the SOS-based architecture, we observe that modular reduction can easily be computed on the data path. On the CIOS-based data path, however, modular reduction is not that straightforward. We solve this problem by computing a SOS-based reduction on our CIOS-based data path. All speed-area trade-off comparisons of our public key coprocessor to published work are in favour of our solution.

The third contribution of the thesis moves away from public key cryptography to focus on symmetric key cryptography. In AES, the component that leaves the most room for area and speed optimization is the S-box. We focus on the area optimization of the AES S-box using composite fields. The composite field approach has been shown to lead to the most compact implementations, but some composite field parameters could be chosen in a more optimal way. That is why we evaluate the influence of the choice of these parameters on the area of the S-box. We show that the previously proposed solution is not the most optimal one and that a suboptimal choice of parameters can lead to a substantial area increase. Although this evaluation only covers a subset of all implementation options, it inspired other researchers to perform a complete evaluation, which resulted in an even more compact solution.

6.2 Open Problems

Related to the implementation of Montgomery multipliers on FPGAs, the following items present some open problems and give directions to future work:

- Our Montgomery multiplication architecture is optimized for implementation on an FPGA due to the considered use of dedicated multipliers on a Virtex-II Pro. However, as can be seen in Table 1.1, more recently available FPGA families, such as Virtex-4 and Virtex-5, do not contain dedicated multipliers, but dedicated DSP blocks [103, 104]. The computation power of these blocks exceeds that of the multiplier blocks: not only can an efficient multiplication be performed, the DSP blocks also provide an efficient multiply-accumulate operation. When looking at the algorithms for Montgomery multiplication, it is clear that this operation can speed up Montgomery multiplication substantially. Kaliski described a cryptographic library for the Motorola DSP56000 processor in [36].

- Another way of computing a modular multiplication consists of an integer multiplication followed by Barrett reduction. Because this method also needs integer multiplications and additions/subtractions, FPGAs could be interesting target platforms.

The following items are worth considering in order to optimize the implementation of public key coprocessors on FPGAs:

- For newer and bigger FPGAs, more parallelism could be exploited in the architecture, which would result in a speed-up of the public key algorithms. Therefore, especially the ECC algorithms would have to be rewritten for a higher level of parallelism.

- Because of the complexity of the instruction hierarchy, it might be better to shift the border between our coprocessor and a microprocessor to a lower level in the hierarchy. Practically, this could be realized with a soft-core microprocessor, implemented on the FPGA, or a separate microprocessor chip.

- On the other hand, the top level in our hierarchy is the basis for protocols at a higher level, so the functionality of our coprocessor can be extended by adding more levels that implement these protocols.

- This work assumes random numbers to be available in the RAM. However, this assumption requires random numbers to be generated outside of the FPGA. The transfer of random numbers could be a very weak point in the security of the system. That is why it is advisable to foresee a random number generator on the FPGA. For Xilinx FPGAs that were launched

before the Virtex-5 family, the best solution for a random number generator is the use of invertor chains with an odd number of invertors in combination with a Linear Feedback Shift Register (LFSR) [26]. FPGAs of the Virtex-5 family provide analog Phase Locked Loops (PLLs), which can also be employed for the implementation of random number generators [104].

- Other public key systems like HyperElliptic Curve Cryptography (HECC) [40], torus-based cryptography [77] and pairing-based implementations [79] are worth considering. Also, since many public key protocols require the use of hash functions, the implementation of hash functions specifically for FPGA should be looked at.

- Besides side channel analysis attacks, another category of implementation attacks is worth taking into account; fault attacks exploit information leakage after inducing a fault in the circuit. Countermeasures against fault attacks at the algorithmic level can be added to our instruction hierarchy. Examples of these countermeasures are computing an operation twice, verifying a signature before sending the signed message, applying tricks like the one of Shamir for CRT-based modular exponentiation [86], etc.

