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

Secure Design Methodology and Implementation for Embedded Public-key Cryptosystems

Promotors:

Prof. dr. ir. Ingrid Verbauwhede
Prof. dr. ir. Bart Preneel

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Kazuo SAKIYAMA

December 2007

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

Secure Design Methodology and Implementation for Embedded Public-key Cryptosystems

Jury:
Prof. Carlo Vandecasteele, voorzitter
Prof. Ingrid Verbauwhede, promotor
Prof. Bart Preneel, promotor
Prof. Francky Catthoor
Prof. Wim Dehaene
Prof. Takao Onoye (Osaka University, Japan)
Prof. Patrick Schaumont (Virginia Tech, USA)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Kazuo SAKIYAMA

U.D.C. 681.3*C3

December 2007

D/2007/7515/122

ISBN 978-90-5682-890-5

Acknowledgments

It is my pleasure to thank many people for helping me to realize my Ph.D. here in Leuven.

First, I would like to express my deepest gratitude to my promoters, Prof. Bart Preneel and Prof. Ingrid Verbauwhede. I am especially thankful for giving me the chance to be a Ph.D. researcher at COSIC. Their constant guidance and support during the past three years were of supreme importance for this thesis to be completed.

I would like to thank Prof. Francky Catthoor and Prof. Wim Dehaene for accepting to be on my advisory committee and for giving me many significant suggestions and comments to improve this dissertation. My gratitude also goes to Prof. Patrick Schaumont and Prof. Takao Onoye for their accepting to be my external committee and for providing valuable remarks on this manuscript. I also would like to thank the chairman of the jury, Prof. Carlo Vandecasteele.

Many thanks to my colleagues for the great time I had at COSIC. I enjoyed their friendship and support. Special thanks to dr. Lejla Batina, dr. Nele Mentens and Benedikt Gierlichs for their insightful discussions through many research projects. I would like to thank dr. Frederik Vercauteren for his precious help on mathematical problems. I would also like to thank dr. Svetla Nikova for her friendliness and help.

I am grateful to Junfeng Fan for his motivation and tenacity in problem-solving. Many people have contributed to this research. I would like to mention especially dr. Kerstin Lemke-Rust, Elke De Mulder, Dries Schellekens, Miroslav Knezevic, Vesselin Velichkov, Joris Plessers, Peter Schreurs, Caroline Vanderheyden and Yong Ki Lee from UCLA. Many thanks to Péla Noé and Elvira Wouters for their administrative support.

Besides my COSIC colleagues, I would like to thank my ESAT colleague Yves Vanderperren for his cooperation in class projects. Furthermore, I would like to express my sincere appreciation to dr. Hiroshi Nakajima for his supervision and kind support during my internship.

With gratitude, I would like to mention that my Ph.D. research was funded by a research grant from the K.U. Leuven, FWO project (EMSEC), EC–FP6 projects

(ECRYPT, SCARD and SESOC) and IBBT project (QoE).

Finally, I would like to thank my family for their thoughtful consideration and unconditional support, especially to my wife Miwa.

Kazuo SAKIYAMA
Leuven, December 2007

Abstract

Efficient embedded systems are implemented taking into account both hardware and software (HW/SW). In the security domain, cryptosystems need to be resistant against Side-Channel Attacks (SCAs) to protect secret information. Therefore trade-offs between cost, performance and security need to be explored when implementing cryptosystems. The goal for this thesis is to find the best architecture by investigating the trade-offs. The first contribution of this thesis focuses on a HW/SW architecture for Public-Key Cryptography (PKC). We introduce a new scalable and flexible Modular Arithmetic Logic Unit (MALU) that can be used for both RSA and curve-based cryptosystems such as Elliptic Curve Cryptography (ECC) and Hyper-Elliptic Curve Cryptography (HECC). The MALU is the main block in the hardware coprocessor and can accelerate modulo n operations and modular operations over GF(2^m) efficiently. We conclude that the proposed HW/SW platform can be used commonly for developing public-key cryptosystems. The second part of this thesis deals with several case studies that explore the cost and performance trade-offs based on the proposed platform. Two extreme examples of public-key implementations will be introduced; one offers very high performance that is necessary for powerful security systems such as banking servers. By exploiting multi-level parallelism, the proposed ECC processor can perform more than 80 000 point multiplications per second. Another one is targeting a low-power application such as passive RFID tags. We show that the compact version of the MALU consumes less than 30 μW @500 kHz. In addition, we discuss a system-level design flow that can be used for evaluating the security level of hardware implementations against power analysis attacks. The design flow offers an environment to get a quick and correct evaluation of the first order attacks. In this way, we can take the cost for SCA resistance into account in an early stage of the design.

Samenvatting

Efficiënte ingebedde systemen kunnen zowel in hardware als in software geïmplementeerd worden (HW/SW). In het domein van databeveiliging moeten cryptosystemen bestand zijn tegen neven-kanaal aanvallen om geheime informatie te beschermen. Daarom is het nodig om afwegingen tussen kost, performantie en veiligheid te evalueren bij de implementatie van cryptosystemen. Het doel van deze thesis is het vinden van de beste architectuur door deze afwegingen te onderzoeken. De eerste bijdrage van deze thesis is een HW/SW architectuur voor publieke-sleutel cryptografie. We introduceren een nieuwe schaalbare en flexibele Modulaire Arithmetische Logische Unit (MALU) die kan gebruikt worden voor zowel RSA als cryptografie gebaseerd op elliptische en hyperelliptische krommen. De MALU is de belangrijkste component in de hardware coprocessor en zorgt voor een efficiënte versnelling van modulo n bewerkingen en modulaire bewerkingen in GF(2^m). We besluiten dat het voorgestelde HW/SW platform makkelijk kan gebruikt worden voor het ontwikkelen van publieke-sleutel cryptosystemen. Het tweede gedeelte van deze thesis behandelt daarom verschillende gevalstudies die kost- en performantie-afwegingen evalueren op basis van het voorgestelde platform. Twee uiteenlopende voorbeelden van publieke-sleutel implementaties worden besproken; de ene implementatie biedt een hoge performantie die nodig is voor krachtige beveiligingssystemen zoals servers voor banken. Door gebruik te maken van meerdere niveaus van parallellisme kan de voorgestelde elliptische kromme processor meer dan 80 000 puntvermenigvuldigingen per seconde berekenen. Een tweede implementatie mikt op laag-vermogen toepassingen zoals passieve RFID tags. We tonen aan dat een compacte versie van de MALU minder dan 30 μW @500 kHz verbruikt. Bovendien bespreken we een ontwerpmethodologie op systeem-niveau die kan gebruikt worden voor de evaluatie van het veiligheidsniveau van hardware implementaties met betrekking tot vermogenaanvallen. De ontwerpmethodologie biedt een omgeving die toelaat om een snelle en correcte evaluatie van eerste-orde aanvallen uit te voeren. Op deze manier kunnen we de kost voor resistentie tegen neven-kanaal aanvallen inschatten in het begin van de ontwerpcyclus.

Contents

List of Abbreviations

AA	Acoustic Analysis
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ATM	Automated Teller Machine
AtoP	Affine to Projective coordinates converter
bps	bit(s) per second
BRAM	Block RAM
CMOS	Complementary Metal Oxide Semiconductor
CPA	Correlation Power Analysis
CPRAM	Coprocessor RAM
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
CP	Carry-Propagate
CS	Carry-Save
CSA	Carry-Save Adder
DA	Design Automation
DBC	Data Bus Controller
DPA	Differential Power Analysis
DSA	Digital Signature Algorithm
DSP	Digital Signal Processor
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
ECC_M	ECC using the Montgomery powering ladder method
ECC_KC	ECC using Koblitz Curves
ECDSA	EC Digital Signature Algorithm
ECM	EC factoring Method
ECPA	EC Point Addition
ECPD	EC Point Doubling
ECPM	EC Point Multiplication
EMA	Electromagnetic Analysis

EX	Execution
F/F	Flip-Flop
FA	Full Adder
FIOS	Finely Integrated Operand Scanning
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GF	Galois Field
HA	Half Adder
HDL	Hardware Description Language
HEC	Hyper-Elliptic Curve
HECC	Hyper-Elliptic Curve Cryptography
HW	Hardware
I/O	Input/Output
IBC	Instruction Bus Controller
IC	Integrated Circuit
IEEE	The Institute of Electrical and Electronics Engineers
IF/D	Instruction Fetch/Decode
ILP	Instruction-Level Parallelism
IOB	Input/Output Block
IQB	Instruction Queue Buffer
IRAM	Internal RAM
ISS	Instruction Set Simulator
LSB	Least Significant Bit
LUT	Look Up Table
MALU	Modular Arithmetic Logic Unit
MALUb	MALU over binary fields
MALUn	MALU for modulo n operations
MALUD	MALU supporting Dual fields
MOF	Mutual Opposite Form
MSB	Most Significant Bit
MUX	Multiplexer
MtoN	Montgomery to Normal representation converter
NAF	Non-Adjacent Form
NIST	National Institute of Standards and Technology
NTRU	N-th degree truncated polynomial ring
NtoM	Normal to Montgomery representation converter
N/A	Not Applicable
OPB	On-chip Peripheral Bus
P-MALUb	Parallelized MALU over binary fields
PA	Power Analysis
PDA	Personal Digital Assistant
PKC	Public-Key Cryptography

PROM	Program ROM
PtoA	Projective to Affine coordinates converter
R/W	Read/Write
RAM	Random Access Memory
RF	Register File
RFID	Radio Frequency Identification
RNG	Random Number Generator
ROM	Read Only Memory
RSA	Rivest-Shamir-Adelman
RT	Register Transfer
SCA	Side-Channel Attack
SPA	Simple Power Analysis
SRAM	Static RAM
STA	Static Timing Analysis
SW	Software
TA	Timing Analysis
TCPC	Toggle Count Per Clock
TDEA	Triple Data Encryption Algorithm
TNAF	τ-NAF
VHDL	Very high speed integrated circuit HDL
WDDL	Wave Dynamic Differential Logic
XOR	Exclusive OR
XRAM	External RAM

List of Notation

p, q	prime numbers
n	product of p and q
$P(x)$	irreducible polynomial
$\mathrm{GF}(p)$	finite field with p elements; prime field
$\mathrm{GF}(2^m)$	finite field with 2^m elements; binary fields
$\mathbb{Z}[\tau]$	the ring of polynomials in τ with integer coefficients
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
$\mathrm{lcm}(x, y)$	least common multiple of x and y
$\gcd(x, y)$	greatest common divisor of x and y
$\deg(f)$	degree of a polynomial f
$\lambda(n)$	Carmichael function of n; $\lambda(n) = \mathrm{lcm}(p-1, q-1)$
$\varphi(n)$	Euler's totient function function of n; $\varphi(n) = (p-1)(q-1)$
$w(k)$	Hamming weight of the positive integer k
E	elliptic curve
C	hyperelliptic curve
$\#E$	the number of rational points on the elliptic curve E
$O(f(x))$	description of how the size of the input x affects computational resources
$\mathcal{O}$	point at infinity on an elliptic curve
P, Q, P_1, P_2	points on an elliptic curve
D, E	divisor on a hyperelliptic curve
k	scalar
kP	point multiplication of an elliptic curve point P with a scalar k
kD	divisor multiplication of a hyperelliptic curve divisor D with a scalar k

(x, y)	affine point
(X, Y, Z)	projective point
MALUb	MALU over GF(2^m)
MALUD	Dual-field MALU
MALUn	MALU for modulo n operation
$\widetilde{S}$	bit inversion of integer S
$\bar{1}$	-1
$X \| Y$	bit concatenation of X and Y
$\ll x$	left-shift operation over x bits
$\gg x$	right-shift operation over x bits
$\lceil x \rceil$	the smallest integer larger than or equal to x

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

1.1 Introduction

A highly developed information society has changed our lives drastically. Many types of data are converted into digital information and we often communicate private information via an open network such as the Internet. Therefore, it becomes very important to protect the digital information appropriately in order to prevent information leakage and to detect impersonation and data substitution. Public-key Cryptography (PKC) plays an important role in many types of cryptographic applications including key agreement and digital signature. Because it allows for secure data transfer via an insecure channel without prior secret key exchange. However, the performance of Elliptic Curve Cryptography (ECC) [43, 60] and RSA [70], the most widely accepted PKC, is several orders of magnitude lower than that of a secret-key cryptography such as AES (Advanced Encryption Standard) [16, 33, 64]. In this sense, it is challenging to implement compact and high-performance public-key cryptosystems in embedded devices such as RFID tags, smart cards and PDAs because they have a limited power budget and limited silicon resources. Throughout the thesis, trade-offs between cost, performance and security are discussed in detail for public-key implementations on hardware and software.

1.1.1 Public-key Cryptography

Cryptography deals with the science of information security and authenticity. Until the middle of 70's, the cryptographic systems (cryptosystems) used for secure communications were based on a symmetric-key (or private-key) cryptosystem as shown in Fig. 1.1. In this system, Alice and Bob own the same secret key k that is exchanged on beforehand. When Alice wants to send a plaintext (or message)

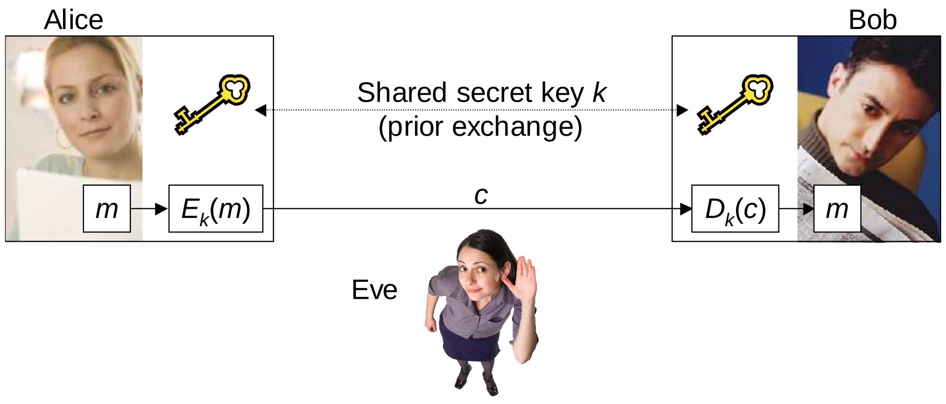

Figure 1.1: Basic model for a symmetric-key cryptosystem.

to Bob in a secure way, she *encrypts* the message m with the secret key k and sends the ciphertext c (or encrypted message) to Bob. The encryption operation can be written as $c = E_k(m)$. When Bob needs to read the message, he *decrypts* the ciphertext with the same secret key k. This decryption process can be written as $D_k(c) = D_k(E_k(m)) = m$. Some examples of symmetric-key cryptosystems are TDEA (Triple Data Encryption Algorithm) [65, 66] and AES [64].

Diffie and Hellman introduced the idea of PKC [18] in 1976. They showed that one can eliminate the need for prior secret key exchange that is necessary for symmetric-key cryptosystems. Since the early 90's, public-key cryptosystems have been an essential building block for digital communication. PKC allows for key exchange and digital signature over insecure channels without prior key exchange. Figure 1.2 shows the basic model for a public-key cryptosystem. When Alice wants to send a message m to Bob in a secure way, she uses Bob's public key B to encrypt a message m by computing $E_B(m) = c$ where c is the ciphertext. Bob can decrypt the ciphertext with his private key b. The decryption is performed by $D_b(c) = D_b(E_B(m)) = m$.

An example of a digital signature scheme is illustrated in Fig. 1.3. When Alice wants to send a message to Bob with her digital signature, she signs the message using her private key a as $S_a(H(m))$ and sends it with the message m. Here, the hash value of the message $H(m)$ is signed in order to avoid signing the complete message. Bob can verify the signature by computing $V_A(S_a(H(m))$ where $V_A(\cdot)$ is a function for the signature verification with Alice's public key A.

The most popular and most widely used public-key cryptosystems are RSA introduced by Rivest, Shamir and Adleman in 1977 [70] and ECC proposed independently by Koblitz and Miller in 1985 [43, 60]. ECC is often preferred for a compact hardware implementation because much shorter key-lengths are needed.

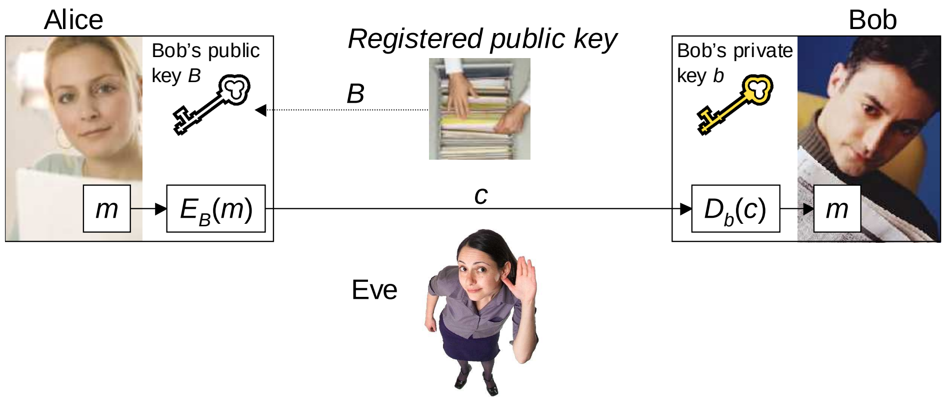

Figure 1.2: Basic model for a public-key cryptosystem.

Blake, Seroussi and Smart argue in [8] that 160-bit ECC has the same security level as 1024-bit RSA. Therefore, ECC can obtain higher performance, lower power consumption, and smaller area on most platforms with the exception of signature verification. Namely, ECC is considered a more suitable choice than RSA in embedded systems. Nevertheless, RSA is still important to support existing RSA-based infrastructure. Therefore, PKC over a prime field GF(p) gives the opportunity to support both RSA and ECC on the *same* hardware platform.

Another appealing candidate for PKC is Hyper-Elliptic Curve Cryptography (HECC). Recently many software and hardware implementations of HECC have been described, while more theoretical work has shown that HECC is also secure for curves with a small genus [98]. Other public-key cryptosystems have been also proposed such as Rabin and NTRU. However, this thesis mainly focuses on RSA, ECC and HECC. An overview of these public-key cryptosystems is introduced in the following sections.

1.1.2 Implementations of Public-key Cryptography

Hardware and software implementations of RSA and curve-based cryptography over GF(p) can be facilitated by common arithmetic operations that perform modular operations. However, the operation sizes are different as mentioned already, *e.g.* 1024 bits for RSA and 160 bits for ECC. This fact makes it difficult to implement a cryptographic processor (cryptoprocessor) that supports both RSA and ECC over GF(p) efficiently.

Curve-based cryptography can also exist over GF(2^m), which means that two different arithmetic operations are needed in order to support both operations over GF(2^m) and GF(p). It is also challenging to support curve-based cryptography

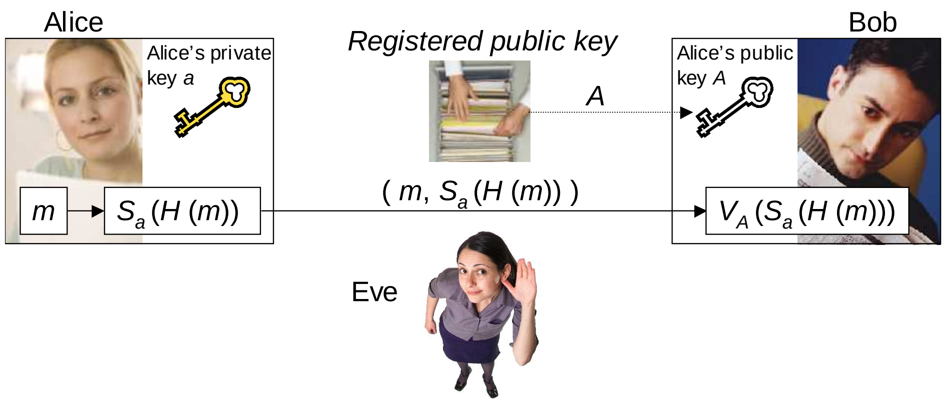

Figure 1.3: Example of a digital signature algorithm.

over both types of fields efficiently. In order to clarify the architectural problems to be solved, we discuss each public-key cryptosystem from an implementation point of view.

RSA

RSA is the first public-key cryptosystem. The security strength of RSA relies on the fact that factorization of large numbers is difficult. A key pair is generated by the following procedure.

- Generate two large random primes p and q.
- Compute $n = pq$.
- Compute $\lambda(n) = \mathrm{lcm}(p-1, q-1)$
- Generate a random integer e with $1 < e < \lambda(n)$ and $\gcd(e, \lambda(n))$=1
- Compute $d \equiv 1/e \bmod \lambda(n)$
- The public key consists of e and n.
- The private key consists of d or p and q.

Encryption can be performed by computing

$$c \equiv E_e(m) \equiv m^e \pmod{n}, \tag{1.1}$$

where c and m are a ciphertext and a plaintext, respectively. The message can be decrypted by performing

$$D_d(c) \equiv c^d \equiv m^{1+k\lambda(n)} \equiv m \pmod{n}\,. \tag{1.2}$$

The most time consuming operation is modular exponentiation. A straightforward algorithm to compute modular exponentiation repeats modular multiplications and modular squarings. For instance, $c^{15}(= c^{(1111)_2}) \bmod n$ can be computed by

$$t = c^2,\quad t = t \cdot c,\quad t = t^2,\quad t = t \cdot c,\quad t = t^2,\quad t = t \cdot c \pmod{n}\,. \tag{1.3}$$

This is called the left-to-right binary method [57]. We need three modular squarings and three modular multiplications in this case. Therefore, it is obvious that the most fundamental approach to accelerate modular exponentiation is to implement a computationally efficient modular squarer and multiplier for large integers (*e.g.* 1024 bits).

Considering that $c^{15} = c^{16-1}(= c^{(1000\bar{1})_2})$, the same result can be obtained by performing

$$t = c^2,\quad t = t^2,\quad t = t^2,\quad t = t^2,\quad t = t/c \pmod{n}\,. \tag{1.4}$$

The main idea of this computational sequence is generalized as the so-called NAF (Non-Adjacent Form) method. If an efficient modular divider is available, this method is also attractive because we could have less modular operations compared to Eq. (1.3). However, modular division is normally computationally expensive and slower than modular multiplications. Therefore the NAF method is not normally used for accelerating modular exponentiation in RSA. On the other hand, the NAF works efficiently for ECC as we will discuss in Sect. 2.4.5.

Another algorithm can also reduce the number of modular squarings and modular multiplications by computing

$$s = c^2,\quad s = s \cdot c,\quad t = s^2,\quad t = t^2,\quad t = s \cdot t \pmod{n}\,. \tag{1.5}$$

By introducing another variable s, the same modular exponentiation can be performed with one less modular multiplication. This algorithm prepares several pre-computed variables ($s = c^3$ is pre-computed in this case) and accelerates the modular exponentiation. This idea is generalized as the so-called k-ary exponentiation method [57] that will be introduced in Sect. 2.3.1 (Alg. 2.8).

There are two important building blocks to optimize for RSA cryptosystems; the first is a datapath block that operates modular operations and the second is a controller part that handles a sequence for modular exponentiation according to one of the modular exponentiation algorithms.

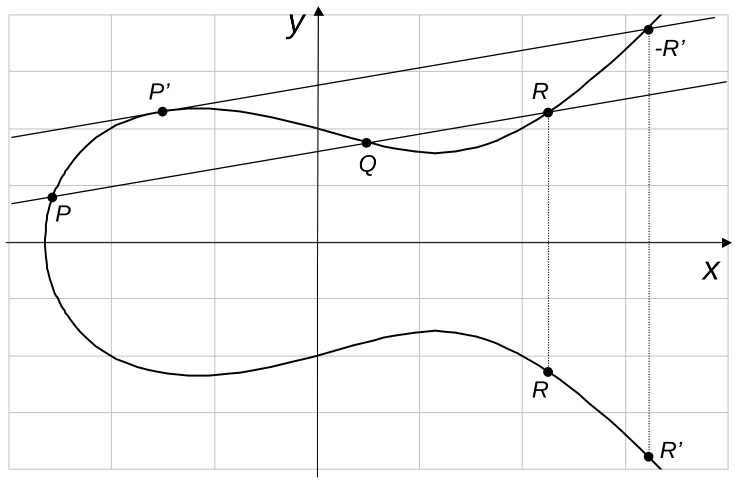

Figure 1.4: Point addition and doubling on an elliptic curve ($y^2 = x^3 - x + 1$).

Curve-based Cryptography

Here, we consider some background information for curve-based cryptography; for HECC we are interested in genus 2 curves over a binary field only. We mention the basic algorithms and the structure of the operations. Good references for the mathematical background are [8, 45, 58].

The main operation in ECC is point multiplication which multiplies a point on an elliptic curve with a scalar, resulting again in a point on the curve. When the point multiplication of a point P with a scalar k results in the point Q, this is denoted by $Q = kP$. The point multiplication can be computed with two basic point operations that are point addition and point doubling. Figure 1.4 graphically explains the point operations, point addition ($R = P + Q$) and point doubling ($R' = 2P'$). The point at infinity $\mathcal{O}$ is the neutral element, similar to the number 0 in ordinary addition. Thus, $P + \mathcal{O} = P$ and $P + (-P) = \mathcal{O}$ for all points P.

For HECC, the main operation is divisor multiplication kD where k is a scalar. A divisor D is a formal sum of points on a hyperelliptic curve, *i.e.* $D = \sum m_P P$ and its degree is $\deg(D) = \sum m_P$. In the same way as elliptic curve scalar multiplication, divisor multiplication can be computed by two divisor operations, divisor addition and divisor doubling. Further details will be discussed in Chapter 2 including the corresponding techniques to Eqs. (1.3), (1.4) and (1.5).

The general hierarchical structure for operations required for implementations of curve-based cryptography is given in Fig. 1.5. Point/divisor multiplication is at the top level. At the next (lower) level are the point/divisor group operations.

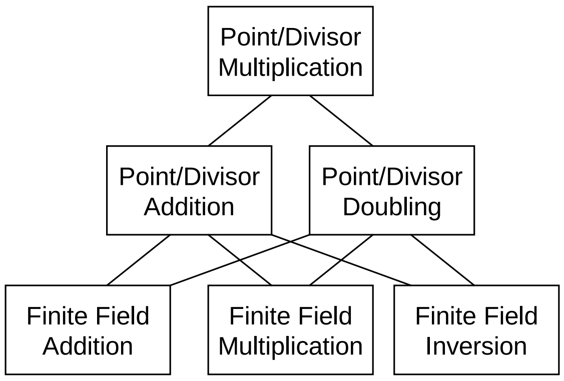

Figure 1.5: Basic scheme of the hierarchy for ECC/HECC operations.

The lowest level consists of finite field operations, such as finite field addition, multiplication and inversion required to perform the group operations. The only difference between ECC and HECC is the sequence of operations at the middle level. The sequence for HECC is more complex compared to the ECC point operations. However HECC uses shorter operands.

1.1.3 Design Flow for a Public-key Cryptoprocessor

This section introduces the design flow that we use for hardware and software co-design in this thesis. The public-key cryptosystems have a controller block which deals with hierarchical instructions. In the case of HW/SW co-design, a micro-controller takes a major role in the controlling operations.

In order to support various PKC, the cryptoprocessor can be separated into two parts, the datapath and the controller part. The datapath performs modular operations with a large integer number (*e.g.* up to 4096-bit modular multiplication). The results and intermediate variables are stored in RAM for RSA and ECC operations.

Our Architecture

There are a lot of design choices to implement PKC depending on applications. As mentioned already, our design starts from HW/SW co-design that offers a programmable environment to explore different algorithms in three levels as introduced in Fig. 1.5: (1) point/divisor multiplication, (2) point/divisor operations, and (3) modular arithmetic datapath. With our design environment, algorithms and implementations for each level can be optimized independently; for instance, we can focus on the datapath block to have an efficient modular multiplier. However, when we determine the optimal operation form (*e.g.* $AB + C \pmod{n}$) in the datapath, the choice of algorithms in the higher levels need to be considered simultaneously. It is also possible to explore this type of optimization with our HW/SW co-design platform.

In this regard, there are several DA (Design Automation) tools available for PKC applications. For example, Target Compiler's tools [93] support a high-level architectural exploration for implementing embedded processors and programmable ASICs. Another example is Tensilica's processor that is configurable with its customizable instruction sets in order to accelerate a specific operation efficiently [96]. They are both attractive for our purpose, however we take a custom design approach to investigate the public-key cryptosystems step by step.

Hierarchical Instructions

We organize FSMs in four different levels from 1st-level to 4th-level (lowest) as shown in Table 1.1. This is because a PKC operation can be made up of a combination of lower-level operations. For instance, modular addition can be computed with addition and subtraction. Point addition can be performed with modular multiplications and modular additions. Thus, each FSM for PKC works in cooperation with other FSMs.

The major instructions of the PKC core hardware are summarized in Table 1.1. The lowest instructions are most tightly coupled with the datapath and compute primitive operations among the instructions. In other words, the FSMs at this level of instructions control datapath, register file and data RAM to execute operations with large integers. The 3rd-level or higher operations are composed of lower-level instructions.

Modeling for Each Abstraction-level

Considering the simulation time, it is not acceptable to pursue the VHDL simulations to verify all levels of the instructions. For instance, the simulation time of a 1024-bit RSA exponentiation results in a bottleneck of the design time. Therefore the verification of higher-level instructions needs higher-level abstraction models with an associated simulation set-up. That is, the design platform combines different abstraction-level designs and provides adaptive simulation models which can be used for quick performance estimation and functional verification, which results in a low-cost design of the cryptosystems.

For that purpose, we prepare three different design models that are Magma [2], GEZEL [28, 83, 84] and VHDL on each modeling accuracy as illustrated in Fig. 1.6. In general, a faster simulation is obtained with a model of lower accuracy and lower instruction level (which has less computational steps). This is also true in our case. Therefore, our strategy to implement public-key cryptoprocessors starts with the functional models running on Magma as shown in the left-side of Fig. 1.6. Once

Table 1.1: Instructions of the PKC core hardware categorized by level.

Level	Instruction	Operation
1st	`RSA`	Modular Exponentiation: $A^B \bmod n$
	`ECC_GFp`	ECC point multiplication over GF(p): kP
	`ECC_GF2m`	ECC point multiplication over GF(2^m): kP
	`HECC_GF2m`	HECC divisor multiplication over GF(2^m): kD
2nd	`PDBL_GFp`	Point doubling over GF(p): $2P$
	`PADD_GFp`	Point addition over GF(p): $P + Q$
	`PDBL_GF2m`	Point doubling over GF(2^m): $2P$
	`PADD_GF2m`	Point addition over GF(2^m): $P + Q$
	`DDBL_GF2m`	Divisor doubling over GF(2^m): $2D$
	`DADD_GF2m`	Divisor addition over GF(2^m): $D + E$
3rd	`MODMUL_Zn`	Multiplication modulo n: $AB \bmod n$
	`MODINV_GFp`	Modular inversion over GF(p): $A^{-1} \bmod p$
	`INVMOD_GF2m`	Modular inversion over GF(2^m): $A^{-1}(x) \bmod P(x)$
	`MODADD_Zn`	Addition modulo n: $(A + B) \bmod n$
	`MODSUB_Zn`	Subtraction modulo n: $(A - B) \bmod n$
4th (Lowest)	`MULT`	Multiplication: AB
	`ADD`	Addition: $A + B$
	`SUB`	Subtraction: $A - B$
	`MODMUL_GF2m`	Modular multiplication over GF(2^m): $A(x)B(x) \bmod P(x)$
	`MODADD_GF2`	Modular addition over GF(2): $A(x) \oplus B(x)$
	`STORE`	Store data to register file
	`LOAD`	Load data from register file
	`SET_KEY`	Store key for RSA/ECC/HECC to key register
	`SHIFT_KEY`	Shift key register

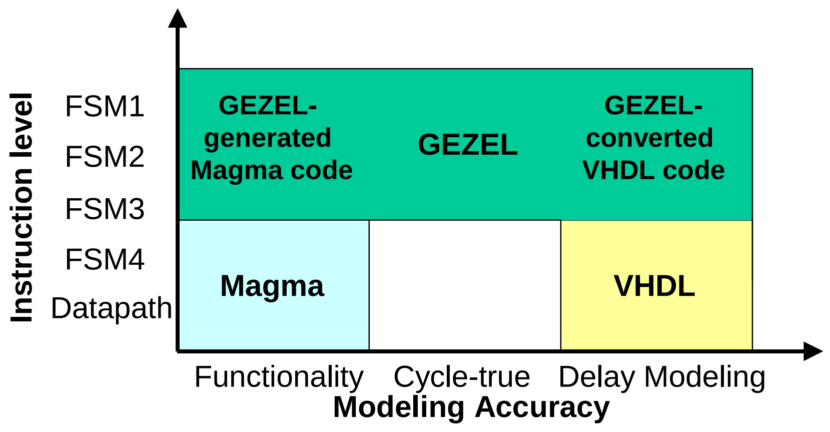

Figure 1.6: Design models with Magma, GEZEL and VHDL.

the functionality is verified, we can estimate the speed performance with the cycle-true simulation with the GEZEL code. The GEZEL code can be automatically translated into VHDL code. Those VHDL code can be used for estimating more accurate design performance including the critical path delay and the gate counts of the design.

However, we use VHDL simulations for the datapath and the level-4 FSM (denoted as FSM4 in Fig. 1.6). This is mainly for further optimization of the datapath that has a significant impact on the system performance. Namely, VHDL code are suitable for optimizing a design at the gate level. Another reason is that we can utilize an existing VHDL-coded hardware library that is highly optimized for a specific platform, *e.g.* dedicated multipliers or carry-propagate adders in FPGA. In this case, we can implement a new datapath simply by instantiating the VHDL libraries.

In addition, we simulate with VHDL some of level-3 instructions controlled by the FSM3 as far as they deal with data-dependent signals, *e.g.* modular addition and subtraction (see Alg. 2.1). On the other hand, the FSM1 and the FSM2 in Fig. 1.6 are performed in a fixed sequence or data-independently. The detailed design flow is explained in the next section.

Design Flow

Our strategy to bridging the modeling gap is to use GEZEL as a mediator between Magma and VHDL as follows:

1. Lower-level instructions are simulated with VHDL including VHDL code converted from GEZEL code.

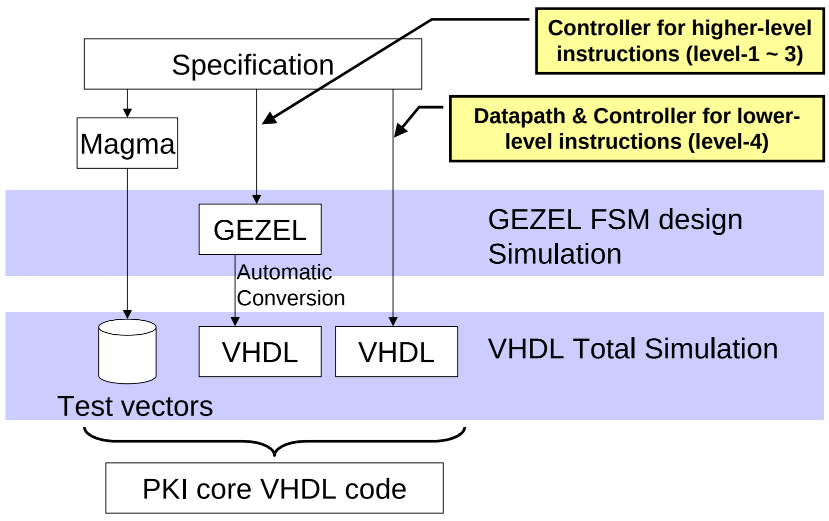
Controller for higher-level instructions (level-1 ~ 3)
Specification
Datapath & Controller for lower-level instructions (level-4)
Magma
GEZEL
GEZEL FSM design Simulation
Automatic Conversion
VHDL
VHDL
VHDL Total Simulation
Test vectors
PKI core VHDL code

Figure 1.7: Design flow for public-key hardware implementation with Magma, GEZEL and VHDL.

2. Higher-level instructions are simulated with Magma by using Magma code generated by GEZEL code.

Magma is a fast software package that is used for solving computationally hard problems in algebra, number theory, and so on. Thus, significantly fast functional simulations can be performed with Magma. Namely, Magma can be used for a behavioral model of the target datapath. At the same time, Magma generates the test vectors of GEZEL and VHDL simulations including RSA exponentiation and EC point multiplication.

We implement higher-level FSMs by using GEZEL which is a hardware description language and can offer a much faster simulation than VHDL. By extracting the computational sequence of an FSM from the GEZEL simulation, a Magma code corresponding to the FSM can be generated. Thus, further speed-up of the simulation is possible by using Magma when verifying the higher-level FSMs.

The GEZEL code is automatically converted to VHDL code too. Only the lowest FSMs and datapath are implemented with VHDL (partially) since it offers area/speed-efficient design by instantiating and combining optimized hardware resources, *i.e.* multipliers, adders and RAMs. The design flow is illustrated in Fig. 1.7.

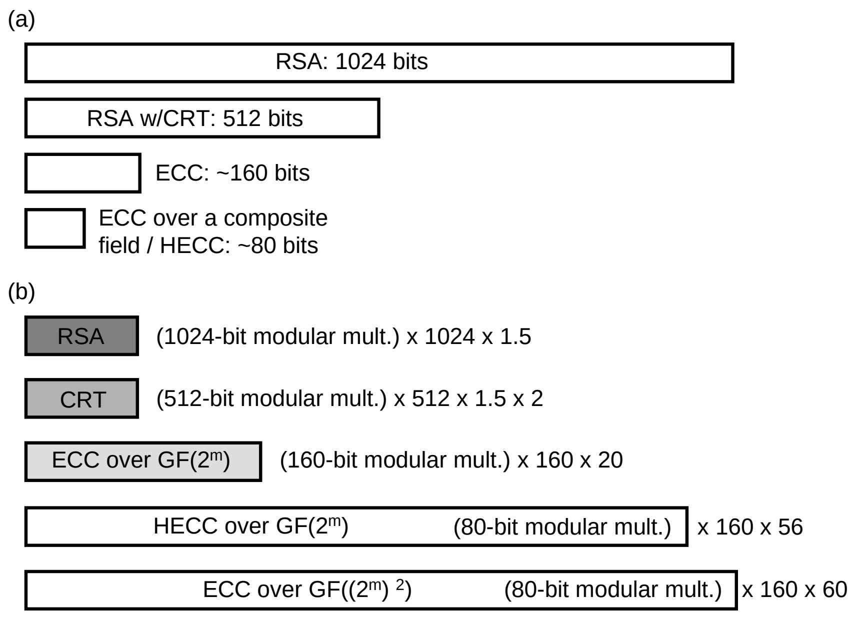

Figure 1.8: Design overview of different public-key cryptosystems. (a) Typical key length. (b) The number of modular multiplications.

1.2 Motivation

PKC is indispensable for secure digital communications in security systems including high performance applications (*e.g.* ATMs) and low power applications (*e.g.* smart cards and RFID tags). In order to satisfy the performance requirements of different public-key cryptosystems, the hardware has to support modular operations over GF(p) and GF(2^m) with different operation sizes (see Fig. 1.8a). Moreover, it is preferable to have scalability in performance, that is, when allocating more hardware resources, a higher performance should be obtained. In this thesis, a flexible and scalable datapath for PKC is proposed.

Having flexibility in the controller block is also necessary for public-key cryptosystems to support different computational sequences in RSA and curve-based cryptography (see Fig. 1.8b). In this regard, HW/SW co-design is the best choice because the computational sequences can be implemented in SW for a CPU and modular operations can be accelerated in HW. However, the communication overhead between SW and HW degrades the performance of PKC. The best HW/SW architecture is explored so that the architecture can provide efficient computations while maintaining a high programmability.

The latency of one RSA exponentiation or one Elliptic Curve (EC) point multiplication can be minimized by computing multiple instructions in parallel on a multi-core architecture. This architecture is attractive especially for curve-based cryptography since it has a more complicated computational sequence as shown in Fig. 1.8b and hence it could have a higher parallelism compared to RSA. On the other hand, PKC for low-power applications requires a compact implementation with a relatively slower performance.

Many datapath implementations have been proposed for modular operations over GF(p) and GF(2^m) [14, 15, 41, 56, 59, 82]. However, the most of them are not flexible enough to support various PKC applications efficiently. We propose a new datapath that enables modular operations modulo n or GF(2^m). We also introduce a dual-field datapath that supports both modular operations modulo n and operations over GF(2^m). The datapath is scalable, flexible and reconfigurable by setting the parameters for the operand size k, the digit size d, the number of the datapaths, etc.

For high performance applications, previous work mostly uses a *fixed* field size and a *hardwired* controller because the datapath can be highly optimized. However, it offers less scalability and flexibility. Considering such background, we investigate a high-speed programmable cryptoprocessor that uses multiple datapaths so that it can accelerate the computational sequences of ECC and HECC by using the datapaths in parallel and can support a wider field size by reconfiguring the connections between several datapaths. Thus, the cryptoprocessor can offer high-speed modular multiplications and additions for arbitrary field sizes with efficient use of hardware resources.

This thesis also includes the research for exploring the smallest possible hardware implementation of PKC for low-power applications. The work of Gaubatz *et al.* [27] discusses the necessity and the feasibility of PKC protocols in sensor networks. However, different from high-performance public-key cryptosystems, very tight constraints are put on implementations in the number of gates, power consumption, communication bandwidth, etc. In [27], the authors claim that NTRUEncrypt is implemented with 3000 gates and a power consumption of less than 20 μW @500 kHz. Inspired by their work, we explore the feasibility of using ECC and HECC for RFID tags by optimizing the arithmetic datapath further.

In addition, the trade-off between area, performance and security is of interest in the thesis. Here security has two meanings; one is security-level realized by changing the key lengths, and the other one is protection against Side-Channel Attacks (SCAs). The trade-offs are explored with several case studies of PKC implementations.

1.3 Thesis Organization and Contributions

The organization of the thesis is illustrated in Fig. 1.9.

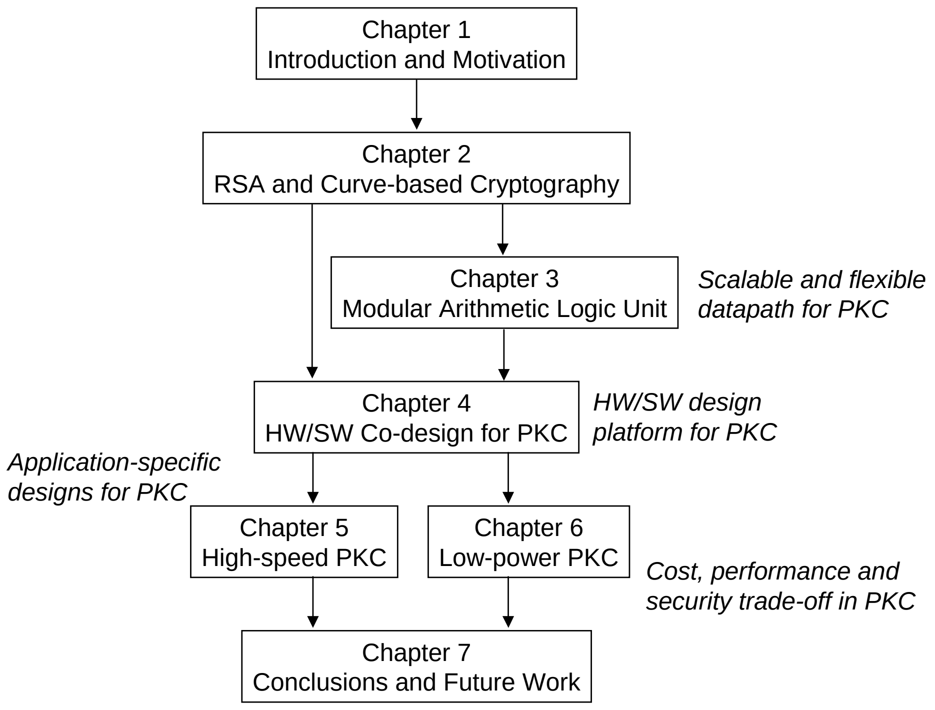
Chapter 1
Introduction and Motivation
Chapter 2
RSA and Curve-based Cryptography
Chapter 3
Modular Arithmetic Logic Unit
Scalable and flexible datapath for PKC
Chapter 4
HW/SW Co-design for PKC
HW/SW design platform for PKC
Application-specific designs for PKC
Chapter 5
High-speed PKC
Chapter 6
Low-power PKC
Cost, performance and security trade-off in PKC
Chapter 7
Conclusions and Future Work

Figure 1.9: Overview of the thesis organization.

Chapter 1 presents the introduction and motivations for this thesis. In Chapter 2, background information is provided on RSA and curve-based cryptography. Chapter 3 presents a scalable and flexible datapath unit that can support the operations for these public-key cryptosystems. The results are published in [71, 76, 77, 79]. Depending on the application, an appropriate PKC is implemented with the proposed datapath in later chapters.

Chapter 4 presents two cases of HW/SW co-design for RSA, ECC and HECC. The first implementation is the co-design for RSA and ECC over GF(p) on the 8051 micro-processor [74]. The second one is the co-design for ECC and HECC over GF(2^m) on the ARM micro-processor [72, 75]. Through these cases, the trade-off between cost and performance is discussed. Based on the proposed HW/SW co-design, a new alternative for high-speed and low-power public-key cryptosystems is explored in Chapters 5 and 6.

Chapter 5 presents high-speed public-key cryptoprocessors. First, we discuss a reconfigurable curve-based cryptoprocessor that accelerates point multiplication of ECC over GF(2^m), and HECC of genus 2 over GF(2^m) [75, 78]. Scalar multiplication of ECC/HECC is accelerated by exploiting Instruction-Level Parallelism

(ILP). Secondly, we present a high-speed public-key cryptoprocessor that exploits three-level parallelism in ECC over GF(2^m) for further speed-up [73].

In Chapter 6, a compact PKC implementation for Radio Frequency IDentification (RFID) tags is discussed. Here, the term "compact" means small die size and low power consumption. In this chapter, the possibility for public-key services for pervasive computing is investigated. We propose a hardware processor supporting ECC over binary fields GF(2^m), ECC over a composite field GF($(2^m)^2$), and HECC over binary fields GF(2^m) on curves of genus 2. The results are published in [4, 5, 71].

The cost, performance and security trade-offs are discussed throughout the thesis and Chapter 7 concludes the thesis and points out future work. Additionally, we describe a simple SCA resistant design flow that is used for identifying potential security bugs of our proposed designs in Appendix A. Efficient countermeasures for side-channel analysis attacks are also mentioned.

Contributions in this thesis are summarized as follows:

- Hardware implementation of scalable, flexible and reconfigurable datapath for PKC.
- HW/SW co-design for PKC.
- High-performance PKC design.
- Low-power PKC design.
- Trade-off between area, performance and security in PKC.

Chapter 2

Arithmetic for RSA and Curve-based Cryptography

2.1 Introduction

This section discusses the basic finite field arithmetic and algorithms used for RSA exponentiation and point/divisor multiplication for curve-based cryptography. The selection of the underlying finite fields and the algorithms has a profound impact on the trade-off between cost, performance and security. Hence, it is possible to discuss the trade-offs of this abstraction level.

