RNS-Flexible hardware accelerators for high-security asymmetric cryptography

Libey Djath

► **To cite this version:**

Libey Djath. RNS-Flexible hardware accelerators for high-security asymmetric cryptography. Cryptography and Security [cs.CR]. Université de Bretagne occidentale - Brest, 2021. English. NNT : 2021BRES0030 . tel-03393289

HAL Id: tel-03393289

https://theses.hal.science/tel-03393289

Submitted on 21 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UBO
Université de Bretagne Occidentale

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE
BRETAGNE OCCIDENTALE

ÉCOLE DOCTORALE Nº 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *Informatique*

Par
Libey DJATH

Accélérateurs matériels RNS flexibles pour la cryptographie asymétrique à haute sécurité

RNS-Flexible Hardware Accelerators for High-Security Asymmetric Cryptography

Thèse présentée et soutenue à Brest, le 25 mai 2021
Unité de recherche : Lab-STICC, CNRS UMR 6285

Rapporteurs avant soutenance :

Laurent-Stéphane DIDIER	Professeur, Université de Toulon, IMATH
Christophe NÈGRE	Maître de conférences HDR, Université de Perpignan, LIRMM

Composition du Jury :

Président :	Jean-Claude BAJARD	Professeur, Sorbonne Université, IMJ-PRG
Examinateurs :	Roselyne CHOTIN	Maître de conférences HDR, Sorbonne Université, LIP6
	Laurent-Stéphane DIDIER	Professeur, Université de Toulon, IMATH
	Christophe NÈGRE	Maître de conférences HDR, Université de Perpignan, LIRMM
Encadrant :	Karim BIGOU	Maître de conférences, Université de Bretagne Occidentale, Lab-STICC
Dir. de thèse :	Arnaud TISSERAND	Directeur de recherche CNRS, Lab-STICC

Invité :

Benoît GÉRARD   Expert cryptographie, DGA

In memory of my parents,
who taught me to think for myself

Acknowledgement

I articulate my appreciation to my advisors, Karim Bigou and Arnaud Tisserand, for their guidance throughout the PhD. Besides, they provide valuable feedback which help improve the clarity of this thesis. I also thank Vianney Lapôtre and Laurent-Stéphane Didier for pointing out what is expected from me at the end of the PhD, in our two annual meetings.

For accepting the task of reviewing my PhD thesis, Laurent-Stéphane Didier and Christophe Nègre have my gratitude. I am also appreciative of Jean-Claude Bajard and Roselyne Chotin agreements to examine my PhD.

I thank my family for their support during the PhD and far beyond. I am particularly grateful to Bughsin' and Pidassa for reading this thesis in its entirety, eagerly seeking ill-formed sentences.

Contents

List of Figures 11

List of Tables 15

List of Algorithms 17

List of Abbreviations 19

Introduction 21

1 State of the Art 27

1.1 Asymmetric Cryptography 27
1.1.1 Overview of Asymmetric Cryptography 27
1.1.2 Elliptic Curve Cryptography 30
1.2 Modular Reduction 38
1.2.1 Montgomery Reduction 38
1.2.2 Barrett Reduction 39
1.2.3 On Pseudo-Mersenne Numbers 40
1.3 Residue Number System 41
1.3.1 Overview of the Residue Number System 41
1.3.2 RNS Inherent Properties and Complexity Measurement 44
1.3.3 Base Extension 46
1.3.4 RNS Modular Reduction 51
1.4 Field Programmable Gate Arrays 53
1.4.1 Overview of FPGAs 53
1.4.2 Some RNS Implementations of Asymmetric Cryptosystems on FPGA from the Literature 56
1.4.3 High-Level Synthesis 58
1.5 Conclusion 59

2 Hierarchical Base Extension **61**
2.1 Notations . . . 62
2.2 Kawamura Base Extension . . . 62
2.2.1 Overview of KBE . . . 62
2.2.2 KBE Algorithm . . . 64
2.2.3 The *Cox-Rower* Architecture . . . 66
2.3 Hierarchical Base Extension . . . 67
2.3.1 Overview of HBE . . . 67
2.3.2 HBE Algorithm . . . 68
2.3.3 A *Cox-Rower* Architecture Adapted for HBE . . . 79
2.4 Application to RNS Modular Multiplications . . . 81
2.5 FPGA Implementation Results . . . 84
2.6 Conclusion . . . 85