Related to compact S-box implementations for AES, the following observation could be worth considering. Because the most important operation to be optimized inside the AES S-box is inversion in GF(2^8), the equation $(X \cdot Y)^{-1} = Y^{-1} \cdot X^{-1}$ could be used to aim for a compact S-box. It would decompose the inverse into two parts of half the length and could therefore reduce the area.

Bibliography

[1] D. Agrawal, J. R. Rao, P. Rohatgi, and K. Schramm. Templates as master keys. In J. R. Rao and B. Sunar, editors, *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 3659 in Lecture Notes in Computer Science, pages 15–29. Springer-Verlag, 2005. 76

[2] M.-L. Akkar and C. Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2162 in Lecture Notes in Computer Science, pages 309–318. Springer-Verlag, 2001. 13

[3] D. N. Amanor, V. Bunimov, C. Paar, J. Pelzl, and M. Schimmler. Efficient hardware architectures for modular multiplication on FPGAs. In *Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL)*, pages 539–542. IEEE, 2005. 61, 97

[4] R. Anderson. A5 – the GSM encryption algorithm. `http://groups.google.be/groups?hl=nl&lr=&selm=2ts9a0%2495r%40lyra.csx.cam.ac.uk`, 1994. 4

[5] E. Barkan, E. Biham, and N. Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. In D. Boneh, editor, *Advances in Cryptology – Proceedings of CRYPTO*, number 2729 in Lecture Notes in Computer Science, pages 600–616. Springer-Verlag, 2003. 4

[6] L. Batina, G. Bruin-Muurling, and S. B. Örs. Flexible hardware design for RSA and elliptic curve cryptosystems. In T. Okamoto, editor, *Proceedings of the RSA Conference – Topics in Cryptography (CT-RSA)*, volume 2964 of *Lecture Notes in Computer Science*, pages 250–263. Springer-Verlag, 2004. 61, 95

[7] L. Batina, S. B. Örs, B. Preneel, and J. Vandewalle. Hardware architectures for public key cryptography. *Elsevier Science Integration the VLSI Journal*, 34(1-2):1–64, 2003. 94

[8] M. Bednara, M. Daldrup, J. Teich, J. von zur Gathen, and J. Shokrollahi. Tradeoff analysis of FPGA based elliptic curve cryptosystems. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 797–800, 2002. 95

[9] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991. 12

[10] I. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press, 1999. 72

[11] Bluetooth S.I.G. Specification of the Bluetooth system, v. 1.2, 2003. http://www.bluetooth.org/spec. 4

[12] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society*, 46(2):203–213, 1999. 6

[13] V. Bunimov, M. Schimmler, and B. Tolg. A complexity-effective version of Montgomery's algorithm. In *Proceedings of the Workshop on Complexity Effective Designs (WCED)*, 2002. 61

[14] D. Canright. A very compact S-box for AES. In J. R. Rao and B. Sunar, editors, *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 3659 in Lecture Notes in Computer Science, pages 441–455. Springer-Verlag, 2005. 117

[15] S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2523 in Lecture Notes in Computer Science, pages 172–186. Springer-Verlag, 2002. 29

[16] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In K. Ohta and D. Pei, editors, *Proceedings of ASIACRYPT*, number 1514 in Lecture Notes in Computer Science, pages 51–65. Springer-Verlag, 1998. 68, 69

[17] J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Ç. K. Koç and C. Paar, editors, *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 1717 in Lecture Notes in Computer Science, pages 292–302. Springer-Verlag, 1999. 76

[18] J.-S. Coron and L. Goubin. On boolean and arithmetic masking against differential power analysis. In Ç. K. Koç and C. Paar, editors, *Proceedings of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 1965 in Lecture Notes in Computer Science, pages 231–237, Worcester, Massachusetts, USA, August 17-18 2000. Springer-Verlag. 13

[19] J. Daemen and V. Rijmen. AES proposal: Rijndael, 2001. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf. 102