2.2 Finite Field Arithmetic

2.2.1 Modulo n Operations

Arithmetic operations performed modulo n are called modular operations, where the result is an element of $0, 1, 2, \cdots, n-1$ and n is a positive integer of $n \geq 2$. They can be computed with integer arithmetic operations followed by a reduction modulo n. Modular addition and subtraction can be performed by Alg. 2.1. A straightforward algorithm to compute modular multiplication is shown in Alg. 2.2.

The reduction step needs a multi-precision divider for multi-precision inputs that are normally used in PKC. In general such a divisor is expensive in hardware, and hence this algorithm is rarely used. One of the most efficient methods to compute modular multiplication is the Montgomery algorithm that can avoid the trial division in the reduction step. The details are discussed in Chapter 3.

One possible approach to computing inversion modulo n is the use of Fermat's (little) theorem, *i.e.*

$$a^{-1} \equiv a^{\varphi(n)-1} \bmod n\,, \tag{2.1}$$

when $\gcd(a,n)=1$. In the case of ECC over prime fields, n is a prime p and $\varphi(p) = p-1$, so this theorem is applicable to ECC over GF(p) as $a^{-1} \equiv a^{p-2} \bmod p$. The theorem indicates that modular inversion over prime fields can be computed by modular exponentiation algorithms that are discussed in Sect. 2.3. The advantage of the use of Fermat's theorem is that cryptosystems can be efficiently implemented by utilizing a hardware resource for modular multiplication. However, because of the high computational costs, cryptosystems have to use as few modular inversions as possible. In this thesis, we use Fermat's theorem for modular inversion, and we aim to give compact inversion-less public-key implementations.

Algorithm 2.1: Algorithms for modular addition and subtraction modulo n.

```
Require: positive integer $n$,
    non-negative integers $a$, $b$
    with $a, b < n$.
Ensure: $(a + b) \bmod n$.
1: $T \leftarrow a + b$
2: if $T \geq n$ then
3:     $T \leftarrow T - n$
4: end if
5: Return $T$
```

```
Require: positive integer $n$,
    non-negative integers $a$, $b$
    with $a, b < n$.
Ensure: $(a - b) \bmod n$.
1: $T \leftarrow a - b$
2: if $T < 0$ then
3:     $T \leftarrow T + n$
4: end if
5: Return $T$
```

Algorithm 2.2: Algorithm for multiplication modulo n.

```
Require: positive integer $n$, non-negative integers $a$, $b$ with $a, b < n$.
Ensure: $ab \bmod n$.
1: $S \leftarrow ab$
2: $T \leftarrow \lfloor S/n \rfloor$
3: $S \leftarrow S - Tn$
4: Return $S$
```

2.2.2 Modular Operations over GF(2^m)

The binary field GF(2^m) is an extension of GF(2). Among various representations for the elements in GF(2^m), a polynomial basis $(1, \alpha, \alpha^2, \ldots, \alpha^{n-1})$, where α is a root of an irreducible polynomial $P(x)$ of degree m over GF(2), is of our interest in this thesis. In some specific cases, other representations show better features, *e.g.* the computational cost for modular squaring in HW is free by using the normal-basis representation. However, the polynomial-basis modular operations can be implemented on HW efficiently, and moreover it enables us to make the hardware architecture scalable and flexible. Reduction is done with an irreducible polynomial, $P(x)$.

Modular addition over GF(2^m) can be implemented simply by XORing the coefficients of the same degree of two operands. This is expressed in Alg. 2.3.

Algorithm 2.3: Algorithm for modular addition over GF(2^m).

```
Require: an irreducible polynomial $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$,
    polynomial-basis operands $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$.
Ensure: $(A(x) + B(x))$ mod $P(x)$.
1: $t_i \leftarrow a_i \oplus b_i$ $(0 \le i \le m-1)$
2: Return $T(x) = \sum_{i=0}^{m-1} t_i x^i$
```

A straightforward algorithm to compute modular multiplication is shown in Alg. 2.4.

Algorithm 2.4: Algorithm for modular multiplication over GF(2^m).

```
Require: an irreducible polynomial $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$,
    polynomial-basis operands $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$.
Ensure: $A(x)B(x)$ mod $P(x)$.
1: $S(x) \leftarrow A(x)B(x)$
2: $T(x) \leftarrow S(x)$ div $P(x)$
3: $S(x) \leftarrow S(x) + T(x)P(x)$
4: Return $S(x)$
```

The reduction step needs to compute the quotient in the polynomial long division of $S(x)$ by $P(x)$. This can be efficiently implemented by using a bit/digit-serial modular multiplier that is discussed in Chapter 3.

Fermat's (little) theorem can be applied to modular inversion algorithm over GF(2^m), *i.e.*

$$a^{-1} \equiv a^{2^m-2} \equiv (a^{2^{m-1}-1})^2 \pmod{P(x)}\,, \tag{2.2}$$

where a is a polynomial of $a \in$ GF(2^m). An optimal way to compute this modular exponentiation is the basis idea of Itoh and Tsujii [36]. This algorithm repeatedly uses modular squarings and multiplications and offers a better result than the binary method for modular exponentiation that will be discussed in Sect. 2.3. The main idea of the algorithm is expressed in Eq. (2.3). Depending on the value of k, a^{2^k-1} can be reformulated in two different ways. The resulting expressions contain

Table 2.1: Example of Itoh-Tsujii algorithm.

modular inversion over GF(2^{163})	Cost	modular inversion over GF(2^{83})	Cost
$a^{-1} = (a^{2^{162}-1})^2$	1S	$a^{-1} = (a^{2^{82}-1})^2$	1S
$a^{2^{162}-1} = (a^{2^{81}-1})^{2^{81}} a^{2^{81}-1}$	81S+1M	$a^{2^{82}-1} = (a^{2^{41}-1})^{2^{41}} a^{2^{41}-1}$	41S+1M
$a^{2^{81}-1} = a\{(a^{2^{40}-1})^{2^{40}} a^{2^{40}-1}\}^2$	41S+2M	$a^{2^{41}-1} = a\{(a^{2^{20}-1})^{2^{20}} a^{2^{20}-1}\}^2$	21S+2M
$a^{2^{40}-1} = (a^{2^{20}-1})^{2^{20}} a^{2^{20}-1}$	20S+1M	$a^{2^{20}-1} = (a^{2^{10}-1})^{2^{10}} a^{2^{10}-1}$	10S+1M
$a^{2^{20}-1} = (a^{2^{10}-1})^{2^{10}} a^{2^{10}-1}$	10S+1M	$a^{2^{10}-1} = (a^{2^{5}-1})^{2^{5}} a^{2^{5}-1}$	5S+1M
$a^{2^{10}-1} = (a^{2^{5}-1})^{2^{5}} a^{2^{5}-1}$	5S+1M	$a^{2^{5}-1} = a\{(a^{2^{2}-1})^{2^{2}} a^{2^{2}-1}\}^2$	3S+2M
$a^{2^{5}-1} = a\{(a^{2^{2}-1})^{2^{2}} a^{2^{2}-1}\}^2$	3S+2M	$a^{2^{2}-1} = a^2 a$	1S+1M
$a^{2^{2}-1} = a^2 a$	1S+1M		
Total	162S+9M	Total	82S+8M

a^{2^j-1} that can be used for the further reformulation by using the same equation. By repeating this procedure, modular inversion over GF(2^m) can be computed.

$$a^{2^k-1} = \begin{cases} a^{2^{2j}-1} = a^{(2^j-1)(2^j+1)} = (a^{2^j-1})^{2^j} a^{2^j-1} \quad (k = 2j : \text{even}) \\ \\ a^{2^{2j+1}-1} = a \cdot a^{2^{2j+1}-2} = a(a^{2^{2j}-1})^2 = a\{(a^{2^j-1})^{2^j} a^{2^j-1}\}^2 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (k = 2j+1 : \text{odd}) \end{cases} \tag{2.3}$$

Table 2.1 shows two examples of Itoh-Tsujii algorithm for modular inversion over GF(2^{83}) and GF(2^{163}).

2.3 RSA Modular Exponentiation

The two most straightforward algorithms to implement RSA modular exponentiation are given in Alg. 2.5, where c is an integer (*e.g.* ciphertext), n is a product of two prime numbers, and d is a t-bit positive integer (*e.g.* private key). The algorithms perform modular exponentiation $c^d \bmod n$ used for the RSA encryption and decryption processes.

The basic operations in both algorithms are modular multiplications and modular squarings. Let $d = (111101)_2$; here $t = 6$. Table 2.2 shows the detailed computational steps for computing $c^{61} \bmod n$ using Alg. 2.5.

For side-channel issues, it is preferable to have a constant operation time for both modular multiplication and modular squaring. To be able to do this, the same datapath is used for both operations. Namely, modular squarings are not performed on dedicated hardware, but on the same multiplier as the one used for modular multiplication. Taking into account an expected value of $\frac{t}{2}$ ones in d, the total number of modular multiplications and modular squarings in both algorithms is $\frac{3t}{2}$. For the left-to-right algorithm, the modular multiplication in

Algorithm 2.5: Algorithms for left-to-right and right-to-left binary modular exponentiation.

```
Require: $c$, $d = (d_{t-1}d_{t-2}\cdots d_1d_0)_2$.
Ensure: $c^d \bmod n$.
1: $T \leftarrow 1$
2: for $i$ from $t-1$ down to 0 do
3:     $T \leftarrow T^2 \bmod n$
4:     if $d_i = 1$ then
5:         $T \leftarrow T \cdot c \bmod n$
6:     end if
7: end for
8: Return $T$
```

```
Require: $c$, $d = (d_{t-1}d_{t-2}\cdots d_1d_0)_2$.
Ensure: $c^d \bmod n$.
1: $S \leftarrow 1$, $T \leftarrow c$
2: for $i$ from 0 to $t-1$ do
3:     if $d_i = 1$ then
4:         $S \leftarrow S \cdot T \bmod n$
5:     end if
6:     $T \leftarrow T^2 \bmod n$
7: end for
8: Return $S$
```

Table 2.2: Example of left-to-right and right-to-left modular exponentiation.

Left-to-right	i	-	5	4	3	2	1	0
	d_i	-	1	1	1	1	0	1
	T	1	c	c^3	c^7	c^{15}	c^{30}	c^{61}
Right-to-left	i	-	0	1	2	3	4	5
	d_i	-	1	0	1	1	1	1
	S	1	c	c	c^5	c^{13}	c^{29}	c^{61}
	T	c	c^2	c^4	c^8	c^{16}	c^{32}	c^{64}

step 5 has to be performed after the modular squaring in step 3 finishes. That is, the modular exponentiation can be performed in $\frac{3t}{2}M$ where M is computation time for one modular multiplication or one modular squaring. This algorithm requires one memory location T for intermediate values. On the other hand, in the right-to-left algorithm the modular multiplication and the modular squaring can be parallelized, which enables one to operate modular exponentiation in time tM. However, the right-to-left algorithm requires two memory locations, S and T for intermediate values.

It is possible to parallelize the modular operations also with the left-to-right algorithm as shown in Alg. 2.6. This algorithm always operates modular multiplication and modular squaring within one for-loop regardless the value of the key bit, which means that the total number of modular operations is $2t$. The performance and the cost for memory is the same as the right-to-left algorithm. Table 2.3 shows the detailed computation steps for computing c^{61} using Alg. 2.6.

Depending on the algorithms, the performance of modular exponentiation can

Algorithm 2.6: Algorithm for parallelized left-to-right binary modular exponentiation (the Montgomery powering ladder [63]).

```
Require: $c$, $d = (d_{t-1}d_{t-2}\cdots d_1d_0)_2$.
Ensure: $c^d \bmod n$.
1: $S \leftarrow 1$, $T \leftarrow c$
2: for $i$ from $t-1$ down to $0$ do
3:   if $d_i = 1$ then
4:     $S \leftarrow S \cdot T \bmod n$, $T \leftarrow T^2 \bmod n$
5:   else
6:     $T \leftarrow T \cdot S \bmod n$, $S \leftarrow S^2 \bmod n$
7:   end if
8: end for
9: Return $S$
```

Table 2.3: Example of parallelized left-to-right modular exponentiation.

	i	-	5	4	3	2	1	0
Parallelized	d_i	-	1	1	1	1	0	1
Left-to-right	S	1	c	c^3	c^7	c^{15}	c^{30}	c^{61}
	T	c	c^2	c^4	c^8	c^{16}	c^{31}	c^{62}

be accelerated up to tM by processing modular operations in parallel. However, the drawback of the use of these parallelized algorithms is that the two-fold parallel computation needs two datapaths and double memory locations for intermediate variables. The next section discusses relatively cheaper solutions to accelerate modular exponentiation.

2.3.1 Exponent Recoding

Another approach to accelerating RSA modular exponentiation without parallel processing is to use a different representation of the exponent d, so-called exponent recoding. This section describes a technique for exponent recoding.

Many techniques for recoding the exponent d have been proposed in the literature starting with Reitwesner [86]. Here we mention the signed-digit representation. Consider an integer representation of the form $d = \sum_{i=0}^{l} d_i 2^i$, where $d_i \in \{-1, 0, 1\}$. This is called the (binary) Signed Digit (SD) representation (see [8, 31, 57]). If an SD representation has no adjacent non-zero digits, it is called a Non-Adjacent Form (NAF). Every integer d has a unique NAF which has the minimum Hamming weight of any signed digit representation of d. Algorithm 2.7 shows a method for recoding the exponent.

Algorithm 2.7: Signed-digit exponent recoding [57].

```
Require: $d = (d_{t+1}d_t d_{t-1}d_{t-2}\cdots d_1 d_0)_2$ with $d_{t+1} = d_t = 0$.
Ensure: $d'$, an SD representation of $d$ (NAF($d$)).
1: $c_0 \leftarrow 0$
2: for $i$ from 0 to $t$ do
3:     $c_{i+1} \leftarrow \lfloor (d_i + d_{i+1} + c_i)/2 \rfloor$, $d'_i \leftarrow d_i + c_i - 2c_{i+1}$
4: end for
5: Return $d'$
```

Table 2.4: Example of signed-digit exponent recoding.

	i	0	1	2	3	4	5	6
Inputs	d_i	1	0	1	1	1	1	0
	c_i	0	0	0	1	1	1	1
	d_{i+1}	0	1	1	1	1	0	0
Outputs	c_{i+1}	0	0	1	1	1	1	0
	d'_i	1	0	$\bar{1}$	0	0	0	1

Let $d = (111101)_2$ for an example of exponent recoding. Table 2.4 shows the detailed computational steps for exponent recoding of d using Alg. 2.7. We obtain $d' = (1000\bar{1}01)_2$ where $\bar{1} = -1$.

When the exponent is given in an SD representation, c^{-1} mod n needs to be pre-computed to perform modular exponentiation with NAF(d). Unless modular inversion can be computed efficiently, only the left-to-right binary method in Alg. 2.5 is worthwhile because it requires no additional modular operations. As a result, we can expect that the average performance of modular exponentiation is $\frac{4t}{3}M + I$ with a single datapath where M and I are denoting modular multiplication and modular inversion c^{-1} mod n, respectively because the average density of non-zero digits in the SD representation is approximately 1/3 [31].

The left-to-right modular exponentiation can be accelerated further by using the k-ary method where k-bit of the exponent are evaluated simultaneously (Alg. 2.8). The total number of modular multiplications for one t-bit modular exponentiation is approximately $(2^k - 2) + u \cdot (k + 1)$ where $u = \lceil \frac{t}{k} \rceil$. For 1024-bit or 2048-bit RSA, the best performance is obtained when $k \simeq 6$ and the computational cost for one modular exponentiation is approximately $1.2tM$. This result is slightly faster than the exponent recoding method. However, this algorithm requires $(2^k - 1)$ memory locations for pre-computed values.

We can further improve the k-ary method by allowing freedom in positioning the windows. This is the so-called sliding-window method [57]. This method needs less memory and fewer modular multiplications compared to the k-ary method.

Algorithm 2.8: Precomputation for k-ary modular exponentiation (left) and algorithms for left-to-right k-ary modular exponentiation.

```
Require: $n$, $k$, $c$.
Ensure: $c^i \bmod n$ $(i = 0, \cdots, 2^k - 1)$.
1: $c_0 \leftarrow 1$, $c_1 \leftarrow c$
2: for $i$ from 2 to $2^k - 1$ do
3:     $c_i \leftarrow c_{i-1} \cdot c \bmod n$
4: end for
5: Return $c_i (i = 0, \cdots, 2^k - 1)$
```

```
Require: $n$, $c_i (i = 0, \cdots, 2^k - 1)$,
    $d = (d_{u-1} d_{u-2} \cdots d_1 d_0)_{2^k}$.
Ensure: $c^d \bmod n$.
1: $T \leftarrow 1$
2: for $i$ from $u - 1$ down to 0 do
3:     $T \leftarrow T^{2^k} \bmod n$
4:     $T \leftarrow T \cdot c_{d_i} \bmod n$
5: end for
6: Return $T$
```

2.4 Curve-based Cryptography

The main operation in any curve-based primitive is point/divisor multiplication as already discussed in Sect. 1.1.2. Here, the arithmetic operations required for curve-based cryptography are introduced.

2.4.1 ECC over GF(p)

When E is an elliptic curve defined over GF(p), the typical equation is

$$E : y^2 = x^3 + ax + b \,, \tag{2.4}$$

where $a, b \in$ GF(p) and $(4a^3 + 27b^2) \bmod p \neq 0$. The sets of points on the elliptic curve together with the point at infinity (denoted as $\mathcal{O}$) can be seen as an additive Abelian group. The main operation, point multiplication is performed by the double-and-add algorithm (or the binary algorithm) that is using the point group operations. On the other hand, finite field operations in GF(p) such as addition, subtraction, multiplication and inversion are required to perform the group operations.

For an arbitrary point P on a curve E, an inverse of the point $P = (x_1, y_1)$ is $-P = (x_1, -y_1)$. The sum $P + Q$ of points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ (assuming that $P, Q \neq \mathcal{O}$, and $P \neq \pm Q$) is a point $R = (x_3, y_3)$ where

$$\begin{aligned} \beta &= \frac{y_2 - y_1}{x_2 - x_1} \\ x_3 &= \beta^2 - x_1 - x_2 \\ y_3 &= (x_1 - x_3)\beta - y_1 \,. \end{aligned} \tag{2.5}$$

This is called point addition. For $P = Q$, we get the following point doubling formulae

$$\begin{aligned}\beta &= \frac{3x_1^2 + a}{2y_1} \\ x_3 &= \beta^2 - 2x_1 \\ y_3 &= (x_1 - x_3)\beta - y_1 \,.\end{aligned} \tag{2.6}$$

Similar to the left-to-right and right-to-left binary algorithms for modular exponentiation in Alg. 2.5, point multiplication can be performed using Alg. 2.9, where P is a point on the elliptic curve and k is a positive integer. The point at infinity $\mathcal{O}$ is the identity element for elliptic curve operations.

Algorithm 2.9: Algorithms for left-to-right and right-to-left binary point multiplication.

```
Require: a point $P = (x, y)$,
    $k = (k_{l-1}k_{l-2} \cdots k_0)_2$.
Ensure: $kP$.
1: $Q \leftarrow \mathcal{O}$
2: for $i$ from $l - 1$ down to 0 do
3:     $Q \leftarrow 2Q$
4:     if $k_i = 1$ then
5:         $Q \leftarrow Q + P$
6:     end if
7: end for
8: Return $Q$
```

```
Require: a point $P = (x, y)$,
    $k = (k_{l-1}k_{l-2} \cdots k_0)_2$.
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:     end if
6:     $S \leftarrow 2S$
7: end for
8: Return $Q$
```

Table 2.5: Example of parallelized left-to-right point multiplication (the Montgomery powering ladder).

	i	-	4	3	2	1	0
Parallelized Left-to-right	k_i	-	1	1	1	0	1
	Q	P	$3P$	$7P$	$15P$	$30P$	$61P$
	S	$2P$	$4P$	$8P$	$16P$	$31P$	$62P$

As for RSA exponentiation, EC point multiplication can be parallelized with the left-to-right algorithm as shown in Alg. 2.10. Table 2.5 shows the computational steps for $k = 61$.

Algorithm 2.10: Algorithm for parallelized left-to-right binary point multiplication (the Montgomery powering ladder [63]).

```
Require: a point $P$, a non-negative integer $k = (1k_{l-2} \cdots k_1 k_0)_2$.
Ensure: $kP$.
1: $Q \leftarrow P$, $S \leftarrow 2P$
2: for $i$ from $l-2$ down to 0 do
3:   if $k_i = 1$ then
4:     $Q \leftarrow Q + S$, $S \leftarrow 2S$
5:   else
6:     $S \leftarrow S + Q$, $Q \leftarrow 2Q$
7:   end if
8: end for
9: Return $Q$
```

There are many types of coordinates in which an elliptic curve may be represented. In Eqs. (2.5) and (2.6), affine coordinates are used, but the so-called projective coordinates have some implementation advantages. More precisely, by using projective coordinates, the point addition and point doubling can be computed without modular inversions, except only one at the end of the point multiplication. Many types of projective coordinates are proposed in the literature. In particular, a weighted projective representation (also referred to as Jacobian representation) is preferred in the sense of faster arithmetic on elliptic curves [8, 35]. In this representation a triplet (X, Y, Z) corresponds to the affine coordinates $(X/Z^2, Y/Z^3)$ for $Z \neq 0$. In this case we have a weighted projective curve equation of the form

$$E : Y^2 = X^3 + aXZ^4 + bZ^6 \,. \tag{2.7}$$

Conversion from projective to affine coordinates costs 1 inversion and 4 multiplications, while vice versa it is trivial. The total cost for point addition is $1I+3M$ in affine coordinates and $16M$ in projective coordinates ($11M$ if using the special case $Z_1 = 1$, *i.e.* one point is given in affine coordinates, and the other one in projective coordinates). Here, I and M denote modular inversion and modular multiplication, respectively. In the case of point doubling (with a special case of $a = p - 3$), this relation is $1I + 4M$ in affine coordinates against $8M$ in projective coordinates. Thus, the choice of coordinates is determined by the ratio $I : M$. Therefore, multiplication in a finite field is the most important operation to focus on when working with projective coordinates. On the other hand, the extra datapath for modular inversion is indispensable for the affine coordinate representation because one modular inversion has to be performed for every point addition or doubling. In this thesis, we use the weighted projective coordinates for an efficient implementation.

Point Addition/Doubling over GF(p)

Here we assume that the two points that will be added, *i.e.* $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2, Z_2)$ are already transformed to the weighted projective coordinates (Jacobian representation), where (X, Y, Z) corresponds to the affine coordinates $(X/Z^2, Y/Z^3)$. We now create a schedule for point addition and doubling using a single datapath. Therefore, the left-to-right binary algorithm shown in Alg. 2.9 can be used, and the result point is stored as Q, *i.e.* $Q = Q + P$ for point addition and $Q = 2Q$ for point doubling. In Alg. 2.11 a possible schedule for point addition and doubling is given including the mixed-addition case, $Z_1 = 1$.

Algorithm 2.11: Scheduling of point addition and doubling over GF(p) reformulated based on [8].

```
Require:
    $P = (X_1, Y_1, Z_1)$,
    $Q = (X_2, Y_2, Z_2)$.
Ensure: $Q = Q + P$.
 1: $t_1 = Z_1 Z_1$;
 2: $t_2 = X_2 t_1$;
 3: $t_3 = Z_2 Z_2$;
 4: $t_4 = X_1 t_3 + t_2$;
 5: $t_2 = X_1 t_3 - t_2$;
 6: $t_5 = t_1 Z_1$;
 7: $t_6 = Y_2 t_5$;
 8: $t_1 = t_3 Z_2$;
 9: $t_3 = t_1 Y_1 + Y_2$;
10: $Y_2 = t_1 Y_1 - Y_2$;
11: $t_5 = t_2 t_2$;
12: $t_1 = t_4 t_5$;
13: $X_2 = Y_2 Y_2 - t_1$;
14: $t_4 = -2X_2 + t_1$;
15: $t_1 = t_5 t_2$;
16: $t_1 = t_3 t_1$;
17: $t_3 = t_4 Y_2 - t_1$;
18: $Y_2 = t_3/2$;
19: $Z_2 = t_2 Z_2$;
20: $Z_1 = Z_1 Z_2$;
21: Return $Q$;
```

```
Require:
    $P = (X_1, Y_1, 1)$,
    $Q = (X_2, Y_2, Z_2)$.
Ensure: $Q = Q + P$.
 1: $t_3 = Z_2 Z_2$;
 2: $t_4 = X_1 t_3 + X_2$;
 3: $t_2 = X_1 t_3 - X_2$;
 4: $t_1 = t_3 Z_2$;
 5: $t_3 = t_1 Y_1 + Y_2$;
 6: $Y_2 = t_1 Y_1 - Y_2$;
 7: $t_5 = t_2 t_2$;
 8: $t_1 = t_4 t_5$;
 9: $X_2 = Y_2 Y_2 - t_1$;
10: $t_4 = -2X_2 + t_1$;
11: $t_1 = t_5 t_2$;
12: $t_1 = t_3 t_1$;
13: $t_3 = t_4 Y_2 - t_1$;
14: $Y_2 = t_3/2$;
15: $Z_2 = t_2 Z_2$;
16: Return $Q$;
```

```
Require:
    $Q = (X_2, Y_2, Z_2)$.
Ensure: $Q = 2Q$.
 1: $t_1 = X_2 X_2$;
 2: $t_1 = 3t_1$;
 3: $t_2 = Z_2 Z_2$;
 4: $t_2 = t_2 t_2$;
 5: $t_2 = a t_2 + t_1$;
 6: $t_1 = 2Y_2$;
 7: $Z_2 = Z_2 t_1$;
 8: $t_3 = t_1 t_1$;
 9: $t_4 = X_2 t_3$;
10: $X_2 = 2t_4$;
11: $X_2 = t_2 t_2 - X_2$;
12: $t_1 = t_1 t_3$;
13: $t_1 = t_1 Y_2$;
14: $t_3 = t_4 - X_2$;
15: $Y_2 = t_2 t_3 - t_1$;
16: Return $Q$;
```

In our case point addition and doubling both consist of 15 steps in the case of $Z_1 = 1$. Here, each step uses only multiply-add/subtract operations in the underlying finite field in order to have a constant operation time and in order to have a compact datapath. Therefore the average computation cost for point multiplication kP with an l-bit scalar k is estimated as $\frac{45l}{2}M_{ab+c}$ where M_{ab+c} denotes one multiply-add/subtract operation over GF(p). If we use an algorithm processing point addition and doubling in parallel (*e.g.* the right-to-left algorithm), the cost becomes $15lM_{ab+c}$.

2.4.2 ECC over GF(2^m)

Here, we consider a finite field of characteristic 2, *i.e.* GF(2^m), a non-supersingular elliptic curve E over GF(2^m) that is defined as the set of solutions $(x, y) \in$ GF(2^m) $\times$ GF(2^m) of the equation,

$$E : y^2 + xy = x^3 + ax^2 + b\,, \tag{2.8}$$

where $a, b \in$ GF(2^m), $b \neq 0$, together with $\mathcal{O}$. The inverse of the point $P_1 = (x_1, y_1)$ is $-P_1 = (x_1, x_1 + y_1)$. The points on the curve and the point at infinity $\mathcal{O}$ form an Abelian group. The sum $P+Q$ of the points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ ($P, Q \neq O$, and $P \neq \pm Q$) is the point $R = (x_3, y_3)$ where

$$\begin{aligned} \beta &= \frac{y_1 + y_2}{x_1 + x_2} \\ x_3 &= \beta^2 + \beta + x_1 + x_2 + a \\ y_3 &= (x_1 + x_3)\beta + x_3 + y_1\,. \end{aligned} \tag{2.9}$$

This operation is called point addition. For $P = Q$, the point doubling formulae are

$$\begin{aligned} \beta &= \frac{y_1}{x_1} + x_1 \\ x_3 &= \beta^2 + \beta + a \\ y_3 &= (x_1 + x_3)\beta + x_3 + y_1\,. \end{aligned} \tag{2.10}$$

We use weighted projective coordinates where an affine point (x, y) is converted to a projective point (X, Y, Z) by computing $x = X/Z^2, y = Y/Z^3$. The projective curve equation corresponding to the affine equation shown in Eq. (2.8) is given by

$$E : Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6\,. \tag{2.11}$$

Point Addition/Doubling

Similar to the case over GF(p), point addition over GF(2^m) is computed by $Q = Q + P$ ($P \neq Q$). For point doubling we use $Q = 2Q$. The computation sequences for point addition and doubling on the projective curve are presented in Alg. 2.12 including the special case $Z_1 = 1$ for point addition.

Algorithm 2.12: Scheduling of point addition over GF(2^m) reformulated based on [8].

```
Require:
    $P = (X_1, Y_1, Z_1)$,
    $Q = (X_2, Y_2, Z_2)$.
Ensure: $Q = P + Q$.
 1: $t_6 = Z_1 Z_1$;
 2: $t_3 = t_6 X_2$;
 3: $t_2 = Z_2 Z_2$
 4: $t_4 = t_2 X_1 + t_3$
 5: $t_1 = t_6 Z_1$;
 6: $t_3 = t_1 Y_2$;
 7: $t_2 = t_2 Z_2$;
 8: $t_1 = t_2 Y_1 + t_3$;
 9: $t_5 = t_4 Z_1$;
10: $t_3 = t_5 Y_2$;
11: $t_3 = t_1 X_2 + t_3$;
12: $Z_2 = t_5 Z_2$;
13: $t_5 = t_1(t_1 + Z_2)$;
14: $Y_2 = t_4 t_4$;
15: $t_2 = t_4 Y_2 + t_5$;
16: $X_2 = a Z_2$;
17: $X_2 = X_2 Z_2 + t_2$;
18: $t_4 = Y_2 t_3$;
19: $t_4 = t_4 t_6$;
20: $Y_2 = X_2(t_1 + Z_2) + t_4$;
21: Return $Q$;
```

```
Require:
    $P = (X_1, Y_1, 1)$,
    $Q = (X_2, Y_2, Z_2)$.
Ensure: $Q = P + Q$.
 1: $t_2 = Z_2 Z_2$;
 2: $t_4 = t_2 X_1 + X_2$;
 3: $t_2 = t_2 Z_2$;
 4: $t_1 = t_2 Y_1 + Y_2$;
 5: $t_3 = t_4 Y_2$;
 6: $t_3 = t_1 X_2 + t_3$;
 7: $Z_2 = t_4 Z_2$;
 8: $t_5 = t_1(t_1 + Z_2)$;
 9: $Y_2 = t_4 t_4$;
10: $t_2 = t_4 Y_2 + t_5$;
11: $X_2 = a Z_2$;
12: $X_2 = X_2 Z_2 + t_2$;
13: $t_4 = Y_2 t_3$;
14: $Y_2 = X_2(t_1 + Z_2) + t_4$;
15: Return $Q$;
```

```
Require:
    $Q = (X_2, Y_2, Z_2)$.
Ensure: $Q = 2Q$.
 1: $t_1 = X_2 X_2$;
 2: $t_2 = t_1 t_1$;
 3: $t_4 = Y_2 Z_2 + t_1$;
 4: $t_3 = Z_2 Z_2$;
 5: $Z_2 = X_2 t_3$;
 6: $t_5 = c t_3 + X_2$;
 7: $t_3 = t_5 t_5$;
 8: $X_2 = t_3 t_3$;
 9: $t_1 = X_2(Z_2 + t_4)$;
10: $Y_2 = t_2 Z_2 + t_1$;
11: Return $Q$;
```

Here, $c = \sqrt[4]{b} = b^{2^{m-2}}$. We assume this curve parameter is pre-computed. The computation sequence for point subtraction $Q - P$ is obtained simply by using

$$t_1 = t_2(X_1 + Y_1) + Y_2 \,, \tag{2.12}$$

in place of step 4 of the point addition in Alg. 2.12.

2.4.3 ECC over a Composite Field

For cryptographic security it is typically recommended to use fields GF(2^m) where m is a prime. As an example we consider the case where $m = 163$. For ECC over composite fields, we should choose GF($(2^{83})^2$) in order to have the equivalent level of security [55]. More precisely, we consider ECC over a field of a quadratic extension of GF(2^{83}), so GF($(2^{83})^2$) = GF(2^{83})$[y]/g(y)$ and $\deg(g) = 2$. In this way one can obtain a speed-up and benefit even more from the parallelism. The reason is that in composite field notation, each element is represented as $c = c_1t+c_0$ where $c_0, c_1 \in$ GF(2^{83}) and the multiplication in this field takes 3 modular multiplications and 4 modular additions in GF(2^{83}) [52]. In [23], the Weil descent attack is introduced against EC defined over binary fields of composite degree. This work puts some doubt on security of composite field implementations of ECC in general. However, further investigations have shown that composite fields with degree $2 \cdot m$ (*i.e.* extension of degree two), where m is prime, remain secure against Weil Descent attacks and its variants [55].

Algorithm 2.13: Modular multiplication over GF($(2^m)^2$).

```
Require: irreducible polynomial $P$, $A = A_1t + A_0$, $B = B_1t + B_0$,
    where $P, A_0, A_1, B_0, B_1 \in$ GF($2^m$).
Ensure: $AB \bmod P = C_1t + C_0$, where $C_0, C_1 \in$ GF($2^m$).
 1: $S \leftarrow A_1B_1 \bmod P$
 2: $T \leftarrow A_0B_0 \bmod P$
 3: $C_0 \leftarrow (S + T) \bmod P$
 4: $S \leftarrow (A_1 + A_0) \bmod P$
 5: $T \leftarrow (B_1 + B_0) \bmod P$
 6: $U \leftarrow ST \bmod P$
 7: $C_1 \leftarrow (U + C_0) \bmod P$
 8: Return $C_1t + C_0$
```

Algorithm 2.14: Modular addition over GF($(2^m)^2$).

```
Require: irreducible polynomial $P$, $A = A_1t + A_0$, $B = B_1t + B_0$,
    where $P, A_0, A_1, B_0, B_1 \in$ GF($2^m$).
Ensure: $(A + B) \bmod P = C_1t + C_0$, where $C_0, C_1 \in$ GF($2^m$).
 1: $C_0 \leftarrow (A_0 + B_0) \bmod P$
 2: $C_1 \leftarrow (A_1 + B_1) \bmod P$
 3: Return $C_1t + C_0$
```

2.4.4 Hyper-Elliptic Curve Cryptography (HECC)

Another appealing candidate for PKC is HECC. Recently many software and hardware implementations of HECC have been described, while more theoretical work has shown HECC also secure for curves with a small genus [98]. As already mentioned, ECC over composite fields allows one to work in a finite field where bit-lengths are shorter with a factor 2, when compared to ECC. That means, for the equivalent level of security of ECC over GF(2^{163}), we should choose GF(2^{83}). We obtain a similar result for HECC on a curve of a genus 2. Nevertheless, the performance is still much slower than the one of private-key cryptography such as AES [33, 34].

The only difference between ECC and HECC is the sequence of operations at the middle level. The sequence for HECC is more complex when compared with the ECC point operations, however HECC uses shorter operands. One can perform inversion also with a chain of multiplications [36] and only provide hardware for finite field addition and multiplication.

Let $\overline{\mathrm{GF}(2^m)}$ be an algebraic closure of the field $\mathrm{GF}(2^m)$. Here we consider a hyperelliptic curve C of genus $g = 2$ over $\mathrm{GF}(2^m)$, which is given by an equation of the form

$$C : y^2 + h(x)y = f(x) \quad \text{in} \quad \mathrm{GF}(2^m)[x, y]\,, \tag{2.13}$$

where $h(x) \in \mathrm{GF}(2^m)[x]$ is a polynomial of degree $\deg(h) \leq g$ and $f(x)$ is a monic polynomial of degree $\deg(f) = 2g + 1$. Also, there are no solutions $(x, y) \in \overline{\mathrm{GF}(2^m)} \times \overline{\mathrm{GF}(2^m)}$ that simultaneously satisfy Eq. (2.13) and the equations: $2v + h(u) = 0, h'(u)v - f'(u) = 0$. These points are called singular points. For the genus 2, in the general case the following equation is used.

$$y^2 + (h_2x^2 + h_1x + h_0)y = x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0\,. \tag{2.14}$$

For our implementation in this thesis, the type II curves proposed by Byramjee and Duquesne [11] are used. The curves are defined with $h_2 = 0, h_1 = 1$ and especially the following form is recommended by the authors from an arithmetic and security point of view.

$$y^2 + xy = x^5 + f_3x^3 + x^2 + f_0\,. \tag{2.15}$$

Instead of point operations used for ECC, we consider divisor operations for HECC. A divisor D is a formal sum of points on the hyperelliptic curve C. We use the Mumford representation denoted as $[u(x), v(x)]$ or $[u, v]$ for short, because it enables fast divisor operations as introduced in [50]. The divisor $[u, v]$ is a pair of polynomials where u is monic of degree 2, $\deg(v) < \deg(u)$ and $u|f - hv - v^2$ (the so-called reduced divisors). We use projective coordinates and therefore the quintuple $[U_1, U_0, V_1, V_0, Z]$ stands for $[x^2 + U_1/Zx + U_0/Z, V_1/Zx + V_0/Z]$. The sequences for divisor addition and doubling on Eq. (2.15) are reformulated as shown in Alg. 2.15 based on work in [11, 50]. For Hyper-Elliptic Curve (HEC) divisor multiplication, the same algorithms of EC point multiplication can be applied.

Algorithm 2.15: Scheduling of divisor addition and doubling over GF(2^m) based on [11, 50].

```
Require: $D = [U_{11}, U_{10}, V_{11}, V_{10}, 1]$,
    $E = [U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$.
Ensure: $E = D + E$.
1: $z_1 = U_{11}Z_2 + U_{21}$
2: $z_2 = U_{10}Z_2 + U_{20}$
3: $z_3 = U_{11}z_1 + z_2$
4: $r = z_1z_1$
5: $r = rU_{10}$
6: $r = z_2z_3 + r$
7: $w_0 = V_{10}Z_2 + V_{20}$
8: $w_1 = V_{11}Z_2 + V_{21}$
9: $w_2 = z_3w_0$
10: $w_3 = z_1w_1$
11: $s_0 = U_{10}w_3 + w_2$
12: $s_1 = z_1(w_0 + w_1) + w_2$
13: $s_1 = z_3(w_0 + w_1) + s_1$
14: $s_1 = w_3(1 + U_{11}) + s_1$
15: $R = Z_2r$
16: $s_0 = s_0Z_2$
17: $s_3 = s_1Z_2$
18: $\tilde{R} = Rs_3$
19: $t = s_1(z_1 + U_{21})$
20: $S_3 = s_3s_3$
21: $S = s_0s_1$
22: $\tilde{S} = s_1s_3$
23: $\tilde{\tilde{R}} = \tilde{R}\tilde{S}$
24: $l_2 = \tilde{S}U_{21}$
25: $l_0 = SU_{20}$
26: $l_1 = \tilde{S}U_{20}$
27: $l_1 = SU_{21} + l_1$
28: $l_2 = s_0s_3 + l_2$
29: $t = s_1t$
30: $t = z_1t$
31: $U_{20} = s_1Z_2$
32: $U_{20} = rz_1 + U_{20}$
33: $U_{20} = RU_{20} + t$
34: $U_{20} = s_0s_0 + U_{20}$
35: $U_{20} = z_2\tilde{S} + U_{20}$
36: $U_{21} = \tilde{S}z_1$
37: $U_{21} = RR + U_{21}$
38: $w_0 = U_{20}(l_2 + U_{21})$
39: $w_0 = S_3l_0 + w_0$
40: $w_1 = U_{21}(l_2 + U_{21})$
41: $w_1 = S_3(U_{20} + l_1) + w_1$
42: $U_{20} = \tilde{R}U_{20}$
43: $U_{21} = \tilde{R}U_{21}$
44: $V_{20} = \tilde{\tilde{R}}V_{20} + w_0$
45: $V_{21} = \tilde{\tilde{R}}(V_{21} + Z_2) + w_1$
46: $Z_2 = \tilde{R}S_3$
47: Return $(U_{21}, U_{20}, V_{21}, V_{20}, Z_2)$
```

```
Require: $E = [U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$.
Ensure: $E = 2E$.
1: $t_0 = Z_2Z_2$
2: $t_1 = U_{21}U_{21}$
3: $r = U_{20}Z_2$
4: $k_1 = f_3t_0 + t_1$
5: $k_0 = V_{21}(Z_2 + V_{21}) + t_0$
6: $k_0 = Z_2k_0$
7: $k_0 = U_{21}k_1 + k_0$
8: $t_2 = k_0U_{21}$
9: $s_1 = k_0Z_2$
10: $s_0 = k_1r + t_2$
11: $t_0 = t_0r$
12: $r = t_0s_1$
13: $t_1 = s_1k_0$
14: $t_3 = U_{20}k_0$
15: $l_2 = s_1t_2$
16: $l_0 = s_0t_3$
17: $l_1 = s_0t_2$
18: $l_1 = s_1t_3 + l_1$
19: $U_{20} = s_0s_0 + r$
20: $U_{21} = t_0t_0$
21: $l_2 = s_1(s_0 + t_2) + U_{21}$
22: $s_1 = s_1s_1$
23: $t_2 = rt_1$
24: $t_0 = U_{20}l_2$
25: $t_0 = l_0s_1 + t_0$
26: $t_1 = U_{21}l_2$
27: $t_1 = s_1(U_{20} + l_1) + t_1$
28: $Z_2 = s_1r$
29: $U_{21} = U_{21}r$
30: $U_{20} = U_{20}r$
31: $V_{20} = t_2V_{20} + t_0$
32: $V_{21} = t_2V_{21} + Z_2$
33: $V_{21} = t_1 + V_{21}$
34: Return $(U_{21}, U_{20}, V_{21}, V_{20}, Z_2)$
```

2.4.5 Scalar Recoding

The exponent recoding method introduced in Alg. 2.7 is also used for scalar recoding to accelerate EC point multiplication and HEC divisor multiplication. Here

Table 2.6: Example of scalar recoding with the window size $w = 3$.

	i	0	1	2	3	4	5	6	7
Input	k	61	32	16	8	4	2	1	0
$\mathrm{NAF}_3(k)$	k_i	$\bar{3}$	0	0	0	0	0	1	-

the NAF method with a wider window size is explained. This method is especially suited for ECC because point subtraction can be computed with a slightly modified sequence from point addition. Therefore, negative integers in the recoded scalar introduced by the NAF methods (*e.g.* $\bar{1}$) can be handled without degrading the performance. Algorithm 2.16 is a generalized algorithm for generating a NAF representation with the w-bit window size or a width-w NAF for a scalar k. Algorithm 2.7 used for exponent recoding is considered the special case for $w = 2$.

Algorithm 2.16: Algorithm for generating width-w NAF [31].

```
Require: window size $w$, positive integer $k$.
Ensure: a width-$w$ NAF representation of $k$ ($\mathrm{NAF}_w(k)$).
 1: $i \leftarrow 0$
 2: while $k \geq 1$ do
 3:   if $k$ is odd then
 4:     $k_i \leftarrow k \bmod 2^w$, $k \leftarrow k - k_i$ $(-2^{w-1} \leq k_i < 2^{w-1})$
 5:   else
 6:     $k_i \leftarrow 0$
 7:   end if
 8:   $k \leftarrow k/2$, $i \leftarrow i + 1$
 9: end while
10: Return $(k_{i-1}k_{i-2}\cdots k_1k_0)$
```

Let $k = (111101)_2$ for an example of scalar recoding with the window size $w = 3$. Table 2.7 shows the detailed computational steps for scalar recoding of k using Alg. 2.16. The result indicates that the scalar is recoded as $\mathrm{NAF}_3(61) = (100000\bar{3})_2$.

Point multiplication with a scalar represented in a width-w NAF can be computed by Alg. 2.17. As can be found in the algorithm, the pre-computed points are used iteratively in the for-loop (steps 5 and 7). However, in the case of the right-to-left and the parallelized left-to-right algorithm, we need to perform point doublings for all pre-computed points in every for-loop (see step 6 of the right-to-left method in Alg. 2.9). The cost for these point doublings is crucial for $w > 2$. Therefore, the width-w NAF method is suitable for the left-to-right algorithm.

The left-to-right algorithm first needs the MSB of a (recoded) scalar although

Algorithm 2.17: Left-to-right point multiplication with $\mathrm{NAF}_w(k)$.

```
Require: window size $w$, $\mathrm{NAF}_w(k) = \sum_{i=0}^{l-1} k_i 2^i$, pre-computed points
    $P_i = iP$ for $i = 1, 3, 5, \cdots, 2^{w-1} - 1$.
Ensure: $kP$.
 1: $Q \leftarrow \mathcal{O}$
 2: for $i$ from $l - 1$ down to 0 do
 3:    $Q \leftarrow 2Q$
 4:    if $k_i > 0$ then
 5:       $Q \leftarrow Q + P_{k_i}$
 6:    else if $k_i < 0$ then
 7:       $Q \leftarrow Q - P_{k_i}$
 8:    end if
 9: end for
10: Return $Q$
```

the result of scalar recoding is generated from the LSB as shown in Table 2.6. Therefore point multiplication can start only after the scalar recoding algorithm finishes. In this sense, it is beneficial to have an algorithm that recodes a scalar from the MSB. Although scalar recoding is not computationally intensive compared to point multiplication, MSB-first (left-to-right) scalar recoding is beneficial because it can reduce the overhead cycles by recoding a scalar on-the-fly. Okeya *et al.* introduce a novel idea for MSB-first scalar recoding by using the MOF (Mutual Opposite Form) [67]. More precisely, the method can generate the result of scalar recoding from the MSB by introducing the MOF representation of a given scalar $k = (k_{t-1} \cdots k_1 k_0)_2$. The MOF representation of k can be obtained simply by applying an SD representation to the value $(2k - k)$. The MOF_w representation is obtained by performing the left-to-right sliding-window method [31] with the support of a look-up table.