3 RNS-Flexible Hardware Accelerators for ECC **89**
3.1 Notations and Definitions . . . 91
3.2 Flexible Kawamura Base Extension . . . 92
3.2.1 Algorithmical Description of the Flexible KBE . . . 92
3.2.2 Architecture of the Flexible KBE . . . 92
3.2.3 FPGA Implementation Results . . . 94
3.3 Flexible Hierarchical Base Extension . . . 99
3.3.1 Algorithmical Description of the Flexible HBE . . . 99
3.3.2 Architecture of the Flexible HBE . . . 99
3.3.3 FPGA Implementation Results . . . 103
3.4 Flexible Elliptic Curve Scalar Multiplication . . . 106
3.4.1 Overview of the Flexible ECSM . . . 107
3.4.2 FPGA Implementation Results . . . 112
3.5 Conclusion . . . 124

Conclusion **127**

A Comparaison d’algorithmes de réduction modulaire en HLS sur FPGA **131**

B Generalized Weierstraß Equation of Elliptic Curves **143**

C Proof of the Chinese Remainder Theorem **145**

Résumé substantiel en français **147**

Bibliography **151**

List of Figures

1.1 Example of an elliptic curve defined over two finite fields. 31
1.2 Geometric description of point addition and point doubling. 32
1.3 Hierarchical description of computations involved in an ECSM. 36
1.4 Hierarchical description of computations involved in an ECSM with RNS as chosen number system. 38
1.5 The *cox-rower* architecture in [Gui10], adapted from [KKSS00]. 51
1.6 Simplified view of an FPGA. 54
1.7 Basic functionalities of the DSP slice in 7-series FPGAs from Xilinx (from [Xil18a]). 55
1.8 Excerpt of C code for HLS to explain the multiplication of arrays elements on two channels. Each of the components `mul0` and `mul1` is dedicated to a channel and performs a multiplication of a value received as input by a precomputed value stored internally. The description made in the component `prod` allows the components `mul0` and `mul1` to run in parallel. `#pragma HLS PIPELINE` allows to pipeline the operations within the loop of the component `prod`. 59

2.1 Simplified description of KBE. 63
2.2 The *cox-rower* architecture in [Gui10], adapted from [KKSS00]. 66
2.3 Simplified description of HBE $c = 2$. 68
2.4 Theoretical cost ratio HBE/KBE of one BE for various estimations EMM/CMR$(2w + 1, w)$ and numbers n of moduli. 79
2.5 Architecture of HBE $c = 2$. 80
2.6 Theoretical cost ratio HBE/KBE of one RNS MM, with optimizations from [GLP+12], for various estimations EMM/CMR$(2w + 1, w)$ and numbers n of moduli. 82
2.7 Theoretical cost ratio HBE/KBE of one RNS MM with *HPR* $d = 2$ [BT15] for various estimations EMM/CMR$(2w + 1, w)$ and numbers n of moduli. . 83

2.8 Theoretical cost ratio HBE/KBE of one RNS MM with *HPR* $d = 4$ [BT16] for various estimations EMM/CMR$(2w + 1, w)$ and numbers n of moduli. . 83

2.9 Comparison of the time, the numbers of DSPs and slices between HBE $c = 2$ (from [DBT19]) solutions and KBE (from [KKSS00]) ones. 86

3.1 Decomposition of the numbers n of VCs and q of PCs in usual RNS implementations of the ECSM, that is, $q = n$. In this example, $q = n = 16$. Computations related to one VC is performed on each PC. 90

3.2 Example of decomposition of the numbers n of VCs and q of PCs in RNS implementations of the ECSM when $q = 4$ and $n = 16$. Computations related to $n/q = 4$ VCs are performed on each PC. At a time, each PC deals with one VC, and the 4 VCs are processed one after another. 90

3.3 The architecture for the flexible KBE (adapted from the *cox-rower* architecture [KKSS00]): example with $q = 4$ PCs and $c = 2$. 94

3.4 Structure of the arithmetic processing performed by a *rower*. The *rower* performs the operation $g \leftarrow (g + a \times b + f) \bmod m$, where $m = 2^w - \varepsilon$ (with $\varepsilon < 2^{w/2}$), and a, b, f and g are of size w bits. 95

3.5 Comparison of FPGA implementation results of the flexible KBE (adapted from [KKSS00]) for $q \in \{1, 2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the width of the PCs is 17 bits. 97