[20] J. Daemen and V. Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer-Verlag, 2002. 102

[21] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976. 4

[22] ECRYPT. Network of Excellence in Cryptography. http://www.ecrypt.eu.org. 4

[23] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2162 in Lecture Notes in Computer Science, pages 255–265. Springer-Verlag, 2001. 13

[24] D. Giry. Keylength.com – cryptographic key length recommendation, v13.3, 2007. http://www.keylength.com. 40

[25] V. D. Gligor and P. Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes. In M. Matsui, editor, *Proceedings of the 8th International Workshop on Fast Software Encryption (FSE)*, number 2355 in Lecture Notes in Computer Science, page 92108. Springer-Verlag, 2001. 4

[26] J. D. Golić. New paradigms for digital generation and post-processing of random data. Cryptology ePrint Archive, Report 2004/254, 2004. http://eprint.iacr.org/. 122

[27] J. Goodman and A. P. Chandrakasan. An energy-efficient reconfigurable public-key cryptography processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, 2001. 95

[28] J. Grosschädl. The Chinese remainder theorem and its application in a high-speed RSA crypto chip. In *Proceedings of the 16th Annual Computer Security Application Conference*, pages 384–393. IEEE Computer Society Press, 2000. 95

[29] J. Guajardo, T. Güneysu, S. S. Kumar, C. Paar, and J. Pelzl. Efficient hardware implementation of finite fields with applications to cryptography. *Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications*, 93:75–118, 2006. 94

[30] J. Guajardo and C. Paar. Efficient algorithms for elliptic curve cryptosystems. In B. S. Kaliski Jr., editor, *Advances in Cryptology – Proceedings of CRYPTO*, number 1294 in Lecture Notes in Computer Science, pages 342–356. Springer-Verlag, 1997. 106

[31] IEEE. Institute of electrical and electronics engineers. http://www.ieee.org. 95

[32] IEEE P1363. Standard Specifications for Public Key Cryptography, 1999. 7

[33] ISO: International Organization for Standardization. Information technology – Security techniques – Digital signatures with appendix – Part 3: Discrete logarithm based mechanisms, 2006. 7

[34] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991. 77

[35] C. S. Jutla. Encryption modes with almost free message integrity. Cryptology ePrint Archive, Report 2000/039, 2000. http://eprint.iacr.org/. 4

[36] B. S. Kaliski. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology – Proceedings of EUROCRYPT*, number 473 in Lecture Notes in Computer Science, pages 230–244. Springer-Verlag, 1990. 121

[37] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. 85

[38] K. Kelley and D. Harris. Parallelized very high radix scalable Montgomery multipliers. In *Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers*, pages 1196–1200, 2005. 61, 62, 63, 95, 97

[39] N. Koblitz. Elliptic curve cryptosystem. *Math. Comp.*, 48:203–209, 1987. 8

[40] N. Koblitz. Hyperelliptic cryptosystems. *Journal of Cryptology*, 1(3):129–150, 1989. 122

[41] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16:26–33, 1996. 35, 36, 38, 61, 62, 63

[42] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In N. Koblitz, editor, *Advances in Cryptology – Proceedings of CRYPTO*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer-Verlag, 1996. 13, 33, 69, 76

[43] P. Kocher, J. Jaffe, and B. Jun. Introduction to differential power analysis and related attacks. http://www.cryptography.com/dpa/technical, 1998. 13, 29, 76

[44] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – Proceedings of CRYPTO*, number 1666 in Lecture Notes in Computer Science, pages 388–397. Springer-Verlag, 1999. 29, 76

[45] T. Kohno, J. Viega, and D. Whiting. The CWC authenticated encryption (associated data) mode. Cryptology ePrint Archive, Report 2003/106, 2003. http://eprint.iacr.org/. 4

[46] H. Ledig, F. Muller, and F. Valette. Enhancing collision attacks. In M. Joye and J.-J. Quisquater, editors, *Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 3156 in Lecture Notes in Computer Science, pages 176–190. Springer-Verlag, 2004. 76