Table 2.7: Example of MSB-first scalar recoding with the window size $w = 3$.

	i	6	5	4	3	2	1	0
Inputs	k_{i-1}	1	1	1	1	0	1	0
	k_i	0	1	1	1	1	0	1
Outputs	$\mathrm{MOF}(k) : d_i$	1	0	0	0	$\bar{1}$	1	$\bar{1}$
	$\mathrm{MOF}_3(k) : d'_i$	1	0	0	0	0	0	$\bar{3}$

Let $k = (111101)_2$ for an example of MSB-first scalar recoding with the window size $w = 3$, *i.e.* $\mathrm{MOF}_3(k)$. Table 2.7 summarizes the procedure of this recoding method based on Alg. 2.18. We need a look-up table when converting a binary

Algorithm 2.18: Algorithm for generating MSB-first scalar recoding with window size w (width-w MOF or $\mathrm{MOF}_w(k)$) [67].

```
Require: window size $w$, positive integer $k = (k_{t-1} \cdots k_1 k_0)$,
    a look-up table $\mathrm{TABLE}_w()$.
Ensure: a width-$w$ MOF representation of $k$ ($\mathrm{MOF}_w(k)$).
 1: $k_{-1} \leftarrow 0$, $k_t \leftarrow 0$, $i \leftarrow t$
 2: while $i \geq w - 1$ do
 3:   if $k_i = k_{i-1}$ then
 4:     $d_i \leftarrow 0$, $d'_i \leftarrow 0$, $i \leftarrow i - 1$
 5:   else
 6:     for $j$ from $i$ down to $i - w + 1$ do
 7:       $d_j \leftarrow k_j - k_{j-1}$
 8:     end for
 9:     $(d'_i, d'_{i-1}, \cdots, d'_{i-w+1}) \leftarrow \mathrm{TABLE}_w(d_i, d_{i-1}, \cdots, d_{i-w+1})$
10:     $i \leftarrow i - w$
11:   end if
12: end while
13: if $i \geq 0$ then
14:   $(d'_i, d'_{i-1}, \cdots, d'_0) \leftarrow \mathrm{TABLE}_{i+1}(d_i, d_{i-1}, \cdots, d_0)$
15: end if
16: Return $(d'_t, d'_{t-1}, \cdots, d'_0)$
```

string in MOF representation into MOF_3. In this example, $\mathrm{TABLE}_3(100) = 100$ and $\mathrm{TABLE}_3(\bar{1}1\bar{1}) = 00\bar{3}$. As a result, $\mathrm{MOF}_3(61) = (100000\bar{3})_2$ is obtained, which is the same result as $\mathrm{NAF}_3(61)$ obtained in Table 2.6.

2.5 Conclusions

This chapter describes algorithms for finite field arithmetic, RSA exponentiation, EC point multiplication and HEC divisor multiplication. We mainly discuss the functionality and the trade-off between cost and performance of different algorithms. As closing remarks of this chapter, we briefly explain another property of an algorithm, *i.e.* algorithm-level countermeasures against Timing Analysis (TA) and Simple Power Analysis (SPA) attacks. The details of these attacks will be described in Appendix A.

Algorithm 2.6, one of the algorithms for RSA exponentiation, always computes one modular multiplication and one modular squaring in the for-loop regardless the value of the key bit d_i. Therefore, the computation time is independent of the value of the exponent d and the same operation patterns are repeated, which means that it is hard to extract bits of exponent, *i.e.* d_i from TA or SPA attacks on an RSA

Algorithm 2.19: Algorithm for add-only right-to-left binary point multiplication [39].

```
Require: a point $P$, a non-negative integer $k = (k_{l-1}k_{l-2} \cdots k_1k_0)_2$.
Ensure: $kP$.
1: $Q_0 \leftarrow P$, $Q_1 \leftarrow P$, $S \leftarrow 2P$
2: for $i$ from 1 to $l-1$ do
3:     $Q_{1-k_i} \leftarrow Q_{1-k_i} + S$
4:     $S \leftarrow Q_0 + Q_1$
5: end for
6: $Q_{k_0} \leftarrow Q_{k_0} - P$
7: Return $Q_0$
```

Table 2.8: Example of parallelized left-to-right point multiplication.

	i	-	1	2	3	4	5	0
Add-only	k_i	-	0	1	1	1	1	1
right-to-left	Q_0	P	P	$5P$	$13P$	$29P$	$61P$	$61P$
	Q_1	P	$3P$	$3P$	$3P$	$3P$	$3P$	$2P$
	S	$2P$	$4P$	$8P$	$16P$	$32P$	$64P$	$64P$

system. Algorithm 2.5 can be a TA/SPA resistant algorithm by slightly modifying the sequence, *e.g.* introducing a dummy modular multiplication if $d_i = 0$. The same holds true for EC point multiplication shown in Algs. 2.9 and 2.10.

In 2007, Joye proposed a new algorithm for EC point multiplication (and for RSA exponentiation) that uses only a point addition sequence [39]. The proposed algorithm is shown in Alg. 2.19 and the computational sequence for the case of $k = 61$ is described in Table 2.8. As can be seen from Table 2.8, memory space for storing three intermediate points is needed. This is relatively expensive compared to the introduced binary methods (see Algs. 2.9 and 2.10). However, there is no need to implement the point doubling sequence in cryptosystems, which may result in area-efficient and TA/SPA-resistant implementations. Further investigation on this new add-only algorithm will be needed.

Chapter 3

Modular Arithmetic Logic Unit (MALU)

3.1 Introduction

The goal of this chapter is to investigate a high-speed and compact hardware implementation of a Modular Arithmetic Logic Unit (MALU) and to explore scalability (*i.e.* trade-offs between area, cost) and flexibility (*i.e.* support of various operand sizes) in PKC. First, we discuss a MALU for modulo n operations. Second, designs for MALUs over GF(2^m) are introduced. Lastly, we introduce another MALU called dual-field MALU that can support both fields.

Finite field operations are fundamental in PKC such as RSA, ECC and HECC. Modular exponentiation in RSA is based on the repeated usage of modular multiplications modulo n that is the product of two distinct large prime numbers. HECC and ECC use modular multiplications and additions over a prime field GF(p) and a binary field GF(2^m) in the point multiplication. For instance, a 160-bit ECC over GF(p) is determined by a prime number p that satisfies $p > 2^{160}$ and ECC over GF(2^m) is specified by an irreducible polynomial $P(x)$ whose degree m is a prime that satisfies $m > 160$ (*e.g.* $m = 163$). Because RSA requires a large integer for its private key (at least 1024 bits), ECC is often preferred for a compact hardware implementation as much shorter key-lengths are sufficient. Blake, Seroussi and Smart argue in [8] that 160-bit ECC has the same security level as 1024-bit RSA.

The field type and the size of the underlying finite field are important parameters that determine the architecture of the data processing block of public-key cryptosystems. Especially, finite fields of characteristic 2 or GF(2^m) operations are considered suitable for hardware implementations due to the carry-free arithmetic. Nevertheless, GF(p) operations are also required for the algorithms such as ECDSA (Elliptic Curve Digital Signature Algorithm) [100]. Moreover, RSA is still important to support the existing RSA-based infrastructure. Therefore, supporting both modulo n operations and operations over a binary field GF(2^m) gives the opportunity to support various cryptosystems on the same hardware configuration.

3.1.1 Previous Work

Much effort has been spent to speed up the performance of modular multiplications. Among previously proposed algorithms, the Montgomery algorithm [62] is considered one of the fastest algorithms for modular multiplication modulo n where n is an odd positive integer. The algorithm can be implemented with normal multiplications and additions, and can avoid time-consuming trial divisions (Alg. 3.1). An integer X is represented as $X \cdot R$ mod n, which is called the Montgomery representation. A power of two is generally used for R, *i.e.* $R = 2^r$ where the radix r determines the number of bits in the reduction. Multiplication is performed in this representation and division by n is replaced with division by R or r-bit right-shift operation. An extended Montgomery algorithm called FIOS (Finely Integrated Operand Scanning) was proposed by Tenca and Koç; it is suitable for a flexible hardware implementation or a software implementation on a w-bit datapath [94].

Algorithm 3.1: Radix-2^r k-bit Montgomery modular multiplication.

```
Require: integers $n = (n_{k-1}, ..., n_0)_R$, $X = (x_{k-1}, ..., x_0)_R$,
    $Y = (y_{k-1}, ..., y_0)_R$, where $0 \leq X, Y < n$, $R = 2^r$ with $\gcd(n, R) = 1$
    and $n^{'} = -n^{-1} \bmod R$.
Ensure: $X \cdot Y \cdot 2^{-k} \bmod n$.
1: $T = (t_k, ..., t_0)_R \leftarrow 0$
2: for $i$ from 0 to $k-1$ do
3:     $m \leftarrow ((t_0 + x_i \cdot y_0) \cdot n^{'}) \bmod R$
4:     $T \leftarrow (T + x_i \cdot Y + m \cdot n)/R$
5: end for
6: if $T \geq n$ then
7:     $T \leftarrow T - n$
8: end if
9: return $T$
```

In another approach, a systolic array has been proposed for RSA and ECC over GF(p) [37, 48, 101]. Its basic idea is to use a multi-dimensional pipeline; therefore it can easily support arbitrary field sizes. Moreover, due to the pipelining, it can use a high clock frequency. The drawback of the systolic array is that it requires a relatively large area of Flip-Flops (F/Fs) to store intermediate variables in processing elements.

Montgomery multiplication can also be performed with a series of additions and divisions by 2 (right-shift operations over 1 bit) for the bit-serial multiplication (see Alg. 3.2). In this case, the carry propagation in the additions is the key to speeding up the performance. A simple and effective solution is to use Carry-Save Adder (CSA) chains. These allow us to ignore a long carry propagation delay in the first phase of operation. For instance, modular exponentiation in RSA can be computed in the Carry-Save (CS) form throughout the operation [56]. In this case, a carry-propagate addition is needed only in the last step of the exponentiation. However, it requires additional F/Fs and additional logic due to the redundant CS form.

Algorithm 3.2: Bit-serial k-bit Montgomery modular multiplication.

```
Require: integers $n = (n_{k-1}, ..., n_0)_2$, $X = (x_{k-1}, ..., x_0)_2$,
    $Y = (y_{k-1}, ..., y_0)_2$, where $0 \leq X, Y < n$, $R = 2^k$ with $\gcd(n, 2) = 1$.
Ensure: $X \cdot Y \cdot R^{-1} \bmod n$.
1: $T = (t_k, ..., t_0)_2 \leftarrow 0$
2: for $i$ from 0 to $k-1$ do
3:     $m \leftarrow t_0 \oplus x_i y_0$
4:     $T \leftarrow (T + x_i \cdot Y + m \cdot n)/2$
5: end for
6: if $T \geq n$ then
7:     $T \leftarrow T - n$
8: end if
9: return $T$
```

Großschädl proposed a bit-serial multiplier performing multiplications in both types of field, GF(p) and GF(2^m) [29]. Based on the datapath for the GF(p) operation, GF(2^m) modular arithmetic is supported on the same hardware datapath by masking the carry-propagate logic. Thus, the datapath is efficiently shared with both types of fields. A similar hardware solution is also presented by Wolkerstorfer in [102]. Wolkerstorfer introduced a low power design which features a short critical path to enable high clock frequencies by using CSAs. The idea of a unified multiplier was first introduced by Savaş, Tenca and Koç in [82]. The most recent published work is the one of Satoh and Takano [81]. They present a dual field multiplier with high performance in both types of fields. The throughput of an EC point multiplication is maximized by use of a Montgomery multiplier and an on-the-fly redundant binary converter. The biggest advantages of their design are the scalability between hardware area and speed and the flexibility in the operand size. As will be explained in the next sections, our proposed datapath also can support both types of fields. Moreover the design is flexible in the field size and scalable in terms of area-performance trade-off.

3.2 MALU for Modulo n Operation

Our proposed architecture of the MALUn is based on an extended bit-serial Montgomery multiplier as shown in Alg. 3.3. As can be seen from the algorithm, we need d times three-input additions for large operands, *e.g.* 160 bits for ECC over GF(p). It is not a good idea to use Carry-Propagate Adders (CPAs) for that algorithm because of the large carry-propagation delay.

Algorithm 3.3: Digit-serial k-bit Montgomery modular multiplication without final subtraction (digit size d).

```
Require: integers $n = (n_{k-1}, ..., n_0)_2$, $X = (x_k, ..., x_0)_2$, $Y = (y_k, ..., y_0)_2$,
    where $0 \le X, Y < 2n - 1$, $R = 2^{k+2}$ with $\gcd(n, 2) = 1$.
Ensure: $X \cdot Y \cdot R^{-1} \bmod n$.
 1: $T = (t_{k+1}, ..., t_0)_2 \leftarrow 0$
 2: for $i$ from 0 to $k+1$ do
 3:     $i \leftarrow i + d$
 4:     $m_i \leftarrow t_0 \oplus x_i y_0$
 5:     $T \leftarrow (T + x_i Y + m_i n)/2$
 6:     $m_{i+1} \leftarrow t_0 \oplus x_{i+1} y_0$
 7:     $T \leftarrow (T + x_{i+1} Y + m_{i+1} n)/2$
            $\vdots$
 8:     $m_{i+d-1} \leftarrow t_0 \oplus x_{i+d-1} y_0$
 9:     $T \leftarrow (T + x_{i+d-1} Y + m_{i+d-1} n)/2$
10: end for
11: return $T$
```

In Sect. 3.3.1, an efficient implementation of Alg. 3.3 is discussed. In order to avoid a long carry propagation delay, we apply the CSA technique to each three-input addition. By preparing d sets of CSA chains, we can explore the trade-off between area and performance. The proposed implementation is scalable in cost-performance trade-off by choosing the digit size d and flexible in the field size k. We denote a MALUn with the digit size d and the field size k as MALUn$_{k\times d}$.

By considering MALUn$_{k\times d}$ as a building block, a more flexible and scalable datapath is explored. $K \times D$ sets of MALUn$_{k\times d}$ can form a new reconfigurable datapath to support high-speed RSA exponentiation and EC point multiplication. The details are discussed in Sect. 3.3.2. Section 3.3.3 summarizes the implementation results of our proposed reconfigurable datapath.

3.2.1 Overview of the MALU for Modulo n Operation

Before explaining the general case, the main functionality of the MALUn is explained for the bit-serial version or $d = 1$. In this configuration, each cell is composed of two Full Adders (FAs) as illustrated in Fig. 3.1a. The cell sums up the four-bit inputs xy, mn, s and c and outputs two bits in the redundant CS form with value $2c_{next} + s_{next}$ where s and c are the virtual sum and carries. The bit multiplications xy and mn are the main inputs for computing the bit-serial Montgomery multiplication in Alg. 3.2, *i.e.* $(T + x_i \cdot Y + m \cdot n)/2$.

Several copies of the schematic of Fig. 3.1a can be concatenated horizontally to support vector inputs. For instance, we allocate 160 copies of Fig. 3.1a to support 160-bit inputs. The signals input from the right side and output to the left side are considered internal signals in this case. Thus, it is possible to make a reduction from four binary vectors to two binary vectors. We call this operation block four-to-two (4-2) CSA, although Fig. 3.1a illustrates the bit-level schematic for 1-bit operands which is defined as the minimal configuration of our proposed MALUn. The connection of the FA arrays in the i-direction (*i.e.* vertical connection) is determined by the bit weights of the outputs of FAs (indicated by numbers in parenthesis in Fig. 3.1a).

One modular multiplication can be computed with time complexity $O(k^2)$ when using bit-level 4-2 CSA operations with $d = 1$. However, as there are no carry propagations in the j-direction because of 4-2 CSA, it is natural to allocate the cells horizontally (in the j-direction) to enhance efficiency. In this case, the time and area complexities of one modular multiplication are $O(k)$. For the purpose of exploring the time and area complexities, we can increase the number of 4-2 CSAs allocated vertically or in the i direction. If preparing d sets of 4-2 CSAs, the time complexity of modular multiplication is reduced to k/d while area complexity increases to $d \cdot k$.

The explanation of the MALUn for a general d is given as follows. As illustrated in Fig. 3.1c, the introduced MALUn with 4-2 CSAs has four types of input vectors, $X = (x_g \cdots x_1 x_0)_{2^d}$, $Y = (y_{k+1} \cdots y_1 y_0)_2$, $n = (n_{k-1} \cdots n_1 n_0)_2$, and $S = (s_{h,k+\gamma-1} \cdots s_{1,k+\gamma-1} s_{0,k+\gamma-1})_{2^d}$ where $g = \lceil (k+1)/d \rceil$ and $h = \lceil k/d \rceil$. Here, X is the multiplier, Y is the multiplicand and n is the modulus. The addend vector S is provided to the MALUn by d bits in every cycle and eventually added to the result of the modular multiplication of X and Y (mod n). Note that we need to allocate additional γ cells in the MALUn in order to avoid an overflow of intermediate values depending on the algorithm of modular multiplication. This will be explained in Lemma. 1.

The intermediate results are stored in $VS = (vs_{i,k+\gamma-1} \cdots vs_{i,1} vs_{i,0})_2$ and $VC = (vc_{i,k+\gamma-1} \cdots vc_{i,1} vc_{i,0})_2$. They are reset to zero when a modular multiplication starts to execute ($i = 0$). The result of a Montgomery multiplication is output from the right-most cell by d bits in every cycle as $Sout = (sout_g \cdots sout_1 sout_0)_{2^d}$. The connection of VS and VC is latched with $(2k + 2\gamma - 1)$ sets of F/Fs as shown in Fig. 3.1c.

The MALUn has two independent stages for modulo n operations. One is the CS stage that implements the Montgomery algorithm in the CS form. The

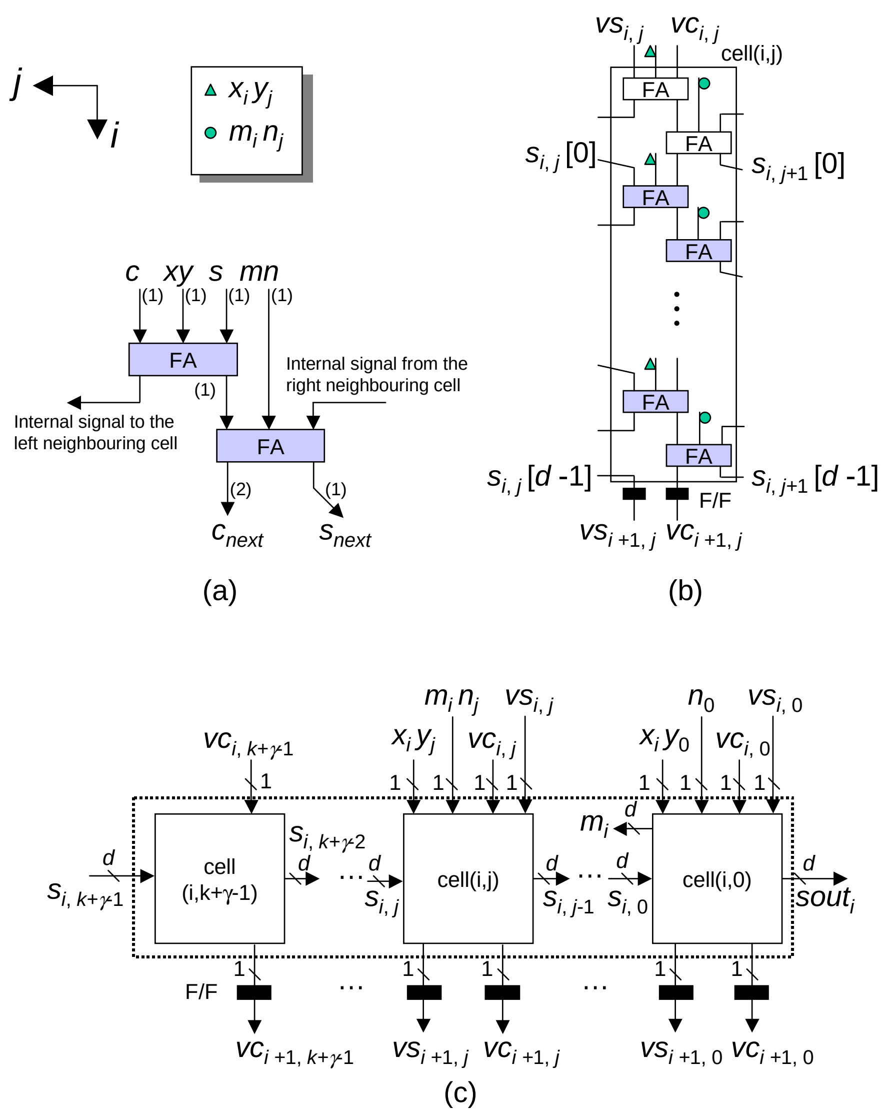

Figure 3.1: Configuration of the MALUn with 4-2 CSAs. (a) Bit-level 4-2 CSA (cell for $d = 1$). (b) Cell: bit-level 4-2 CSA with the digit size d. (c) Datapath of the MALUn.

second stage, the Carry-Propagate (CP) stage converts the CS from a redundant representation to a normal integer by propagating carries. Moreover the CP stage is capable of adding/subtracting S to/from the result of the CS stage. When subtracting S from XY, we use the 2's complement of S. In order to reduce the hardware cost and the critical path delay, the CP calculations are executed in the same datapath of the MALUn as the CS calculations. The operation of the MALUn can be explained as

$$\mathrm{MALUn}(XR, YR, SR) = (XY \pm S)R \bmod n\,, \tag{3.1}$$

where R is selected as $R = 2^{k+\gamma}$, k is the bit-length of the secret key (*i.e.* $n < 2^k$) and γ is a value determined so that the final reductions can be avoided. In our case, we chose $\gamma = 4$. The details are explained by the following Lemma based on work by Batina and Muurling [7].

Lemma 1. *If the Montgomery parameter R satisfies the inequality $R > 16n$, for inputs $X, Y < 4n$, $S < 2n$ and $M < R$, the result T will satisfy: $T < 4n$ (as required).*

Proof: Montgomery multiplication as implemented in the MALUn calculates

$$\begin{aligned} T = \mathrm{MALUn}(X, Y, S) &= \frac{XY + Mn}{R} + S \\ &< \frac{4n \cdot 4n}{16n} + n + 2n \le 4n\,. \end{aligned} \tag{3.2}$$

□

Although the reduction step was needed in the original notation of the Montgomery algorithm, we use a method which does not require reduction. For convenience of repeating usage of Eq. (3.1), the so-called Montgomery form is applied because the output is in the Montgomery form as well. The latency to calculate a MALUn operation is $2 \cdot \lceil (k+\gamma)/d \rceil$ cycles in total.

With regard to the critical path delay of the MALUn, we discuss it for a general d. The cell, a column of the datapath of the MALUn, uses d sets of 4-2 CSAs (Fig. 3.1b), *i.e.* the inputs and outputs of the cell are presented in the CS form during the operation. One 4-2 CSA is composed of two Full Adders (FAs), and a cell of the digit size d needs $2d$ sets of FAs. Therefore, the critical path of the datapath is estimated based on the critical path delay of the cells as

$$T_{4-2CSAs} = 2dT_{FA}\,. \tag{3.3}$$

Here, we assume that the delay for the sum and carry calculations are the same. The propagation of the signal $s_{i,j}$ goes through d sets of the cells where two FAs are used. The right-most cell, `cell(i,0)` provides the m_i vector for the rest of the cells. As expressed in Eq. (3.4), the path for generating a bit of m_i only consists of a 3-input XOR in the right-most cell.

$$\begin{aligned} m_i[0] &= vs_{i,0} \oplus vc_{i,0} \oplus x_i[0]y_0 \,, \\ m_i[1] &= s_{i,0}[1] \oplus c0_{next}[0] \oplus x_i[1]y_0 \,, \\ &\vdots \\ m_i[d-1] &= s_{i,0}[d-1] \oplus c0_{next}[d-1] \oplus x_i[d-1]y_0 \,. \end{aligned} \tag{3.4}$$

The worst combination of the paths through the logic generating m_i results in

$$dT_{m_i} + dT_{FA} \,. \tag{3.5}$$

This path delay is assumed to be equivalent to or shorter than $T_{4-2CSAs}$. As can be seen from Eq. (3.5), the MALUn has also an area and delay trade-off that can be adapted with the size of d. In this way, the proposed MALUn is scalable regarding the field size k and the digit size d; a trade-off can be made between cost and performance by considering system requirements, *i.e.* system constraints for performance, area and clock frequency.

3.2.2 Reconfigurable MALU for Modulo n Operation

In order to perform a high-performance modular operation in Eq. (3.1), we need to allocate $(k+\gamma-1)$ cells for the datapath of the MALUn. For instance, the case of a 256-bit ECC over GF(p) (denoted as ECC256p) and 2048-bit RSA (denoted as RSA2048) needs 260 and 2052 cells, respectively. For the general case ($K \times D$ sets of MALUn$_{k\times d}$) the block diagram of the reconfigurable datapath is illustrated in Fig. 3.2.

The datapath is configured by changing the interconnection of the MALUn$_{260\times 1}$ that is determined by three multiplexers as shown in Fig. 3.3 and two- or three-bit registers for selecting them. The signals *sel1* and *sel2* are used for configuring the datapath, and *sel3* is used for configuring the F/Fs. Those multiplexers and F/Fs for the selector signals are considered as the area overhead (denoted as AO) introduced by this reconfigurable feature. It is approximately estimated as follows.

$$\begin{aligned} AO_{base} &= KD(A_{MUX(sel1)} + A_{MUX(sel2)} + A_{MUX(sel3)} + A_{FF(config)}) \\ &= \{(k+6d)KD - 2Kd\} \cdot A_{3-1MUX} + (3KD - K) \cdot A_{FF} \,, \end{aligned} \tag{3.6}$$

where A_{3-1MUX} and A_{FF} denote the area of a 3-to-1 multiplexer and k-bit F/Fs. In addition to AO_{base}, some more F/Fs are not used depending on the configuration. For example, we consider the case of using eight copies of MALUn$_{260\times 1}$ ($k = 260, d = 1, K = 2$ and $D = 4$). When supporting RSA2048, the datapath is configured as MALUn$_{2080\times 1}$. In this configuration, the horizontal connections of

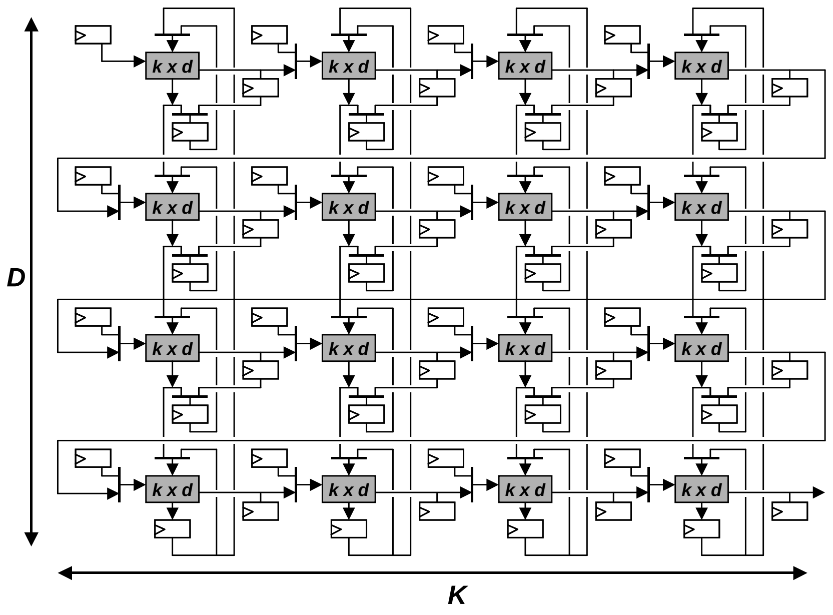

Figure 3.2: Reconfigurable datapath using $K \times D$ sets of $\mathrm{MALUn}_{k\times d}$.

the MALUn cores are not re-timed by F/Fs for S and R. That is, we need only two registers for S and R. Therefore, the total area overhead becomes

$$\begin{aligned} AO_{RSA2048} &= AO_{base} + (16-2)A_{FF} \\ &= 11\,448A_{3-1MUX} + 36A_{FF}\,. \end{aligned} \tag{3.7}$$

Since we target a platform which supports both ECC256p and RSA2048 (with or without CRT), we introduce a coarse grain datapath of the $\mathrm{MALUn}_{260\times 1}$ ($k = 256$ and $d = 1$) and allocate its copies. Figure 3.4 shows the interconnection for ECC and RSA in this case.

Likewise, for RSA2048 with CRT (Chinese Remainder Theorem) [69], the datapath is configured to have two sets of $\mathrm{MALUn}_{1040\times 1}$ as shown in Fig. 3.4b. $\mathrm{MALUn}_{1040\times 1}$ needs two registers for S and R. Therefore, the area overhead for RSA2048 with CRT is estimated as

$$\begin{aligned} AO_{RSA2048(CRT)} &= AO_{base} + (16-2-2)A_{FF} \\ &= 11\,448A_{3-1MUX} + 34A_{FF}\,. \end{aligned} \tag{3.8}$$

For ECC256p, we configure the datapath so that two sets of $\mathrm{MALUn}_{260\times 4}$ can be used in parallel. This configuration uses vertical series of $\mathrm{MALUn}_{260\times 1}$ as shown in Fig. 3.4c. The intermediate values for the virtual carry and sum are

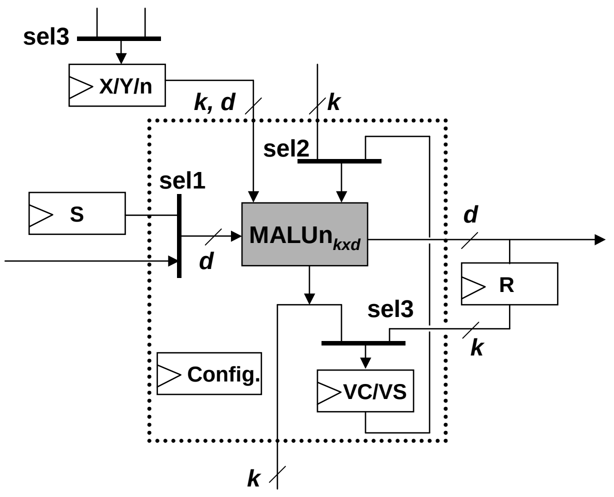

Figure 3.3: F/Fs and multiplexers of the MALUn$_{k\times d}$.

stored in the F/Fs located at the bottom of Fig. 3.4. In this configuration, only one fourth of the registers for X, Y and n are used in each MALUn$_{260\times 1}$. Therefore, the total area of the overhead becomes

$$\begin{aligned} AO_{ECC256p} &= AO_{base} + (260 \times 12 + 260 \times 24 \times 3/4)A_{FF} \\ &= 11\,448A_{3-1MUX} + 7822A_{FF}\,. \end{aligned} \tag{3.9}$$

In the case of the RSA configuration, we can utilize the F/Fs with almost no waste, while the configuration of ECC cannot use them effectively.

In order to exploit the unused F/Fs, we introduce another kind of reconfigurability. Different from RSA, ECC needs to store the intermediate variables during point operations. For this purpose, two sets of 14 entries of 260-bit RAM (28×260-bit RAM) can be configured with the unused F/Fs. More precisely, the registers controlled by the selectors *sel3* can be accessed from a system controller and used for storing the intermediate variables for ECC. In this case, the area overhead is considered as

$$AO_{ECC256p} = 11\,448A_{3-1MUX} + 542A_{FF}\,. \tag{3.10}$$

Thus, we can make the best use of the hardware resources also for the ECC configuration.

The critical path delay for each configuration is as follows:

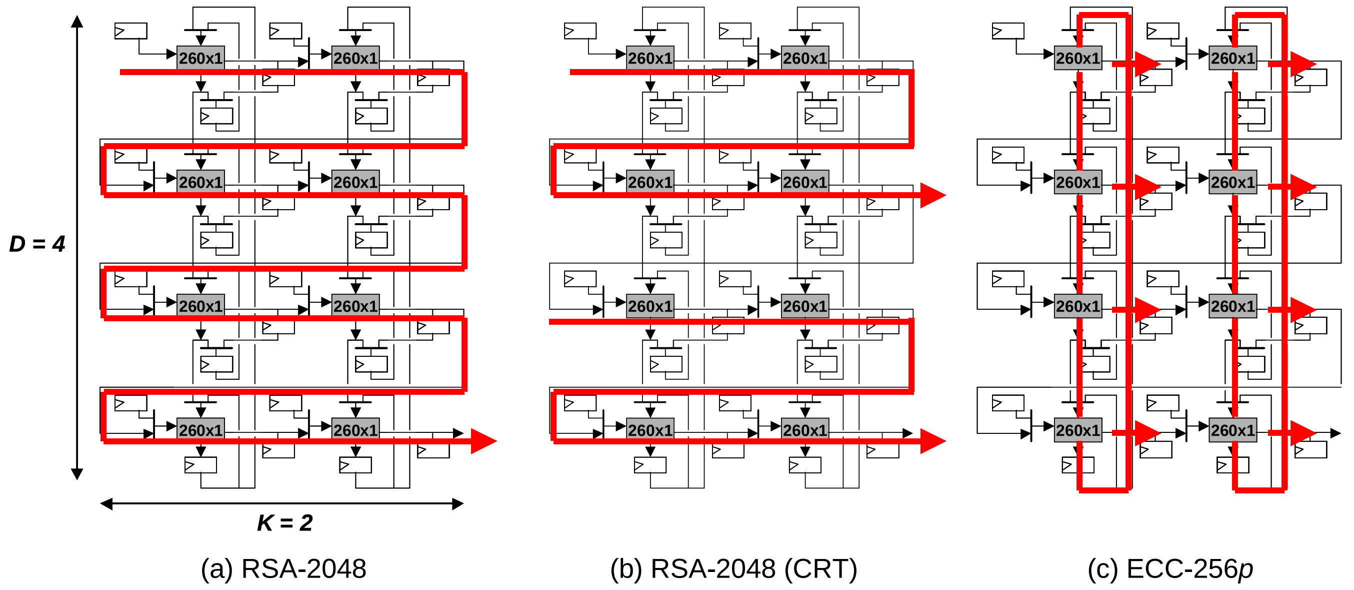

Figure 3.4: Reconfigurable datapath using 2×4 sets of $\mathrm{MALUn}_{260\times1}$. (a) The case of RSA2048: $\mathrm{MALUn}_{2080\times1}$. (b) The case of RSA2048(CRT): $2 \cdot \mathrm{MALUn}_{1040\times1}$. (c) The case of ECC256: $2 \cdot \mathrm{MALUn}_{260\times4}$.

Table 3.1: Implementation result of the proposed datapath.

Target Platform	Number of slices	Critical Path $[ns]$	
		RSA config.	ECC config.
XC3S5000-5	27 597 slices	10.6	25.3
0.25-μm CMOS	243 Kgates	4.0	6.3

$$
\begin{aligned}
T_{RSA2048} = T_{RSA2048(CRT)} &= 2T_{FA} + 2T_{3-1MUX} + T_{FF} + T_{wiring} \\
T_{ECC256p} &= 8T_{FA} + 4T_{3-1MUX} + T_{FF} + 4T_{wiring}\,.
\end{aligned}
\tag{3.11}
$$

We assume that the wiring delay in the critical path of ECC is four times longer than that of RSA. As can be seen from Eq. (3.11), the critical path delay of ECC is about four times longer than RSA. Therefore, we need to assume that we can use two different clocks, *e.g.* providing a divided clock for the ECC configuration, in order to facilitate high performance for both configurations.

3.2.3 Implementation Results of the Reconfigurable MALUn

We implemented the proposed datapath on a Xilinx FPGA [104]. The results after the place-and-route from the Xilinx ISE tool are shown in Table 3.1. The same design is also synthesized with a 0.25-μm CMOS technology by using Synopsys Design Vision and the pre-layout netlist is used for the cost and performance estimation. As for the constraints of the synthesis, the design is first set to the ECC configuration and then the critical path delay for the RSA configuration is estimated by performing STA (Static Timing Analysis).

3.3 MALU over GF(2^m)

In this section, we discuss the implementation of the MALUb supporting GF(2^m) operations. One way to implement an efficient datapath for modular multiplication over GF(2^m) is to use a specific irreducible polynomial, *e.g.* a trinomial such as $P(x) = x^{193} + x^{15} + 1$. In this case, modular multiplications can be implemented efficiently, and the squaring operation needs only several modular adders if it is implemented separately from the multiplier. The critical path delay of the squarer is low enough to use it as a one-cycle operation. Likewise, the modular inversion can be efficiently implemented [80]. Therefore, three different modular operations need to be supported in a datapath block. However, this dedicated approach makes the datapath inflexible in the operand size (*i.e.* the field size) in addition to the fixed irreducible polynomial.

In contrast, our proposed MALUb supports a single operation on a finite field over GF(2^m), *e.g.* $A(x) \cdot B(x) + C(x) \bmod P(x)$. The irreducible polynomial, $P(x)$ can be chosen arbitrarily. All modular operations necessary for point/divisor multiplication of the curve-based cryptography can be processed by using the single core iteratively, including modular inversions.

3.3.1 Overview of the MALU over GF(2^m)

The MALUb is based on an MSB-first bit-serial polynomial-basis GF(2^m) multiplier as illustrated in Fig. 3.5a. This is a hardware implementation that computes $A(x) \cdot B(x) \bmod P(x)$ where $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$, and $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$. The intermediate result, $T(x) = \sum_{i=0}^{m} t_i x^i$ is stored in a register as shown in Alg. 3.4.

Algorithm 3.4: MSB-first bit-serial modular multiplication over GF(2^m).

```
Require: $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$, $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$.
Ensure: $A(x) \cdot B(x) \cdot 2^{-m} \bmod P(x)$.
1: $T(x) = \sum_{i=0}^{m+1} t_i x^i \leftarrow 0$
2: for $i$ from $m-1$ down to 0 do
3:     $m_i \leftarrow t_m$;  $q_i \leftarrow t_{m+1}$;
4:     $T(x) \leftarrow (T(x) + a_i B(x) + m_i P(x))x$;
5: end for
6: return $T(x)/x$
```

The MALUb XORs three inputs which are $a_i B(x), m_i P(x)$ and $T(x)$, and then outputs the next intermediate result, $T(x)$ by computing

$$T(x) = (T(x) + a_i B(x) + m_i P(x))x \,, \tag{3.12}$$

where $m_i = t_m$. By providing $T(x)$ as the next input and repeating the same computation n times, one can obtain the result, $A(x) \cdot B(x)$ (see Sakiyama *et al.* [79]). Moreover, by providing $B(x) + D(x)$ instead of $B(x)$ and XORing the result with $C(x)$, the operation form, $(A(x) \cdot (B(x) + D(x)) + C(x)) \bmod P(x)$ can also be supported.

The proposed datapath is scalable in the digit size d (vertical direction in Fig. 3.5b. The corresponding algorithm can be obtained by loop-unrolling Alg. 3.4 as shown in Alg. 3.5. In this case, one operation finishes in $\lceil m/d \rceil$ cycles. Thus, the appropriate digit size can be parameterized in the datapath design and can be determined by exploring the best combination of cost and performance.

The proposed MALUb can offer wider field sizes by reconfiguring the datapath. Namely, our proposed MALUb core has additional ports that are used for interconnecting neighboring MALUb cores by setting a configuration register. Thus,

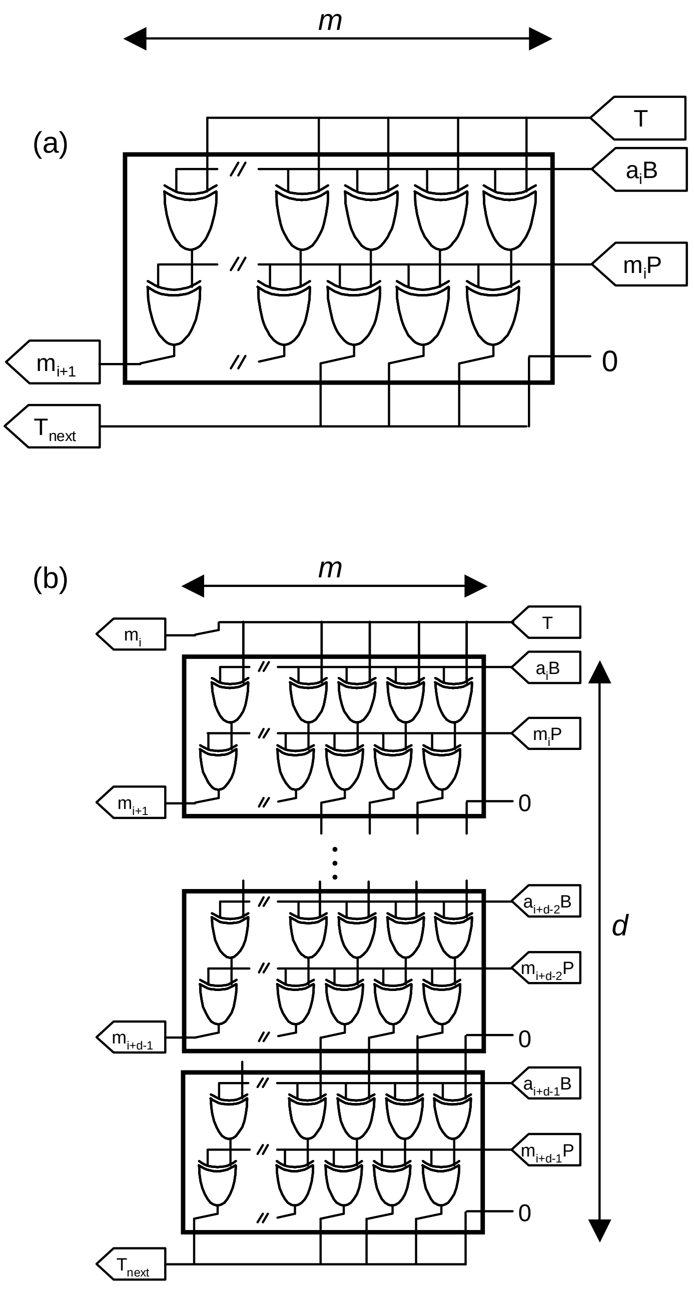

Figure 3.5: Block diagram for one MALUb – extension of the digit size. (a) The building block corresponding to Alg. 3.4. (b) GF(2^m) multiplier with the digit size d.

Algorithm 3.5: MSB-first digit-serial modular multiplication over GF(2^m) (digit size d).

```
Require: $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$, $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$.
Ensure: $A(x) \cdot B(x) \cdot 2^{-m} \bmod P(x)$.
1: $T = \sum_{i=0}^{m+1} t_i x^i \leftarrow 0$;
2: for $i$ from $d\lceil \frac{m}{d} \rceil - 1$ down to 0 do
3:     $i \leftarrow i - d$
4:     $m_i \leftarrow t_m, \quad q_i \leftarrow t_{m+1}$
5:     $T(x) \leftarrow (T(x) + a_i B(x) + m_i P(x))x$
6:     $m_{i+1} \leftarrow t_m, \quad q_{i+1} \leftarrow t_{m+1}$
7:     $T(x) \leftarrow (T(x) + a_{i+1} B(x) + m_{i+1} P(x))x$
           $\vdots$
8:     $m_{i+d-1} \leftarrow t_m, \quad q_{i+d-1} \leftarrow t_{m+1}$
9:     $T(x) \leftarrow (T(x) + a_{i+d-1} B(x) + m_{i+d-1} P(x))x$
10: end for
11: return $T(x)/x$
```

we can construct a new datapath that can handle larger operands. The details are explained in the next section.

3.3.2 Reconfigurable MALU over GF(2^m)

The field size m is determined by the key-length. A larger field size can also be obtained by interconnecting several MALUb cores in the horizontal direction. Hence, various implementation options can be chosen with the MALUb. For instance, the cryptoprocessor can support arbitrary field sizes up to 587 bits when using six copies of the MALUb cores each of which support a field size of 97 bits.

The schematic circuit diagram illustrated in Fig. 3.6 describes how the MALUb cores are reconfigured for supporting different field sizes. When each of the MALUb cores is used independently without interconnections for the purpose of parallel processing, *cfg1* is set to zero so that a d-bit vector $\mathbf{m} = (m_{i+d-1}, \cdots, m_i)$ in Alg. 3.5 can be used for the modular reduction in its own core. More precisely, the vector $\mathbf{m}$ determines whether the irreducible polynomial should be XORed with the intermediate result so that the degree of $T(x)$ can be at most n or $deg(T) \leq n$. In this way, the LSBs of $T(x)$ can be always 0 because they are provided by another d-bit vector $\mathbf{q} = (q_{i+d-1}, \cdots, q_{i+1}, q_i)$ (see Alg. 3.5) of the neighboring MALUb core. This corresponds to the 1-bit left-shift operation.

On the other hand, when supporting a wider field-size datapath by interconnecting several MALUb cores, each vector $\mathbf{m}$ is exchanged with one from the neighboring core. For instance, α copies of the MALUb over GF(2^m) with digit

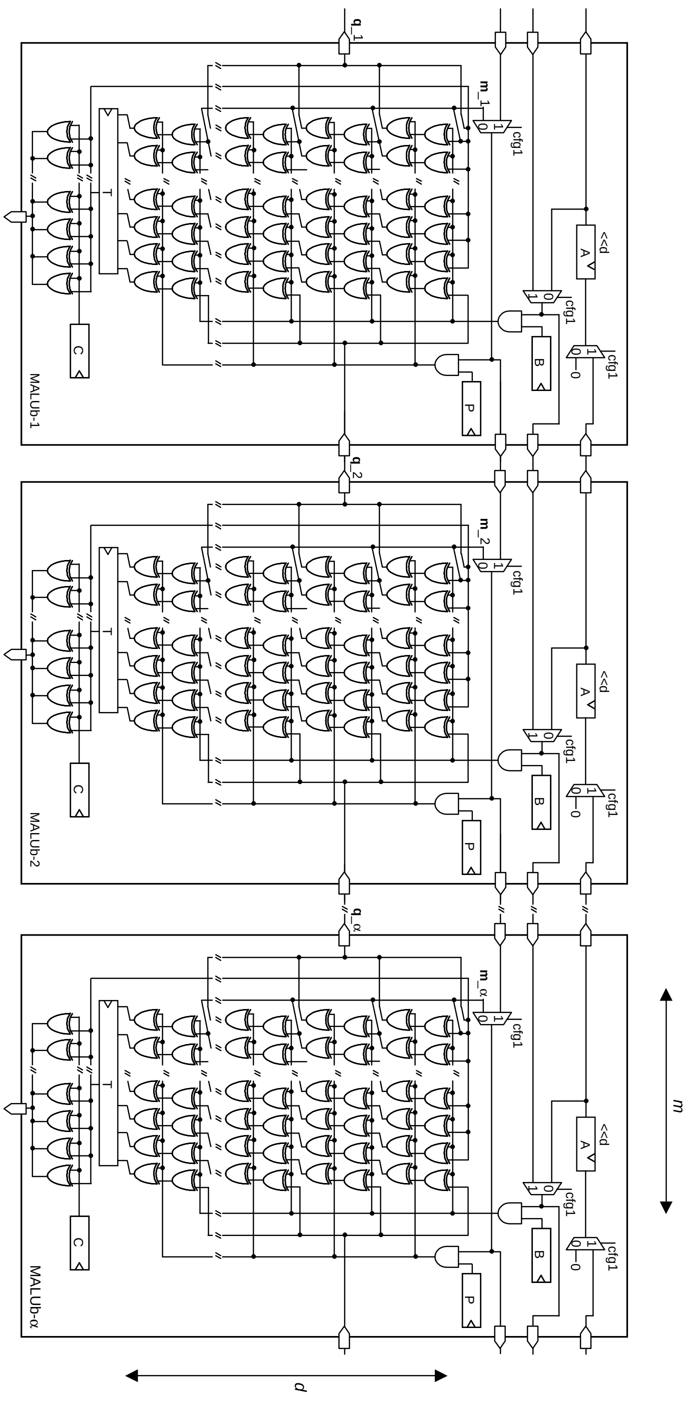

Figure 3.6: Block diagram for the reconfigurable datapath – extension of parallelism and the field size. Depending on the setting of the interconnections, various datapaths can be reconfigured, *e.g.* α copies of the MALUb over GF(2^m) with the digit size d or one MALUb over GF$(2^{\alpha(m+1)-1})$ with the digit size d.

size d ($6 \cdot \text{MALUb}_{m\times d}$) can be reconfigured as one $\text{MALUb}_{(\alpha(m+1)-1)\times d}$. Suppose that the MALUb-1, MALUb-2 and MALUb-3 are reconfigured to make a datapath for the triple field size (more precisely $3m+2$), the configuration signal, $cfg1$ should be set to 0, 1 and 1 respectively for the MALUb-1, MALUb-2 and MALUb-3.