3.6 Example of numbers cycles used by one PC to perform all computations related to the $16/q$ VCs it handles. The PC sequentially 'uploads' the $16/q$ VCs. Computations on the same PC are pipelined. In this typical example of loops at lines 1–4 of Algorithm 13, the computations related to the $rc = 16$ VCs are equally shared by the q PCs in use. At the left figure, $q = 2$ PCs are used and each PC handles 8 VCs. At the first cycle (top to bottom), computations related to the first VC start. At the next cycle, computations related to the next VC start, and so on until computations related to all 8 VCs (mapped onto the PC) start. Meanwhile, it is possible that the PC has terminated the computations related to the VCs that started earlier. Still, a few cycles are needed by the PC to terminate the remaining computations related to its ongoing VCs, that is, to empty out the pipeline. At the right figure, $q = 4$ PCs are used and each PC handles 4 VCs. The filling and emptying of the pipeline are similar to the ones at the left figure. 98

3.7 The flexible architecture for HBE (adapted from the HBE architecture in [DBT19], itself inspired by the *cox-rower* architecture [KKSS00]): example with $q = 4$ PCs and $c = 2$. 101

3.8 Structure of the upper part of the arithmetic processing performed by a *rower* in the flexible HBE ($c = 2$). The upper part of the *rower* performs the operation $G \leftarrow G + a \times b + f$, where a, b and f are of size w bits, and G of size at most $2w + 1$ bits. 102

3.9 Structure of the lower part of the arithmetic processing performed by a *rower* in the flexible HBE ($c = 2$). The lower part of the *rower* performs the operation $g \leftarrow G \bmod m$, where $m = 2^w - \varepsilon$ (with $\varepsilon < 2^{w/2}$), g is of size w bits, and G of size at most $2w + 1$ bits. 102

3.10 Comparison of FPGA implementation results of the flexible HBE (adapted from [DBT19]) for $q \in \{2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the width of the PCs is 17 bits. 104

3.11 Comparison of the time, the numbers of cycles, DSPs and slices between the flexible HBE solutions and the flexible KBE ones for various numbers of PCs. HBE is not implemented for one PC. 106

3.12 Hierarchical description of RNS computations of the ECSM. The ECSM is computed from a few hundreds point doublings and additions. If Algorithm 15 of the Mongomery ladder [Mon87] is used, each iteration of its loop requires two dozens basic operations and a dozen RNS mod p reductions. Each mod p reduction is made of two BEs and a few basic operations. The cost of the ECSM is mainly driven by that of the mod p reduction, which is itself driven by that of the BEs. 107

3.13 Comparison of FPGA implementation results of the flexible ECSM-KBE for $q \in \{1, 2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the PCs 17-bit wide. 114

3.14 Comparison of FPGA implementation results of the flexible ECSM-HBE for $q \in \{2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the PCs 17-bit wide. 117

3.15 Comparison of the time, the numbers of cycles, DSPs and slices between the flexible ECSM-HBE solutions and the flexible ECSM-KBE ones for various numbers of PCs. ECSM-HBE is not implemented for one PC. . . . 119

3.16 Comparison of the area and area × time trade-offs of our ECSM implementations with the one in [BM14] and the best one in [MLPJ13]. ECSM implementations from [BM14, MLPJ13], and generally from the literature, are not *flexible*. 123

A.1 Extraits de codes C de notre bibliothèque (haut) et de son utilisation (bas). 136
A.2 Comparaison des différents algorithmes de réduction pour $w \in \{13, 17, 23, 30\}$ bits pour le motif M2 RSF avec $N = 20$. 138
A.3 Impact de la taille N des vecteurs pour MSR avec $w = 23$. 142

List of Tables

1.1 Some RNS implementations of asymmetric cryptosystems on FPGA from the literature. The size$^{(\star)}$ is the operand size (for example the modulus size for RSA or the size of the underlying field of the curve for ECC). 57

2.1 FPGA implementation results for CMR($2w+1, w$) and EMM operations on a Xilinx XC7Z020 FPGA. 79

2.2 HLS implementation results on a XC7Z020 FPGA for HBE $c = 2$ (from [DBT19]) and KBE (from [KKSS00]) algorithms for two widths of prime field elements and four RNS channel widths w. 85

3.1 HLS implementation results on a XCZU7EV-FFVC1156 FPGA for the flexible KBE (adapted from [KKSS00]) using $q \in \{1, 2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the width of the PCs is 17 bits. 96

3.2 HLS implementation results on a XCZU7EV-FFVC1156 FPGA for the flexible HBE (adapted from [DBT19]) using $q \in \{2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the width of the PCs is 17 bits. 103

3.3 Comparison of FPGA implementation results of the flexible HBE (adapted from [DBT19]) with those of the flexible KBE (adapted from [KKSS00]) for various number of PCs. The flexible HBE is not implemented for one PC. Finite fields are of 256-bit elements and the width of the PCs is 17 bits. 105