[47] Y. K. Lee, H. Chan, and I. Verbauwhede. Throughput optimized SHA-1 architecture using unfolding transformation. In *Proceedings of the 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 354–359. IEEE, 2006. 92, 94

[48] H. Lidl and H. Niederreiter. *Introduction to Finite Fields*. Cambridge University Press, 1986. 19

[49] S. Mangard, N. Pramstaller, and E. Oswald. Successfully attacking masked AES hardware implementations. In J. R. Rao and B. Sunar, editors, *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 3659 in Lecture Notes in Computer Science, pages 157–171. Springer-Verlag, 2005. 76

[50] K. Manochehri and S. Pourmozafari. Fast montgomery modular multiplication by pipelined CSA architecture. In *Proceedings of the International Conference on Microelectronics (ICM)*, pages 144–147, 2004. 61

[51] M. Matsui. Linear cryptanalysis method for DES cipher. In T. Helleseth, editor, *Advances in Cryptology – Proceedings of EUROCRYPT*, number 765 in Lecture Notes in Computer Science, pages 386–397. Springer-Verlag, 1993. 12

[52] M. Matsui. The first experimental cryptanalysis of the Data Encryption Standard. In Y. Desmedt, editor, *Advances in Cryptology – Proceedings of CRYPTO*, number 839 in Lecture Notes in Computer Science, pages 1–11. Springer-Verlag, 1994. 12

[53] C. McIvor, M. McLoone, and J. V. McCanny. FPGA Montgomery multiplier architectures – a comparison. In *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 279–282. IEEE Computer Society, 2004. 61, 63

[54] C. McIvor, M. McLoone, J. V. McCanny, A. Daly, and W. Marnane. Fast Montgomery modular multiplication and RSA cryptographic processor architectures. In *Proceedings of the 37th Annual Asilomar Conference on Signals, Systems and Computers*, pages 379–384, 2003. 63, 95, 97

[55] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. 1, 2, 3

[56] T. S. Messerges. Using second-order power analysis to attack DPA resistant software. In Ç.K. Koç and C. Paar, editors, *Proceedings of the 2nd International Workshop on Cryptograpic Hardware and Embedded Systems (CHES)*, number 1965 in Lecture Notes in Computer Science, pages 238–252. Springer-Verlag, 2000. 29, 76

[57] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology – Proceedings of CRYPTO*, number 218 in Lecture Notes in Computer Science, pages 417–426. Springer-Verlag, 1985. 8

[58] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985. 16, 31, 33

[59] National Bureau of Standards. FIPS Pub. 197: Data Encryption Standard, 1977. 3

[60] National Institute of Standards and Technology. FIPS 186-2: Digital signature standard, 2000. 7

[61] NIST. FIPS Pub. 46-3: Data Encryption Standard, 1999. 102

[62] NIST. FIPS Pub. 197: Specification for the AES, Nov. 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. 3, 101, 102

[63] NIST. NIST Special Publication 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, 2004. 4

[64] NIST. NIST Special Publication 800-67: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher, 2004. 3

[65] NIST. NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication, 2006. 4

[66] K. Okeya, K. Schmidt-Samoa, C. Spahn, and T. Takagi. Signed binary representations revisited. In M. K. Franklin, editor, *Advances in Cryptology – Proceedings of CRYPTO*, number 3152 in Lecture Notes in Computer Science, pages 123–139. Springer-Verlag, 2004. 74

[67] G. Orlando and C. Paar. A scalable GF(p) elliptic curve processor architecture for programmable hardware. In Ç.K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of the 3rd International Workshop on Cryptograpic Hardware and Embedded Systems (CHES)*, number 2162 in Lecture Notes in Computer Science, pages 356–371. Springer-Verlag, 2001. 61, 95, 97