3.3.3 Implementation Results of the Reconfigurable MALU over GF(2^m)

The performance of the reconfigurable MALUb over GF(2^m) is heavily dependent on the number of MALUb cores and the datapath configuration in each MALUb core. They also determine the supported range of field sizes. The field sizes of interest in this thesis are 163 and 193 bits for ECC because they offer a security level larger than or equal to 1024-bit RSA [8]. The corresponding field sizes for HECC are 83 and 97 bits. Therefore, it is reasonable to use the MALUb cores with a datapath of length $m = 97$. As for the digit size $d = 12$ is chosen as an example case. This datapath is denoted as $\text{MALUb}_{97\times 12}$ and one modular multiplication over GF(2^{97}) can be computed in $\lceil m/d \rceil$ or 9 cycles.

Suppose that six cores with the datapath, $\text{MALUb}_{97\times 12}$ are allocated in the cryptoprocessor, various datapath configurations can be supported. Table 3.2 summarizes selected hardware configurations and the number of clock cycles for one MALUb instruction ($\lceil m/d \rceil$) over different field sizes including the clock cycles for memory accesses. The memory accesses cost $l+1$ cycles as will be explained in Sect. 5.2.2, where l is the number of parallel executions (*i.e.* execution of l-way parallelism). The throughput of the $\text{MALUb}_{m\times d}$ instructions can be estimated with

$$\frac{l \cdot m}{\lceil m/d \rceil + l + 1} \quad \text{[bits/clock]} \quad (l = 1, 2, 3, 4)\,. \tag{3.13}$$

Although the number of datapaths used in parallel varies from 1 to 6 depending on the field lengths, we show the throughput of the MALUb with at most four-way parallelism. Figure 3.7 illustrates the minimal and maximal throughput of the MALUb operation as a function of the field size. As can be seen from the figure, this configuration can offer a high throughput for a field size around 97, 195 and 293 bits because we use $\text{MALUb}_{97\times 12}$ as a building block of the datapath. The maximum throughput can be obtained only if all the datapaths are used in parallel. When no parallelism can be found in MALUb instructions (*i.e.* in the case of single execution of the MALUb instruction), our cryptoprocessor performs at the minimum throughput. In other words, by exploiting parallelism for $m \leq 293$ in the MALUb instructions, the throughput can be improved depending on the number of parallel executions. However, the throughput is almost constant for $m > 293$.

3.3.4 Parallelized MALU over GF(2^m)

For further speed-up, we propose a parallel processing architecture using two MALUb cores in parallel. The parallelized MALUb or the P-MALUb is based on the implementation presented by Batina *et al.* in [3]. The datapath of the P-MALUb deals with a polynomial basis (see Sect. 2.2.2. It consists of a conventional bit-serial MSB-first multiplier and a Montgomery multiplier over GF(2^m) as illustrated in Fig. 3.8 (the XOR chain in the left and right side, respectively). This datapath implementation computes

$$(A(x) \cdot B(x) \cdot x^{-N} + C(x)) \bmod P(x)\,, \tag{3.14}$$

where $N = \lceil m/2 \rceil$, $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$, $C(x) = \sum_{i=0}^{m-1} c_i x^i$ and $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$. This can be explained as follows. First, the MSB-first multiplier performs modular multiplication by providing the coefficients of

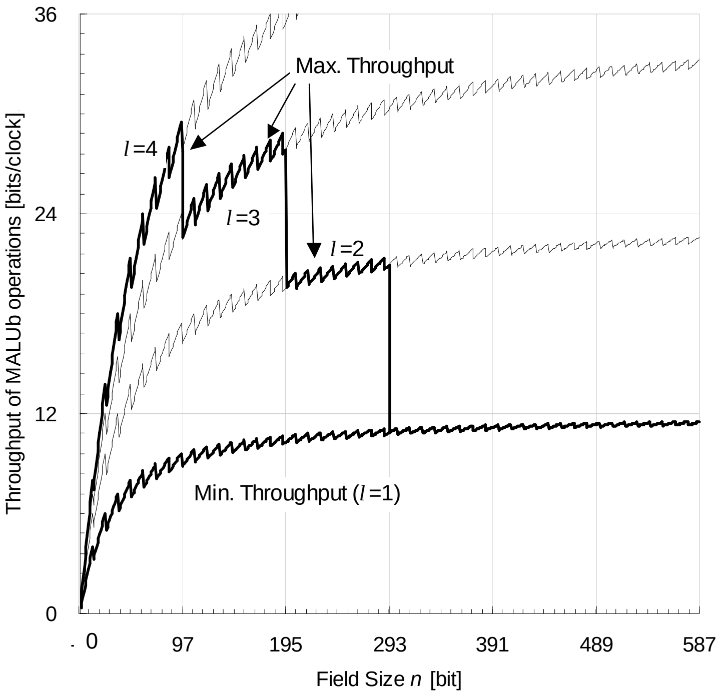

Figure 3.7: Throughput for different configurations of the datapath with six MALUb$_{97\times12}$.

Table 3.2: Possible configurations of the datapath with six cores of $m = 97$ and $d = 12$ (MALUb$_{97\times12}$) and the execution cycles for one MALUb instruction over a binary field.

Configuration	Supportable Field	Execution Cycles	l-way Parallelism
$6 \cdot \text{MALUb}_{97\times12}$	GF(2^{97})	9 cycles	4
$3 \cdot \text{MALUb}_{195\times12}$	GF(2^{193})	17 cycles	3
$2 \cdot \text{MALUb}_{293\times12}$	GF(2^{283})	24 cycles	2
$1 \cdot \text{MALUb}_{587\times12}$	GF(2^{571})	48 cycles	1

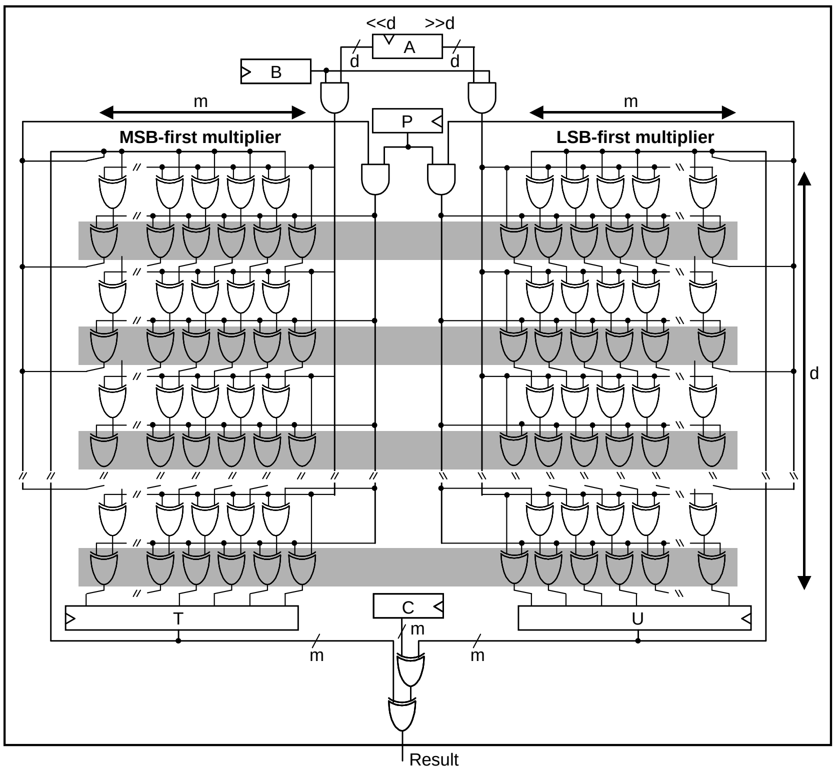

Figure 3.8: Parallelized datapath for GF(2^m) operation. MSB-first digit-serial multiplier (left). Montgomery modular multiplier (right). Most of the XORs in the gray blocks can be removed when fixing the irreducible polynomial, *e.g.* $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$.

$A(x)$ from the MSB up to N bits. Thus we obtain the partial modular multiplication result as

$$\sum_{i=N}^{m-1} a_i x^{i-N} \cdot B(x) \bmod P(x)\,. \tag{3.15}$$

Second, a Montgomery multiplication is executed by providing the coefficients of $A(x)$ from the LSB as follows.

$$\sum_{i=0}^{N-1} a_i x^{i} \cdot B(x) \cdot x^{-N} \bmod P(x)\,. \tag{3.16}$$

Lastly, the above two multiplication results and $C(x)$ are added, and Eq. (3.14) is obtained. The advantage of this method lies in the fact that modular multiplication can be accelerated by computing two different multiplications *in parallel*. As the operation can be completed in N cycles, the latency of the modular multiplication can be improved compared to the case of using individual multipliers that take m cycles.

Although $B(x)$ can be divided into different bit sizes, it is simple to separate it in half (*i.e.* $\lceil m/2 \rceil$ bits) so that the computation in Eq. (3.14) is distributed equally to two different type of multipliers. This thesis also uses this strategy. For instance, $m = 163$ can be used for GF(2^{163}). Then a 163-bit$\times$163-bit modular multiplication is divided into two 83-bit$\times$163-bit multiplications.

For convenience of repeated usage of Eq. (3.14), the modified Montgomery form or $\tilde{A}(x) = A(x) \cdot x^N \bmod P(x))$ is applied because the output is in the Montgomery form as well.

$$\begin{aligned}&(\tilde{A}(x) \cdot \tilde{B}(x) \cdot x^{-N} + \tilde{C}(x)) \bmod P(x)\\ =&(A(x) \cdot B(x) + C(x)) \cdot x^{N} \bmod P(x)\,.\end{aligned} \tag{3.17}$$

Next, the digit-serial implementation of two multipliers is explained in detail. The MSB-first multiplier sums up three inputs that are $a_iB(x), m_iP(x)$, and $T(x)$, and then outputs the intermediate result, $T_{next}(x)$ by computing

$$T_{next}(x) = (T(x) + a_iB(x) + m_iP(x))x\,, \tag{3.18}$$

where $m_i = t_m$. By providing the most significant d bits of $A(x)$ and $T_{next}(x)$ as the next-cycle input $T(x)$, one can obtain the result of Eq. (3.15) in $\lceil N/d \rceil$ cycles. The detailed explanation is also discussed in [79].

As for the Montgomery multiplier, $A(x)$ is sent to the datapath from the LSB. In this way,

$$U_{next}(x) = (U(x) + a_iB(x) + w_iP(x))/x \tag{3.19}$$

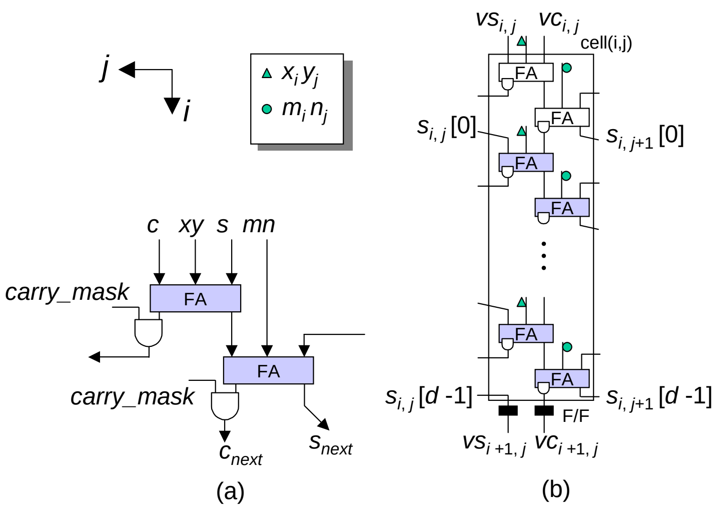

Figure 3.9: Dual-field MALU, MALUD. (a) 4-2 CSA with carry-mask logic. (b) Cell component for 4-2 CSA with carry-mask logic.

is computed where $w_i = u_0 + a_i b_0$. The result of Eq. (3.16) can be obtained in $\lceil N/d \rceil$ cycles for the case of d-bit digit-serial implementation. Those results of $T(x)$ and $U(x)$ are XORed with $C(x)$ at the last step in multiplication.

The proposed datapath is scalable in the digit size d which can be decided by exploring the best combination of system performance and cost. The field size m is determined by the key-length. Figure 3.8 illustrates the block diagram of the digit-serial P-MALUb.

3.4 Dual-field MALU

Based on the implementation by Großschädl [29], the MALUn is modified to support modular operations over both GF(p) and GF(2^m). The so-called dual-field MALU or MALUD is also scalable in the digit size d and can be reconfigurable as mentioned in the previous sections.

The proposed MALUD is based on a digit-serial Montgomery multiplier for modulo n operations as introduced in Alg. 3.3. In order to support GF(2^m) operations, the logic for the carry propagation is gated with a mode signal as shown in Fig. 3.9.

As introduced in Alg. 3.5, an MSB-first multiplier is employed for GF(2^m) operations instead of the Montgomery algorithm for modulo n operations. This

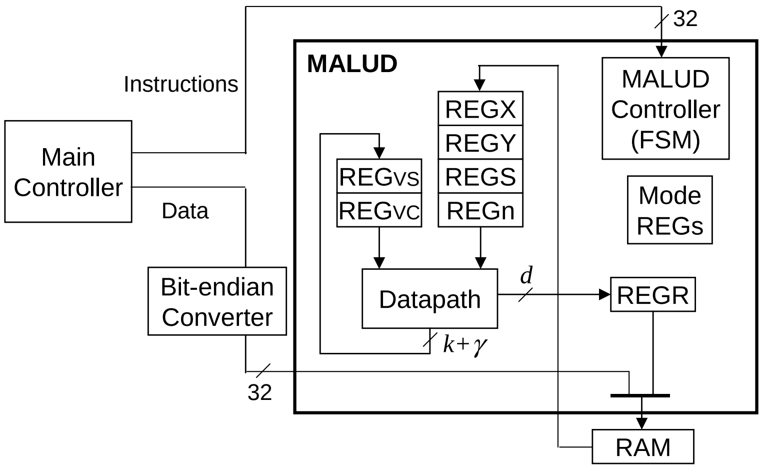

Figure 3.10: Block diagram of a system with the MALUD.

can be easily implemented by simply changing the bit endianness of the data, *i.e.* reversing the bit-order of data when setting modulus n and input data. The permutation logic can be implemented in glue logic between the MALUD and a data controller as shown in Fig. 3.10. As a result, we need no conversions from/to the Montgomery representation in the case of operating GF(2^m) arithmetic.

3.4.1 Implementation Results of the Dual-field MALU

The hardware blocks were described with GEZEL and VHDL, synthesized with Synplify Pro, place-and-routed with Project Navigator by Xilinx, and implemented

Table 3.3: Synthesis results for area and performance of the MALUD$_{256\times d}$ with $k = 256$.

Digit Size (d)	Max. Freq. [MHz]	Area [slices]	Required Clock Cycles for the MALUD$_{256\times d}$ GF(p)	GF(2^{239})	Throughput Rate [Mbps] GF(p)	GF(2^{239})
1	132.6	2430	522	247	65.0	128.3
2	123.9	2932	266	128	119.2	231.3
4	110.4	4836	138	68	204.8	388.0
8	58.9	8867	74	38	203.8	370.5

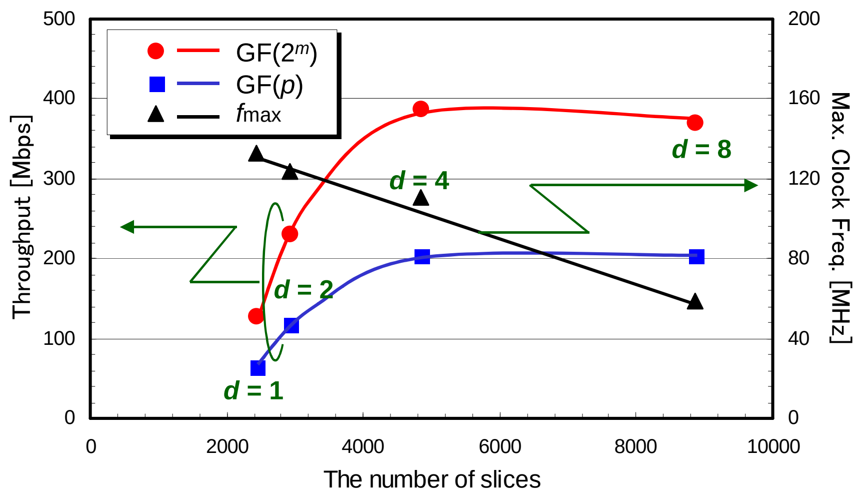

Figure 3.11: Area-performance trade-off of the $\mathrm{MALUD}_{256\times d}$ for different digit sizes ($d = 1, 2, 4, 8$).

on a Virtex-II PRO (XC2VP30). Note that our logic design is not dedicated to FPGAs, *i.e.* all the combinatorial logics and F/Fs are mapped on slices. As an example, we chose 256 bits as the field size targeting ECC256p or ECC239b (ECC over GF(2^{239})). The implementation results are summarized in Table 3.3. The performance increases for $d \leq 4$ as d increases. If we set d larger than 4, the performance does not increase any further.

If the critical path delay is proportional to the digit size d, the throughput of the MALUD should be constants at a maximum clock frequency in theory. However this is not true in our FPGA design. As one of the reasons, we consider the timing margins needed for synchronous designs such as clock uncertainty and setup/hold time constrain for F/Fs. They have a profound effect on the maximum clock frequency especially for small d. Another reason is due to the fact that the MALUD controller logic realized by a Finite State Machine (FSM) is related to the critical path for smaller d. This means that the potential performance of the MALUD is concealed by the critical path delay of the FSM. In other words, it is difficult to make best use of the datapath in an actual system with a small digit size d.

As shown in Table 3.4, our design shows a faster result than earlier work except the design by Satoh and Takano [81] and Mentens [59]. The design of [81] uses a scalable r-bit $\times$ r-bit multiplier and shows better performance with $r = 32, 64$. In

Table 3.4: Performance comparison of modulo n operations.

Reference Design	Field Size [bit]	Target Platform	Throughput [Mbps]	Modular Operation
This Work	256	Virtex-II Pro	205	$XY \pm Z$ $(d = 4)$
Satoh [81]	256	0.13-μm CMOS	103	XY $(r = 16)$
			241	XY $(r = 32)$
			534	XY $(r = 64)$
McIvor [56]	512	Virtex-II	105	XY, only GF(p)
Savaş [82]	256	1.2-μm CMOS	39	XY
Kaihara [41]	256	0.35-μm CMOS	174	XY, only GF(p)
Crowe [14]	256	Virtex-II	45	XY
Harris [15]	256	Virtex-II	132	XY
Mentens [59]	256	Virtex-II Pro	883	XY

order to achieve their best performance of $r = 64$, we need a factor of 2.5 times faster clock frequency or a deeper digit size ($d \geq 10$) with the same clock frequency as the case of $d = 4$. Recent work by Mentens [59] shows significantly faster results on a Xilinx FPGA (XC2VP30). The design makes best use of the FPGA resource to accelerate Montgomery multiplications. As a result, the throughput of 883 Mbps is achieved with 68 dedicated 16-bit multipliers and 2216 slices.

However, as our performance also includes addition/subtraction, the difference of the performance is regarded smaller in the ECC and HECC applications. In fact, our implementation of an ECC processor based on the proposed MALUD shows that 256-bit ECC over GF(p) can be computed in 2.7 ms [78], and this result is equivalent to the 256-bit ECC results implemented by Satoh and Takano [81], and by Mentens [59], which are 2.7 ms and 2.3 ms respectively.

The most relevant work was done by Mclvor *et al.* [56]. They also implemented on a Xilinx FPGA and used 4-2 CSAs. Although their result is based on 512-bit operations, it seems impossible to obtain the double performance with the field size of 256 bits because the critical path is not related to the field size.

3.5 Conclusions

We have proposed a new datapath, the MALU that enables modular operations modulo n or GF(2^m). We also have presented a dual-field MALU that supports modular operations modulo n and operations over GF(2^m) by modifying the 4-2 CSAs. The MALU is scalable and flexible by setting the parameters that are the operand size k, the digit size d and the number of the MALU cores. By reconfiguring the connections between several coarse-grain MALU cores, the datapath offers high-speed modular multiplications and additions for arbitrary field sizes with efficient use of hardware resources.

The proposed reconfigurable datapath is suitable for cryptosystems including RSA and curve-based cryptography. Thus, it can be commonly used as a datapath component in various public-key cryptosystems such as a high-performance server, a low-power smart card, etc. This unique feature of the MALU makes it possible to implement efficient public-key cryptosystems. Therefore, we use the MALU for all implementations presented in the following chapters.

Chapter 4

HW/SW Co-design for Public-key Cryptography

4.1 Introduction

Implementing PKC is a challenge for most application platforms varying from software to hardware. The reason is that one has to deal with very long numbers in conditions that are often constrained in area and performance. For the choice of the implementation platform, several factors have to be taken into account. Hardware solutions provide the speed and more physical security, however the flexibility is limited. Software solutions offer flexibility, but a pure software solution is not a feasible option in most resource-limited environments. Although software implementations are flexible, the performance of PKC is very slow on a compact 8-bit CPU and even on a 32-bit low-power embedded CPU. On the other hand, ASIC implementations of PKC show better performance than software ones. This is mainly due to the following reasons:

- **Operation size:** Public-key cryptosystems need repeated modular operations with much longer bit integers than the operation size of a CPU (*e.g.* RSA needs 1024 bits or larger and ECC needs 160 bits or larger). ASICs don't have this bit-width limitation.

- **Latency for memory accesses:** In ASICs, the memory organization and the memory access latency can be tuned to the application. Allocating F/Fs or dedicated RAM, it is possible to use the best of clock cycles for transferring intermediate data.

- **Tightly coupled controller:** A dedicated controller in ASICs can handle operations in a datapath without any stalls.

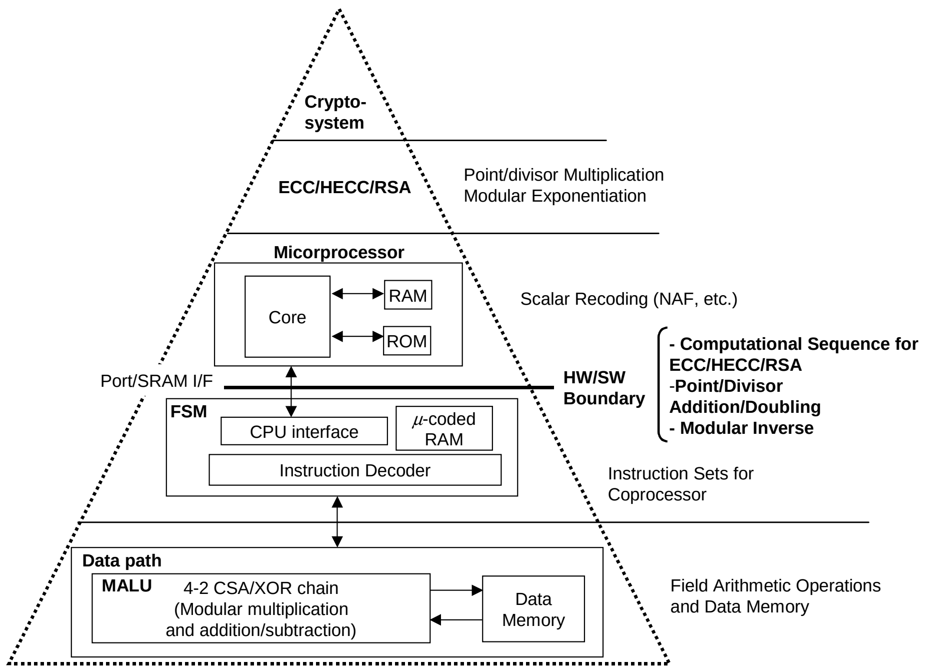

Figure 4.1: Scheme of the hierarchical HW/SW co-design for public-key cryptosystems.

Such ASICs are the best for mass production considering trade-off between cost and performance. However, a fixed hardware solution has only a limited application range and cannot be easily reused.

Hence, HW/SW co-designs are attractive since they can offer the advantage of both a SW flexibility and a HW performance when a proper partition of HW/SW is given. A simultaneous HW and SW optimization can be facilitated by GEZEL. GEZEL is not only a hardware description language, but it offers a design environment for HW/SW co-designs by interfacing with an Instruction Set Simulator (ISS) of a CPU. The combination of GEZEL and ISS makes it possible to simulate the system-level functionality fast, distribute the computations between HW and SW and even optimize the HW coprocessor for a given CPU. This shortens the design time for both HW and SW.

The performance of PKC is primarily determined by the efficient realization of the arithmetic operations in the underlying finite field, *i.e.* modular addition, multiplication and inversion. If projective coordinates are used for an elliptic curve, modular inversion can be neglected because it is needed only when converting the projective point back to the affine point at the last step of point multiplication. Therefore, the system architecture for PKC is normally designed to accelerate the field multiplication and addition. Figure 4.1 shows a scheme of hierarchical HW/SW co-design for PKC. The software part deals with high-level PKC instructions as mentioned in Sect. 1.1.3 (see Table 1.1). Based on this scheme, the best HW/SW architecture for PKC is discussed by using two different types of CPUs: an 8-bit 8051 and a 32-bit ARM.

4.2 HW/SW Co-design for RSA and ECC on the 8051

4.2.1 Cryptosystem Architecture with the 8051

In this section, we discuss the details of HW/SW co-design with the 8051 microprocessor, especially focusing on the memory location for storing intermediate variables since it has a great impact on the performance of the cryptosystem. A 12 MHz 8051 was chosen as a controller for our proposed HW/SW co-design. The modular multiplications, which are the most critical computation components, are implemented on the coprocessor part.

The Coprocessor for RSA and ECC over GF(p)

For the purpose of supporting both RSA and ECC over GF(p), the MALUn is allocated in the coprocessor, where n is a product of two large prime numbers for RSA and p is a prime number for ECC. The MALUn operations need seven sets of $(k + \alpha)$-bit F/Fs that store inputs (X, Y, S and N), outputs (R), and intermediate variables (VC, VS), where k is the field length and $\alpha = 4$ as mentioned in Sect. 3.2.1. The computational sequences for RSA and ECC include the consecutive use of the MALUn operations. The intermediate results need to be stored during the RSA or ECC operations. One possible location for storing the intermediate results is XRAM attached to the 8051 core. This is the simplest and cheapest solution and facilitated without additional hardware storage. Another solution is realized by allocating a register file or SRAM for storing the intermediate variables in the coprocessor, called CPRAM. This is an expensive solution from a cost point of view, especially for large k. However the use of CPRAM can offer a much higher performance since the communication overhead between the 8051 and the coprocessor can be reduced significantly by using CPRAM.

Communication Between the 8051 and the Coprocessor

The 8051 is an 8-bit micro-controller originally designed by Intel that consists of several components: a CPU, 128 bytes of internal memory (IRAM), 4 Kbytes of internal program memory (PROM), up to 64 Kbytes of external RAM (XRAM). The coprocessor is controlled by opcodes sent via one of the 8-bit ports, P0 of the 8051. Here, we assume that P1, P2, and P3 are connected to the address and data pins of the XRAM. All the communications between the 8051 and the coprocessor are via the port. The FSM (Finite State Machine) block in the coprocessor decodes the coprocessor instructions and executes the required operations. A 12 MHz 8051 issues one CPU instruction only once every 12 clock cycles and one port access takes several CPU instruction cycles. This results in huge intervals between consecutive coprocessor instructions. Figure 4.2 illustrates the system architecture with the 8051 and the coprocessor.

In order to make the communication simple, neither interrupt nor register poll schemes are not used here. Namely, the 8051 keeps on sending the coprocessor instructions without checking the completion of the operations in the coprocessor. Therefore, we need to define an instruction set that can complete an operation before the next coprocessor instruction arrives at the coprocessor.

Instruction Set for the Coprocessor

Table 4.1 shows some of the primary instructions for the coprocessor. They are all based on using XRAM in the 8051. The input registers of the MALUn are

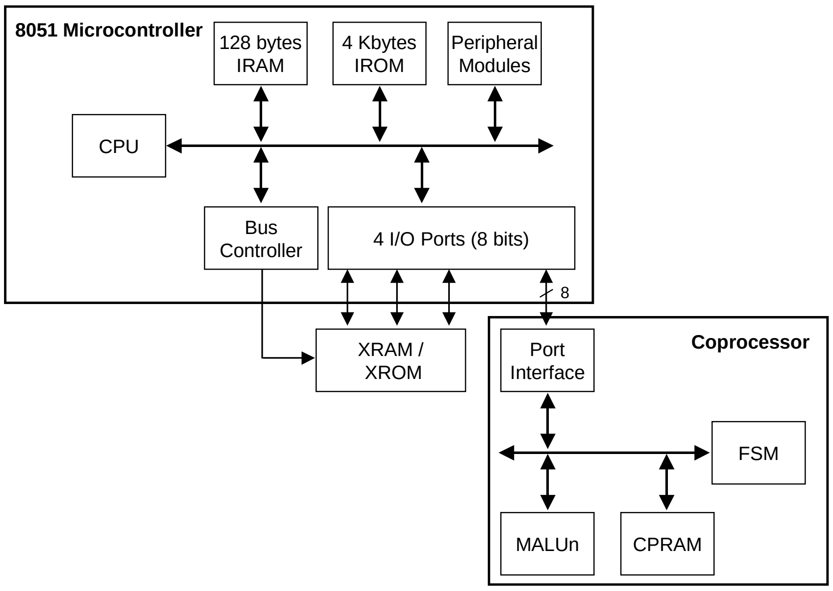

Figure 4.2: Block diagram for the system architecture with the 8051 and the MALUn coprocessor.

Table 4.1: Primary instructions for the coprocessor.

INSTRUCTION	DESCRIPTION	OPERATION
`ST_@dst(din)`	Store data (`din`) to a register by 8 bits from LSB	`REG@dst =(din || REG@dst>>8);`
`LD_@src()`	Load data from a register by 8 bits from LSB	`P2=REG@src[7:0]; REG@src=REG@src>>8;`
`RPT(din)`	Set the number of iteration cycles for CS and CP-stage	`# of iteration =din;`
`MALUn()`	Execute MALUn operation	`MALUn();`
`CS()`	Execute CS-stage of MALUn	`CS();`
`CP()`	Execute CP-stage of MALUn	`CP();`

Table 4.2: Composite instructions for the coprocessor using CPRAM.

INSTRUCTION	DESCRIPTION	OPERATION
`MALUn_@dst(@src1-3)`	Set operands, execute `MALUn()` and store the result	`REG@dst <= MALUn(REG@src1-3);`
`CS(@src1-3)`	Set operands and execute CS-stage	`CS(REG@src1-3);`
`CP_@dst()`	Execute CP-stage and store the result	`REG@dst <= CP();`

set via ports in unit of 8 bits. For instance, setting modulus n to the register requires 20 executions of `SETN` instruction in the case of 160-bit ECC. After all the input values are set to the registers in the MALUn, the `MALUn()` instructions are executed. As explained previously in Sect. 3.2.1, the MALUn operation needs $2\cdot\lceil(k+4)/d\rceil$ cycles. In addition to the primary instructions, composite instructions for the use of CPRAM are defined in Table 4.2. These instructions can reduce the memory access cycles by taking advantage of a fast access to CPRAM.

The 8051 needs four port accesses to issue a coprocessor instruction whose type is `INST(din,din2,din3)` for instance. This is illustrated in Fig. 4.3. The four port accesses require at least 96 clock cycles.

Optimal Digit Size of the MALUn

In order to investigate the best configuration of the MALUn (when CPRAM is used), the performance of EC point multiplication and RSA exponentiation is

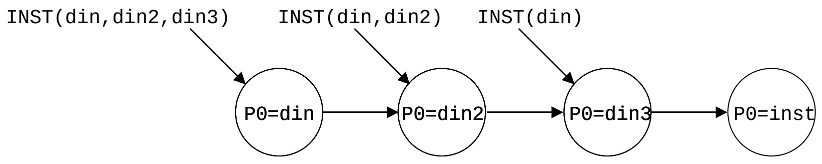

Figure 4.3: Port accesses for coprocessor instructions.

simulated for different digit sizes d by using GEZEL. As shown in Fig. 4.4, the performance of ECC160p increases with the value of d if $d \leq 4$. The performance of RSA1024 increases even for $d > 4$, however only a small performance gain is observed for $d > 4$. This can be explained as follows.

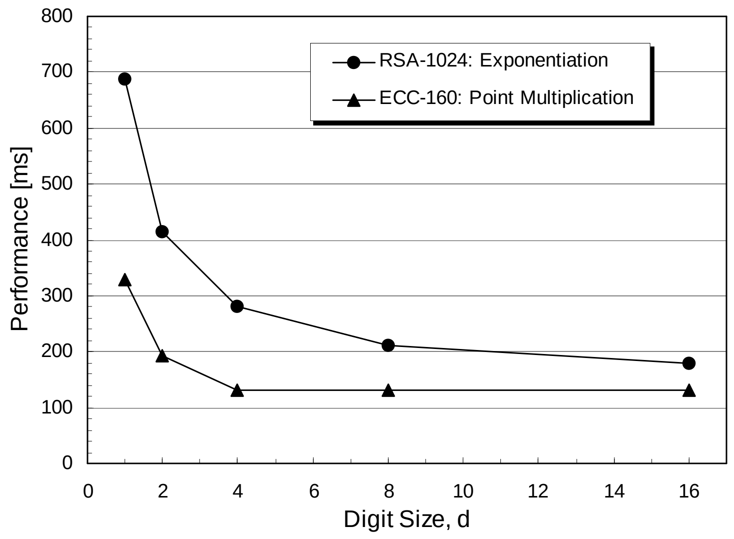

Figure 4.4: Performance for point multiplication of ECC160p and exponentiation of RSA1024.

When the digit size d is large enough to complete the `MALUn(din, din2, din3)` instructions within 96 clock cycles, the number of the MALUn instructions determines the performance of RSA and ECC. This fact indicates that the use of a large digit size does not always improve performance because the 8051 cannot dispatch instructions to keep the coprocessor busy computing. For the case of ECC160p, $2\lceil(160 + 4)/d\rceil \leq 96$ or $d \geq 4$ is the corresponding case. For the case of RSA1024,

$2\lceil(1024+4)/d\rceil \leq 96$ or $d \geq 22$ is derived.

On the other hand, if $d < 4$ for ECC160p or $d < 22$ for RSA1024, we need to split one MALUn instruction into several CS and CP instructions because the MALUn instruction cannot be performed in 96 cycles. Therefore, we need to send two or more instructions for operating one MALUn operation. As the value d decreases, the number of instructions increases, which means that the performance degrades for small d.

4.2.2 Implementation Results

Based on the performance estimation with GEZEL, we implemented ECC160p on a Xilinx FPGA. The software C codes are compiled with Vision2 by Keil [42] with an Intel 8051AH. The coprocessor block was synthesized with Project Navigator by Xilinx and implemented on a Virtex-II PRO (XC2VP30).

The results of two different configurations are summarized in Table 4.3. The table contains the details of each processing step including the representation conversions. The use of CPRAM leads to a much better performance with a small software size. However, the coprocessor with CPRAM requires additional single-port SRAMs (Block RAMs in the Xilinx FPGAs). Considering the significant improvement of the performance, the extra hardware cost for CPRAM is worth paying. The average performance of ECC160p for point multiplication is about 130 ms including the initial and final data transfer from/to CPU and the representation conversions (MtoN and NtoM). The reason why the coordinate conversion of PtoA costs 21 ms is the modular inversion.

The result of using CPRAM is about 40 times faster than the result of an optimized software implementation reported in [30] (5.6 s @12 MHz). Therefore, the introduced HW/SW co-design provides high performance with a flexiblity of SW. On the other hand, our result of using XRAM shows 5 times lower performance than [30]. This fact indicates that the misuse of HW/SW co-design even degrades the SW performance.

Seeing the results from a different perspective, the MALUn is activated for 0.14 % and 30 % of the total cycles if XRAM and CPRAM are used respectively. This observation implies that the computational efficiency of the MALUn is the key to achieving high performance.

4.3 HW/SW Co-design for Curve-based Cryptography on the ARM

4.3.1 Cryptosystem Architecture with the ARM

The coprocessor used in HW/SW co-design with ARM for curve-based cryptography over GF(2^m) is composed of the main controller, the datapath MALUb,

Table 4.3: ECC160p implementation results for different configurations in the case of $d = 4$.

Location of Intermediate Variables	**XRAM**	**CPRAM**
Total Latency [ms]	**27 042.2**	**129.8**
Scalar Multiplication (ECPM)	**25 898.8**	**96.3**
Point Addition (ECPA)	142.3	0.5
Point Doubling (ECPD)	90.7	0.3
Representation Conversion	**1143.4**	**33.5**
Normal to Montgomery (NtoM)	17.7	9.2
Montgomery to Normal (MtoN)	7.0	3.4
Affine to Projective (AtoP)	21.3	0.1
Projective to Affine (PtoA)	1097.3	20.8
System Cost		
8051 Resource [Bytes]		
XRAM	372	211
ROM	4785	2867
FPGA Mapping of Coprocessor		
Number of 4-input LUTs	6023	6666
Number of slice F/Fs	1188	1195
Number of Block RAMs	0	6

the coprocessor memory and the μ-coded RAM. The block diagram of the cryptosystem is illustrated in Fig. 4.5. The configuration of the coprocessor is flexible to provide from the smallest to the fastest implementation depending on a target application. Some components can be added or removed as will be explained next.

The main CPU communicates with the coprocessor through memory-mapped I/O (*e.g.* an SRAM interface) and has three types of 32-bit in- and outputs; one of them is a signal that tells the controller to stop sending instructions when the instruction buffer is full. A 32-bit input/output passes data back and forward between the main CPU and the coprocessor and a 32-bit output is used to send instructions. The data transfer between the main CPU and the coprocessor is controlled by a Data Bus Controller (DBC). When using SRAM attached to the main CPU for storing intermediate variables for HECC/ECC operations, the coprocessor can be constructed without use of the coprocessor memory. Alternatively, for the purpose of reducing the I/O transfer overhead, the data memory can be embedded in the coprocessor. In this case, the path through the DBC is only activated when an initial point and the parameters of an elliptic curve are sent to the RAM, or when the result is retrieved.

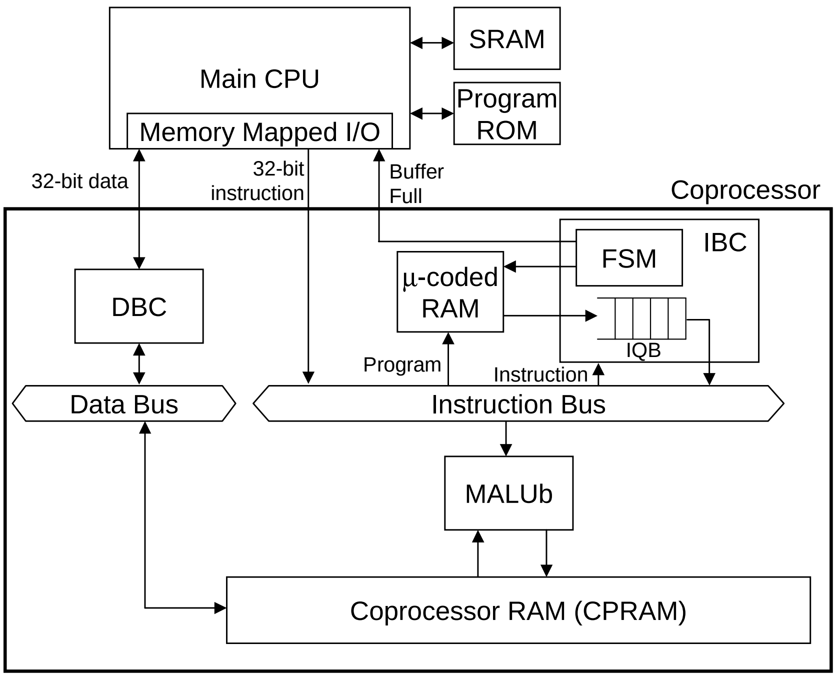

Figure 4.5: Block diagram for the system architecture with the ARM and a coprocessor.

Instructions are sent to the MALUb either from the main CPU or from pre-set micro codes in the μ-coded RAM. When the main CPU is in charge of dispatching instructions, the μ-coded RAM can be detached from the coprocessor. However, in this case, it occurs that the throughput of issuing instructions is not high enough for the MALUb to be utilized effectively. On the contrary, when the μ-coded RAM is used for assisting the main CPU, the Instruction Bus Controller (IBC) can handle even one instruction per cycle. For instance, the sequence of point doubling instructions is stored in the μ-coded RAM and the main CPU calls it as one instruction. Thus the MALUb can be activated in parallel without any instruction stalls. During point multiplication, the IBC keeps on reading instructions from the μ-coded RAM and stores them to an Instruction Queue Buffer (IQB) unless the IQB is full.

4.3.2 The MALU Instruction for GF(2^m) Coprocessor

Here, an instruction called MALUb is defined. It is worth mentioning that this is the only instruction that operates on the datapath in this case study.

$$\text{MALUb}(A, B, C, D) = (A(x) \cdot (B(x) + D(x)) + C(x)) \bmod P(x)\,. \tag{4.1}$$

When using the $(A(x) \cdot B(x) + C(x)) \bmod P(x)$ operation, one can ignore $D(x)$ as $D(x) = 0$. The whole procedure to execute MALUb starts from an instruction fetch and decode. Then, variables for $A(x), B(x), C(x)$ and $D(x)$ are loaded via CPRAM for the succeeding execution stage. The result is written back to the coprocessor memory at the last step.

4.3.3 Instruction Set for Coprocessor

Table 4.4 shows some of the primary instructions for the coprocessor from the ARM. The input registers of the MALUb are set via data-bus ports. As we use a 32-bit ARM, setting a register whose address is `src1` requires three `STORE(@dst)` instructions for HECC over GF(2^{83}). After all operands are set in corresponding registers, an instruction, `MALUb(@dst,@src1-4)` is executed. When using the configuration of using the μ-coded RAM, it is possible to define an instruction that consists of a series of `MALUb(@dst,@src1-4)` instructions (*e.g.* `PDBL_GF2m()`). In this thesis, point/divisor operations are all composed of the MALUb instructions only.

4.3.4 Implementation Results for Different Coprocessor Configurations

The system configurations are explored in two steps. First, in order to make the best use of the coprocessor, four different coprocessor configurations are selected

Table 4.4: Primary instructions for the ARM coprocessor.

INSTRUCTION	DESCRIPTION	OPERATION
`STORE(@dst)`	Data storing to the coprocessor	`R@dst <= din;`
`LOAD(@src)`	Data loading from the coprocessor	`dout <= R@src;`
`MALUb(@dst,@src1-4)`	Operate MALUb	`R@dst <= MALUb(R@src1-4)`
`PDBL_GF2m()`	Point doubling over GF(2^m)	`P <= 2P`
`PADD_GF2m()`	Point addition over GF(2^m)	`P <= P+Q`
`DDBL_GF2m()`	Divisor doubling over GF(2^m)	`D <= 2D`
`DADD_GF2m()`	Divisor addition over GF(2^m)	`D <= D+E`

Table 4.5: Coprocessor configurations for the vertical exploration.

	MALUb	μ -coded RAM	CPRAM
TYPE I	✓		
TYPE II	✓	✓	
TYPE II	✓		✓
TYPE IV	✓	✓	✓

as listed in Table 4.5. This is the so-called vertical exploration of the HW/SW co-design. Secondly, the further performance improvement can be made by using multiple MALUb cores. Thus the coprocessor can also be investigated from a parallel processing point of view. The so called horizontal exploration will be discussed in Chapter 5.

Figure 4.6 compares the performance of divisor multiplication of HECC over GF(2^{83}) for different coprocessor configurations. For the case of TYPE I and II, the I/O transfer overhead between the ARM and the coprocessor takes up the majority of the cycles (about 98%). The reason for this is that the intermediate variables are stored in SRAM attached to the main CPU and travel through the CPU to the coprocessor for processing. As for TYPE III, the I/O transfer overhead is reduced significantly due to the effect of the data memory allocated in the coprocessor. However, the I/O overhead is still dominant because the main CPU issues instructions via the slow communication channel. The parallel processing feature is hence useless to obtain further improvement of the performance in such system settings. Note that the ratio of the I/O transfer overhead is reduced ostensibly by introducing smaller d since the datapath consumes more clock cycles for one MALUb instruction. In this way, it is important to find the best digit size, d that can hide the I/O transfer overhead with TYPE III. This thesis, however, investigates further on TYPE IV for high-performance implementations by exploring parallelism, because the speed of dispatching instructions in TYPE IV assures the highest parallelism regardless the value of d. This will be discussed in detail in Chapter 5.

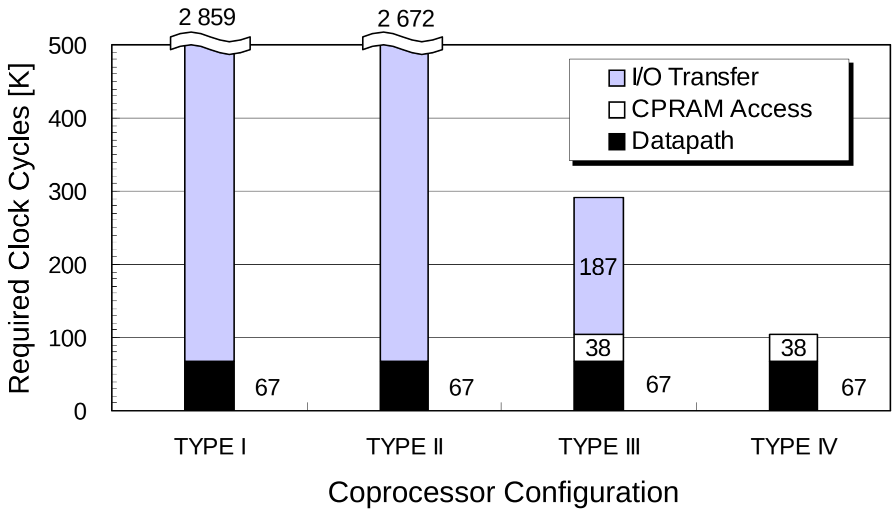

Figure 4.6: Required clock cycles of HEC divisor multiplication for different coprocessor configuration ($d = 12$).

4.4 Conclusions

In this chapter, two HW/SW co-designs are discussed to explore the best coprocessor configuration. In the design with the 8051, two different coprocessor configurations are implemented on an FPGA. The results show that the performance improves significantly by allocating CPRAM in the coprocessor. Without CPRAM, the performance becomes even worse than the result of a software implementation. This is mainly because of the slow data transfers between the 8051 and the coprocessor via the 8-bit ports. Furthermore, the best digit size d in the MALUb is explored for RSA1024 and ECC160p. As a result, we figure out that the speed of dispatching the instructions from the 8051 determines the best digit size for a configuration without interrupt and register poll.