3.4 Generalized Montgomery curve formulas to curves in short Weierstraß equation [BJ02, IT02]. The formulas are in projective coordinates. The value $\mathrm{x_Q}$ denotes the affine x-coordinate of Q, the input point of Algorithm 15 about the Montgomery ladder. 108

3.5 Steps to compute point addition and doubling formulas from [BJ02, IT02] reported in Table 3.4. 108

3.6 HLS implementation results on a XCZU7EV-FFVC1156 FPGA for the ECSM-KBE using $q \in \{1, 2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the PCs 17-bit wide. 113

3.7 HLS implementation results of the ECSM-KBE according to introduced metrics of area *vs.* time trade-offs for $q \in \{1, 2, 4, 8, 16\}$ PCs. 115

3.8 HLS implementation results on a XCZU7EV-FFVC1156 FPGA for the ECSM-HBE using $q \in \{2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the PCs 17-bit wide. 116

3.9 HLS implementation results of the ECSM-HBE according to introduced metrics of area *vs.* time trade-offs for $q \in \{2, 4, 8, 16\}$ PCs. 117

3.10 Comparison of flexible ECSM-HBE solutions with flexible ECSM-KBE ones for the various numbers of PCs. ECSM-HBE is not implemented for one PC. 118

3.11 Comparison of our FPGA implementation results of the ECSM with the ones from the literature. 122

A.1 Impact des directives d'optimisation pour M2 RSF, MSR, $w = 23$ et $N = 20$.139

A.2 Impact des stratégies de réduction pour $w = 23$ et $N = 20$. 140

LIST OF ALGORITHMS

1 Diffie-Hellman key-exchange protocol [DH76]. 28
2 Private- and public-key generation in RSA [RSA78]. 29
3 Encryption and decryption procedures in RSA [RSA78]. 30
4 Elliptic curve version of the Diffie-Hellman key-exchange protocol [Mil85]. . 35
5 *Double-and-add* algorithm (see, for example [HMV04]). 36
6 Montgomery ladder algorithm [Mon87]. 37
7 Montgomery reduction algorithm [Mon85]. 39
8 RNS-to-MRS conversion algorithm [Val56, Gar59]. 47
9 Base extension $BE_{\mathcal{M}\to\mathcal{M}'}(x)$ [KKSS00]. 50
10 RNS Montgomery reduction algorithm [PP95]. 52

11 Kawamura base extension $BE_{\mathcal{M}\to\mathcal{M}'}(x)$ adapted from [KKSS00] 65
12 Hierarchical base extension $BE_{\mathcal{M}\to\mathcal{M}'}(x)$ [DBT19] 69

13 Flexible KBE $BE_{\mathcal{M}\to\mathcal{M}'}(x)$ adapted from [KKSS00]. The function $f(v,i)$ is given by $f(v,i) = \frac{q}{c}(v-1)+i$. 93
14 Flexible HBE $BE_{\mathcal{M}\to\mathcal{M}'}(x)$ adapted from [DBT19]. The function $f(v,i)$ is given by $f(v,i) = \frac{q}{c}(v-1)+i$. 100
15 Montgomery ladder algorithm [Mon87] 108
16 RNS Montgomery reduction algorithm [PP95] 109
17 Flexible basic operation in the RNS base $\mathcal{M}$. The function $f(v,i)$ is given by $f(v,i) = \frac{q}{c}(v-1)+i$. 110

List of Abbreviations

ADD	elliptic curve point addition
ANSSI	Agence nationale de la sécurité des systèmes d'information (France)
ASIC	application specific integrated circuit
BE	base extension
BRAM	block RAM
CAD	computer-aided design
CADD	cost of an addition/subtraction
CLB	configurable logic block
CMUL	cost of a multiplication
CMR	cost of a modular reduction
CRT	Chinese remainder theorem
DBL	elliptic curve point doubling
DCM	digital clock manager
DLP	discrete logarithm problem (in finite fields)
DPA	differential power analysis
DSA	digital signature algorithm
DSP slice	digital signal processing slice
ECC	elliptic curve cryptography
ECDH	elliptic curve Diffie-Hellman protocol
ECDLP	elliptic curve discrete logarithm problem
ECDSA	elliptic curve digital signature algorithm
ECSM	elliptic curve scalar multiplication
EMA	elementary modular addition
EMM	elementary modular multiplication
FF	flip-flop
FPGA	field programmable gate array
HBE	hierarchical base extension
HDL	hardware description language
HECC	hyperelliptic curve cryptography