[68] S. B. Örs, E. Oswald, and B. Preneel. Power-analysis attacks on an FPGA – first experimental results. In C. Walter, Ç. K. Koç, and C. Paar, editors, *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2779 in Lecture Notes in Computer Science, pages 35–50. Springer-Verlag, 2003. 74

[69] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994. 106, 108

[70] T. Popp and S. Mangard. Masked dual-rail pre-charge logic: DPA-resistance without routing constraints. In J. R. Rao and B. Sunar, editors, *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 3659 in Lecture Notes in Computer Science, pages 172–186. Springer-Verlag, 2005. 29

[71] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronic Letters*, 18(21):905–907, 1982. 66

[72] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smard cards. In I. Attali and T. P. Jensen, editors, *Smart Card Programming and Security (E-smart)*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, 2001. 13

[73] G. W. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960. 72, 74

[74] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. 5

[75] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):365–403, 2003. 4

[76] RSA laboratories. PKCS #1 v2.1: RSA cryptography standard, 2002. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf. 6

[77] K. Rubin and A. Silverberg. Torus-based cryptography. In D. Boneh, editor, *Advances in Cryptology – Proceedings of CRYPTO*, number 2729 in Lecture Notes in Computer Science, pages 349–365. Springer-Verlag, 2003. 122

[78] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of the 3rd International Workshop on Cryptograpic Hardware and Embedded Systems (CHES)*, number 2162 in Lecture Notes in Computer Science, page 171 184. Springer-Verlag, 2001. 107

[79] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems based on pairings. In *Proceedings of the Symposium on Cryptography and Information Security (SCIS)*, 2000. 122

[80] K. Sakiyama, E. De Mulder, B. Preneel, and I. Verbauwhede. A parallel processing hardware architecture for elliptic curve cryptosystems. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages III–904–III–907, 2006. 95, 96, 97

[81] T. Sakurai. Perspectives on power-aware electronics. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, pages 26–29. IEEE, 2003. 27

[82] A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A compact Rijndael hardware architecture with S-box optimization. In C. Boyd, editor, *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security – Advances in Cryptology*, number 2248 in Lecture Notes in Computer Science, pages 239–254. Springer-Verlag, 2001. xviii, xx, 107, 109, 110, 111

[83] P. Schaumont. GEZEL version 2. http://rijndael.ece.vt.edu/gezel2/index.php/Main_Page, 2006. 62

[84] W. Schindler, K. Lemke, and C. Paar. A stochastic model for differential side channel cryptanalysis. In J. R. Rao and B. Sunar, editors, *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 3659 in Lecture Notes in Computer Science, pages 30–46. Springer-Verlag, 2005. 76

[85] K. Schramm, G. Leander, P. Felke, and C. Paar. A collision-attack on AES. In M. Joye and J.-J. Quisquater, editors, *Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 3156 in Lecture Notes in Computer Science, pages 163–175. Springer-Verlag, 2004. 76

[86] A. Shamir. Method and apparatus for protecting public key schemes from timing and fault attacks. US patent number 5991415, 1999. 122

[87] J. A. Solinas. Efficient arithmetic on Koblitz curves. *Codes and Cryptography*, 19:195–249, 2000. 72

[88] F.-X. Standaert, S. B. Örs, and B. Preneel. Power analysis of an FPGA: Implementation of Rijndael: Is pipelining a DPA countermeasure? In M. Joye and J.-J. Quisquater, editors, *Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 3156 in Lecture Notes in Computer Science, page 3044. Springer-Verlag, 2004. 74

[89] S. H. Tang, K. S. Tsui, and P. H. W. Leong. Modular exponentiation using parallel multipliers. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, pages 52–59, 2003. 95, 96, 97

[90] A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. In Ç.K. Koç and C. Paar, editors, *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 1717 in Lecture Notes in Computer Science, pages 94–108. Springer-Verlag, 1999. 61