In the second case study with the ARM, a much faster 32-bit SRAM interface is used for the HW/SW communication. However, we also see a huge I/O overhead caused by data transfers between the ARM and the coprocessor. More precisely, the MALUb is activated for 2% of the total cycles in the case of TYPE I and II (*i.e.* the configuration without CPRAM in the coprocessor). Furthermore, the μ-coded RAM is used for assisting the ARM to dispatch the coprocessor instructions. As a result, almost no I/O overheads are observed and the datapath is active during more than 60% of the cycles.

As can be seen from the result of TYPE IV, the number of clock cycles for accessing CPRAM in the coprocessor needs to be reduced for further speed-ups. This kind of optimization is important especially in multi-core systems where several data processing elements are used in parallel. The details will be discussed in the following chapter.

Chapter 5

High-speed Public-key Cryptoprocessor

5.1 Introduction

This chapter presents two high-speed public-key cryptoprocessors. First, we discuss a reconfigurable curve-based cryptoprocessor that accelerates point multiplication of ECC over GF(2^m) and divisor multiplication of HECC of genus 2 over GF(2^m). By allocating α copies of processing cores (MALUb cores), the point/divisor multiplication of ECC/HECC can be accelerated by exploiting Instruction-Level Parallelism (ILP). The MALUb cores are reconfigurable and can support a wider field size. The supported field size can be arbitrary up to $\alpha(m+1)-1$. The superscaling feature is facilitated by defining a single instruction that can be used for all field operations and point/divisor operations. In addition, the cryptoprocessor is fully programmable and it can handle various curve parameters and arbitrary irreducible polynomials. The cost, performance and security trade-offs are thoroughly discussed for different hardware configurations and software programs.

Secondly, we present a high-speed public-key cryptoprocessor that exploits three levels parallelism in ECC over GF(2^{163}). The first level parallelism is in the datapath of the MALUb, the second is in the modular operations performed in the P-MALUb and the third is in the instructions determined by the computational sequence of EC point multiplication. The proposed cryptoprocessor employs a P-MALUb (see Sect. 3.3.4) that exploits another parallelism in the datapath (MALUb) for accelerating modular operations. Based on the first reconfigurable cryptoprocessor, the sequence of point multiplication is also accelerated by exploiting ILP and processing multiple instructions in parallel. The system is programmable and hence independent of the type of the elliptic curves and point multiplication algorithms.

5.1.1 Related Work

This section lists some relevant previous work. As already mentioned, there is a considerable amount of research performed on hardware implementations on FPGAs as well as ASIC implementations for ECC since Agnew *et al.* [1] reported in 1989 the first result for performing elliptic curve operations on hardware. The work can be classified into the following categories. First, mathematical investigations have been reported for various types of elliptic curves, *e.g.* Koblitz curves [44]. Second, various algorithms for point multiplication have been proposed and criteria for improvements include performance as well as side-channel security. One of the best-known examples that meet both requirements is the Montgomery powering ladder [63]. Various types of coordinates have been prepared and also a number of approaches to speed up finite field arithmetic. Lastly, architecture-level improvements can be considered from a hardware implementations' point of view.

The majority of previous hardware ECC implementations have been over binary fields. Gao *et al.* proposed an elliptic curve cryptosystem coprocessor with variable key sizes, which utilizes the internal SRAM/registers in an FPGA in [25]. In 2000 Orlando and Paar proposed a scalable elliptic curve processor architecture which operates over binary finite fields in [68]. Eberle, Gura and Shantz [19] have introduced a programmable hardware accelerator for ECC over GF(2^n), which can handle arbitrary field sizes up to 255. Satoh and Takano [81] present a dual field multiplier with high performance in both binary and prime fields. The throughput of an elliptic curve point multiplication is maximized by using a Montgomery modular multiplier and an on-the-fly redundant binary converter. The biggest advantages of their design are the flexibility in the operand size and scalability in trade-off between speed and hardware. The work by Tenca and Koç [95] also introduces a scalable architecture for the computation of modular multiplication, based on a Montgomery multiplier. Their proposed multiplier works with any precision of the input operands, limited only by memory or control constraints. Harris *et al.* [15] improve the version of the Tenca-Koç Montgomery multiplier and achieve half the latency and half the queue memory requirement. Recently significantly faster implementations are proposed in [80, 91].

The first contribution of this chapter is in the acceleration of curve-based cryptosystems by deploying a superscalar architecture. The solution is algorithm-independent and can be applied to any point multiplication algorithm. We discuss the improvement of the performance for ECC and HECC over binary fields. Some previous work reported parallel use of modular arithmetic units to accelerate point multiplication [32, 38, 40, 61, 88, 103]. In these papers, point/divisor doubling and addition operations are reformulated so that they can take advantage of the parallel processing. On the other hand, our proposed architecture embeds an instruction scheduler that explores the highest level of parallelism and assigns tasks for the processing units in an optimal way. In this way the parallelism within the operations can be found *on-the-fly* by *dynamically* checking the data dependencies in the instructions.

The second contribution is the design of a reconfigurable datapath over binary fields. In order to support multiple curve-based cryptosystems and various field sizes for them, it is necessary to provide different field-length modular operations, *e.g.* 193 bits for ECC and 97 bits for HECC. In [81], Satoh and Takano solve this problem by using an r-bit $\times$ r-bit multiplier and by applying an algorithm originally used for software implementations. In other words, the advantage of their solution is that the operand size can be freely chosen, limited only by the size of memory for storing intermediate variables. The area and time complexities of modular multiplications are respectively $O(r^2)$ and $O(m^2)$, where $n = m \cdot r$ is the field size. For high-speed modular multiplications, the parameter r needs to be large and the datapath delay of the multiplier becomes longer. In contrast, the critical path delay is independent of the field size in our solution where an m-bit$\times$$d$-bit digit-parallel multiplier is used. Furthermore, the organization of the multiplier can be reconfigured by changing the interconnections between processing cores that have the MALUb$_{m\times d}$ and memory. The so-called coarse-grained reconfigurable datapath offers high-speed modular multiplications, efficient use of hardware resource for various field sizes and support for arbitrary field sizes by providing enough MALUb cores.

The third contribution discussed in this chapter is a fair comparison between ECC and HECC. For HECC of genus 2 the field size is two times smaller than the one for ECC for the same level of security [8]. Our programmable architecture enables one to use the same hardware design for the two curve-based cryptosystems. Moreover, we also explore different arithmetic operations in the MALUb and examine the effects on the level of the parallelism in ECC and HECC. As a result, we discuss the trade-offs between cost, performance and security-level of curve-based cryptography. As the last contribution, based on the proposed reconfigurable cryptoprocessor, we explore the fastest possible performance of ECC by using a fixed irreducible polynomial, $x^{163} + x^7 + x^6 + x^3 + 1$ and by applying a fast point multiplication algorithm.

5.2 High-speed Curve-based Cryptoprocessor with Reconfigurable Datapath

Here, we extend the basic algorithms for ECC and HECC introduced in Chapters 1 and 2, and discuss the best architecture of a high-speed curve-based cryptography.

Again, the main operation in any curve-based primitive is point multiplication. The general hierarchical structure for operations required for implementations of curve-based cryptography is given in Fig. 5.1a. One can perform modular inversion

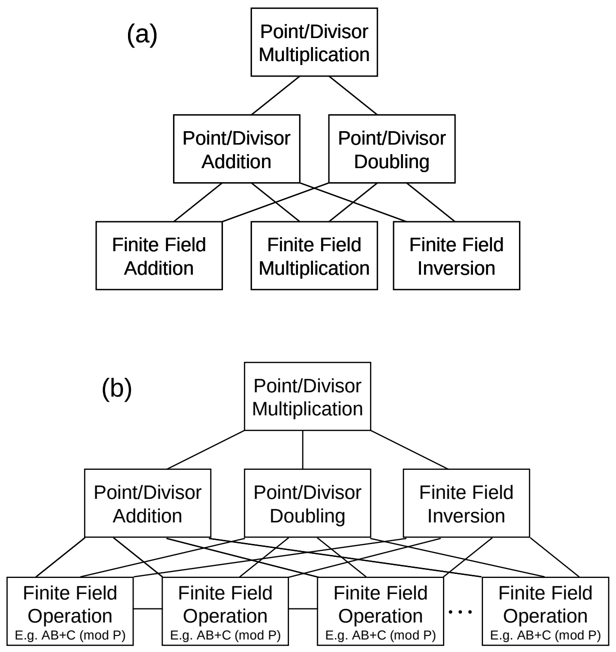
(a)
Point/Divisor Multiplication
Point/Divisor Addition
Point/Divisor Doubling
Finite Field Addition
Finite Field Multiplication
Finite Field Inversion
(b)
Point/Divisor Multiplication
Point/Divisor Addition
Point/Divisor Doubling
Finite Field Inversion
Finite Field Operation E.g. AB+C (mod P)
Finite Field Operation E.g. AB+C (mod P)
Finite Field Operation E.g. AB+C (mod P)
...
Finite Field Operation E.g. AB+C (mod P)

Figure 5.1: Scheme of the hierarchy for ECC/HECC operations. (a) Conventional hierarchy. (b) Proposed hierarchy using a single finite field operation at the lowest level.

also with a chain of multiplications as discussed in Sect. 2.2.2. and only provide hardware for finite field addition and multiplication. The corresponding hierarchy is illustrated in Fig. 5.1b. The hierarchy uses several copies of operation units at the lowest level to accelerate point/divisor group operations and inversions by parallel computation. We use this structure for our cryptoprocessor.

5.2.1 System Architecture

The proposed architecture of the curve-based cryptoprocessor is composed of the main controller, several MALUb cores and the Register Files (RFs) that store intermediate variables and share them with the MALUb cores. The block diagram of the cryptoprocessor is illustrated in Fig. 5.2. The hardware configuration of the cryptoprocessor is flexible to provide from the smallest to the fastest implementation depending on the target application. Some components can be added or removed as will be explained in the next sections.

The main CPU communicates with the cryptoprocessor via memory-mapped

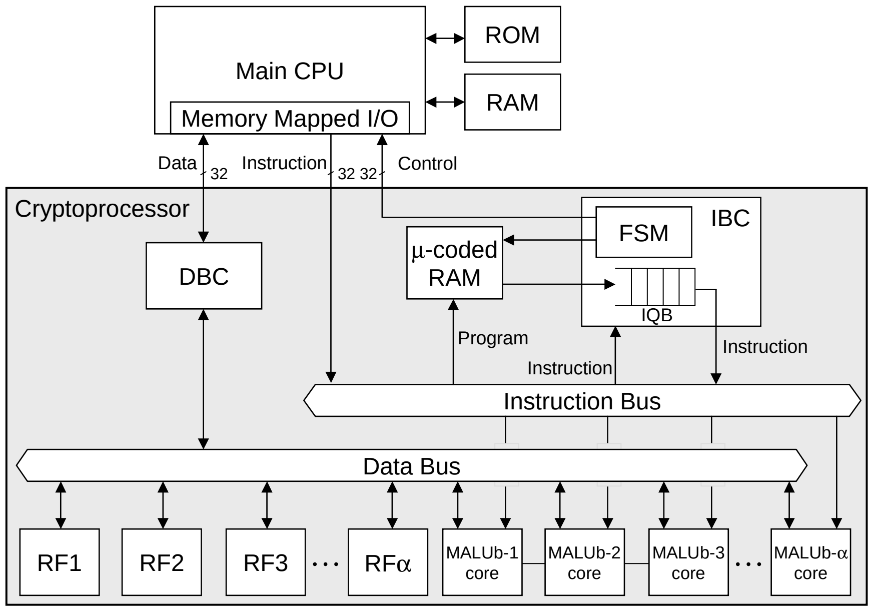

Figure 5.2: Block diagram for the system architecture with the curve-based cryptoprocessor.

I/O (*e.g.* an SRAM interface) and has three types of 32-bit inputs and outputs: one of them is a signal that tells the controller to stop sending instructions when the instruction buffer is full. A 32-bit input/output passes data back and forward between the main CPU and the cryptoprocessor and a 32-bit output is used to send instructions. The data transfer between the main CPU and the cryptoprocessor is controlled by a Data Bus Controller (DBC). If the intermediate variables for ECC/HECC operations are stored in SRAM attached to the main CPU, the cryptoprocessor can be constructed without RFs. However, the I/O transfer overhead becomes the bottleneck of the performance. Hence, the RFs have to be embedded in the cryptoprocessor for the purpose of reducing the data transfer overhead. In this way, the path through the DBC is only activated when an initial point and the curve parameters are sent to the RFs, or when the result of a point multiplication is retrieved.

Instructions are sent to the MALUb cores either from the main CPU or from pre-set micro codes in the μ-coded RAM. When the main CPU is in charge of dispatching instructions, the Instruction Bus Controller (IBC) block can be detached from the cryptoprocessor. In this case, typically the throughput of issuing instructions is not high enough for the MALUb cores to be utilized effectively. However, if the μ-coded RAM is used for assisting the main CPU, the IBC can handle one instruction per cycle. For instance, the sequence of point doubling is stored in the μ-coded RAM and the main CPU calls it as a single instruction. Thus, multiple MALUb cores can be activated in parallel without any instruction stalls. During point/divisor multiplications, the IBC keeps on reading instructions from the μ-coded RAM and stores them in an Instruction Queue Buffer (IQB) unless the IQB is full. The IBC checks if there is instruction-level parallelism (ILP) by checking the data-dependency of instructions in the IQB and forwards them to the MALUb core(s) (see Sect. 5.2.2).

Architecture of the datapath

We use the reconfigurable datapath discussed in Sect .3.3.2. Therefore, the MALUb cores can be reconfigurable to support different field sizes and high performance by executing multiple MALUb instructions in parallel.

Architecture of the RF

If the MALUb supports the operation $(A(x)(B(x) + D(x)) + C(x)) \bmod P(x)$ or $A(B + D) + C$ for short, four different operands need to be read from the RF and the result is written back to the RF after completing the execution. When using three MALUb cores for instance, twelve read and three write operations occur for a three-way parallel execution. This heavy memory-access is one of the bottlenecks in multi-core systems. In order to reduce the memory-access cycles especially in read operations, a multi-port RF is implemented as illustrated in Fig. 5.3a.

The multi-port RF supports four simultaneous read operations at four different addresses per cycle (4R). This allows one to read all necessary operands for the operation form, $A(B + D) + C$ in a single cycle. In this way, the number of the read-access cycles can be reduced by 3/4 or 75%. The read cycle is reduced to only three cycles for a three-way parallel execution. The write operation from three MALUb cores can be done in three cycle (1W).

Note that one RF can be shared with multiple MALUb cores. Only when supporting a wider field size, multiple RFs should be allocated in the cryptoprocessor. In addition, the required number of entries in the RF differs from ECC to HECC in that ECC needs 16 registers while HECC uses 32 registers as mentioned previously. This difference can be a problem when the cryptoprocessor needs to support both cryptosystems. A simple solution is to prepare 32 entries in each RF (denoted as $\mathrm{RF}_{m\times 32}$ in Fig. 5.3a). Another solution is to make one 32-entry RF from two 16-entry RFs as shown in Fig. 5.3b. The figure illustrates how an $\mathrm{RF}_{m\times 32}$ can be configured with $\mathrm{RF1}_{m\times 16}$ and $\mathrm{RF2}_{m\times 16}$. As will be investigated in detail in Sect. 5.2.5, both solutions have drawbacks and advantages.

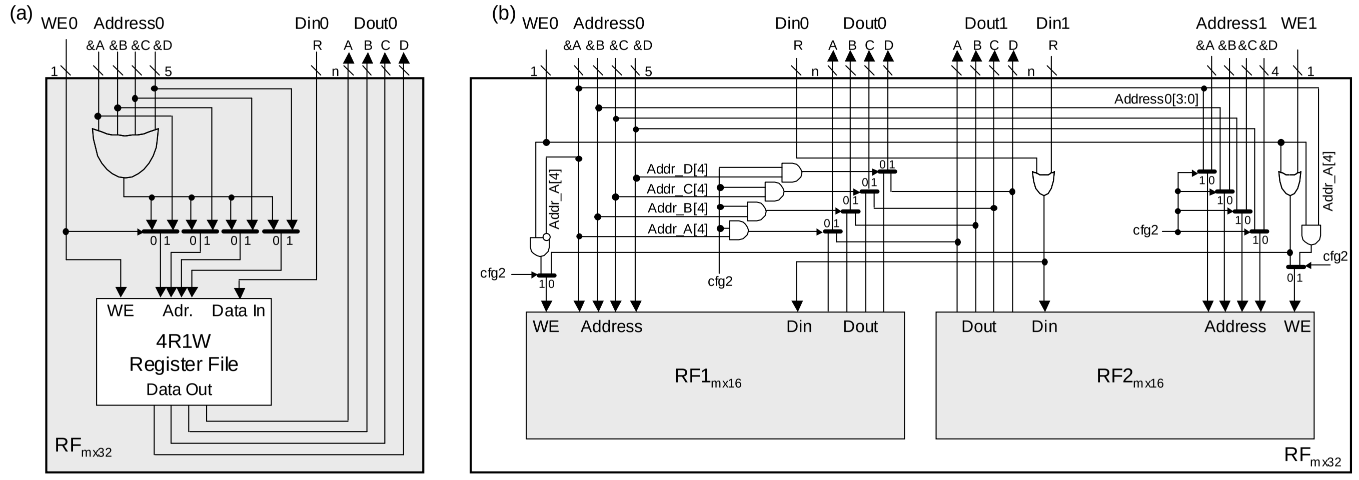

Figure 5.3: Hardware architecture of the register files. (a) Four-ported register file supporting 4-read and 1-write (4R1W) to support simultaneous read of four operands. 32-entry m-bit Register File ($\mathrm{RF}_{m\times32}$). (b) A pair of $\mathrm{RF}_{m\times16}$ can be configured as $\mathrm{RF}_{m\times32}$ by setting the signal $\mathit{cfg2} = 1$.

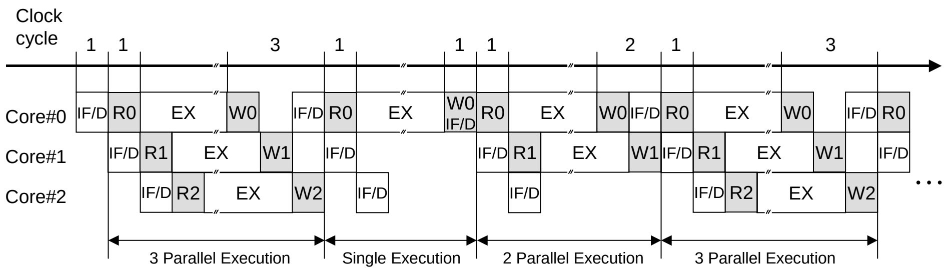

Figure 5.4: Example of parallel issue of instructions for three MALUb instructions. (IF/D: Instruction Fetch/Decode, EX: Execution of MALUb, R/W: Read/Write from/to the RF). The consecutive write cycles depend on the number of instructions issued in parallel. The execution cycle is determined by the configuration of MALUb cores.

5.2.2 Dynamic Scheduling for Multi-core Architecture

ILP is exploited for all `MALUb()` instructions as long as two or more instructions are buffered in the IQB. Here, the strategy to find ILP is introduced.

We now design a new instruction called `MALUb()`. It is worth mentioning again that this is the only instruction that operates on the datapath.

$$\begin{aligned}&\texttt{MALUb(\&R,\&A,\&B,\&C,\&D) :}\\&R(x) = (A(x)\cdot(B(x)+D(x))+C(x)) \bmod P(x)\,.\end{aligned} \tag{5.1}$$

Here, `&A, &B, &C, &D` denote the addresses for four inputs of the instruction and `&R` denotes the address where the result is stored. As illustrated in Fig. 5.4, the whole procedure to execute `MALUb()` starts from an instruction fetch and decode (IF/D). Then, the variables for $A(x), B(x), C(x)$ and $D(x)$ are loaded via the RF (R) for the succeeding execution stage. The result is stored to the RF (W) at the last step. When performing parallel processing, the write operations from every MALUb core should be sequential in order to avoid memory-write conflicts. More precisely, in order to keep data integrity between the RFs, only one data item can be written to the RFs within a cycle; this is a consequence of the way in which the 4R1W RF is embedded in our cryptoprocessor. From another viewpoint, the operands for different MALUb cores can be also read sequentially, which means that one RF can be shared with multiple MALUb cores.

The instruction has four source operands and outputs the result to the RF, *i.e.* `MALUb(&R,&A,&B,&C,&D)` deals with five types of addresses for the operation $(A(x)(B(x)+D(x))+C(x)) \bmod P(x)$. They are expressed as

$$\texttt{MALUb : \&R = \&A, \&B, \&C, \&D}\,. \tag{5.2}$$

The `MALUb` instruction also refers to $P(x)$ that is stored in the RF. To include out-of-order executions, the following two types of dependencies need to be checked between two instructions, $\texttt{MALUb}^i$ and $\texttt{MALUb}^j$ (i and j are labels indicating the order of the instruction in the IQB). For all i and j that satisfy $0 \leq i < j < \text{ILP}_D$, where ILP_D is the size of the instruction window to exploit ILP, one can determine the number of instructions to be issued in parallel, by checking the following two dependencies.

Read-After-Write (RAW) Dependency check for in-order execution ($\texttt{\&R}^i = \texttt{\&A}^j$ or $\texttt{\&R}^i = \texttt{\&B}^j$ or $\texttt{\&R}^i = \texttt{\&C}^j$ or $\texttt{\&R}^i = \texttt{\&D}^j$): If the result of the instruction $\texttt{MALUb}^i$, $\texttt{R}^i$ is used as input of the instruction $\texttt{MALUb}^j$, $\texttt{MALUb}^j$ cannot be issued before $\texttt{MALUb}^i$ completes its operation. In other words, if the condition above in parenthesis is false, $\texttt{MALUb}^j$ can be issued with $\texttt{MALUb}^i$.

However, when the parallel issue of $\texttt{MALUb}^i$ and $\texttt{MALUb}^j$ includes out-of-order execution, the next condition has to be verified as well.

RAW Dependency check for out-of-order execution ($\texttt{\&R}^j = \texttt{\&A}^i$ or $\texttt{\&R}^j = \texttt{\&B}^i$ or $\texttt{\&R}^j = \texttt{\&C}^i$ or $\texttt{\&R}^j = \texttt{\&D}^i$): As a result of checking the conditions for in-order execution, it is possible that the instruction $\texttt{MALUb}^j$ can be issued while some preceding instructions cannot. In this case, we need to check if the result of the instruction $\texttt{MALUb}^j$, $\texttt{R}^j$ is used for the input of the preceding instructions that cannot be issued. The corresponding condition is described above in parenthesis.

The proposed architecture needs no check for Write-After-Read and Write-After-Write dependencies in contrast to a general superscalar machine. Indeed, the instruction `MALUb()` is a fixed-length multi-cycle instruction, and hence we can skip those dependencies in checking the sequence of point/divisor operations.

As the 0th instruction $\texttt{MALUb}^0$ is issued unconditionally, the number of conditions to check ILP becomes $4(\text{ILP}_D - 1)^2$. This fact indicates that the hardware complexity for ILP grows quadratically in a large ILP_D, but in exchange further parallelism can be exploited. The choice of the ILP_D is discussed in Sect. 5.2.4.

Instruction Set for the Cryptoprocessor

Table 5.1 shows some of the primary instructions for the cryptoprocessor. For a 32-bit CPU such as the ARM, storing data to the address, `dst` requires four `STORE()` instructions for HECC over GF(2^{97}). After all operands are set at the corresponding addresses of the RF, the main CPU sends the instructions `MALUb()`. By using the μ-coded RAM in the cryptoprocessor, it is possible to define an instruction that consists of a series of `MALUb()` instructions. In our design, all necessary point/divisor operations are preprogrammed in the μ-coded RAM, and the main CPU uses these instructions (*e.g.* `ECC_PA()` and `ECC_PD()`) for point multiplication.

The number of instructions that can be issued in parallel decides consecutive write cycles. In total, an l-way parallel execution takes $l + 1$ cycles in addition to

Table 5.1: Primary instructions for the proposed cryptoprocessor and the computational costs in the case of the operation form $A(B+D)+C$. Here $w(k)$ denotes the Hamming weight of the positive integer k.

INSTRUCTION	OPERATION	COMPUTATIONAL COST
`STORE(data,num,@dst)`	Storing `data` in the RF`#num`	-
`LOAD(num,@src)`	Loading `data` from the RF`#num`	-
`CFG(data,num)`	Setting `cfg*` for the MALUb core`#num`	-
`MALUb(&R,&A,&B,&C,&D)`	MALUb: $R = A(B+D)+C$	$\lceil n/d \rceil$ cycles
`PADD()`	Point Addition for ECC	14 `MALUb()` instructions
`PDBL()`	Point Doubling for ECC	10 `MALUb()` instructions
`PADD_M()`	Point Addition for ECC_M	6 `MALUb()` instructions
`PDBL_M()`	Point Doubling for ECC_M	6 `MALUb()` instructions
`DADD()`	Divisor Addition for HECC	46 `MALUb()` instructions
`DDBL()`	Divisor Doubling for HECC	33 `MALUb()` instructions
`INV()`	Modular Inverse (by Itoh-Tsujii [36])	$\{\lfloor \log_2(n-1) \rfloor + w(n-1) - 1$ $+(n-1)\}$ `MALUb()` instructions

the execution cycles that depend on the MALUb configuration.

5.2.3 Algorithms for the Implementations

In our implementations, point multiplication of ECC is achieved by two different computational sequences: the first is from the recommendation of IEEE P1363 (see Alg. 2.12) and the second is based on the idea of the Montgomery powering ladder of López and Dahab (denoted as ECC_M in this thesis) [53]. The sequence of the Montgomery powering ladder will be explained in Sect. 5.3.2. With regard to divisor multiplication of HECC, we use the formulae as shown in Alg. 2.15. All of the sequences use projective coordinates, and we apply the binary NAF or the windowed NAF method for point multiplication [8, 31] except ECC_M. In this way the scalar is decomposed as a NAF and point multiplication is performed with a lower cost than the binary method. Modular inversion is performed with a chain of modular multiplications repeatedly [36]. The total number of modular multiplications required for the modular inverse is $\{\lfloor \log_2(n-1) \rfloor + w(n-1) - 1 + (n-1)\}$. Since we need only one modular inversion for each point multiplication of ECC and HECC, the inversion cost is not a serious bottleneck. The details will be discussed in Sect. 5.2.5.

As our datapath performs one basic operation, $AB+C$ or $A(B+D)+C$ over a binary field, we have rewritten the sequences of point/divisor doubling and addition to obtain an optimal usage of our new datapath. For example, the formulae for the mixed-addition of HECC includes 48 operations of $A(B+D)+C$ instead of 6 squarings, 34 multiplications and a lot of additions. Note that our strategy for refining the sequences also minimizes the number of intermediate variables to save hardware resource. As a result, point multiplication can be performed with at most 16 and 32 registers, respectively for ECC (including ECC_M) and HECC.

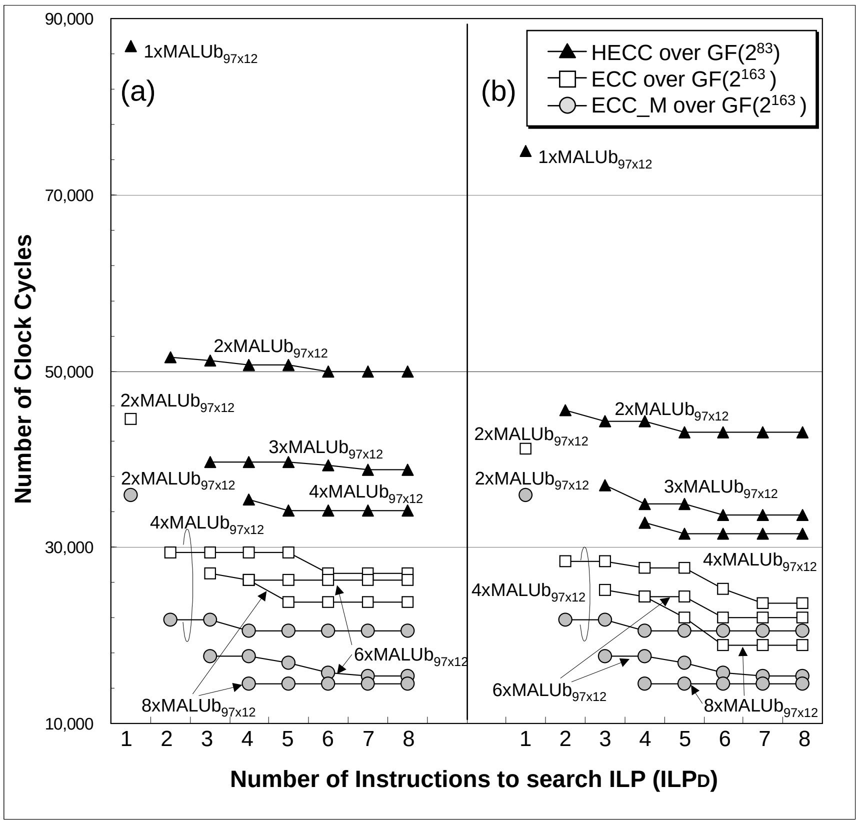

Figure 5.5: The number of clock cycles for point multiplication of ECC163, ECC_M163 and HECC83 for different ILP$_D$ when allocating 1 to 8 MALUb$_{97\times12}$. (a) Operation form is $AB + C$. (b) Operation form is $A(B + D) + C$.

5.2.4 Performance Improvement by Exploiting ILP

As the performance of the superscalar architecture is dependent on the degree of parallelism, it is also important to determine ILP$_D$, an appropriate number of instructions to search for ILP, as well as the number of MALUb cores. Figure 5.5 shows the number of clock cycles for point/divisor multiplication when setting ILP$_D$ from 1 to 8.

Up to eight copies of the MALUb cores with MALUb$_{97\times12}$ are instantiated in the cryptoprocessor to evaluate the performance improvement by the superscaling feature for ECC163, ECC_M163 and HECC83. Here, we assume that enough RFs are allocated in the cryptoprocessor, *i.e.* RF$_{97\times32}$ is assigned for each MALUb core with MALUb$_{97\times12}$ so that the cryptosystem has no limitations on the supported field sizes and the type of cryptosystem.

As a result of the GEZEL system-level simulation, we can see the effectiveness of the operation form of $A(B+D)+C$. We also observe that for both operation forms the overall performance improves as the number of MALUb$_{97\times12}$ increases. For instance, the performance of ECC_M163 can be improved by a factor of 2.5 by using eight copies of MALUb$_{97\times12}$. Also a large ILP$_D$ helps to exploit more parallelism and leads to higher performance. However, the degree of improvement is not the same as performance improvement by increasing the number of MALUb$_{97\times12}$. The results of using this operation with ILP$_D$ = 6 are also summarized in Table 5.2.

In order to investigate the performance bottleneck of ECC and HECC, the number of clock cycles for a point multiplication is split by the degree of parallelism. Figure 5.6 shows the results for ECC163, ECC_M163 and HECC83 by changing the number of MALUb cores and the type of the operation. Note that the computational sequence for ECC_M163 is the same regardless of the operation type, therefore we use the operation type $AB+C$ for ECC_M163.

For example, point multiplication of ECC_M163 can be performed in 36 Kcycles. By allocating two copies MALUb$_{97\times12}$ (denoted as 2·MALUb in Fig. 5.6), the number of clock cycles is reduced to 21 Kcycles. In this case, all operations in the ECC_M sequence are performed with two-way parallelism. The cases of using three and four copies MALUb$_{97\times12}$ can reduce the clock cycles to 16 and 15 Kcycles, respectively. However, the MALUb cores are not fully utilized in those cases because of the limitation of parallelism inherent in the sequence of ECC_M.

We consider the utilization of the MALUb cores in order to know if the datapaths are effectively used in parallel. If a parallel execution utilizes all datapaths, the utilization is defined as 100%. On the other hand, the utilization is 50% when half of the datapaths are used, *e.g.* a two-way computation is executed on a configuration with four datapaths. Note that the cycles for memory accesses are excluded for computing the utilization. We compute the average of the utilization for ECC and HECC with the operation type $AB+C$ and $A(B+D)+C$. The formula is defined as

$$\frac{\sum_{i=1}^{l_{max}} i \cdot R_i}{l_{max}\sum_{i=1}^{l_{max}} R_i} \quad (l_{max} = 1,2,3,4)\,, \tag{5.3}$$

where l_{max} is the maximum degree of parallelism under the given hardware configuration (*i.e.* the number of MALUb cores) and R_i is the number of clock cycles required for i-way computations. As can be seen from Fig. 5.6, the proposed superscalar feature can reduce the overall number of clock cycles. However, the utilization of the MALUb cores (marked on the bars in percent figures) decreases as the value l_{max} increases. This fact indicates that area and performance trade-offs

Table 5.2: The number of clock cycles for a point multiplication with $d = 12$, ILP$_D$ = 6 and the operation form $A(B+D)+C$ and $AB+C$ for ECC_M. The numbers in parenthesis are the speed-up ratio compared to the single-scalar configuration.

Cryptoprocessor Configuration	HECC 83	ECC 163	ECC_M 163	HECC 97	ECC 193	ECC_M 193	HECC 139	ECC 283	ECC_M 283	HECC 283	ECC 571	ECC_M 571
1·MALUb$_{97\times12}$	74 970 (1.00)	-	-	107 086 (1.00)	-	-	-	-	-	-	-	-
2·MALUb$_{97\times12}$	43 036 (1.74)	41 136 (1.00)	35 936 (1.00)	60 413 (1.77)	57 840 (1.00)	50 528 (1.00)	195 303 (1.00)	-	-	-	-	-
4·MALUb$_{97\times12}$	31 548 (2.38)	25 225 (1.63)	20 530 (1.75)	43 630 (2.45)	35 196 (1.64)	28 600 (1.77)	108 321 (1.80)	106 134 (1.00)	85 624 (1.00)	733 460 (1.00)	-	-
6·MALUb$_{97\times12}$	31 548 (2.38)	22 030 (1.87)	15 730 (2.28)	43 630 (2.45)	30 656 (1.89)	21 779 (2.32)	82 827 (2.36)	58 980 (1.80)	47 815 (1.79)	397 663 (1.84)	450 319 (1.00)	393 394 (1.00)
8·MALUb$_{97\times12}$	31 548 (2.38)	18 850 (2.18)	14 530 (2.47)	43 630 (2.45)	26 138 (2.21)	20 074 (2.52)	82 827 (2.36)	58 980 (1.80)	47 815 (1.79)	397 663 (1.84)	450 319 (1.00)	393 394 (1.00)

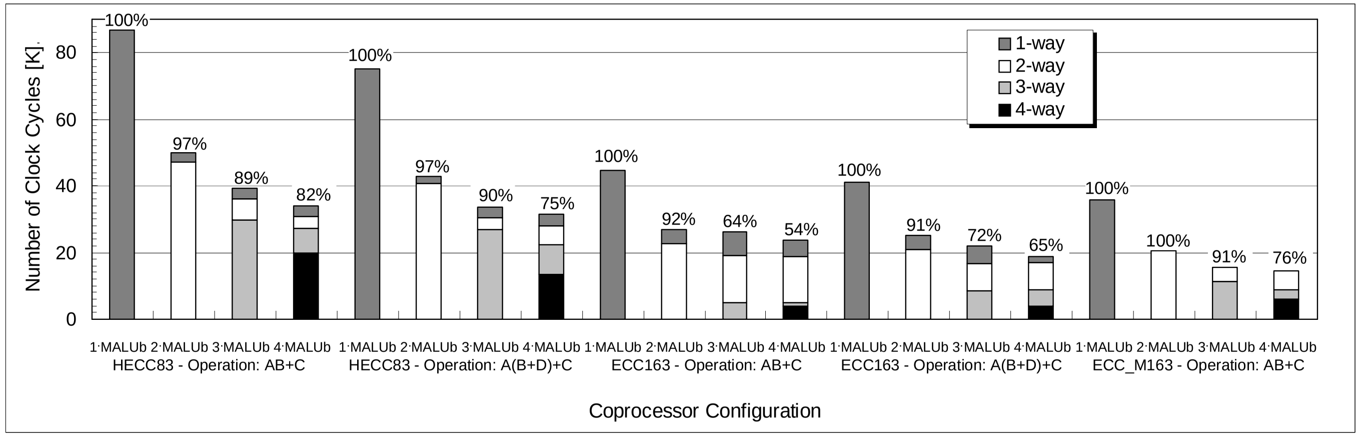

Figure 5.6: The profile graphs of the number of clock cycles in EC point multiplication and HEC divisor multiplication for different hardware settings of the cryptoprocessor ($d = 12$). The figures in percentage marked on the bars indicate the utilization of the MALUb cores.

Table 5.3: Configurations of the datapath with six cores, each of which has $\text{MALUb}_{97\times12}$.

Configuration	HECC over GF(2^m)			
	$m \leq 97$	$97 < m \leq 195$	$195 < m \leq 293$	$293 < m \leq 587$
I: $6 \cdot \text{MALUb}_{97\times12}$ $+6 \cdot \text{RF}_{97\times32}$	Supported (4-way)	Supported (3-way)	Supported (2-way)	Supported (1-way)
II: $6 \cdot \text{MALUb}_{97\times12}$ $+6 \cdot \text{RF}_{97\times16}$	Supported (3-way)	Supported (2-way)	Supported (1-way)	Not Supported

Configuration	ECC over GF(2^m)		
	$m \leq 195$	$195 < m \leq 293$	$293 < m \leq 587$
I: $6 \cdot \text{MALUb}_{97\times12}$ $+6 \cdot \text{RF}_{97\times32}$	Supported (3-way)	Supported (2-way)	Supported (1-way)
II: $6 \cdot \text{MALUb}_{97\times12}$ $+6 \cdot \text{RF}_{97\times16}$			

are getting worse for a larger l_{max}; it can be explained by inherent data dependencies in ECC and HECC. From this observation, we decide to employ $l_{max} \leq 3$ for ECC and $l_{max} \leq 4$ for HECC to maintain high utilization of the MALUb cores.

5.2.5 Implementation Results

The cryptoprocessor discussed in Sect. 5.2.1 has been synthesized with a 0.13-μm CMOS technology by using Synopsys Design Vision. From the performance evaluation discussed in Sect. 5.2.4, we allocate up to eight instantiations of the MALUb cores with $\text{MALUb}_{97\times12}$ with an operation, $A(B + D) + C$. For the size of the instruction window, $\text{ILP}_D = 6$ is selected. Trade-offs between cost and performance are discussed for ECC, ECC_M and HECC with different RF configurations. The synthesis results show that all designs meet with the timing constraint of 292 MHz.

A pair of $\text{MALUb}_{97\times12}$ can be reconfigured as one $\text{MALUb}_{195\times12}$ by changing the interconnection between the MALUb cores. In this way, both HECC97 and ECC193 can be supported by allocating $2{\cdot}\text{MALUb}_{97\times12}$ in the cryptoprocessor. As for the RF, we need to prepare either a pair of $\text{RF}_{97\times32}$ or a pair of $\text{RF}_{97\times16}$ to support the field lengths.

As explained previously, HECC requires $\text{RF}_{m\times32}$ while ECC can be computed with $\text{RF}_{m\times16}$ where m is the field size of ECC and HECC. Therefore, depending on the configuration of the RF, the supported field lengths and the degree of parallelism are determined differently as shown in Table 5.3. In other words, the configuration using $6{\cdot}\text{RF}_{97\times32}$ offers enough registers for both ECC and HECC, and hence a better degree of parallelism can be expected. Moreover, we can apply the NAF_4 (NAF with a width-4 window) for ECC by utilizing the 16 redundant entries in the RF. In contrast, $6{\cdot}\text{RF}_{97\times16}$ can be considered as an ECC-centric configuration, because it can save redundant registers when ECC is performed.

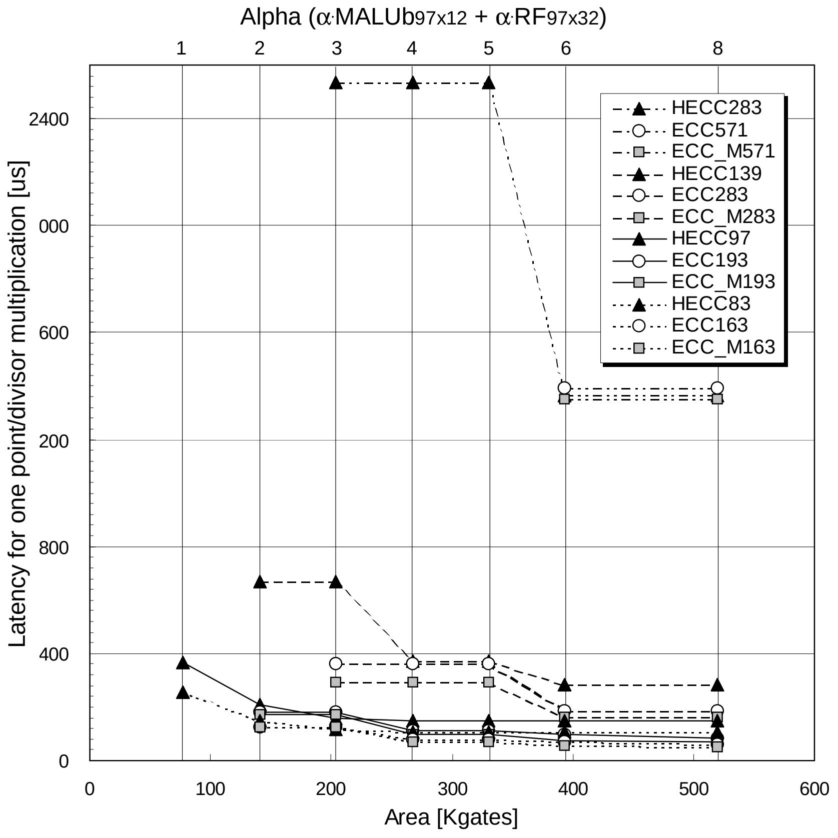

Figure 5.7: The latency of point/divisor multiplication for different variants of CONFIG-I, *i.e.* $\alpha\cdot$ MALUb$_{97\times12}$ + $\alpha\cdot$RF$_{97\times32}$.

The drawback of this configuration is that the degree of parallelism in HECC is restricted by the number of RF$_{97\times16}$, *i.e.* the cryptoprocessor can exploit at most three-way parallelism for HECC in this case.

Figs. 5.7 and 5.8 show the average time for an EC point multiplication and HEC divisor multiplication for the configuration of $\alpha\cdot$ MALUb$_{97\times12}$ + $\alpha\cdot$ RF$_{97\times32}$ where $\alpha = 1, 2, 3, 4, 5, 6, 8$ (CONFIG-I). Figs. 5.9 and 5.10 show the result for the configuration of $\alpha\cdot$ MALUb$_{97\times12}$ + $\alpha\cdot$ RF$_{97\times16}$ where $\alpha = 2, 3, 4, 5, 6, 8$ (CONFIG-II). Here, we use 1-KByte μ-coded RAM in order to support ECC and HECC.

As can be seen from the figures, the cost of supporting ECC571 with CONFIG-I is 393 Kgates, which is more expensive than for CONFIG-II whose gate size is

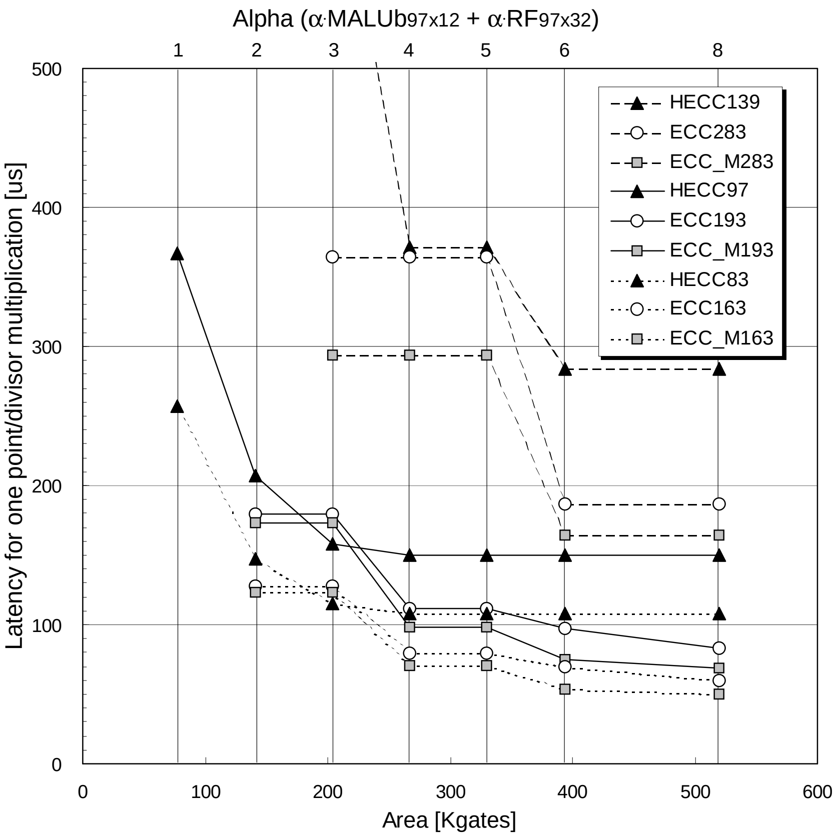

Figure 5.8: Magnified image of Fig. 5.7.

244 Kgates. However, CONFIG-II shows slightly lower performance for HECC. This performance degradation is more apparent for a larger field size, *e.g.* in the case of $\alpha = 6$, the performance of HECC139 decreases from 284 μs to 670 μs when changing the configuration.

The computational cost for modular inversion is summarized in Table 5.4 for CONFIG-I. The ratio of the inversion cost to the computational cost for point multiplication varies from 7 % to 18% in ECC and ECC_M. This is because we use the MALUb instructions for modular squarings in the Itoh-Tsujii algorithm. However, considering the flexibility of the proposed hardware architecture, the inversion cost can be regarded low enough. In the case of HECC, the cost for modular inversion is negligible.