[91] K. Tiri, M. Akmal, and I. Verbauwhede. A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proceedings of the 28th European Solid-State Circuits Conference (ESSCIRC)*, pages 403–406, 2002. 29

[92] K. Tiri and I. Verbauwhede. Securing encryption algorithms against DPA at the logic level: Next generation smart card technology. In C. Walter, Ç. K. Koç, and C. Paar, editors, *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2779 in Lecture Notes in Computer Science, pages 125–136. Springer-Verlag, 2003. 13

[93] K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 246–251, 2004. 29

[94] W. van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4:269–286, 1985. 12

[95] C. D. Walter. Montgomery exponentiation needs no final subtraction. *Electronic letters*, 35(21):1831–1832, October 1999. 33

[96] C. D. Walter. Montgomery's multiplication technique: How to make it smaller and faster. In Ç.K. Koç and C. Paar, editors, *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 1717 in Lecture Notes in Computer Science, pages 80–93. Springer-Verlag, 1999. 33

[97] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES S-boxes. In B. Preneel, editor, *Proceedings of the RSA Conference – Topics in Cryptography (CT-RSA)*, number 2271 in Lecture Notes in Computer Science, pages 67–78. Springer-Verlag, 2002. xviii, xx, 107, 108, 111

[98] T. Wollinger and C. Paar. *Security aspects of FPGAs in cryptographic applications*, chapter in New Algorithms, Architectures, and Applications for Reconfigurable Computing, pages 265–278. Kluwer, 2004. 74

[99] P. Wright. *Spy Catcher: The Candid Autobiography of a Senior Intelligence Officer*. Viking Press, 1987. 12

[100] Xilinx. The programmable logic company. http://www.xilinx.com, 2007. 14

[101] Xilinx. Spartan-3 publications by part. http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-1209726&sGlobalNavPick=&sSecondaryNavPick=, 2007. 15

[102] Xilinx. Spartan/XL publications by part. http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-19223&sGlobalNavPick=&sSecondaryNavPick=, 2007. 15

[103] Xilinx. Virtex-4 publications by part. http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-1210768&sGlobalNavPick=&sSecondaryNavPick=, 2007. 15, 121

[104] Xilinx. Virtex-5 publications by part. http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-1212262&sGlobalNavPick=&sSecondaryNavPick=, 2007. 15, 121, 122

[105] Xilinx. Virtex-II Pro publications by part. http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-19228&sGlobalNavPick=&sSecondaryNavPick=, 2007. xv, 14, 15, 40, 43, 44, 47, 58, 95

[106] Xilinx. Virtex-II publications by part. http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-19227&sGlobalNavPick=&sSecondaryNavPick=, 2007. 14

[107] Xilinx. Virtex publications by part. http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-19224&sGlobalNavPick=&sSecondaryNavPick=, 2007. 15

[108] Xilinx. XC4000 publications by part. http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-19230&sGlobalNavPick=&sSecondaryNavPick=, 2007. 15

List of Publications

International Journals

1. N. Mentens, S. B. Örs, B. Preneel, J. Vandewalle, "An FPGA Implementation of a Montgomery Multiplier over GF(2^m)", International Journal on Computing and Informatics, Special Issue on Design and Test 23(5-6), pp. 487-499, 2005.

2. L. Batina, N. Mentens, B. Preneel, I. Verbauwhede, "Balanced Algorithms for Side-channel Aware Design of Elliptic Curve Cryptography", IEE Proceedings on Information Security, Special Issue on Cryptographic Algorithms and Architectures for System-on-Chip 152(1), pp. 57-65, 2005.

3. K. Sakiyama, N. Mentens, L. Batina, B. Preneel, I. Verbauwhede, "Reconfigurable Modular Arithmetic Logic Unit Supporting High-performance RSA and ECC over GF(p)", International Journal of Electronics 94(5), pp. 497-510, 2007.