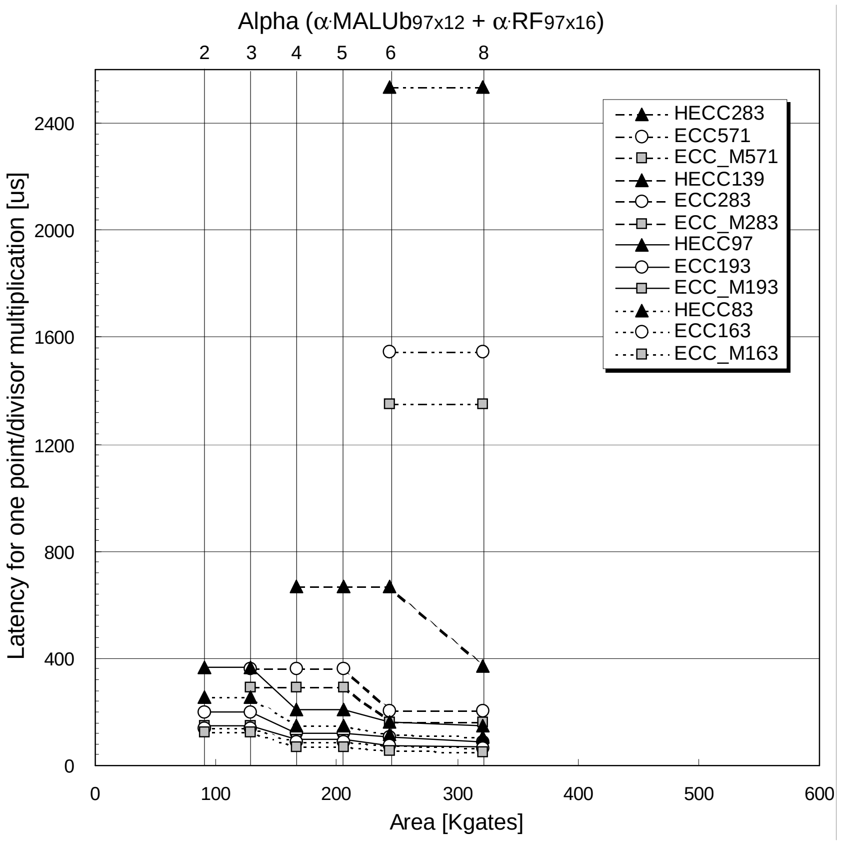

Figure 5.9: The latency of point/divisor multiplication for different variants of CONFIG-I, *i.e.* $\alpha\cdot$ MALUb$_{97\times12}$ + $\alpha\cdot$ RF$_{97\times16}$.

Table 5.4: Computational cost of modular inversion with CONFIG-I (@292 MHz).

Field size [bits]	83	97	139	163	193	283	571
Time for modular inversion [μs]	2.8	3.9	7.1	9.4	13.2	26.1	99.8
Ratio in ECC	-	-	-	13%	15%	14%	7%
Ratio in ECC_M	-	-	-	17%	18%	16%	7%
Ratio in HECC	3%	3%	2%	-	-	2%	-

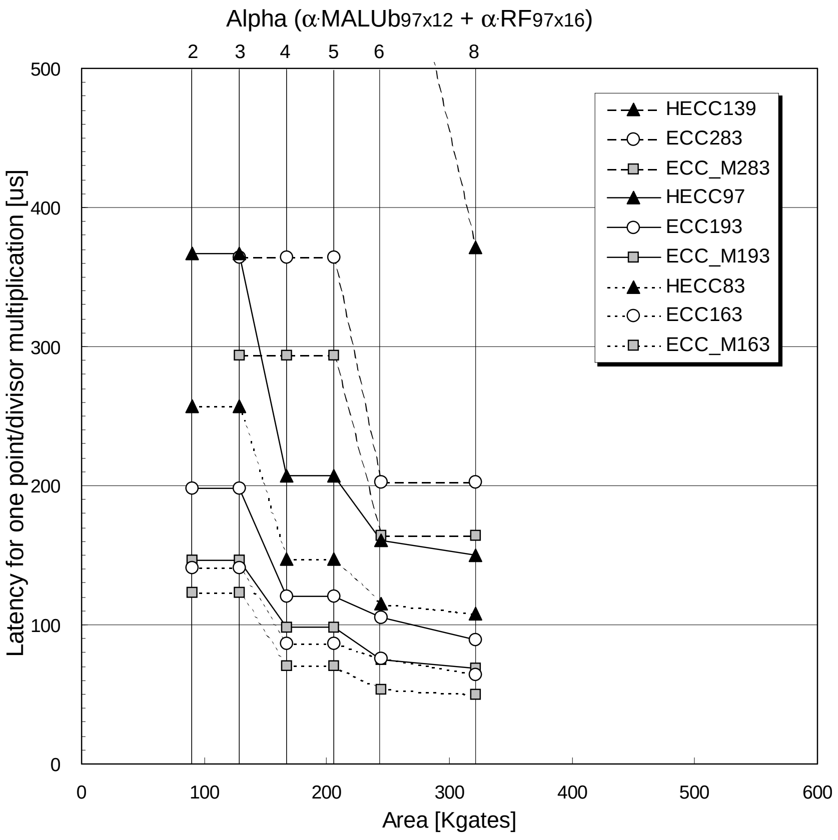

Figure 5.10: Magnified image of Fig. 5.9.

For achieving faster performance, other configurations are also considered by fixing the field length and the irreducible polynomial and supporting either ECC or HECC only. In these configurations, one fixed-size RF can be shared with the MALUb cores. For instance, we can consider the configuration of 4· $\text{MALUb}_{83\times12}$ + $\text{RF}_{83\times32}$ for HECC83. In addition, we use ROM for storing the μ-coded program. These configurations offer higher performance with lower cost compared to CONFIG-I and -II at the cost of reduced flexibility and programmability. The results of this configuration are also discussed in Sect. 5.4.

5.3 High-speed ECC over GF(2^{163}) with the P-MALUb

From the results of the fair performance comparison for curve-based cryptography discussed in Sect. 5.2, ECC is considered as the best choice for a high-speed implementation. Here in addition to ILP, another form of parallelism is introduced in the datapath, *i.e.* the parallelized MALUb (P-MALUb) is used for exploring the fastest possible ECC implementation. Namely, the proposed cryptoprocessor exploits ILP in an EC point multiplication algorithm and executes instructions on several P-MALUb cores. The field size is fixed to 163 bits and a fixed irreducible polynomial is used. The computational sequence of EC point multiplication is also optimized in order to achieve the speed limitation of ECC over binary fields.

5.3.1 System Architecture

The proposed architecture of the cryptoprocessor consists of the main controller, several P-MALUb cores and the RF that shares intermediate variables between the P-MALUb cores, *i.e.* the so-called shared memory. The block diagram of the cryptoprocessor is illustrated in Fig. 5.11. As for the implementation discussed in Sect. 5.2, one can perform all necessary modular operations for ECC with the P-MALUb that for a single operation of the form $A(B+D)+C$. This architecture facilitates parallel processing for EC point multiplication.

5.3.2 High-speed Options for ECC over GF(2^m)

Many interesting computational sequences for ECC over binary fields exist including the following sequences.

- The binary NAF method (Alg. 2.17).
- The Montgomery powering ladder [53, 63].
- The τ-NAF or TNAF method on a Koblitz curve [90].

In this sense, a domain specific programmable architecture is an attractive choice for an Elliptic Curve (EC) cryptoprocessor because it offers the equivalent performance of an ASIC while maintaining the flexibility to support the wide range of options for EC point multiplication. As summarized in Table 5.5, we denote the computational sequences of point multiplication with ECC, ECC_M and ECC_KC depending on the type of the curve and the point multiplication algorithm. Although further performance improvement is possible by applying the width-w NAF method (Alg. 2.17) to ECC and ECC_KC, the cost and performance trade-off degrades compared to the binary NAF method. We discuss the details in Sect. 5.3.4.

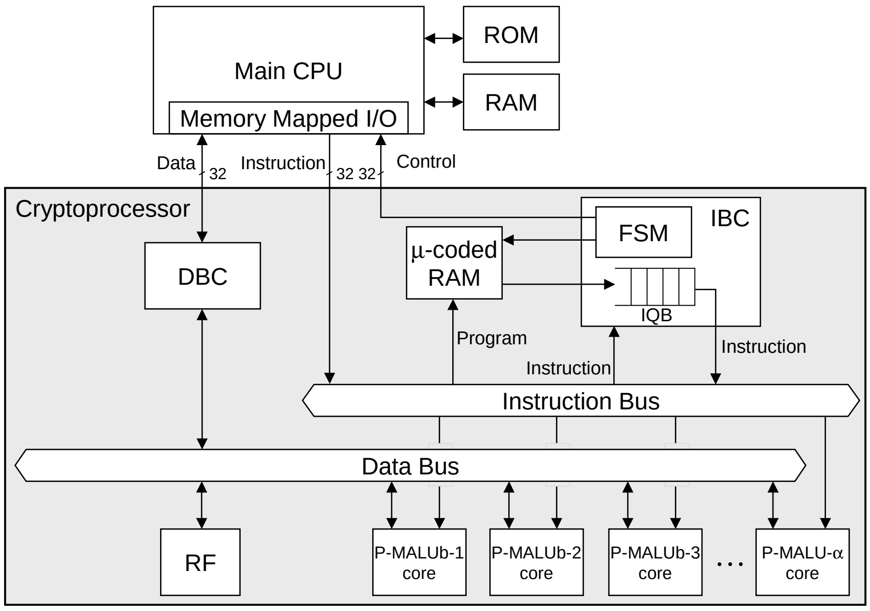

Figure 5.11: Block diagram for programmable EC cryptoprocessor that embeds multiple P-MALUb cores.

Table 5.5: Various methods to compute EC point multiplication.

	Type of Curve	Point Multiplication Algorithm
ECC	Generic Curves	Binary NAF
ECC_M	Generic Curves	Montgomery powering ladder
ECC_KC	Koblitz Curves	TNAF method

The Montgomery Powering Ladder

The EC point multiplication over GF(2^m) can be efficiently executed with the method of the Montgomery powering ladder that maintains the relationship $P_2 - P_1$ as invariant (Alg. 5.1) [63]. All computations are performed on the x-coordinate only in affine coordinates and hence the point multiplication can be performed with less computational sequences. This algorithm is considered as a special case of the parallelized left-to-right algorithm introduced in Alg. 2.10.

If $P_1 \neq P_2$, the x-coordinate of the point addition ($P_3 = P_2 + P_1$) is computed

Algorithm 5.1: Algorithm for point multiplication: the Montgomery powering ladder [63].

```
Require: a point $P$, a non-negative integer $k = (1k_{l-2} \cdots k_1 k_0)_2$.
Ensure: $x(kP)$.
1: $P_1 \leftarrow P$, $P_2 \leftarrow 2P$
2: for $i$ from $l-2$ down to 0 do
3:   if $k_i = 1$ then
4:     $x(P_1) \leftarrow x(P_1) + x(P_2)$, $x(P_2) \leftarrow x(2P_2)$
5:   else
6:     $x(P_2) \leftarrow x(P_2) + x(P_1)$, $x(P_1) \leftarrow x(2P_1)$
7:   end if
8: end for
9: Return $x(P_1)$
```

as

$$x_3 = x + \left(\frac{x_1}{x_1 + x_2}\right)^2 + \frac{x_1}{x_1 + x_2}\,, \tag{5.4}$$

where

$$x = \frac{x_1 y_2 + x_2 y_1}{(x_1 + x_2)^2} + \frac{x_1 x_2}{(x_1 + x_2)^2} + \frac{x_1 x_2}{(x_1 + x_2)}\,. \tag{5.5}$$

If $P_1 = P_2$,

$$x_3 = x_1^2 + \frac{b}{x_1^2}\,, \tag{5.6}$$

In order to avoid computationally expensive modular inversions, López and Dahab applied this method to the projective coordinates [53]. In this algorithm, all necessary computations are performed on the X and Z coordinates in a projective representation, where $x = X/Z$. The computational sequences for point addition ($P_1 = P_1 + P_2$) and point doubling ($P_1 = 2P_1$) are described in Alg. 5.2.

Koblitz Curve

A Koblitz curve is an elliptic curve defined over GF(2) that is given by the equation

$$E : y^2 + xy = x^3 + ax^2 + 1\,, \tag{5.7}$$

where $a \in \{0, 1\}$. Koblitz curves are of interest because point doublings in the binary method can be replaced with computationally cheaper operations. Namely, consider the Frobenius map $\tau : E :\rightarrow E$ defined by $\tau(x, y) = (x^2, y^2)$ for all points on a curve E except $\tau(\mathcal{O}) = \mathcal{O}$ [31]. This map can be efficiently computed

Algorithm 5.2: Algorithms for the Montgomery powering ladder in projective coordinates [53]. Here, $c = \sqrt{b} = b^{2^{m-1}}$.

```
Require:
    $P_1 = (X_1, Z_1)$,
    $P_2 = (X_2, Z_2)$,
    $X_3 = X_3(P_2 - P_1)$.
Ensure: $P_1 = P_1 + P_2$.
 1: $X_1 = X_1 Z_2$;
 2: $Z_1 = X_2 Z_1$;
 3: $t_1 = X_1 Z_1$;
 4: $Z_1 = X_1 + Z_1$;
 5: $Z_1 = Z_1 Z_1$;
 6: $X_1 = xZ_1 + t_1$;
 7: Return $P_1$;
```

```
Require:
    $P_1 = (X_1, Z_1)$.
Ensure: $P_1 = 2P_1$.
 1: $t_2 = X_1 X_1$;
 2: $t_3 = Z_1 Z_1$;
 3: $Z_1 = t_2 t_3$;
 4: $t_2 = t_2 t_2$;
 5: $t_3 = t_3 t_3$;
 6: $X_1 = ct_2 + t_3$;
 7: Return $P_1$;
```

since it relies on the squaring operation in GF(2^m). It can be shown from the point addition operation that three points (x, y), (x^2, y^2) and (x^4, y^4) satisfy the following relation:

$$(x^4, y^4) + 2(x, y) = (-1)^{1-a}(x^2, y^2)\,. \tag{5.8}$$

From the definition of the Frobenius map we get

$$(\tau^2 + 2)P = \mu\tau(P)\,, \tag{5.9}$$

where $\mu = (-1)^{1-a}$.

So, the Frobenius map can be viewed as a complex number τ that satisfies $\tau^2 + 2 = \mu\tau$, for which we choose $\tau = (\mu + \sqrt{-7})/2$. Let $\mathbb{Z}[\tau]$ denote the ring of polynomials in τ with coefficients from $\mathbb{Z}$. Then we write as

$$\begin{aligned}&(u_{t-1}\tau^{t-1} + \ldots + u_1\tau + u_0)P \\ =&u_{t-1}\tau^{t-1}(P) + \ldots + u_1\tau(P) + u_0(P)\,.\end{aligned} \tag{5.10}$$

Therefore, one has to find a decomposition of a scalar k in the following form $k = u_{t-1}\tau^{t-1} + \ldots + u_1\tau + u_0$ (the so-called τ-adic expansion) and then use the previous equation to compute kP. This point multiplication is called TNAF method and it is computed with point additions/subtractions and τ multiplications (τP) (see Alg. 5.3). Solinas proposed an efficient τ-adic expansion of k by recoding the scalar in this form [90].

Table 5.6 summarizes the cost of the three different computational sequences for point multiplication. Numbers in the table indicate the number of operations

Algorithm 5.3: Algorithm for point multiplication: TNAF method [31].

```
Require: a point $P$, a positive integer $\rho' = \sum_{i=0}^{l-1} u_i \tau^i$, $u_i \in \{-1, 0, 1\}$.
Ensure: $kP$.
1: $P_1 \leftarrow P$, $P_2 \leftarrow \mathcal{O}$
2: for $i$ from $l-1$ down to 0 do
3:     $P_2 \leftarrow \tau P_2$
4:     if $u_i = 1$ then
5:         $P_2 \leftarrow P_2 + P_1$
6:     else if $u_i = -1$ then
7:         $P_2 \leftarrow P_2 - P_1$
8:     end if
9: end for
10: Return $x(P_2)$
```

by the P-MALUb. The averaged cost of point multiplication is estimated for 163-bit key.

Table 5.6: Computational cost of EC point multiplication.

	ECC (NAF)	ECC_M	ECC_KC (TNAF)
Point addition	14	6	14
Point doubling (τ multiplication)	10	6	2
Point multiplication	2391	1956	1087

5.3.3 Performance Improvement by Exploiting Multi-level Parallelism

Our proposed cryptoprocessor has three levels parallelism: the first is in the datapath of the MALUb, the second is in the modular operations performed in the P-MALUb and the third is in the instructions determined by the computational sequence of the EC point multiplication. Figure 5.12 illustrates performance improvement of the proposed cryptoprocessor for different hardware configurations and different point multiplication algorithms to evaluate the effect of each level of parallelism. Although the performance cannot be doubled by introducing the P-MALUb due to the memory accesses, the performance of point multiplication is improved by a factor of 1.8 for all three point multiplication algorithms. The performance also improves as the number of the P-MALUb or MALUb increases

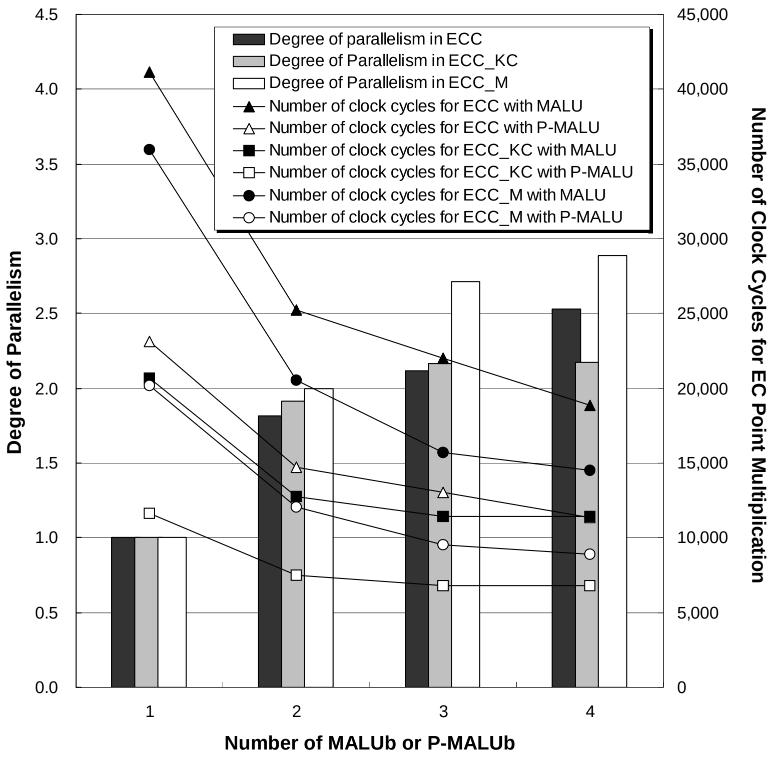

Figure 5.12: Degree of parallelism and required clock cycles for different hardware configurations of 163-bit ECC, ECC_M and ECC_KC.

because of the parallel execution of instructions. In order to see the effectiveness of this parallelism, we define *the degree of parallelism* as

$$\frac{\sum_{i=1}^{l_{max}} M_i \cdot i}{\sum_{i=1}^{l_{max}} M_i} \quad (l_{max} = 1, 2, 3, 4)\,, \tag{5.11}$$

where M_i is the number of instructions executed in i-way parallel processing and l_{max} is the number of P-MALUb or MALUb cores. For instance, the degree of parallelism becomes 2.0 for ECC_M with two copies of the datapath since almost all instructions are processed in two-way parallel executions. The degree of parallelism is almost proportional to the number of the processing elements up to three-way parallelism for ECC_M. In this way, we obtain 2.7 for the degree of parallelism in ECC_M with three copies of the datapath. However, no effective performance improvement is observed for four P-MALUb or MALUb cores. For the degree of parallelism in ECC and ECC_KC, we obtain 2.1 and 2.2 with three copies of the datapath, respectively. These results indicate that the computational sequence in ECC and ECC_KC cannot utilize more than two processing elements effectively.

5.3.4 Implementation Results

In order to develop the fastest implementation possible, the cryptoprocessor is synthesized with 0.13-μm CMOS technology with the P-MALUb configuration of $d = 12$. The synthesis results show that point multiplication of ECC over GF(2^{163}) on a generic curve can be computed in 20 and 16 μs respectively for the binary NAF (Non-Adjacent Form) and the Montgomery powering ladder algorithm. The performance can be accelerated furthermore on a Koblitz curve and achieves a point multiplication of 12 μs with the TNAF (τ-adic NAF) method. This fast performance allows us to perform over 80 000 point multiplications per second.

Cost-performance Trade-offs

So far, we have focused on reducing the time for one point multiplication. More precisely, our optimization in the cryptoprocessor is for reducing the latency of one point multiplication. This type of improvement is important for applications that perform point multiplications sequentially. For example, when an ECC-RFID reader verifies the identity of an ECC-RFID tag, the critical operation can be point multiplication in the reader and tag. For speeding up an identification protocol such as Schnorr identification protocol [85], point multiplication in the reader should be as fast as possible. Because it is hard to accelerate the operations in the tags and the communication speed between the tags and the reader is determined by the carrier frequency.

On the other hand, we can assume other kinds of PKC applications that need point multiplications in parallel. When computing two independent point multiplications (*e.g.* k_1P_1 and k_2P_2), the performance can be doubled by using two EC cryptoprocessors (denoted as two channels) in parallel. Namely, this type of improvement is for increasing the throughput.

Figures 5.13 and 5.14 show trade-offs between area and performance respectively for ECC_M and ECC_KC with up to four channels. The figures marked at the plot indicate the number of P-MALUb or MALUb cores. One channel of the cryptoprocessor with one MALUb is the smallest hardware configuration in our study, and the gate size is 78 Kgates. By increasing the number of MALUb cores, the performance improves up to 38 200 and 48 600 kP/s respectively for ECC_M and ECC_KC. When using the P-MALU instead of the MALUb, the performance

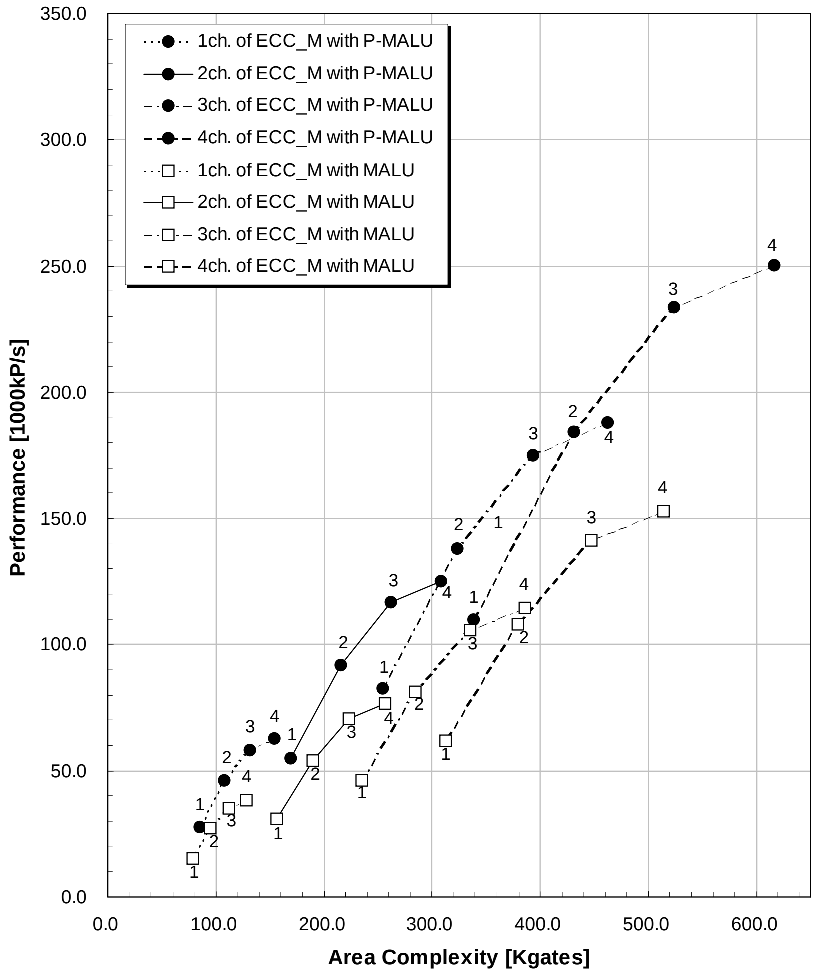

Figure 5.13: Performance of ECC_M for area complexity. The numbers labeled on the plots indicate the number of the P-MALUb or the MALUb cores.

achieves 82 000 and 62 600 kP/s respectively for ECC_M and ECC_KC with four P-MALUb cores.

As discussed in Sect. 5.3.3, the best cost-performance trade-off is obtained from a cryptoprocessor with two P-MALUb cores for ECC and ECC_KC, and with three P-MALUb cores for ECC_M. Figure 5.15 shows the estimated cost-performance curve when preparing up to 10 channels of ECC, each of which has two P-MALUb

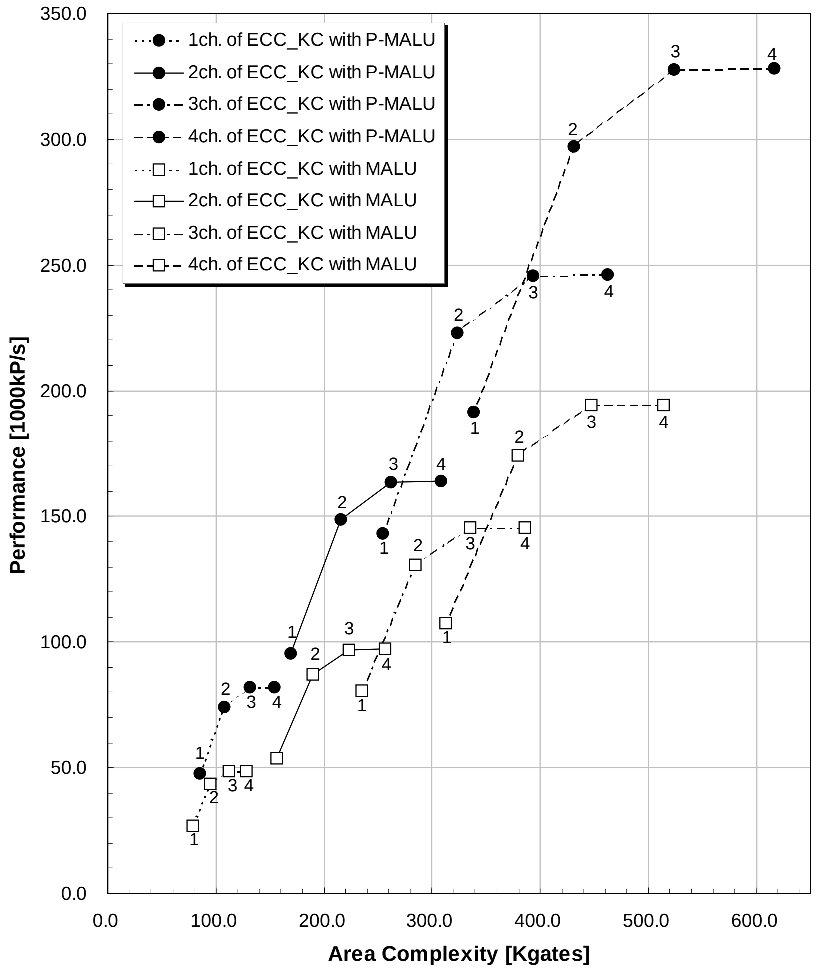

Figure 5.14: Performance of ECC_KC for area complexity. The numbers labeled on the plots indicate the number of the P-MALUb or the MALUb cores.

cores for ECC and ECC_KC, and three P-MALUb cores for ECC_M. As can be seen from this figure, our 163-bit ECC implementation offers a performance of 689, 446 and 350 kP/s per 1 Kgates, respectively for ECC_KC, ECC_M and ECC on the same cyptoprocessor.

The result of the NAF method with windows of width 4 (NAF_4) for ECC and the TNAF method with windows of width 4 (TNAF_4) for ECC_KC are also

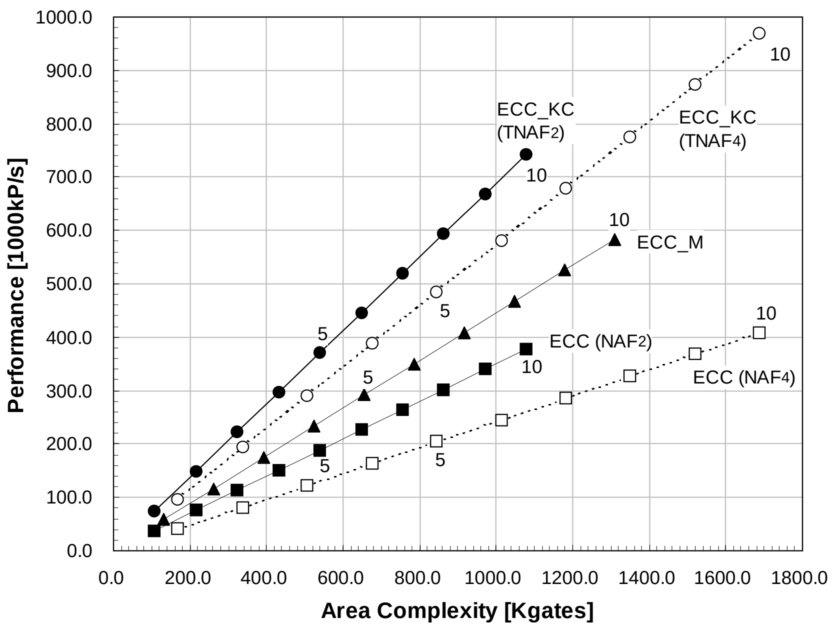

Figure 5.15: Cost and performance trade-offs of ECC, ECC_M and ECC_KC with the P-MALUb. The window method is also applied for ECC and ECC_KC (width-4). The numbers labeled on the plots indicate the number of channels.

plotted to compare to the binary method and the TNAF method, respectively. As is visible for Fig. 5.15, NAF_4 and $TNAF_4$ show worse area-performance trade-offs in our implementation. This is because we use a F/F-based multi-port RF, and hence additional memory space for pre-computed points is expensive relative to the performance gain.

5.4 Comparison with Previous Work

Tables 5.7 and 5.8 summarize the performance of EC point multiplication and HEC divisor multiplication for selected field sizes and different hardware configurations. Our proposed cryptoprocessor can provide various choices of area and performance. The observed performance maintains high for all supported field sizes.

When implementing the cryptoprocessor based on CONFIG-I, the highest flexibility and performance can be obtained for both ECC and HECC with 393 Kgates. On the other hand, for CONFIG-II, the gate size becomes 244 Kgates with some performance penalty for ECC and HECC. The cryptoprocessor can also support

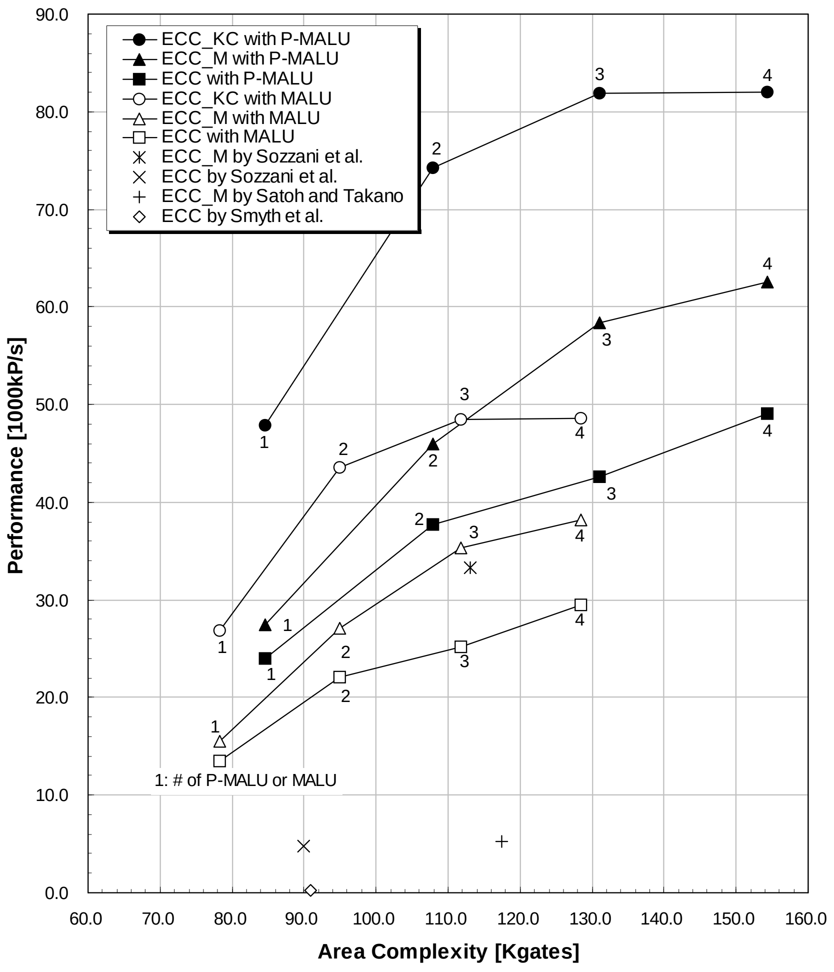

Figure 5.16: Performance of EC point multiplication for area complexity. The numbers labeled on the plots indicate the number of the P-MALUb or the MALUb cores.

high security-level ECC and HECC including ECC over GF(2^{571}) and HECC over GF(2^{283}) on the same hardware.

Comparing with previous work, our HECC implementation results are faster than the implementation reported by Wollinger [103] that was one of the fastest HECC implementations. Furthermore, our reconfigurable cryptosystem can support both ECC and HECC. Our results for the ECC implementation also show a better performance than other previous work except an ECC_M implementation of Sozzani *et al.* [91]. This is because their ASIC design uses the 163-bit *fixed* field size and a hardwired controller, which offers less scalability and flexibility than our reconfigurable design. In fact, our design with a fixed irreducible polynomial shows better performance than their result while supporting both ECC and ECC_M.

Moreover, Table 5.7 lists the results for the compact and fastest EC cryptoprocessor that uses the P-MALUb cores and a fixed irreducible polynomial $x^{163} + x^7 + x^6 + x^3 + 1$. As a result, point multiplication on a generic curve ECC over GF(2^{163}) can be performed in 16 μs and 20 μs, respectively with the binary NAF method and the Montgomery algorithm. On a Koblitz curve, point multiplication is accelerated up to 12 μs by using the TNAF method. These results outperform any other designs proposed previously in terms of performance and the trade-off between cost and performance. We also summarize the most important comparison results in Fig. 5.17.

5.5 Conclusions

This chapter presented a multi-core cryptoprocessor for ECC and HECC to support a wide range of field sizes and to accelerate point/divisor multiplication of ECC and HECC of genus 2 over GF(2^m) by exploiting ILP on-the-fly. The superscaling feature is facilitated by defining a single instruction that is flexibly defined as $AB + C$ or $A(B + D) + C$; it can be used for all field operations such as modular multiplications, modular additions and point/divisor operations. We conclude that the operation $A(B+D)+C$ is effective to decrease the number of clock cycles for point/divisor multiplication.

The fully programmable cryptoprocessor can handle various curve parameters and an arbitrary irreducible polynomial. In addition, a wide range of the field size of modular operations can be supported by reconfiguring the datapath in the MALUb cores. Thus the trade-off between performance and security-level of ECC and HECC can be obtained simply by changing the program and reconfiguring the MALUb cores. The synthesis results show that point/divisor multiplication can be performed at 292 MHz with 244 Kgates, while supporting ECC over GF(2^{571}) and HECC over GF(2^{283}). In our design, ECC offers a better area-performance trade-off than HECC.

Furthermore, we explored a high-speed cryptoprocessor for ECC. The implementation results on a generic curve showed that point multiplication of ECC over GF(2^{163}) were performed in 16 μs and 20 μs, respectively with the binary NAF method and the Montgomery algorithm. Further more, point multiplication was accelerated up to 12 μs on a Koblitz curve by using the TNAF method. This speed-up was achieved by thoroughly exploiting parallelism in the system architecture of ECC.

The results of NAF$_4$ and TNAF$_4$ showed a better performance and worse area-performance trade-offs compared to the results of NAF and TNAF, respectively. This is because we use a F/F-based multi-port RF, and hence additional memory space for pre-computed points is expensive for the performance gain.

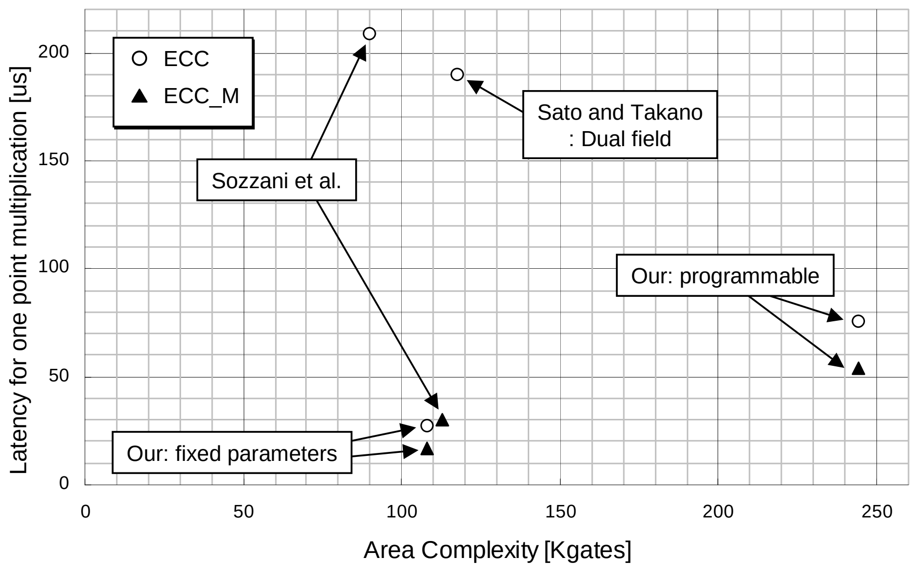

Figure 5.17: Comparison of the latency and area of EC cryptoprocessors over GF(2^{163}) synthesized with a 0.13-μm CMOS.

Table 5.7: Performance comparison of ECC hardware implementations over binary fields.

Ref. Design	Technology / FPGA	f_{max} [MHz]	Area [slices/Gates]	Galois Field	Irreducible Polynomial	Performance† [μs]	Comments
This work (Reconfig-urable)	0.13-μm CMOS	292	393 Kgates	GF(2^{163})	Arbitrary	70 / 54	CONFIG-I ($\alpha = 6$) 6·MALUb$_{97\times12}$ + 6·RF$_{97\times32}$.
				GF(2^{193})		90 / 75	
				GF(2^{283})		187 / 164	
				GF(2^{571})		1394 / 1349	
			244 Kgates	GF(2^{163})	Arbitrary	76 / 54	CONFIG-II ($\alpha = 6$) 6·MALUb$_{97\times12}$ + 6·RF$_{97\times16}$.
				GF(2^{193})		105 / 75	
				GF(2^{283})		202 / 164	
				GF(2^{571})		1545 / 1349	
This work (Fastest)	0.13-μm CMOS	555.6	154 Kgates	GF(2^{163})	$x^{163} + x^7 + x^6 + x^3 + 1$	20	4·P-MALUb$_{163\times12}$, Binary NAF.
						12	4·P-MALUb$_{163\times12}$, TNAF.
						16	4·P-MALUb$_{163\times12}$, Montgomery.
This work (Best trade-off)	0.13-μm CMOS	555.6	108 Kgates	GF(2^{163})	$x^{163} + x^7 + x^6 + x^3 + 1$	27	2·P-MALUb$_{163\times12}$, Binary NAF.
						14	2·P-MALUb$_{163\times12}$, TNAF.
			131 Kgates			17	3·P-MALUb$_{163\times12}$, Montgomery.
[91]	0.13-μm CMOS	416.7	90 Kgates♯	GF(2^{163})	Arbitrary	209 / -	Mult., divider and squarer.
			113 Kgates♯			- / 30	Two mult., divider and squarer.
[81]	0.13-μm CMOS	510.2	117.5 Kgates	GF(2^{163})	Arbitrary	190 / -	Support of GF(p). 64-bit × 64-bit mult.
[89]	0.13-μm CMOS	166	90.9 Kgates 32K-bit RAM	GF(2^{163})	Arbitrary	3720	Support of GF(2^m) and GF(p) up to 2016-bit fields.
[19]	Virtex-E (xcv2000E-7)	66.4	10 034 slices‡	GF(2^{163})	Arbitrary	- / 300	MSD first mult., divider and squarer for field sizes up to 255.
				GF(2^{193})		- / 420	
				GF(2^{233})		- / 510	
				GF(2^{163})	Fixed	- / 140	
				GF(2^{193})		- / 190	
				GF(2^{233})		- / 230	
[12]	Virtex-II (xc2v6000)	54	-	GF(2^{162})	Fixed	- / 60	Optimal normal basis mult. for a fixed field size.
		35	-	GF(2^{270})	Fixed	- / 170	
[54]	Virtex-E (xcv2000E)	66	5009 slices‡	GF(2^{163})	$x^{163} + x^7 + x^6 + x^3 + 1$	233 / -	Mult., div. and squarer (Generic).
						75 / -	Mult., div. and squarer (Koblitz).
[68]	Virtex-E (xcv400E)	76.7	3002 slices + 10 BRAMs	GF(2^{167})	$x^{167} + x^6 + 1$	- / 210	167-bit×16-bit mult. and 167-bit×167-bit squarer.
[80]	Virtex-E (xcv3200E)	9.99	19 626 slices + 26 BRAMs	GF(2^{191})	$x^{191} + x^9 + 1$	- / 59.26	Two binary Karatsuba mult., squarer and inverter.

†: The ECC performance denoted as - / - indicates the performance of ECC / ECC_M.
‡: Estimate based on their results of the number of LUTs.
♯: Estimate based on their results of 0.47 mm^2 and 0.59 mm^2.

Table 5.8: Performance comparison of HECC hardware implementations over binary fields.

Ref. Design	Technology / FPGA	f_{max} [MHz]	Area [slices/Gates]	Galois Field	Irreducible Polynomial	Performance [μs]	Comments
This work	0.13-μm CMOS	292	393 Kgates	GF(2^{83})	Arbitrary	108	CONFIG-I ($\alpha = 6$) 6·MALUb$_{97\times 12}$ + 6·RF$_{97\times 32}$.
				GF(2^{97})		150	
				GF(2^{139})		284	
				GF(2^{283})		1364	
			244 Kgates	GF(2^{83})	Arbitrary	115	CONFIG-II ($\alpha = 6$) 6·MALUb$_{97\times 12}$ + 6·RF$_{97\times 16}$.
				GF(2^{97})		160	
				GF(2^{139})		670	
				GF(2^{283})		2533	
		500	65 Kgates	GF(2^{83})	$x^{83} + x^7 + x^4 + x^2 + 1$	63	4·MALUb$_{83\times 12}$ + RF$_{83\times 32}$.
[103]	Virtex-II Pro (xc2vp20-7)	57.0	4039 slices	GF(2^{81})	Fixed	787	Two mult. / inverter.
		60.7	7737 slices			387	Three mult. / two inverters.

Chapter 6

Low-power MALU for RFID Tags

6.1 Introduction

An emerging example of PKC applications is Radio Frequency IDentification (RFID) tag in sensor networks. Different from high-performance systems introduced in Chapter 5, very tight constraints are put on public-key implementations in the number of gates, power consumption, communication bandwidth, etc. In this chapter we show that the arithmetic unit for curve-based cryptosystems can be further optimized for these new challenging applications. We investigate the feasibility of public-key services for pervasive computing. We show that ECC and HECC processors can be designed for lightweight applications suitable for RFID tags and wireless sensor networks. Here, the term lightweight corresponds to small die size and low power consumption. Therefore, we propose a hardware processor supporting ECC and HECC that features very low footprint and low-power. We investigate two types of solutions, one of which can be applied to ECC over binary fields GF(2^m) where m is a prime and the other one to ECC over a composite field GF($(2^m)^2$) or for HECC on curves of genus 2. The latter implies the same arithmetic unit for both cases which is a factor 2 smaller than for the first ECC option.

The work of Gaubatz *et al.* [27] discusses the necessity and the feasibility of PKC protocols in sensor networks. In [27], the authors investigated implementations of two algorithms for this purpose, *i.e.* Rabin's scheme and NTRUEncrypt. The results for NTRUEncrypt are very appealing with 3000 gates and a power consumption of less than 20 μW @500 kHz. In [26], the authors presented an architecture of an ECC processor which occupies an area of 18 720 gates and consumes less than 400 μW of power 500 kHz. The field used is a prime field with

$p = (2^{101} + 1)/3$. Entity authentication for RFIDs can be achieved by symmetric as well as asymmetric primitives. Most of the previous work dealt with implementations of symmetric ciphers. The most notable example is the work of Feldhofer *et al.* [22], who consider the implementation of AES on an RFID tag. Recently, Fürbass and Wolkerstorfer presented a low-cost hardware design for ECC that is suitable for implementations of ECDSA on a small IC [24]. They also conjectured that ECC might be feasible on high-end RFID-tags.

6.2 Design Strategy of the MALU for RFID Tags

As discussed in Sect. 3.3, the MALU over GF(2^m) can support composite operations such as $AB+C$ or $A(B+D)+C$ by changing the hardware architecture of the datapath. However, the composite operations are not suitable for low-power and compact implementations. Here, we proposed a new datapath architecture based on the simplest MALUb operating AB. The MALUb is a bit-serial multiplier; it can also operate as a modular adder by changing a mode setting in the MALUb. Thus, the compact MALUb can support both the AB and the $A + B$ operations on the same hardware.

6.2.1 Modular Operations by the Compact MALUb

The architecture of the compact MALU over GF(2^m) is implemented based on the MSB-first bit-serial multiplier as illustrated in Fig. 3.5. The procedure of modular multiplication is the same as the original MALUb (see Alg. 3.5). Modular addition, $(B(x) + C(x)) \bmod P(x)$ can also be supported on the same hardware logic by setting $C(x)$ to the register for $T(x)$ instead of resetting register $T(x)$ when initializing the MALUb. This operation requires additional multiplexers and XORs; however, the cost of this solution is lower compared to the case of having a separate modular adder. This type of hardware sharing is very important for such low-cost applications. The compact MALUb is also scalable in the digit size d which can be selected by exploring the best combination of performance and cost.

In Fig. 6.1 the architecture of the compact MALUb is shown for finite field operations in GF(2^{163}). To perform modular multiplication, the *cmd* value should be set to 1 and the operands should be loaded into the registers A and B. The value stored in A is evaluated digit per digit from MSB to LSB. The result of the multiplication will be provided in register T after $\lceil \frac{163}{d} \rceil$ clock cycles. Modular addition is performed by giving *cmd* the value 0, resetting register A and loading the operands into registers B and T. The value that is loaded into T is denoted by C in Fig. 6.1. After one clock cycle, the result of the addition is provided in register T. The *cmd* value makes sure only the last cell is used for this addition.

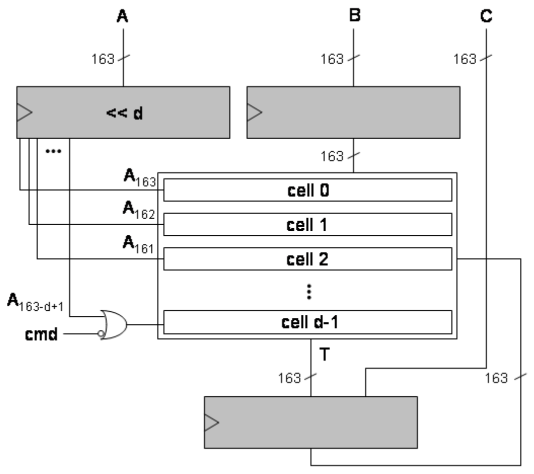

Figure 6.1: Architecture of the compact MALUb over GF(2^{163}) [71].

The cells inside the compact MALUb all have the same structure, which is depicted in Fig. 6.2. A cell consists of a full-length array of AND-gates, a full-length array of XOR-gates and a smaller array of XOR-gates. The position of the XOR-gates in the latter array depends on the irreducible polynomial; in this case, the polynomial $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$ is used. The *cmd* value determines whether the reduction needs to be done or not. For modular multiplication, the reduction is needed. For modular addition, the reduction will not be performed. The output value T_{out} is either given (in a shifted way) to the next cell or to the output register T in Fig. 6.2. The input value T_{in} is either coming from the previous cell or from the output register T.

The strong part of this architecture is that it uses the same cell(s) for modular multiplication and addition without a big overhead in multiplexers. This is achieved by using T as an output register as well as an input register. The F/Fs in T are provided with a load input, which results in a smaller area overhead compared to a solution that would use separate datapaths for modular multiplication and addition.

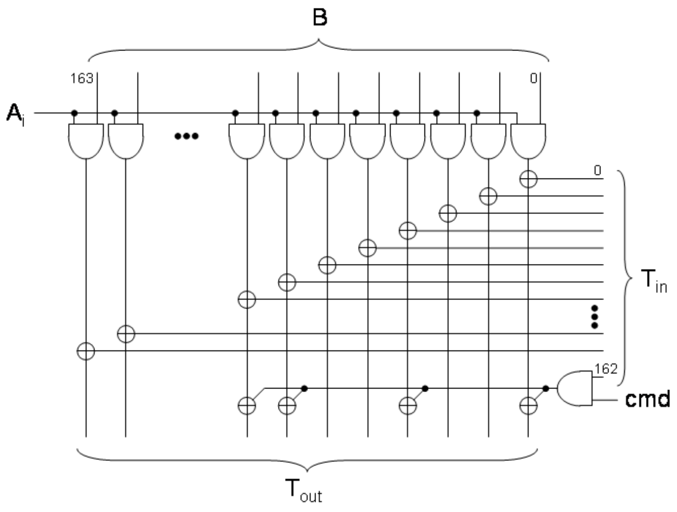

Figure 6.2: Combinatorial logic inside one cell of the compact MALUb [71].

6.3 Implementation Results

In this section, we discuss the results for area complexity, power consumption and latency for ECC and HECC processors. As mentioned already, the core part of each curve-based protocol is one point/divisor multiplication. For example the protocol of Schnorr allows for authentication at the cost of one point multiplication in a tag besides the communication with a reader [85]. Therefore, the performance will be evaluated based on the number of clock cycles required for one point/divisor multiplication.

6.3.1 Area Complexity

The results for area complexity are given here. All of the designs are synthesized by Synopsys Design Vision using a 0.13-μm CMOS library. We use binary fields from bit-size 131 to 163 as recommended by NIST.

The results for various architectures of the compact MALUb with respect to the choice of fields and the size of d for ECC and HECC are given in Table 6.1. For the case of ECC over composite fields and HECC the MALUb shrinks in size but some speed-up is then necessary which we obtain by means of digit-serial multiplications (instead of a bit-serial one, *i.e.* $d = 1$).

Table 6.1: The area complexity in gates of the compact MALUb various filed and digit sizes.

Field size	d=1	d=2	d=3	d=4	d=6	d=8
67	2171	2421	-	2901	3420	3899
71	2299	2563	-	5535	3576	4083
79	2564	2854	-	6016	4012	4530
83	2693	2997	-	6486	4168	4794
131	4281	4758	5219	5685	-	-
139	4549	5043	5535	6028	-	-
151	4955	5472	6016	6540	-	-
163	5314	5900	6486	7052	-	-

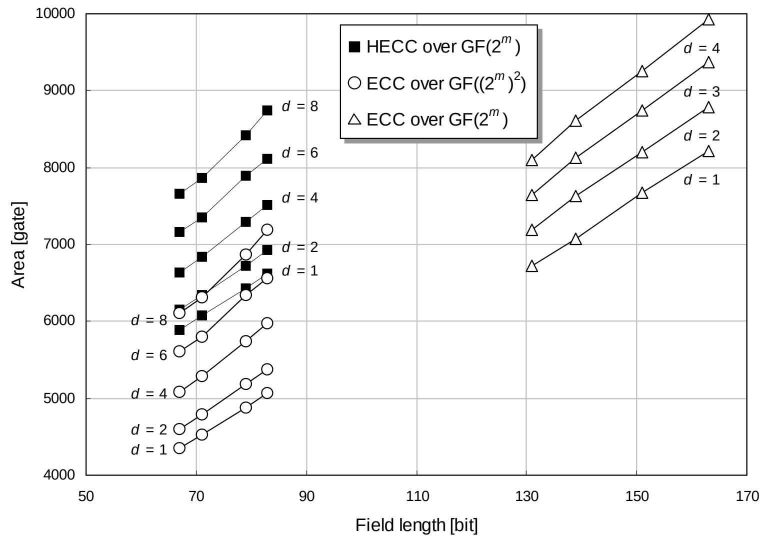

Figure 6.3: The area complexity of ECC over GF(2^m), ECC over GF($(2^m)^2$) and HECC over GF(2^m).

Figure 6.3 shows the results of area complexity including the controller for all the cases of the ECC and HECC processor. In this figure results are given for the complete processors excluding the data memory block where intermediate values are stored. Although they use the same MALUb, the gate size of HECC processor is approximately 2000 gates bigger than that of ECC processor over a composite

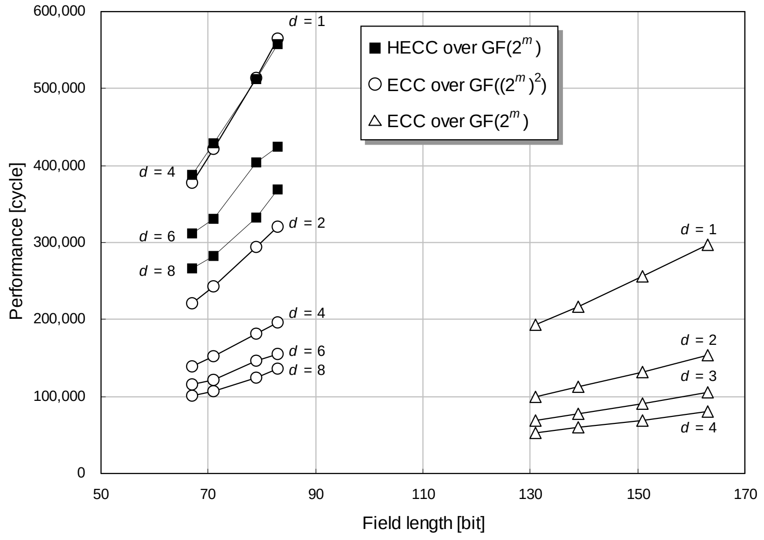

Figure 6.4: Performance in the number of cycles for ECC over GF(2^m), ECC over GF($(2^m)^2$) and HECC over GF(2^m).

field. This is due to the complicated sequence of HECC that makes the controller block bigger than the controller of ECC.

6.3.2 Performance Estimation

The performance for ECC is calculated by the algorithm for point multiplication in Alg. 5.2. We calculate the total number of cycles for each field operation with the following formulae. The total number of cycles for one modular multiplication is $\lceil \frac{l}{d} \rceil + 3$ where l denotes the number of bits of the scalar k, *e.g.* the secret key and d is the digit size in the MALUb. On the other hand, one field addition takes

Table 6.2: The number of cycles required for one point multiplication for ECC over GF(2^{131}), ECC over GF($(2^{67})^2$) and HECC over GF(2^{67}).

PKC	d=1	d=2	d=3	d=4	d=6	d=8
ECC over GF(2^{131})	193 180	100 230	68 770	53 040	-	-
ECC over GF($(2^{67})^2$)	378 252	220 248	-	138 852	114 912	100 548
HECC over GF(2^{67})	1 153 243	648 508	-	388 493	312 018	266 133

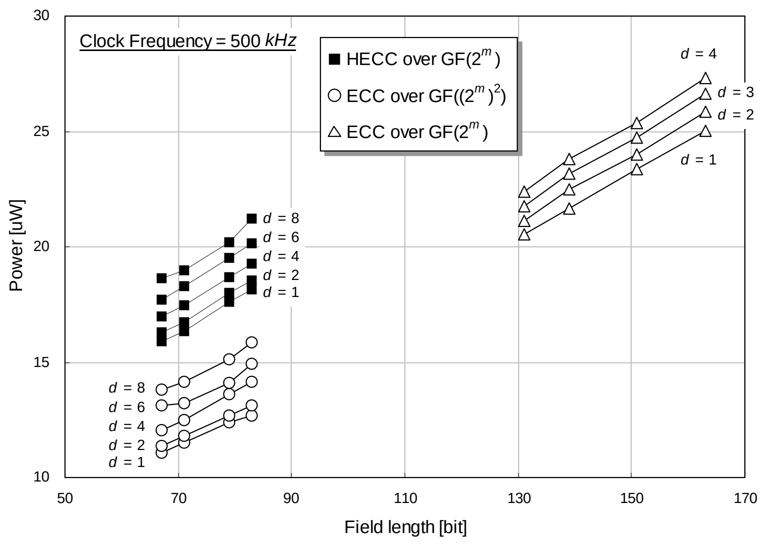

Figure 6.5: Results of power consumption for ECC over GF(2^m), ECC over GF($(2^m)^2$) and HECC over GF(2^m) (@500 kHz).

4 cycles. The number of cycles required for one EC point multiplication can be estimated with Eq. (6.1):

$$(l-1)\left[11\left(\left\lceil\frac{l}{d}\right\rceil+3\right)+12\right] . \tag{6.1}$$

The number of cycles required for one point multiplication in the case of ECC over a composite field GF$((2^m)^2)$ and the case of HECC over GF(2^m) are given by

$$(2l-1)\left[36\left(\left\lceil\frac{l}{d}\right\rceil+3\right)+324\right] , \tag{6.2}$$

and

$$(2l-1)\left[115\left(\left\lceil\frac{l}{d}\right\rceil+3\right)+621\right] . \tag{6.3}$$

The results for the total number of cycles of one point multiplication for ECC over GF(2^{131}), ECC over GF($(2^{67})^2$) and HECC over GF(2^{67}) are given in Table 6.2. Performance results are also summarized in Fig. 6.4.

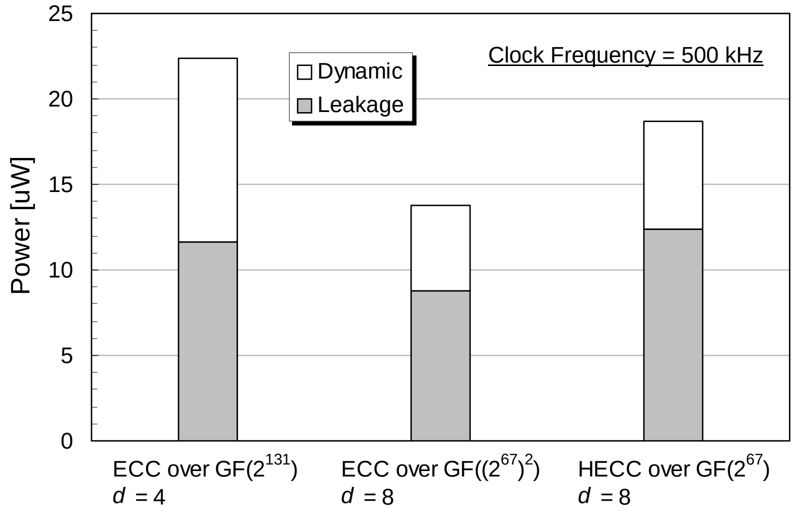

Figure 6.6: Power consumed by leakage current and dynamic current for ECC over GF(2^{131}), ECC over GF($(2^{67})^2$) and HECC over GF(2^{67}) synthesized with a 0.13-μm CMOS library (@500 kHz).

6.3.3 Power Estimation

Another figure of merit is power as we target applications where consumed power or energy is of utmost importance. In order to estimate the power consumption, we use Synopsys PrimePower for our designed gate-level netlist synthesized with a 0.13-μm CMOS library and the conservative wire load model (pre-layout netlist). All results for power consumption for the proposed processors are given in Fig. 6.5. The results for power consumption show a similar evolution as the results of area complexity.

Figure 6.6 shows the profile of the power consumption for the three processors with a clock frequency 500 kHz. As can be seen from the figure, the amount of power consumed by leakage current is more than a half of the total power for all the cases.

6.3.4 Energy Estimation

Although we mainly discussed a low-power implementation assuming passive RFID tags, the energy consumption is also interesting for a PKC application with a battery backup. Figure 6.7 shows that with regard to the energy per point/divisor

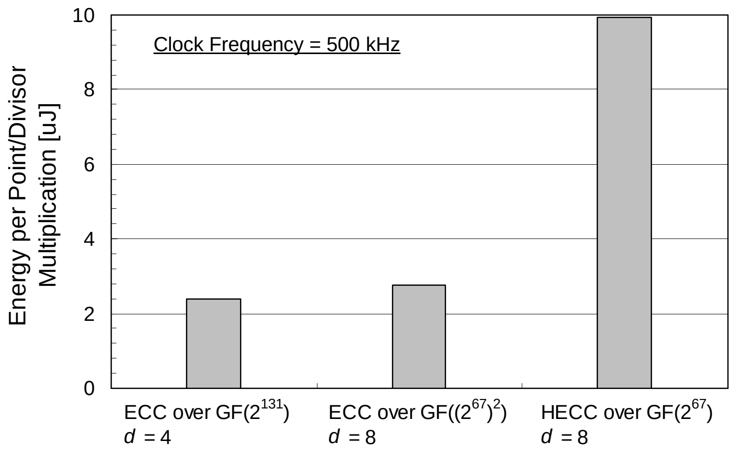

Figure 6.7: Results of energy per point/divisor multiplication for ECC over GF(2^{131}), ECC over GF($(2^{67})^2$) and HECC over GF(2^{67}) synthesized with a 0.13-μm CMOS library (@500 kHz).

multiplication, ECC implementations are better than HECC in this regard.

6.4 Comparison with Previous Work

To calculate the time for one point multiplication we need an operating clock frequency. However, the clock frequency that can be used is strictly influenced by the total power. We assumed an operating clock frequency of 500 kHz in order to estimate the actual timing as our power results showed to be reasonable for RFID applications.

We obtain 106 ms and 201 ms for the best case of ECC over GF(2^{131}) for $d = 4$ and GF($(2^{67})^2$) for $d = 8$ respectively, and 532 ms for the best case of HECC over GF(2^{67}) for $d = 8$. The results of all relevant works are compared in Table 6.3.

We underline again that the results for the area complexity presented above do not include data memory. The amount of storage that is required for the implementations is to store $13m$ bits and $28m$ bits for ECC over a composite field and HECC respectively, where m is the number of bits of elements in a subfield. For ECC, the required storage is $5m$ bits.

Table 6.3: Comparison of the compact PKC implementations.

Ref. Design	PKC	Area [gates]	Tech. [μm]	Freq. [kHz]	Perf. [ms]	Power [μW]	Energy per operation [μJ]	Comments
This work	ECC GF(2^{131})	8104	0.13	500	106	22	2.4	Operation: Point/divisor Mult. (No data memory included).
	ECC GF($(2^{67})^2$)	6103			201	14	2.8	
	HECC GF(2^{67})	7652			532	19	10.0	
[49]	ECC GF(2^{131})	11 970	0.35	13 560	18	N/A	N/A	Operation: Point Mult.
[26]	ECC GF(p), $p = (2^{101} + 1)/3$	18 720	0.13	500	410	394	161.5	Operation: ECDSA
[24]	ECC GF(p_{192})	23 600	0.35	83 333	6.0	141 000	846.0	Operation: ECDSA

6.5 Conclusions

The conclusion from this chapter is that ECC over composite fields is the best choice for low-power and area but for the performance it is better to use the regular ECC. Therefore, ECC over composite fields is a promising for the main computational part of RFID tags by choosing an appropriate digit size d as well as ECC over a binary field.

In terms of a low-power implementation, the leakage current is considered one of the bottlenecks in our experiments. For each processor, we only see small differences in their power consumption for different digit sizes d. This indicates that the combinatorial logic part of the MALUb (*i.e.* cells in Fig. 6.1 has less impact for the total power consumption. Since the performance improvement is almost proportional to the digit size d (see Fig. 6.4), a better power-performance trade-off can be obtained with a large d.

This part of our research is still work in progress. The data memory block needs to be taken into consideration to estimate the performance of the actual implementation. In addition, the power consumption needs to be estimated with a net-list with a physical lay-out information. Recent work presented in [51] shows that the MALU-based ECC processor over GF(2^{163}) can be implemented with 13.2 Kgates including the memory block. With our results, even a smaller area can be obtained using a design of ECC over composite fields.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis explored the trade-off between cost, performance and security by implementing several HW/SW co-designs for Public-Key Cryptography (PKC). For each abstraction level of cryptosystems, we discussed a various options and HW/SW implementations. We dealt with RSA, Elliptic Curve Cryptography (ECC) and Hyper-Elliptic Curve Cryptography (HECC) for PKC, and considered three different underlying fields, GF(p), GF(2^m) and GF($(2^m)^2$) for ECC. We also discussed the use of projective coordinates and the algorithms for point/divisor multiplication.

In order to explore the options for HW/SW implementations, we first focus on constructing a "soft" public-key cryptoprocessor that has a reconfigurable datapath and a programmable controller, *e.g.* the 8051 or the ARM microprocessor. The datapath or the MALU is scalable and flexible, and hence various options can be determined depending on the system requirements including the type of underlying fields (*e.g.* GF(p) or GF(2^m)), the parallelism in the digit size d (related to the performance) and the supported field size (*i.e.* security level). The proposed cryptoprocessor can be configured as a multi-core system to perform several instructions for the MALUs in parallel. In this way, our proposed cryptoprocessor plays a role as a development platform in this thesis.

Second, we discussed two extreme cases of implementations for high-speed and low-power in Chapters 5 and 6. For high-speed implementations, the synthesis results of a programmable cryptoprocessor show that a high-speed EC point multiplication and HEC divisor multiplication can be performed on the same cryptoprocessor with a gate size of 244 Kgates, while supporting ECC over GF(2^{571})

and HECC over GF(2^{283}). For achieving faster performance, other configurations are also considered by fixing the field length, fixing the irreducible polynomial and supporting either ECC or HECC only. In these configurations, the point multiplication over GF(2^{163}) was accelerated up to 12 μs @556 MHz on a Koblitz curve by using the τ-NAF (TNAF) method.

For a low-power PKC implementation, ECC over GF(2^{131}) can be performed in 106 ms with the 8-Kgate MALUb$_{131\times4}$ (Modular Arithmetic Logic Unit over GF(2^{131}) with the digit size $d = 4$). The power consumption is 22 μW with a 500 kHz clock. The results for ECC over a composite field GF(2^{131}) showed a lower power consumption of 14 μW with a smaller gate size of 6 Kgates. Although the performance of 201 ms is twice slower than that of ECC over GF(2^{131}), ECC over composite fields is a possible candidate for a low power application.

7.2 Future Work

In 2007, Suzuki introduced a high-speed FPGA implementation for modular exponentiation [92]. The design uses seventeen Xilinx hardware-macro DSPs (Digital Signal Processors) that are embedded in a Virtex-4 FPGA device. The embedded DSP called Xtreme DSP accelerates the algorithm for modular multiplication, and a single 1024-bit modular exponentiation is computed in 1.7 ms. Fan *et al.* proposed an efficient SW implementation of Montgomery multiplication for multi-core systems and achieved a higher throughput than previous SW implementations on a single core [21]. Thus, a new direction for a high-speed cryptosystem would be based on the use of multiple DSPs, CPUs or other types of processing cores [20]. The organization of memory and the RF need to be investigated further more because it is one of the system bottlenecks in accelerating the performance of multi-core systems.

It is of great interest to implement PKC by use of available DA tools and platforms such as TI's multi-core DSPs [97], Target Compiler's tools [93] and Tensilica's processors in order to compare their performance results with ours. Especially, power and energy need to be compared with our introduced architecture since they are significantly important in embedded systems.

Our research for secure RFID (Radio Frequency IDentification) tags is still work in progress. The RFID tag needs a controller, a serial communication block, RAM and ROM for facilitating a system-level algorithm. In addition, we need to support modulo n operations for entity authentication protocols. Our proposed dual-field MALU can be used for this purpose, however it is still expensive for the RFID tags. Further investigation for the trade-off between area, performance and power is needed with a post-layout netlist in order to determine the best system architecture of the RFID tags.

We used a secure design flow to verify that the design offers simple Side-Channel Attack (SCA) resistance in an early design stage (Appendix A). This flow was facilitated by the GEZEL system-level language. In order to protect the ECC processor from DPA attacks, the use of a Random Number Generator (RNG) [17] is regarded one of the most effective countermeasures. Our future work includes a SCA-resistant FPGA implementation by using the RNG (Random Number Generator) block [17]. The research goal is to explore the trade-off between area, performance and security deeply on an FPGA and to minimize the area of the security-sensitive part in an ECC implementation.

In 2007, Yu and Schaumont proposed a method called Double Wave Dynamic Differential Logic (Double WDDL) that overcomes the difficulty of balanced routing of the complementary cells and wires by allocating the copy of a WDDL [99] module additionally and swapping the complementary logics of the copied module [105]. This method is effective for FPGA implementations and hence can be used for our FPGA implementation.

Appendix A

Secure Design Flow in PKC

Secure systems have to be resistant against various malicious attacks in order to prevent leakage of information or an unexpected use of the system. For silicon devices, the most straightforward attack is a physical attack directly to the silicon. This is a powerful method if a probing point is available. For instance, it is easy to retrieve secret information by probing the data on a bus. The fault induction attack [9] is a well-known technique that works by disturbing the device to induce errors in the computation. These attacks are named active attacks after the technique.

The recent threats are Side-Channel Attacks (SCAs) which are attacks that observe in a non-intrusive way computational timing, power variants or electromagnetic radiation of the device. By simple observation or mathematical processing of the observed physical phenomena, one can retrieve secret data out of the device. We introduce a system-level design flow that can be used for evaluating the security level of hardware implementations against power analysis attacks. The design flow offers an environment to get a quick and correct evaluation of the first order attacks. In this way, we can take the cost for SCA resistance into account in an early stage of the design.

The passive attack is based on measuring physical characteristics leaking from side-channels of the embedded system. Timing Analysis (TA) checks the computation time. If the execution time varies with the data or the key used in the computations, this can be detected by the attacker [46]. Simple Power Analysis (SPA) measures the power fluctuations during cryptographic operations and guesses the actual types of computations. Furthermore, in [47], Kocher, Jaffe and Jun introduced Differential Power Analysis (DPA), a type of differential SCA that also considers effects correlated to data values. Brier, Clavier and Oliver introduced Correlation Power Analysis (CPA) that aims at enhancing DPA by improving the Hamming weight model [10]. Electromagnetic Analysis (EMA) and Acoustic Analysis (AA) were also introduced as effective SCA examples [87].

At the cycle-true Register Transfer level (RT-level), simple SCAs can be detected and countermeasures can be implemented. Of course, once a design is made resistant to the simple SCA, the differential and higher order ones need to be addressed. They require solutions such as adding noise, introducing random delays or using a special style of logic [99].

A.1 Simple SCA Resistant Design Flow

We use a simulation environment GEZEL, which allows us to estimate instantaneous dynamic power consumption. We use Toggle Count Per Clock (TCPC) as an approximation for this power. The toggle count is obtained directly out of the RT-level model and does not require synthesis of the model. Despite this approximation, our experiments have shown that it is sufficient to build cryptosystems that are resistant to simple SCA. As mentioned before, higher order attacks, such as DPA, need to be addressed in an actual implementation. However, the system-level design gives the designer an environment to get a quick and correct evaluation of first order attacks. It doesn't make sense to address higher order attacks (which are much more difficult and time consuming to mount) if simple first order attacks are possible. This is especially needed for cryptographic algorithms that contain multiple levels of complex arithmetic, such as ECC and HECC. Because of the size and complexity of the design, these SCA resistance tests are very time consuming at gate level and impossible at SPICE level.

The GEZEL design environment allows us to try out different alternatives for the performance, area and SCA resistance at a cycle-accurate level. For each alternative, functional tests and simple SCA resistance tests are verified as illustrated in Fig. A.1. We obtain the toggle count of the HW modules as a function of the clock cycle in order to evaluation the power pattern. These toggle counts allow us to analyze the risk for simple SCAs (especially SPA) and, if detected, to rewrite the HW module. After completing both tests by using cycle-accurate simulation, the HW model is converted into VHDL, and synthesized by a secure back-end. In the next section, we briefly discuss the mechanism for toggle counting at the RT-level.

A.2 RT-level Toggle Counting

The HW descriptions in GEZEL are expressions of cycle-accurate register transfers, containing operations and assignments on signals and registers. We obtain toggle count estimates directly on these expressions, by means of a simple set of rules:

- The toggle count per cycle (TCPC) for a register or a signal is the Hamming distance between the value during the previous clock cycle and the value

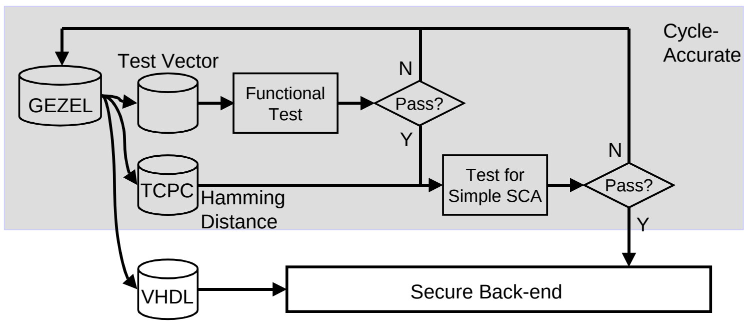

Figure A.1: GEZEL system-level design flow for security applications.

during the current clock cycle. For a register, this toggle count is measured at the register input.

- The TCPC of a simple operation is given by the Hamming distance of the previous operation result and the current result. The toggle count is measured at the operation output. Simple operations includes additions, subtractions, shifts, multiplications, and so on.
- The TCPC of an expression composed of simple operations is given by the sum of TCPC of individual operations.

For example, assume an RT-level expression as follows:

```
signal   a;
register b;
a = 3;
b = b + a;
```

This piece of code contains three operations: two assignments and an addition. In the first clock cycle, signal `a` changes from 0 to 3. The addition operation output will be 3 as well, and this value will be assigned to register `b`. The total toggle count for the first clock cycle thus equals 6. In the second clock cycle, signal `a` does not change value. The output of the addition operator will now change from 3 to 6 however (Hamming distance '110' - '011' = 2), and register `b` will change from 3 to 6 as well. The toggle count for the second clock cycle thus equals 4. We also choose way of the toggle count depending on the type of the data change, *i.e.* all transitions (0-to-1 and 1-to-0), 0-to-1 or 1-to-0 transitions. In the above case, the 0-to-1 Hamming distance will be 1. We can continue in this way to obtain an approximation of the immediate dynamic power consumption. This methodology is very simple; for example it does not take into account glitching nor the implementation complexity of operators. For the purpose of simple SCA on the other hand, it is adequate.

A.3 Countermeasure by Using Balanced Point Operations

Point multiplication can be implemented as a repeated combination of point addition and point doubling. The simplest algorithm, the left-to-right binary method (Alg. 2.9) is vulnerable to SCA because of a different computation time in the for-loop. One of the countermeasures is the double-and-add-always method proposed by Coron [13]. The idea is that SCA resistance is ensured by computing both point addition and point doubling in one for-loop. This method increases the HW cost or decreases the performance because of the additional dummy point additions.

In order to resist SCAs with a small penalty for performance, we create a countermeasure by letting them have the same type of the computational sequence in point addition and doubling. To do this, dummy MALUn, CS and CP instructions are added to the point operations. Namely, the imbalance of the binary method can be solved by inserting dummy instructions such that intervals of point operations can be constant and toggle counts can be similar within the intervals. The original idea is introduced by Batina *et al.* for ECC over binary fields [6].

We also evaluate another countermeasure where the coprocessor is implemented to have constant execution time for each MALUn or CP instruction. Moreover, a hardware controller is embedded in the coprocessor to ensure a constant issue of coprocessor instructions. Thus, this countermeasure aims at balancing modular operations.

Each of the proposed countermeasure is implemented on the HW/SW co-design platform using the 8051 as introduced in Sect. 4.2. It is commonly accepted that increasing the resistance against SCAs is at the cost of system performance and/or resources. Therefore, the trade-off between cost, performance, and security is of great interest, which can be explored easily by the proposed HW/SW co-design environment described in Chapter 4.

A.3.1 Results of Balanced Point Operations

Here, the result for a case study of an ECC160p implementation is discussed. Three metrics, namely cost, performance, and security for ECC160p are estimated using the proposed design flow of GEZEL system-level co-simulation. First, HW and SW are implemented based on a strategy to find the best trade-off between performance and cost. The performance/cost-optimized result tends to be vulnerable to simple SCAs. To confirm the fact, power estimation for point multiplication is simulated with GEZEL. More precisely, the toggle counts of the coprocessor is collected

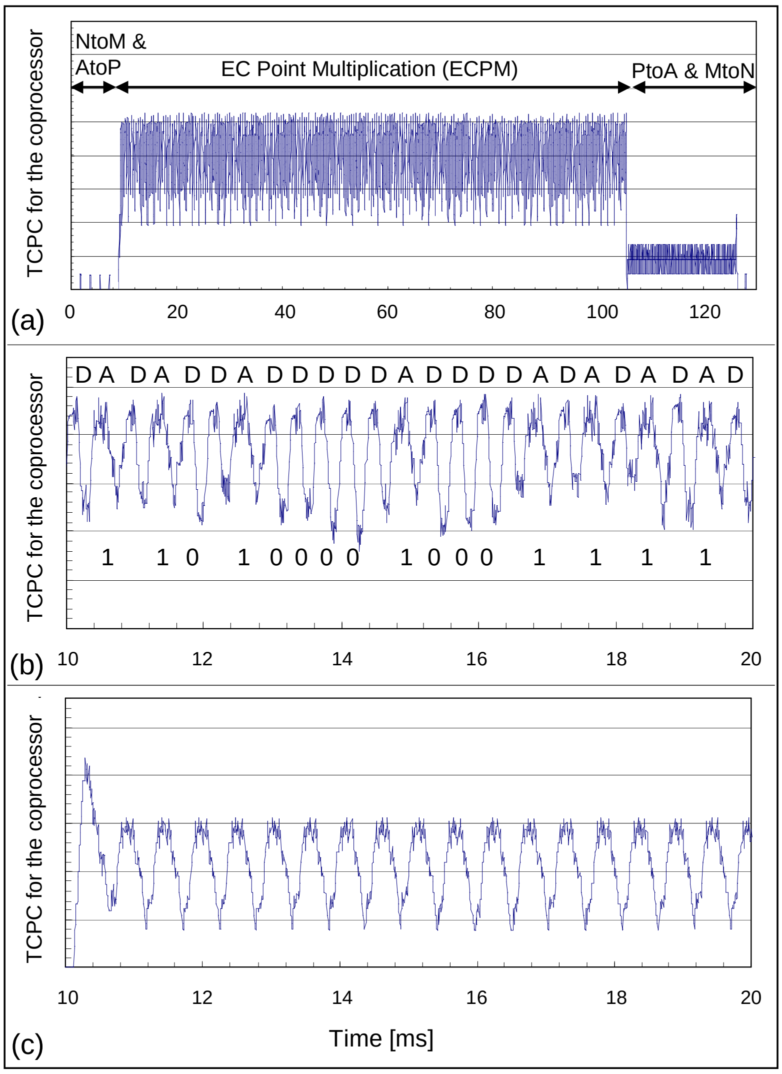

Figure A.2: Results of TCPC for ECC160p. (a) Estimated power consumption for the whole ECC160p operation. (b) The enlarged trace of (a). "D" and "A" means point doubling and point addition respectively. The secret key is written below the signal. (c) Simple SCA resistant result. Point doubling and point addition are indistinguishable.

Table A.1: Result of point multiplication of ECC160p for different security configurations on the same coprocessor.

Simple SCA resistance	**No**	**Double-and-add-always**	**Balanced Operation**
Total Latency [ms]	129.8	172.1 (+33 %)	160.4 (+24 %)
8051 Resource [Bytes]			
XRAM	211	211	211
PROM	2867	2867	2895
FPGA Mapping of Coprocessor			
Number of 4-input LUTs		6666	
Number of slice F/Fs		1195	
Number of Block RAMs		6	

through the ECC operations and traced in smoothing format (average of 2000 cycles). The result is shown in Fig. A.2. AtoP and PtoA represent coordinate conversions between affine and projective coordinates. The representation of values is converted with NtoM and MtoN that convert variables from the normal form to the Montgomery representation and vice versa. The trace in Fig. A.2a clearly shows what operations are executed. The highest risk exists in its enlarged figure, Fig. A.2b. By carefully observing the trace, different shapes of peaks and valleys can be identified. The difference is caused by the imbalanced point operations. Thus, the secret key is easily found (the secret key is added in Fig. A.2b).

In the second place, HW and SW are modified based on our proposed simple SCA resistant implementation. The result is shown in Fig. A.2c. From the trace, it is hard to distinguish the point operation type. One of the side-effects of the countermeasures is a longer execution time because dummy instructions are added to point operations. Hence, this resistance comes at the price of lower performance. The more detailed observation is given in the following.

Three different configurations, performance/area-oriented design and simple SCA resistant design are summarized in Table A.1. The non-balanced binary method (no simple SCA resistance) shows on the left side of the table. The right side of the table describes the simple SCA resistant result with the double-and-add-always method and the proposed countermeasure, *i.e.* the balanced point operation method. In the countermeasure implementations, the latency of point multiplication increases by 33% and 24%, respectively. In order to see the performance-security trade-offs, the same coprocessor is used for all three cases. Although the program size of the 8051 becomes slightly larger (1%) for the case of the balanced operation, this cost impact is negligible. Our proposed countermeasure shows a

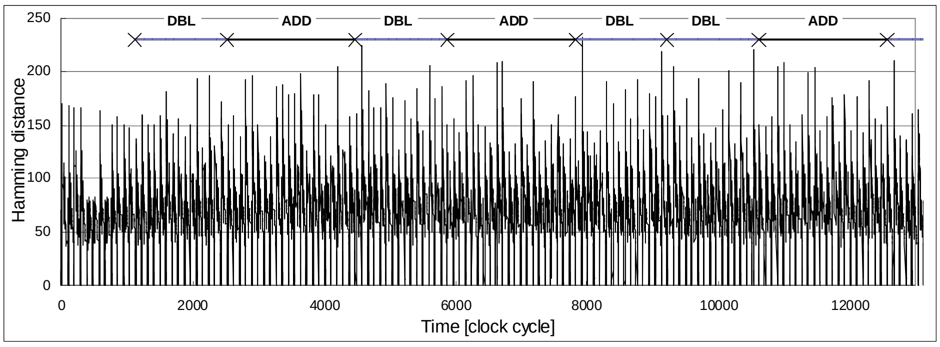

Figure A.3: Result of TCPC with only functional correctness. Trace of Hamming distance for all registers in the ECC160p coprocessor. DBL and ADD denote point doubling and point addition respectively.

better result than the double-and-add-always method in terms of the trade-off between performance and security. Although we cannot see obvious differences in point operations, there are still two potential risks in these countermeasures for simple SCA as follows: (1) nonuniform communication between HW and SW and (2) stalling cycles in the 8051 caused by the branch condition in the binary method. In order to overcome the risks, we discuss another countermeasure in the next section.

A.4 Countermeasure by Using Single MALUn Operations

Next we describe another countermeasure to protect an ECC160p implementation against SCA; we unify the modular multiplication and addition instructions, *i.e.* we have a unified modular multiply-and-add instruction which completes the $R = (XY \pm S) \bmod n$ (where n is a prime) operation in constant execution time.

In addition, a dedicated HW controller for the binary method is implemented in the coprocessor. As discussed in Chapter 4, this additional micro-coded controller improves the performance, but also enhances simple SCA resistance, since it can dispatch the the instructions $R = (XY \pm S) \bmod n$ continuously to the MALUn datapath part by using a queue buffer. In this way, the imbalance of the binary method can be solved without extra dummy instructions, at the price of an increased hardware cost. Thus this countermeasure explores the cost and security trade-offs.

The countermeasure is verified with GEZEL tool. As a result, the trace of

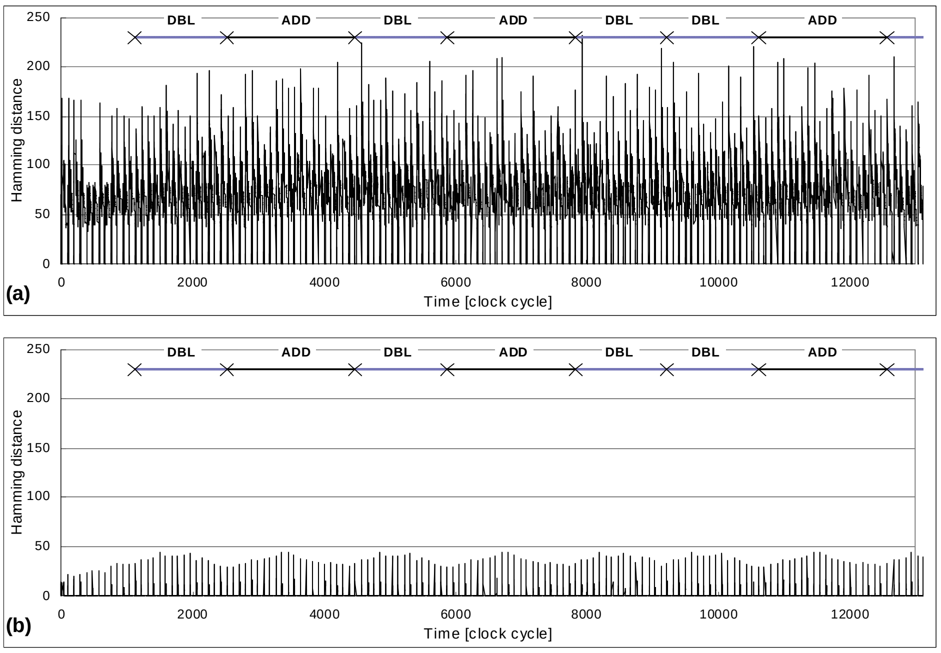

Figure A.4: Results of partial TCPC for simple SCA verification. (a) TCPC for the MALUn. (b) TCPC for the controller block.

the GEZEL simulation shows peaks which have constant intervals because of the constant execution time of the MALUn (Fig. A.3). The period corresponds precisely to one MALUn's operation time (93 cycles). We can therefore claim that TA resistance is successfully implemented as intended. In other words, TA resistance can be checked with the cycle-true functional simulations. However, some peaks are higher than others (*e.g.* the fourth and fifth MALUn operations in the point addition). They are always located at the same position in point doubling and addition. Therefore, this design might have a problem in terms of SPA resistance although its functionality and TA resistance are correctly implemented.

In order to identify the reason of the "big peaks", we evaluate the partial TCPC. First we separate the toggle counts into two parts; one is obtained from the MALUn and the other one is from the controller. From the results shown in Fig. A.4, we know the "big peaks" are mainly caused by the MALUn. Moreover, the risk of vulnerability against SPA also exists in the controller block because the periodical character of the peaks in the trace is corresponding to point operations. Thus, by doing partial simulation of the TCPC, we can find the hidden security

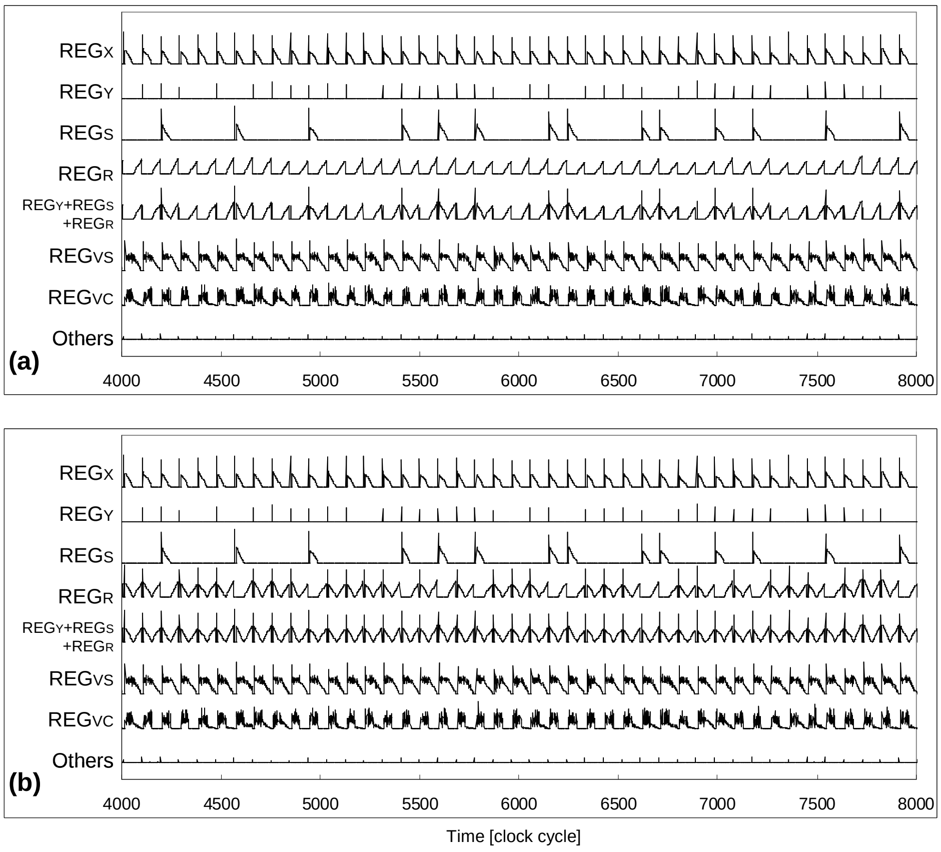

Figure A.5: Results of TCPC for the register in the MALUn. (a) No simple SCA resistance. (b) After fixing the security bug.

bugs which cannot be found by the conventional functional simulation. In the following sections, we focus on explaining how to remove the former security bug with partial toggle counts.

A.5 Security Bug in the MALUn

The MALUn is composed of several registers and the datapath for computing $(XY \pm S) \bmod n$ as shown in Fig. 3.3. All registers are cleared to 0 before an operation. The registers provide the input data for the datapath (REG_X, REG_Y, REG_S), store the intermediate values (REG_{VS}, REG_{VC}), and collect the result (REG_R).

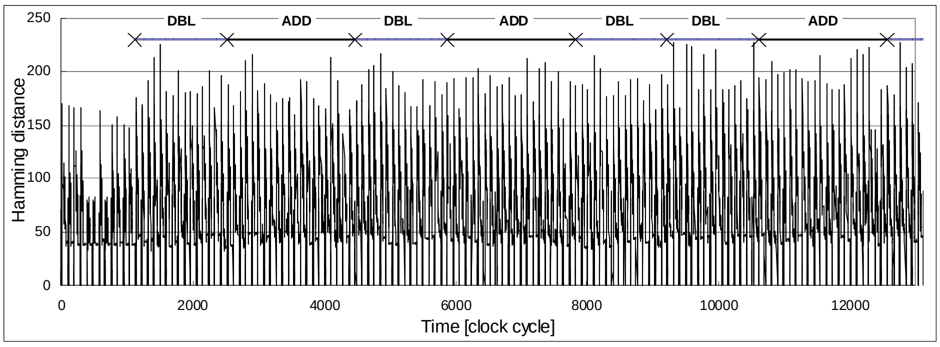

Figure A.6: Result of TCPC for the ECC160p coprocessor after simple SCA verification and countermeasure.

The TCPC results for the registers in the MALUn are shown in Fig. A.5a. From the trace, we can find that the TCPC for REG_Y and REG_S are irregular. Especially, the irregular TCPC for REG_S affects the height of the peaks and eventually leads to the "big peaks" in the first simulation. The reason of the irregularity is explained as follows. When setting the input values for each modular multiplication-and-addition, the MALUn has three different procedures depending on the type of operation as

(1) $(XY + S) \bmod n$: $REG_{X/Y} \leftarrow X/Y$ and $REG_S \leftarrow S$,

(2) $(XY - S) \bmod n$: $REG_{X/Y} \leftarrow X/Y$ and $REG_S \leftarrow \widetilde{S}$,

(3) $XY \bmod n$: $REG_{X/Y} \leftarrow X/Y$ and $REG_S \leftarrow 0$.

$\widetilde{S}$ denotes the bit inversion of S. It is used for the subtraction in the 2's complement representation. Unless the assumption that the probability that each bit of S has a value 1 is 0.5, the Hamming distance in setting REG_S can be regarded the same. However, no toggle occurs with REG_S in the operation (3). This observation agrees with the simulation result completely.

A.6 Fixing the Security Bug

The simplest way of fixing the bug is to add a register that compensates the lack of toggle counts for REG_S. Fortunately, the output register, REG_R, is not used when REG_S is used for the datapath. Therefore, we decide to reuse REG_R to fix this bug. In this way, we can add a SCA resistance to the design without area and performance penalties. The simulation result after fixing the bug is shown in

Fig. A.5b. The subtotal of the TCPC for REG_S, REG_R and REG_Y shows a good regularity in the trace. As a result, Fig. A.6 shows the TCPC trace corresponding to Fig. A.3. It is hard to distinguish point operations from the trace by checking the "big peaks", *i.e.* simple SCA resistance is enhanced with our proposed design method.

There is still another risk in the controller block that has to be solved in order to increase the simple SCA resistance. However, from Fig. A.6, it is difficult to find the periodical character of the peaks as shown in Fig. A.4. Here, a question arises, how we can determine whether design is SPA resistant based on the TCPC trace that is an approximation of dynamic power consumption. Although this remains an open question, the SPA resistant system-level design here is assumed to be a design whose total TCPC has no outstanding peaks or has outstanding peaks (*e.g.* "big peaks" in this case) independent on the secret key in terms of the timing intervals.

Appendix B

Magma Code for Verifying ECC/HECC Sequences

All the sequences for ECC and HECC implementations based on the MALU were checked with the following Magma software. These programs were also used to generate test vectors to verify the HW/SW co-designs introduced in this thesis.