National Journals

4. L. Batina, P. Buysschaert, E. De Mulder, N. Mentens, S. B. Örs, B. Preneel, G. Vandenbosch, I. Verbauwhede, "Side Channel Attacks and Fault Attacks on Cryptographic Algorithms", Revue HF Tijdschrift 2004(3), pp. 36-45, 2004.

LNCS Publications

5. N. Mentens, L. Batina, B. Preneel, I. Verbauwhede, "A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-box", In the Proceedings of the RSA Conference on Topics in Cryptology (CT-RSA),

A. Menezes (ed.), Lecture Notes in Computer Science 3376, pp. 323-333, Springer Verlag, 2005.

6. N. Mentens, L. Batina, B. Preneel, I. Verbauwhede, "Time-memory Trade-off Attack on FPGA Platforms: UNIX Password Cracking," In the Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC), K. Bertels, J. M. P. Cardoso, S. Vassiliadis (eds.), Lecture Notes in Computer Science 3985, pp. 323-334, Springer-Verlag, 2006.

7. K. Sakiyama, N. Mentens, L. Batina, B. Preneel, I. Verbauwhede, "Reconfigurable Modular Arithmetic Logic Unit for High-performance Public-key Cryptosystems", In the Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC), K. Bertels, J. M. P. Cardoso, S. Vassiliadis (eds.), Lecture Notes in Computer Science 3985, pp. 323-334, Springer-Verlag, 2006.

8. L. Batina, N. Mentens, K. Sakiyama, B. Preneel, I. Verbauwhede, "Low-cost Elliptic Curve Cryptography for Wireless Sensor Networks", In the Proceedings of the 3rd European Workshop on Security and Privacy in Ad Hoc and Sensor Networks (ESAS), L. Buttyan, V. Gligor, D. Westhoff (eds.), Lecture Notes in Computer Science 4357, pp. 6-17, Springer-Verlag, 2006.

International Conference Articles

9. N. Mentens, S. B. Örs, B. Preneel, J. Vandewalle, "An FPGA Implementation of a Montgomery Multiplier over GF(2^m)", In the Proceedings of the 7th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS), pp. 121-128, 2004.

10. N. Mentens, S. B. Örs, B. Preneel, "An FPGA Implementation of an Elliptic Curve Processor over GF(2^m)", In the Proceedings of the 14th ACM Great Lakes Symposium on VLSI (GLSVLSI), pp. 454-457, 2004.

11. L. Batina, N. Mentens, S. B. Örs, B. Preneel, "Serial Multiplier Architectures over GF(2^n) for Elliptic Curve Cryptosystems", In the Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference (MELECON), pp. 779-782, 2004.

12. L. Batina, N. Mentens, B. Preneel, I. Verbauwhede, "A New Systolic Architecture for Multiplication in GF(2^n)", In the Proceedings of the IFAC Workshop on Programmable Devices and Systems (PDS), pp. 461-466, 2004.

13. N. Mentens, L. Batina, B. Preneel, I. Verbauwhede, "An FPGA Implementation of Rijndael: Trade-offs for Side-channel Security", In the Proceedings of

the IFAC Workshop on Programmable Devices and Systems (PDS), pp. 493-498, 2004.

14. L. Batina, N. Mentens, I. Verbauwhede, "Side-channel Issues for Designing Secure Hardware Implementations", In the Proceedings of the 11th IEEE International On-Line Testing Symposium (IOLTS), C. Metra, K. Roy, L. Anghel, M. Nicolaidis (eds.), pp. 118-121, IEEE Computer Society Press, 2005.

15. L. Batina, N. Mentens, B. Preneel, I. Verbauwhede, "Side-channel Aware Design: Algorithms and Architectures for Elliptic Curve Cryptography over GF(2^n)", In the Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP), S. Vassiliadis, N. Dimopoulos, S. Rajopadhye (eds.), pp. 350-355, IEEE Computer Society Press, 2005.

16. L. Batina, N. Mentens, B. Preneel, I. Verbauwhede, "Flexible Hardware Architectures for Curve-based Cryptography", In the Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), pp. 4839-4842, 2006.

17. N. Mentens, K. Sakiyama, L. Batina, I. Verbauwhede, B. Preneel, "FPGA-oriented Secure Data Path Design: Implementation of a Public Key Coprocessor", In the Proceedings of the 16th IEEE International Conference on Field Programmable Logic and Applications (FPL), pp. 133-138, 2006.

18. N. Mentens, K. Sakiyama, B. Preneel, I. Verbauwhede, "Efficient Pipelining for Modular Multiplication Architectures in Prime Fields", In the Proceedings of the 17th ACM Great Lakes Symposium on VLSI (GLSVLSI), pp. 534-539, 2007.

19. L. Batina, J. Guajardo, T. Kerins, N. Mentens, P. Tuyls, I. Verbauwhede, "Public-Key Cryptography for RFID-Tags", In the Proceedings of the IEEE International Workshop on Pervasive Computing and Communication Security (PerSec), 6 pages, 2007.

20. L. Batina, N. Mentens, K. Sakiyama, B. Preneel, I. Verbauwhede, "Public-Key Cryptography on the Top of a Needle", In the Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1831-1834, 2007.

21. N. Mentens, K. Sakiyama, L. Batina, B. Preneel, I. Verbauwhede, "A Side-channel Attack Resistant Programmable PKC Coprocessor for Embedded Applications", To appear in the Proceedings of the International Symposium on Systems, Architectures, Modeling and Simulation (SAMOS), 8 pages, 2007.

Other Articles

22. J. Lano, N. Mentens, B. Preneel, I. Verbauwhede, "Power Analysis of Synchronous Stream Ciphers with Resynchronization Mechanism", In the Workshop Record of the ECRYPT Workshop on the State of the Art of Stream Ciphers (SASC), pp. 327-333, 2004.

23. L. Batina, J. Lano, N. Mentens, B. Preneel, I. Verbauwhede, S. B. Örs, "Energy, Performance, Area versus Security Trade-offs for Stream Ciphers", In the Workshop Record of the ECRYPT Workshop on the State of the Art of Stream Ciphers (SASC), pp. 302-310, 2004.

24. N. Mentens, L. Batina, B. Preneel, I. Verbauwhede, "Cracking Unix Passwords using FPGA Platforms", In the Workshop Record of the ECRYPT Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS), pp. 83-91, 2005.

25. A. Braeken, J. Lano, N. Mentens, B. Preneel, I. Verbauwhede, "SFINKS: A Synchronous Stream Cipher for Restricted Hardware Environments", In the Workshop Records of the ECRYPT Symmetric Key Encryption Workshop (SKEW), 19 pages, 2005.

26. L. Batina, S. Kumar, J. Lano, K. Lemke, N. Mentens, C. Paar, B. Preneel, K. Sakiyama, I. Verbauwhede, "Testing Framework for eSTREAM Profile II Candidates", In the Workshop Record of the ECRYPT Workshop on the State of the Art of Stream Ciphers (SASC), 9 pages, 2006.

27. K. Sakiyama, L. Batina, N. Mentens, B. Preneel, I. Verbauwhede, "Small-footprint ALU for Public-key Processors for Pervasive Security," In the Workshop Record of the ECRYPT Workshop on RFID Security, 12 pages, 2006.

28. L. Batina, J. Guajardo, T. Kerins, N. Mentens, P. Tuyls, I. Verbauwhede, "Public Key Cryptography for RFID-Tags", In the Workshop Record of the ECRYPT Workshop on RFID Security, 16 pages, 2006.

29. L. Batina, J. Guajardo, T. Kerins, N. Mentens, P. Tuyls, I. Verbauwhede, "An Elliptic Curve Processor Suitable for RFID-tags", In the Workshop Record of the 1st Benelux Workshop on Information and System Security (WISSec), 14 pages, 2006.