B.1 Weighted Projective Point Addition of ECC over GF(p)

```
p  := RandomPrime(160);
a  := RandomPrime(160);
b  := RandomPrime(160);
E  := EllipticCurve([GF(p)| a, b]);
P1 := Random(E);
P2 := Random(E);

/// Point Addition Test
x1 := P1[1];
y1 := P1[2];
Z1 := RandomPrime(160);
x2 := P2[1];
y2 := P2[2];
Z2 := RandomPrime(160);

/// Magma Result (Expected Values)
P2 := P1 + P2;
```

```
EXPx2 := P2[1];
EXPy2 := P2[2];

/// Sequence for the MALU
/// ECC-160p point addition sequence (normal to weighted projective)
X1 := x1 * Z1^2;
Y1 := y1 * Z1^3;
X2 := x2 * Z2^2;
Y2 := y2 * Z2^3;

/// ECC-160p point addition sequence (Algorithm 2.11)
if (Z1 eq 1) then
  t3 := Z2 * Z2;
  t4 := X1 * t3 + X2;
  t2 := X1 * t3 - X2;
  t1 := t3 * Z2;
  t3 := t1 * Y1 + Y2;
  Y2 := t1 * Y1 - Y2;
  t5 := t2 * t2;
  t1 := t4 * t5;
  X2 := Y2 * Y2 - t1;
  t4 := -2 * X2 + t1;
  t1 := t5 * t2;
  t1 := t3 * t1;
  t3 := t4 * Y2 - t1;
  Y2 := t3 / 2;
  Z2 := t2 * Z2;
else
  t1 := Z1 * Z1;
  t2 := X2 * t1;
  t3 := Z2 * Z2;
  t4 := X1 * t3 + t2;
  t2 := X1 * t3 - t2;
  t5 := t1 * Z1;
  t6 := Y2 * t5;
  t1 := t3 * Z2;
  t3 := t1 * Y1 + t6;
  Y2 := t1 * Y1 - t6;
  t5 := t2 * t2;
  t1 := t4 * t5;
  X2 := Y2 * Y2 - t1;
  t4 := -2 * X2 + t1;
  t1 := t5 * t2;
  t1 := t3 * t1;
  t3 := t4 * Y2 - t1;
  Y2 := t3 / 2;
  Z2 := t2 * Z2;
  Z2 := Z1 * Z2;
end if;

/// ECC-160p point addition sequence (weighted projective to normal)
x2 := X2 / (Z2^2);
y2 := Y2 / (Z2^3);

/// Compare the resutls with expected values (shoudl be 0 if correct)
x2 - EXPx2;
y2 - EXPy2;
```

B.2 Weighted Projective Point Doubling of ECC over GF(p)

```
p  := RandomPrime(160);
a  := RandomPrime(160);
b  := RandomPrime(160);
E  := EllipticCurve([GF(p)| a, b]);
P2 := Random(E);

/// Point Doubling Test
x2 := P2[1];
y2 := P2[2];
Z2 := RandomPrime(160);

/// Magma Result (Expected Values)
P2 := 2 * P2;
EXPx2 := P2[1];
EXPy2 := P2[2];

/// Sequence for the MALU
/// EC point doubling sequence over GF(p) (normal to weighted projective)
X2 := x2 * Z2^2;
Y2 := y2 * Z2^3;
```

```
/// EC point doubling sequence over GF(p) (Algorithm 2.11)
t1 := X2 * X2;
t1 :=  3 * t1;
t2 := Z2 * Z2;
t2 := t2 * t2;
t2 :=  a * t2 + t1;
t1 :=  2 * Y2;
Z2 := Z2 * t1;
t3 := t1 * t1;
t4 := X2 * t3;
X2 :=  2 * t4;
X2 := t2 * t2 - X2;
t1 := t1 * t3;
t1 := t1 * Y2;
t3 := t4 - X2;
Y2 := t2 * t3 - t1;

/// EC point doubling sequence over GF(p) (weighted projective to normal)
x2 := X2 / (Z2^2);
y2 := Y2 / (Z2^3);

/// Compare the results with expected values (should be 0 if correct)
x2 - EXPx2;
y2 - EXPy2;
```

B.3 Weighted Projective Point Addition of ECC over GF(2^m)

```
RR<T> := PolynomialRing(GF(2));
F<u>  := ext<GF(2) | T^163+T^7+T^6+T^3+1>;
R<x>  := PolynomialRing(F);
E     := EllipticCurve([1,Random(F),0,0,Random(F)]);
P1    := Random(E);
P2    := Random(E);

/// Point Addition Test
x1 := P1[1];
y1 := P1[2];
Z1 := Random(F);
x2 := P2[1];
```

```
y2 := P2[2];
Z2 := Random(F);

/// Magma Result (Expected Values)
P2 := P1 + P2;
EXPx2 := P2[1];
EXPy2 := P2[2];

/// Sequence for the MALU
a := Eltseq(E)[2];
/// ECC over GF(2^163) point addition sequence (normal to projective)
X1 := x1 * Z1^2;
Y1 := y1 * Z1^3;
X2 := x2 * Z2^2;
Y2 := y2 * Z2^3;

/// ECC over GF(2^163) point addition sequence (Algorithm 2.12)
if (Z1 eq 1) then
  t2 := Z2 * Z2;
  t4 := t2 * X1 + X2;
  t2 := t2 * Z2;
  t1 := t2 * Y1 + Y2;
  t3 := t4 * Y2;
  t3 := t1 * X2 + t3;
  Z2 := t4 * Z2;
  t5 := t1 *(t1 + Z2);
  Y2 := t4 * t4;
  t2 := t4 * Y2 + t5;
  X2 :=  a * Z2;
  X2 := X2 * Z2 + t2;
  t4 := Y2 * t3;
  Y2 := X2 *(t1 + Z2) + t4;
else
  t6 := Z1 * Z1;
  t3 := t6 * X2;
  t2 := Z2 * Z2;
  t4 := t2 * X1 + t3;
  t1 := t6 * Z1;
  t3 := t1 * Y2;
  t2 := t2 * Z2;
  t1 := t2 * Y1 + t3;
  t5 := t4 * Z1;
  t3 := t5 * Y2;
  t3 := t1 * X2 + t3;
  Z2 := t5 * Z2;
  t5 := t1 *(t1 + Z2);
  Y2 := t4 * t4;
  t2 := t4 * Y2 + t5;
  X2 :=  a * Z2;
  X2 := X2 * Z2 + t2;
  t4 := Y2 * t3;
  t4 := t4 * t6;
  Y2 := X2 *(t1 + Z2) + t4;
end if;

/// ECC over GF(2^163) point addition sequence (projective to normal)
x2 := X2 / (Z2^2);
y2 := Y2 / (Z2^3);

/// Compare the results with expected values (should be 0 if correct)
x2 + EXPx2;
y2 + EXPy2;
```

B.4 Weighted Projective Point Doubling of ECC over GF(2^m)

```
RR<T> := PolynomialRing(GF(2));
F<u>  := ext<GF(2) | T^163+T^7+T^6+T^3+1>;
R<x>  := PolynomialRing(F);
E     := EllipticCurve([1,Random(F),0,0,Random(F)]);
P2    := Random(E);

/// Point Doubling Test
x2 := P2[1];
y2 := P2[2];
Z2 := Random(F);

/// Magma Result (Expected Values)
P2 := 2 * P2;
EXPx2 := P2[1];
EXPy2 := P2[2];
```

```
/// Sequence for the MALU
c  := (Eltseq(E)[5])^(2^161);
/// ECC over GF(2^163) point doubling sequence (normal to projective)
X2 := x2 * Z2^2;
Y2 := y2 * Z2^3;

/// ECC over GF(2^163) point doubling sequence (Algorithm 2.12)
t1 := X2 * X2;
t2 := t1 * t1;
t4 := Y2 * Z2 + t1;
t3 := Z2 * Z2;
Z2 := X2 * t3;
t5 :=  c * t3 + X2;
t3 := t5 * t5;
X2 := t3 * t3;
t1 := X2 *(Z2 + t4);
Y2 := t2 * Z2 + t1;

/// ECC over GF(2^163) point doubling sequence (projective to normal)
x2 := X2 / (Z2^2);
y2 := Y2 / (Z2^3);

/// Compare the results with expected values (should be 0 if correct)
x2 + EXPx2;
y2 + EXPy2;
```

B.5 Weighted Projective Divisor Addition of HECC over GF(2^m)

```
R<T>  := PolynomialRing(GF(2));
F<u>  := ext<GF(2) | T^83 + T^7 + T^4 + T^2 + 1>;
RR<x> := PolynomialRing(F);

/// f3 = 0x0057F00A9FD2F9F42A624
f3 := u^74 + u^72 + u^70 + u^69 + u^68 + u^67 + u^66 + u^65 + u^64 + u^55
    + u^53 + u^51 + u^48 + u^47 + u^46 + u^45 + u^44 + u^43 + u^42 + u^40
    + u^37 + u^35 + u^34 + u^33 + u^32 + u^31 + u^28 + u^27 + u^26 + u^25
    + u^24 + u^22 + u^17 + u^15 + u^13 + u^10 + u^9  + u^5  + u^2;

/// f0 = 0x189309A7E5FEEC1868F65
```

```
f0 := u^80 + u^79 + u^75 + u^72 + u^69 + u^68 + u^63 + u^60 + u^59 + u^57
    + u^54 + u^53 + u^52 + u^51 + u^50 + u^49 + u^46 + u^44 + u^43 + u^42
    + u^41 + u^40 + u^39 + u^38 + u^37 + u^35 + u^34 + u^33 + u^31 + u^30
    + u^24 + u^23 + u^18 + u^17 + u^15 + u^11 + u^10 + u^9  + u^8  + u^6
    + u^5  + u^2  + 1;

f := x^5 + f3*x^3 + x^2 + f0;
C := HyperellipticCurve(f,x);
J := Jacobian(C);
D := 2*Random(J);
E := 2*Random(J);

/// Divisor Addition Test
u10 := Eltseq(D[1])[1];
u11 := Eltseq(D[1])[2];
v10 := Eltseq(D[2])[1];
v11 := Eltseq(D[2])[2];
Z1  := 1;
u20 := Eltseq(E[1])[1];
u21 := Eltseq(E[1])[2];
v20 := Eltseq(E[2])[1];
v21 := Eltseq(E[2])[2];
Z2  := Random(F);

/// Magma Result (Expected Values)
E := D + E;
EXPu20 := Eltseq(E[1])[1];
EXPu21 := Eltseq(E[1])[2];
EXPv20 := Eltseq(E[2])[1];
EXPv21 := Eltseq(E[2])[2];

/// Sequence for the MALU
/// HECC divisor doubling sequence (normal to projective)
U10 := u10 * Z1;
U11 := u11 * Z1;
V10 := v10 * Z1;
V11 := v11 * Z1;
U20 := u20 * Z2;
U21 := u21 * Z2;
V20 := v20 * Z2;
V21 := v21 * Z2;
```

```
/// HECC divisor addition sequence (Algorithm 2.15)
 z1 := U11 *  Z2 + U21;
 z2 := U10 *  Z2 + U20;
 z3 := U11 *  z1 +  z2;
  r :=  z1 *  z1;
  r :=   r * U10;
  r :=  z2 *  z3 +   r;
 w0 := V10 *  Z2 + V20;
 w1 := V11 *  Z2 + V21;
 w2 :=  z3 *  w0;
 w3 :=  z1 *  w1;
 s0 := U10 *  w3 +  w2;
 s1 :=  z1 *( w0 +  w1) + w2;
 s1 :=  z3 *( w0 +  w1) + s1;
 s1 :=  w3 *(  1 + U11) + s1;
  R :=  Z2 *   r;
 s0 :=  s0 *  Z2;
 s3 :=  s1 *  Z2;
 cR :=   R *  s3;
  t :=  s1 *( z1 + U21);
 S3 :=  s3 *  s3;
  S :=  s0 *  s1;
 cS :=  s1 *  s3;
ccR :=  cR *  cS;
 l2 :=  cS * U21;
 l0 :=   S * U20;
 l1 :=  cS * U20;
 l1 :=   S * U21 + l1;
 l2 :=  s0 *  s3 + l2;
  t :=  s1 *   t;
  t :=  z1 *   t;
U20 :=  s1 *  Z2;
U20 :=   r *  z1 + U20;
U20 :=   R * U20 +   t;
U20 :=  s0 *  s0 + U20;
U20 :=  z2 *  cS + U20;
U21 :=  cS *  z1;
U21 :=   R *   R + U21;
 w0 := U20 *( l2 + U21);
 w0 :=  S3 *  l0 +  w0;
 w1 := U21 *( l2 + U21);
 w1 :=  S3 *(U20 +  l1) + w1;
```

```
U20 :=  cR * U20;
U21 :=  cR * U21;
V20 := ccR * V20 +  w0;
V21 := ccR *(V21 +  Z2) + w1;
 Z2 :=  cR *  S3;

/// HECC divisor doubling sequence (projective to normal)
u20 := U20/Z2;
u21 := U21/Z2;
v20 := V20/Z2;
v21 := V21/Z2;

/// Compare the results with expected values (should be 0 if correct)
u20 + EXPu20;
u21 + EXPu21;
v20 + EXPv20;
v21 + EXPv21;
```

B.6 Weighted Projective Divisor Doubling of HECC over GF(2^m)

```
/// Same hyperelliptic curve as what is used for divisor addition
E := 2*Random(J);

/// Divisor Doubling Test
u20 := Eltseq(E[1])[1];
u21 := Eltseq(E[1])[2];
v20 := Eltseq(E[2])[1];
v21 := Eltseq(E[2])[2];
Z2  := Random(F);

/// Magma Result (Expected Values)
E := 2 * E;
EXPu20 := Eltseq(E[1])[1];
EXPu21 := Eltseq(E[1])[2];
EXPv20 := Eltseq(E[2])[1];
EXPv21 := Eltseq(E[2])[2];

/// Sequence for the MALU
/// HECC divisor doubling sequence (normal to projective)
```

```
U20 := u20 * Z2;
U21 := u21 * Z2;
V20 := v20 * Z2;
V21 := v21 * Z2;

/// HECC divisor doubling sequence (Algorithm 2.15)
 t0 :=  Z2 *  Z2;
 t1 := U21 * U21;
  r := U20 *  Z2;
 k1 :=  f3 *  t0 +  t1;
 k0 := V21 *( Z2 + V21) +  t0;
 k0 :=  Z2 *  k0;
 k0 := U21 *  k1 +  k0;
 t2 :=  k0 * U21;
 s1 :=  k0 *  Z2;
 s0 :=  k1 *   r +  t2;
 t0 :=  t0 *   r;
  r :=  t0 *  s1;
 t1 :=  s1 *  k0;
 t3 := U20 *  k0;
 l2 :=  s1 *  t2;
 l0 :=  s0 *  t3;
 l1 :=  s0 *  t2;
 l1 :=  s1 *  t3 +  l1;
U20 :=  s0 *  s0 +   r;
U21 :=  t0 *  t0;
 l2 :=  s1 *( s0 +  t2) + U21;
 s1 :=  s1 *  s1;
 t2 :=   r *  t1;
 t0 := U20 *  l2;
 t0 :=  l0 *  s1 +  t0;
 t1 := U21 *  l2;
 t1 :=  s1 *(U20 +  l1) +  t1;
 Z2 :=  s1 *   r;
U21 := U21 *   r;
U20 := U20 *   r;
V20 :=  t2 * V20 +  t0;
V21 :=  t2 * V21 +  Z2;
V21 :=  t1 + V21;

/// HECC divisor doubling sequence (projective to normal)
u20 := U20/Z2;
```

```
u21 := U21/Z2;
v20 := V20/Z2;
v21 := V21/Z2;

/// Compare the results with expected values (should be 0 if correct)
u20 + EXPu20;
u21 + EXPu21;
v20 + EXPv20;
v21 + EXPv21;
```

B.7 Projective Point Addition of ECC over GF(2^m) with the Montgomery Powering Ladder

```
RR<T> := PolynomialRing(GF(2));
F<u>  := ext<GF(2) | T^163+T^7+T^6+T^3+1>;
R<x>  := PolynomialRing(F);
E     := EllipticCurve([1,Random(F),0,0,Random(F)]);
P1    := Random(E);
P2    := Random(E);

/// Point Addition Test
x1 := P1[1];
Z1 := Random(F);
P2 := 2 * P1;
x2 := P2[1];
Z2 := Random(F);
x  := Eltseq(P2-P1)[1];

/// Magma Result (Expected Values)
P1 := P1 + P2;
EXPx1 := P1[1];

/// Sequence for the MALU

/// ECCM over GF(2^163) point addition sequence (normal to projective)
X1 := x1 * Z1;
X2 := x2 * Z2;

/// ECCM over GF(2^163) point addition sequence (Algorithm 5.2)
X1 := X1 * Z2;
```

```
Z1 := X2 * Z1;
t1 := X1 * Z1;
Z1 := X1 + Z1;
Z1 := Z1 * Z1;
X1 :=  x * Z1 + t1;

/// ECCM over GF(2^163) point addition sequence (projective to normal)
x1 := X1 / Z1;

/// Compare the results with expected values (should be 0 if correct)
x1 + EXPx1;
```

B.8 Projective Point Doubling of ECC over GF(2^m) with the Montgomery Powering Ladder

```
RR<T> := PolynomialRing(GF(2));
F<u>  := ext<GF(2) | T^163+T^7+T^6+T^3+1>;
R<x>  := PolynomialRing(F);
E     := EllipticCurve([1,Random(F),0,0,Random(F)]);
P1    := Random(E);

/// Point Doubling Test
x1 := P1[1];
Z1 := Random(F);

/// Magma Result (Expected Values)
P1 := 2 * P1;
EXPx1 := P1[1];

/// Sequence for the MALU
b  := Eltseq(E)[5];
/// ECCM over GF(2^163) point doubling sequence (normal to projective)
X1 := x1 * Z1;

/// ECCM over GF(2^163) point doubling sequence (Algorithm 5.2)
t2 := X1 * X1;
t3 := Z1 * Z1;
Z1 := t2 * t3;
t2 := t2 * t2;
t3 := t3 * t3;
```

```
X1 :=  b * t3 + t2;

/// ECCM over GF(2^163) point doubling sequence (projective to normal)
x1 := X1 / Z1;

/// Compare the results with expected values (should be 0 if correct)
x1 + EXPx1;
```

Bibliography

[1] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. A fast elliptic curve cryptosystem. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology – Proc. EUROCRYPT'89*, number 434 in Lecture Notes in Computer Science, pp. 706–708, Springer-Verlag, 1989.

[2] The Computational Algebra Group at the School of Mathematics and Statistics of the University of Sydney. The Magma computational algebra system. `http://magma.maths.usyd.edu.au/magma/`.

[3] L. Batina, N. Mentens, S. B. Örs, and B. Preneel. Serial multiplier architectures over $GF(2^n)$ for elliptic curve cryptosystems. In *Proc. 12th IEEE Mediterranean Electrotechnical Conference (MELECON'04)*, pp. 779–782, IEEE, 2004.

[4] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede. Low-cost elliptic curve cryptography for wireless sensor networks. In L. Buttyán, V. D. Gligor, and D. Westhoff, editors, *Proc. 3rd European Workshop on Security and Privacy in Ad hoc and Sensor Networks (ESAS'06)*, number 4357 in Lecture Notes in Computer Science, pp. 6–17, Springer-Verlag, 2006.

[5] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede. Public-key cryptography on the top of a needle. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS'06)*, pp. 1831–1834, IEEE, 2007.

[6] L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede. Balanced point operations for side-channel protection of elliptic curve cryptography. In *Proc. IEE proceedings – Information Security*, 152(1):57–65, 2007.

[7] L. Batina and G. Muurling. Montgomery in practice: how to do it more efficiently in hardware. In B. Preneel, editor, *Topics in Cryptology – Proc. Cryptographers' Track at the RSA Conference (CT-RSA'02)*, number 2271 in Lecture Notes in Computer Science, pp. 40–52, Springer-Verlag, 2002.

[8] I. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography.* London Mathematical Society Lecture Note Series 265, Cambridge University Press, 1999.

[9] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In W. Fumy, editor, *Advances in Cryptology – Proc. EUROCRYPT'97*, number 1233 in Lecture Notes in Computer Science, pp. 37–51, Springer-Verlag, 1997.

[10] E. Brier, C. Clavier, and F. Oliver. Correlation power analysis with a leakage model. In M. Joye and J.-J. Quisquater, editors, *Proc. 6th International Workshop on Cryptographic Hardware in Embedded Systems (CHES'04)*, number 3156 in Lecture Notes in Computer Science, pp. 16–29, Springer-Verlag, 2004.

[11] B. Byramjee and S. Duquesne. Classification of genus 2 curves over F_{2^n} and optimization of their arithmetic. Cryptology ePrint Archive: Report 2004/107.

[12] R. C. C. Cheung, N. J. Telle, W. Luk, and P. Y. K. Cheung. Customizable elliptic curve cryptosystems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(9):1048–1059, 2005.

[13] J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Ç. K. Koç and C. Paar, editors, *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, number 1717 in Lecture Notes in Computer Science, pp. 292–302, Springer-Verlag, 2007.

[14] F. Crowe, A. Daly, and W. Marnane. A scalable dual mode arithmetic unit for public key cryptosystems. In *Proc. IEEE International Symposium on Information Technology: Coding and Computing (ITCC'05), Vol. 1*, pp. 568–573, IEEE, 2005.

[15] D. Harris, R. Krishnamurthy, M. Anders, S. Mathew, and S. Hsu. An improved unified scalable radix-2 Montgomery multiplier. In *Proc. IEEE Symposium on Computer Arithmetic (ARITH-17)*, pp. 172–178, IEEE, 2005.

[16] J. Daemen and V. Rijmen. *The design of Rijndael: AES – the Advanced Encryption Standard.* Springer-Verlag, 2002.

[17] M. Dichtl and J. Dj. Golic. High-speed true random number generation with logic gates only. In P. Paillier and I. Verbauwhede, editors, *Proc. 9th International Workshop on Cryptographic Hardware in Embedded Systems (CHES'07)*, number 4727 in Lecture Notes in Computer Science, pp. 45–62, Springer-Verlag, 2007.

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

[19] H. Eberle, N. Gura, and S. C. Shantz. Cryptographic processor for arbitrary elliptic curves over GF(2^m). In *Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'03)*, pp. 444–454, IEEE, 2003.

[20] J. Fan, K. Sakiyama, L. Batina, and I. Verbauwhede. FPGA design for algebraic tori based public key cryptography. In *Proc. Design, Automation and Test in Europe Conference (DATE'08)*, 6 pages, ACM, 2008 (to appear).

[21] J. Fan, K. Sakiyama, and I. Verbauwhede. Montgomery modular multiplication algorithm on multi-core systems. In *Proc. IEEE Workshop on Signal Processing Systems: Design and Implementation (SIPS'07)*, pp. 261–266, IEEE, 2007.

[22] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In M. Joye and J.-J. Quisquater, editors, *Proc. 6th International Workshop on Cryptographic Hardware in Embedded Systems (CHES'04)*, number 3156 in Lecture Notes in Computer Science, pp. 357–370, Springer-Verlag, 2004.

[23] G. Frey. How to disguise an elliptic curve (Weil descent). *Presentation given at the 2nd Elliptic Curve Cryptography (ECC'98)*, 1998.

[24] F. Fürbass and J. Wolkerstorfer. ECC processor with low die size for RFID applications. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS'07)*, pp. 1835–1838, IEEE, 2007.

[25] L. Gao, S. Shrivastava, H. Lee, and G. E. Sobelman. A compact fast variable key size elliptic curve cryptosystem coprocessor. In *Proc. Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, pp. 304–305, IEEE, 1999.

[26] G. Gaubatz, J.-P. Kaps, E. Öztürk, and B. Sunar. State of the art in ultra-low power public key cryptography for wireless sensor networks. In *2nd IEEE International Workshop on Pervasive Computing and Communication Security (PerSec'05)*, pp. 146–150, IEEE, 2005.

[27] G. Gaubatz, J.-P. Kaps, and B. Sunar. Public key cryptography in sensor networks – revisited. In C. Castelluccia, H. Hartenstein, C. Paar and D. Westhoff, editors, *Proc. 1st European Workshop on Security in Ad-Hoc and Sensor Networks (ESAS'04)*, number 3313 in Lecture Notes in Computer Science, pp. 2–18, Springer-Verlag, 2004.

[28] GEZEL Version 2. `http://rijndael.ece.vt.edu/gezel2/index.php/Main_Page`.

[29] J. Grosschädl. A bit-serial unified multiplier architecture for finite fields GF(p) and GF(2^n). In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proc. 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, number 2162 in Lecture Notes in Computer Science, pp. 206–223, Springer-Verlag, 2001.

[30] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Proc. 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'04)*, number 3156 in Lecture Notes in Computer Science, pp. 119–132, Springer-Verlag, 2004.

[31] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curves Cryptography.* Springer-Verlag, 2004.

[32] A. Hodjat, L. Batina, D. Hwang, and I. Verbauwhede. HW/SW co-design of a hyperelliptic curve cryptosystem using a microcode instruction set coprocessor. *Elsevier Integration, the VLSI Journal, special issue on Embedded Cryptographic Hardware*, 40(1):45–51, 2006.

[33] A. Hodjat and I. Verbauwhede. Area-throughput trade-offs for fully pipelined 30 to 70 Gbits/s AES processors. *IEEE Transactions on Computers*, 55(4):366–372, 2006.

[34] D. Hwang, K. Tiri, A. Hodjat, B.-C. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. AES-based security coprocessor IC in 0.18-μm CMOS with resistance to differential power analysis side-channel attacks. *IEEE Journal of Solid-State Circuits* , 41(4):781–792, 2006.

[35] IEEE P1363. Standard specifications for public key cryptography, November 2000. `http://grouper.ieee.org/groups/1363/`

[36] T. Itoh and S. Tsujii. Effective recursive algorithm for computing multiplicative inverses in GF(2^m). *Electronics Letters*, 24(6):334–335, 1988.

[37] K. Iwamura, T. Matsumoto, and H. Imai. Systolic-arrays for modular exponentiation using Montgomery method. In R. A. Rueppel, editor, *Advances in Cryptology – Proc. EUROCRYPT'92*, number 658 in Lecture Notes in Computer Science, pp. 477–481, Springer-Verlag, 1992.

[38] T. Izu and T. Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In D. Naccache and P. Paillier, editors, *Proc. 5th International Workshop on Practice and Theory in Public Key Cryptosystems (PKC'02)*, number 3027 in LNCS, pp. 280–296, Springer-Verlag, 2002.

[39] M. Joye. Highly regular right-to-left algorithms for scalar multiplication. In P. Paillier and I. Verbauwhede, editors, *Proc. 9th International Workshop on Cryptographic Hardware in Embedded Systems (CHES'07)*, number 4727 in Lecture Notes in Computer Science, pp. 135–147, Springer-Verlag, 2007.

[40] M. Joye and S.-M. Yen. The Montgomery powering ladder. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, number 2523 in Lecture Notes in Computer Science, pp. 291–302, Springer-Verlag, 2002.

[41] M. E. Kaihara and N. Takagi. Hardware algorithm for modular multiplication/division. *IEEE Transactions on Computers*, 54(1): 12–21, 2005.

[42] Keil, an ARM company. Embedded development tools. `http://www.keil.com/`.

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

[44] N. Koblitz. CM-curves with good cryptographic properties. In J. Feigenbaum, editor, *Advances in Cryptology – Proc. CRYPTO'91*, number 576 in Lecture Notes in Computer Science, pp. 279–287, Springer-Verlag, 1991.

[45] N. Koblitz. *Algebraic Aspects of Cryptography.* Springer-Verlag, first edition, 1998.

[46] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In N. Koblitz, editor, *Advances in Cryptology – Proc. CRYPTO'96*, number 1109 in Lecture Notes in Computer Science, pp. 104–113, Springer-Verlag, 1996.

[47] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – Proc. CRYPTO'99*, number 1666 in Lecture Notes in Computer Science, pp. 388–397, Springer-Verlag, 1999.

[48] P. Kornerup. A systolic, linear-array multiplier for a class of right-shift algorithms. *IEEE Transactions on Computers*, 43(8):892–898, 1994.

[49] S. Kumar and C. Paar. Are standards compliant elliptic curve cryptosystems feasible on RFID? In *the Workshop Record of the ECRYPT Workshop on RFID Security 2006*, 19 pages, 2006.

[50] T. Lange. Formulae for arithmetic on genus 2 hyperelliptic curves. *Applicable Algebra in Engineering, Communication and Computing*, 15(5):295–328, 2005.

[51] Y. K. Lee and I. Verbauwhede. A compact architecture for Montgomery elliptic curve scalar multiplication processor. In S. Kim, M. Yung, and H.-W. Lee, editors, *Proc. 8th International Workshop on Information Security Applications (WISA'07)*, number 4867 in Lecture Notes in Computer Science, pp. 115–127, Springer-Verlag, 2007.

[52] R. Lidl and H. Niederreiter. *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications.* Cambridge University Press, second edition, 2000.

[53] J. López and R. Dahab. Fast multiplication on elliptic curves over GF(2^m). In Ç. K. Koç and C. Paar, editors, *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, number 1717 in Lecture Notes in Computer Science, pp. 316–327, Springer-Verlag, 1999.

[54] J. Lutz and A. Hasan. High performance FPGA based elliptic curve cryptographic co-processor. In *Proc. IEEE International Symposium on Information Technology: Coding and Computing (ITCC'04), Vol. 2*, pp. 486–492, IEEE, 2004.

[55] M. Maurer, A. Menezes, and E. Teske. Analysis of the GHS Weil descent attack on the ECDLP over characteristic two finite fields of composite degree. C. P. Rangan and C. Ding, editors, *Proc. 2nd International Conference on Cryptology in India (INDOCRYPT'01)*, number 2247 in Lecture Notes in Computer Science, pp. 195–213, Springer-Verlag, 2001.

[56] C. McIvor, M. McLoone, J. McCanny, A. Daly, and W. Marnane. Fast Montgomery modular multiplication and RSA cryptographic processor architectures. In *Proc. 37th Annual Asilomar Conference on Signals, Systems and Computers*, pp. 379–384, IEEE, 2003.

[57] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[58] A. Menezes, Y.-H. Wu, and R. Zuccherato. *An Elementary Introduction to Hyperelliptic Curves, Appendix to Algebraic Aspects of Cryptography, by N. Koblitz*, pp. 155–178, Springer-Verlag, 1998.

[59] N. Mentens. *Secure and Efficient Coprocessor Design for Cryptographic Applications on FPGAs.* Ph.D. thesis, Katholieke Universiteit Leuven, Belgium, 2007.

[60] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology – Proc. CRYPTO'85*, number 218 in Lecture Notes in Computer Science, pp. 417–426, Springer-Verlag, 1985.

[61] P. K. Mishra and P. Sarkar. Parallelizing explicit formula for arithmetic in the Jacobian of hyperelliptic curves. In J. Hartmanis G. Goos and J. van Leeuwen, editors, *Proc. ASIACRYPT'03*, number 2894 in Lecture Notes in Computer Science, pp. 93–110, Springer-Verlag, 2003.

[62] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[63] P. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[64] NIST. FIPS PUB 197: Advanced Encryption Standard (AES), November, 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[65] NIST. FIPS PUB 46-3: Data Encryption Standard (DES), October, 1999. `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`

[66] NIST. SP 800-67: Recommendation for the Triple Data Encryption Algorithm (TDEA) block cipher, May, 2004. `http://csrc.nist.gov/publications/nistpubs/800-67/SP800-67.pdf`

[67] K. Okeya, K. Schmidt-Samoa, C. Spahn, and T. Takagi. Signed binary representations revisited. In M. K. Franklin, editor, *Advances in Cryptology – Proc. CRYPTO'04*, number 3152 in Lecture Notes in Computer Science, pp. 123–139, Springer, 2004.

[68] G. Orlando and C. Paar. A high-performance reconfigurable elliptic curve processor for GF(2^m). In Ç. K. Koç and C. Paar, editors, *Proc. 2nd International Workshop on Cryptograpic Hardware and Embedded Systems (CHES'00)*, number 1965 in Lecture Notes in Computer Science, pp. 41–56. Springer-Verlag, 2000.

[69] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18:905–907, 1982.

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

[71] K. Sakiyama, L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede. Small-footprint ALU for public-key processors for pervasive security. In *the Workshop Record of the ECRYPT Workshop on RFID Security 2006*, 12 pages, 2006.

[72] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Superscalar coprocessor for high-speed curve-based cryptography. In Louis Goubin and Mitsuru Matsui, editors, *Proc. 8th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'06)*, number 4249 in Lecture Notes in Computer Science, pp. 415–429, Springer-Verlag, 2006.

[73] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. High-performance public-key cryptoprocessor for wireless mobile applications. *Mobile Networks and Applications (MONET)*, 12(4):245–258, 2007.

[74] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. HW/SW co-design for public-key cryptosystems on the 8051 micro-controller. *Computers & Electrical Engineering*, 33(5-6):324–332, 2007.

[75] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Multicore curve-based cryptoprocessor with reconfigurable modular arithmetic logic units over GF(2^n). *IEEE Transactions on Computers*, 56(9):1269–1282, 2007.

[76] K. Sakiyama, N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. Reconfigurable modular arithmetic logic unit for high-performance public-key cryptosystems. In K. Bertels, J. M. P. Cardoso, and S. Vassiliadis, editors, *Proc. International Workshop on Applied Reconfigurable Computing (ARC'06)*, number 3985 in Lecture Notes in Computer Science, pp. 347–357, Springer-Verlag, 2006.

[77] K. Sakiyama, N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. Reconfigurable modular arithmetic logic unit supporting high-performance RSA and ECC over GF(p). *International Journal of Electronics*, 94(5):501–514, 2007.

[78] K. Sakiyama, E. De Mulder, B. Preneel, and I. Verbauwhede. A parallel processing hardware architecture for elliptic curve cryptosystems. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'06)*, pp. III–904–III–907, IEEE, 2006.

[79] K. Sakiyama, B. Preneel, and I. Verbauwhede. A fast dual-field modular arithmetic logic unit and its hardware implementation. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS'06)*, pp. 787–790, IEEE, 2006.

[80] N. A. Saqib, F. Rodríguez-Henríquez, and A. Díaz-Prez. A reconfigurable processor for high speed point multiplication in elliptic curves. *International Journal of Embedded Systems 2005*, 1, No. 3/4:237–249, 2005.

[81] A. Satoh and K. Takano. A scalable dual-field elliptic curve cryptographic processor. *IEEE Transactions on Computers, special issue on cryptographic hardware and embedded systems*, 52(4):449–460, 2003.

[82] E. Savaş, A. F. Tenca, and Ç. K. Koç. A scalable and unified multiplier architecture for finite fields GF(p) and GF(2^m). In C. Paar and Ç. K. Koç, editors, *Proc. 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'00)*, number 1965 in Lecture Notes in Computer Science, pp. 281–296, Springer-Verlag, 2000.

[83] P. Schaumont and I. Verbauwhede. Interactive cosimulation with partial evaluation. In *Proc. Design, Automation and Test in Europe Conference (DATE'04)*, pp. 642–647, ACM, 2004.

[84] P. Schaumont and I. Verbauwhede. A Component-Based Design Environment for ESL Design. In *IEEE Design & Test of Computers*, 23(5):pp. 338–347, 2006.

[85] C. P. Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology – Proc. CRYPTO'89*, pp. 239–252, Springer-Verlag, 1989.

[86] G. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960.

[87] A. Shamir and E. Tromer. Acoustic cryptanalysis on nosy people and noisy machines. *Preliminary proof-of-concept presentation.* `http://www.wisdom.weizmann.ac.il/~tromer/acoustic/`.

[88] N. P. Smart. The Hessian form of an elliptic curve. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proc. 3rd International Workshop on Cryptograpic Hardware and Embedded Systems (CHES'01)*, number 2162 in Lecture Notes in Computer Science, pp. 121–128, Springer-Verlag, 2001

[89] N. Smyth, M. McLoone, and J. V. McCanny. An adaptable and scalable asymmetric cryptographic processor. In *Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'06)*, pp. 341–346, IEEE, 2006.

[90] J. Solinas. Efficient arithmetic on Koblitz curves. *Journal of Designs, Codes and Cryptography*, 19:195–249, 2000.

[91] F. Sozzani, G. Bertoni, S. Turcato, and L. Breveglieri. A parallelized design for an elliptic curve cryptosystem coprocessor. In *Proc. IEEE International Symposium on Information Technology: Coding and Computing (ITCC'05), Vol. 1*, pp. 626–630, IEEE, 2005.

[92] D. Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. In P. Paillier and I. Verbauwhede, editors, *Proc. 9th International Workshop on Cryptographic Hardware in Embedded Systems (CHES'07)*, number 4727 in Lecture Notes in Computer Science, pp. 272–288, Springer-Verlag, 2007.

[93] Target Compiler Technologies. Retargettable compiler. `http://www.retarget.com/index.html`.

[94] A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. In Ç. K. Koç and C. Paar, editors, *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, number 1717 in Lecture Notes in Computer Science, pp. 94–108, Springer-Verlag, 1999.

[95] A. F. Tenca and Ç. K. Koç. A scalable architecture for modular multiplication based on Montgomery's algorithm. *IEEE Transactions on Computers*, 52(9):1215–1221, 2003.

[96] Tensilica Inc. Configurable and standard processor cores for SOC design. `http://www.tensilica.com/`.

[97] Texas Instruments Inc. Analog Technologies; Semicondictor, Digital Signal Processing. `http://www.ti.com/`.

[98] N. Thériault. Index calculus attack for hyperelliptic curves of small genus. In C. S. Laih, editor, *Proc. Advances in Cryptology – Proc. ASIACRYPT'03*, number 2894 in Lecture Notes in Computer Science, pp. 75–92, Springer-Verlag, 2003.

[99] K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proc. Design, Automation and Test in Europe Conference (DATE'04)*, pp. 246–251, ACM, 2004.

[100] U.S. Department of Commerce and National Institute of Standards and Technology. Digital Signature Standard (DSS), FIPS PUB 186-2, January 2000.

[101] C. D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42:376–378, 1993.

[102] J. Wolkerstorfer. Dual-field arithmetic unit for $GF(p)$ and $GF(2^m)$. In B. S. Kaliski Jr., Ç. Koç, and C. Paar, editors, *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, number 2523 in Lecture Notes in Computer Science, Springer-Verlag, 2002.

[103] T. Wollinger. *Software and Hardware Implementation of Hyperelliptic Curve Cryptosystems*. Ph.D. thesis, Ruhr-University Bochum, Germany, 2004.

[104] Xilinx, Inc. FPGA and CPLD Solutions from Xilinx, Inc. `http://www.xilinx.com/`.

[105] P. Yu, and P. Schaumont. Secure FPGA circuits using controlled placement and routing. In *Proc. IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*, pp. 45–50, ACM, 2007.

List of Publications

International Journals

1. K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Multi-core curve-based cryptoprocessor with reconfigurable modular arithmetic logic units over GF(2^n). *IEEE Transactions on Computers*, 56(9):1269–1282, 2007.

2. K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. High-performance public-key cryptoprocessor for wireless mobile applications. *Mobile Networks and Applications*, 12(4):245–258, 2007.

3. K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. HW/SW co-design for public-key cryptosystems on the 8051 micro-controller. *Computers and Electrical Engineering*, 33(5–6):324–332, 2007.

4. K. Sakiyama, N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. Reconfigurable modular arithmetic logic unit supporting high-performance RSA and ECC over GF(p). *International Journal of Electronics*, 94(5):501–514, 2007.

5. S. Yang, K. Sakiyama, and I. Verbauwhede. Efficient and secure fingerprint verification for embedded devices. *EURASIP Journal on Applied Signal Processing*, 2006:1–11, 2006.

Book Chapters

6. L. Batina, K. Sakiyama and I. Verbauwhede. Architectures for public-key cryptography. In V. G. Oklobdzija, editor, *Digital Systems and Applications*, The Computer Engineering Handbook, Second Edition, CRC press, 2007.

7. L. Batina, K. Sakiyama. Compact public-key implementations for RFID and sensor nodes. In I. Verbauwhede, editor, *Secure Integrated Circuits and Systems*, Springer-Verlag (to appear).

LNCS Publications

8. L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede. Low-cost elliptic curve cryptography for wireless sensor networks. In L. Buttyán, V. Gligor and D. Westhoff, editors, *Third European Workshop on Security and Privacy in Ad hoc and Sensor Networks (ESAS'06)*, number 4357 in Lecture Notes in Computer Science, pp. 6–17, Springer-Verlag, 2006.

9. K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Superscalar coprocessor for high-speed curve-based cryptography. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems (CHES'06)*, number 4249 in Lecture Notes in Computer Science, pp. 415–429, Springer-Verlag, 2006.

10. K. Sakiyama, N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. Reconfigurable modular arithmetic logic unit for high-performance public-key cryptosystems. In K. Bertels, J. M. P. Cardoso and S. Vassiliadis, editors, *International Workshop on Applied Reconfigurable Computing (ARC'06)*, number 3985 in Lecture Notes in Computer Science, pp. 347–357, Springer-Verlag, 2006.

International Conference Articles

11. J. Fan, L. Batina, K. Sakiyama, and I. Verbauwhede, FPGA design for algebraic tori based public key cryptography. In *Proc. Design, Automation and Test in Europe Conference (DATE'08)*, 6 pages, ACM, 2008 (to appear).

12. J. Fan, K. Sakiyama, and I. Verbauwhede, Montgomery modular multiplication algorithm on multi-core systems. In *Proc. IEEE Workshop on Signal Processing Systems (SIPS'07)*, pp. 261–266, IEEE, 2007.

13. N. Mentens, K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. A side-channel attack resistant programmable PKC coprocessor for embedded applications. In *Proc. 2007 International Conference on Systems, Architectures, Modeling and Simulation (IC-SAMOS'07)*, pp. 194–200, IEEE, 2007.

14. L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede. Public-key cryptography on the top of a needle. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS'07), Special Session on Novel Cryptographic Architectures for Low-Cost RFID*, pp. 1831–1834, IEEE, 2007.

15. K. Sakiyama, E. De Mulder, B. Preneel, and I. Verbauwhede. Side-channel resistant system-level design flow for public-key cryptography. In *Proc. 2007*

ACM Great Lakes Symposium on VLSI (GLSVLSI'07), pp. 144–147, ACM, 2007.

16. N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede. Efficient pipelining for modular multiplication architectures in prime fields. In *Proc. 2007 ACM Great Lakes Symposium on VLSI (GLSVLSI'07)*, pp. 534–539, ACM, 2007.

17. L. Batina, A. Hodjat, D. Hwang, K. Sakiyama, and I. Verbauwhede. Reconfigurable architectures for curve-based cryptography on embedded microcontrollers. In *Proc. 16th International Conference on Field Programmable Logic and Applications (FPL'06)*, pp. 667–670, IEEE, 2006.

18. N. Mentens, K. Sakiyama, L. Batina, I. Verbauwhede, and B. Preneel. FPGA-oriented secure data path design: implementation of a public key coprocessor. In *Proc. 16th International Conference on Field Programmable Logic and Applications (FPL'06)*, pp. 133–138, IEEE, 2006.

19. K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. HW/SW codesign for accelerating public-key cryptosystems over GF(p) on the 8051 μ-controller. In *Proc. World Automation Congress (WAC'06), Special Session on Information Security and Hardware Implementations*, 6 pages, 2006.

20. K. Sakiyama, E. De Mulder, B. Preneel, and I. Verbauwhede. A parallel processing hardware architecture for elliptic curve cryptosystems. In *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'06)*, pp. III-904–III-907, IEEE, 2006.

21. K. Sakiyama, B. Preneel, and I. Verbauwhede. A fast dual-field modular arithmetic logic unit and its hardware implementation. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS'06)*, pp. 787–790, IEEE, 2006.

22. P. Schaumont, K. Sakiyama, A. Hodjat, and I. Verbauwhede. Embedded software integration for coarse-grain reconfigurable architectures, In *Proc. IEEE 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, pp. 137–142, IEEE, 2004.

23. S. Yang, K. Sakiyama, and I. Verbauwhede. A secure and efficient fingerprint verification system for an embedded platform. In *Proc. 37th Asilomar Conference on Signals, Systems and Computers*, pp. 2058–2062, IEEE, 2003.

24. P. Schaumont, K. Sakiyama, Y. Fan, D. Hwang, A. Hodjat, B. Lai, I. Verbauwhede. and S. Yang, Testing ThumbPod: softcore bugs are hard to find. In *Proc. IEEE International High Level Design Validation and Test Workshop (HLDVT'04)*, pp. 77–82, IEEE, 2003.

25. D. Hwang, P. Schaumont, Y. Fan, A. Hodjat, B. Lai, K. Sakiyama, S. Yang, and I. Verbauwhede. Design flow for HW/SW acceleration transparency in the Thumbpod secure embedded system. In *Proc. 40th Design Automation Conference (DAC'03)*, pp. 60–65, ACM, 2003 (Honorable Mention).

26. K. Sakiyama, P. Schaumont, D. Hwang, and I. Verbauwhede. Teaching trade-offs in system-level design methodologies. In *Proc. IEEE Microelectronic Systems Education (MSE'03)*, pp. 62–63, IEEE, 2003.

27. K. Sakiyama, P. Schaumont, and I. Verbauwhede. Finding the best system design flow for a high-speed JPEG encoder. In *Proc. 8th Asia and South Pacific Design Automation Conference (ASP-DAC'03)*, pp. 577–578, ACM, 2003.

Patents

28. I. Verbauwhede, P. Schaumont, D. Hwang, B. Lai, S. Yang, K. Sakiyama, Y. Fan, and A. Hodjat. System for biometric signal processing with hardware and software acceleration. Patent number US2007/0038867, University of California (applicant), 2007.

Other Articles

29. J. Fan, K. Sakiyama, and I. Verbauwhede, Elliptic curve cryptography on embedded multicore systems. In *the Workshop Record of the Workshop on Embedded Systems Security (WESS'07)*, 6 pages, 2007.

30. J. Fan, K. Sakiyama, and I. Verbauwhede, Montgomery modular multiplication algorithm for multi-core systems. In *the Workshop Record of the ECRYPT Workshop, Software Performance Enhancement for Encryption and Decryption (SPEED'07)*, 12 pages, 2007.

31. C. Vanderheyden, J. Fan, K. Sakiyama, and I. Verbauwhede, Exploring trade-offs between area, performance and security in HW/SW co-design of ECC. In *the Workshop Record of Western European Workshop on Research in Cryptology (WEWoRC'07)*, 2 pages, 2007.

32. K. Sakiyama, L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede. Small-footprint ALU for public-key processors for pervasive security. In *the Workshop Record of the ECRYPT Workshop on RFID Security 2006*, 12 pages, 2006.

33. L. Batina, S. Kumar, J. Lano, K. Lemke, N. Mentens, C. Paar, B. Preneel, K. Sakiyama, and 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'06)*, 9 pages, 2006.

34. K. Sakiyama, L. Batina, P. Schaumont, and I. Verbauwhede. HW/SW co-design for TA/SPA-resistant public-key cryptosystems. In *the Workshop Record of the ECRYPT Workshop on Cryptographic Advances in Secure Hardware (CRASH'05)*, 8 pages, 2005.

Biography

Kazuo Sakiyama was born in Hyogo, Japan on August 18th in 1971. He received the B.Eng. and M.Eng. degrees in Electrical Engineering from Osaka University, Japan in 1994 and 1996, respectively. From 1996 to 2004, he was with the Semiconductor and IC Division of Hitachi, Ltd. (now Renesas Technology Corp.). During this time, he obtained the M.Sc. degree in Electrical Engineering from the University of California, Los Angeles (UCLA). His master thesis was on secure embedded systems. He started working on a Ph.D. program in the research group COSIC (Computer Security and Industrial Cryptography) at the Katholieke Universiteit Leuven (K.U.Leuven), Belgium in 2005.