HLS	high-level synthesis
HPR	modular multiplication algorithm proposed in [BT15, BT16]
HTTPS	hypertext transfer protocol secure
I/O	input/output
KBE	Kawamura base extension
LSB	least significant bit
LUT	look-up table
MM	modular multiplication
MR	modular reduction
MRS	mixed-radix system
MSB	most significant bit
NIST	National Institute of Standards and Technology (United States)
OC	number of operation cycles
PC	physical channel
RAM	random access memory
RNS	residue number system
RSA	Rivest Shamir Adleman (the cryptosystem proposed in [RSA78])
RTL	register transfer level
SPA	simple power analysis
TLS	transport layer security
VC	virtual channel
VHDL	VHSIC hardware description language
VHSIC	very high speed integrated circuits
WC	number of wait cycles

Introduction

Context

Hardware accelerators of asymmetric cryptosystems with flexible utilization of resources are proposed in this thesis. Asymmetric cryptography is used in various applications in order to, among others, securily exchange a secret key, digitally sign documents or authenticate for example, to a server. For instance, the transport layer security (TLS) protocol, used notably in the hypertext transfer protocol secure (HTTPS), relies on asymmetric cryptography for the server/client authentication during the TLS handshake. Asymmetric cryptosystems are implemented in many devices used on a daily basis such as smartphones, personal computers, TV sets and smart cards. These devices have various security requirements to protect the sensitive data they handle. The implementations of the cryptosystems have to be secure and efficient to protect the involved data and to avoid poor performance of the devices.

Computations in current asymmetric cryptosystems involve large operands. For example, operands in RSA [RSA78] are integers of more than 2000-bit size, and operands in *elliptic curve cryptography* (ECC) [Mil85, Kob87] are finite field elements of more than 200-bit size. Therefore, an efficient arithmetic suitable for large operands is needed.

The *residue number system* (RNS) [Val56, Gar59] is a nonpositional number system wherein large operands are represented by their residues over a set (called base) of small coprime moduli m_i (a few dozens-bit size). Basic operations such as multiplication, addition and subtraction are *independently* performed on the small residues. Computations on large operands are then replaced by parallel computations on small operands, leading to faster basic operations. This independence also induces carry-free operations between the moduli [ST67]. These advantages recently motivate uses of RNS in implementations of asymmetric cryptosystems; see, for example [NMSK01, SFM$^+$09, Gui10, BM14].

On the drawback side, position-related operations such as modular reductions (MRs), divisions and comparisons are difficult since the representation is nonpositional. Indeed, the order of magnitude of operands is more difficult to evaluate than in a positional representation.

Numerous MRs have to be performed in current asymmetric cryptosystems owing to

the modular operations therein. For instance, the main operation in RSA is the modular exponentiation. For ECC applications, the main operation is the elliptic curve scalar multiplication (ECSM), which itself is computed through numerous operations on finite field elements. To reduce the cost of the RNS MR in [PP95], the base extension (BE) proposed in [KKSS00] is used. The BE becomes an important operation in RNS implementations of asymmetric cryptosystems because the cost of the RNS MR is substantially the cost of the two BEs it comprises.

A channel is the hardware support of basic operations on small residues modulo an element m_i of the RNS base. In current RNS implementations of asymmetric cryptosystems (see, for example [NMSK01, SFM$^+$09, Gui10, BM14]), the number of used channels is the number of moduli needed to represent the large operands. The quantity of hardware resources used in implementations is then related to the size of the large operands involved in the computations of asymmetric cryptosystems. The quantity of needed hardware resources becomes a problem whenever it is greater than the one available on the integrated circuit.

Most integrated circuits are primarily used for noncryptographic applications such as signal processing. After taking into account the hardware resources for the primary applications, the remaining hardware resources can be insufficient to implement the desired asymmetric cryptosystem. In such cases, the circuit designer is forced to choose between lowering the performance of the primary applications and reducing the level of security needed to protect the involved data. However, in many domains such as defense or aerospace industries, this choice is not acceptable.

The lack of hardware resources (on an integrated circuit) to implement a desired asymmetric cryptosystem can also occur over time. The recommended levels of security for asymmetric cryptosystems increase over some time. This increase is usually followed by an increase in the sizes of the large operands involved in computations of asymmetric cryptosystems. For instance, the recommended minimal level of security for ECC applications grows recently from 80 bits to 128 bits, resulting in a growth of the minimal size of finite field elements from 160 bits to 256 bits; see, for example [oST09, ndlsdsd11, oST13]. The increase in the sizes of the operands is translated into an increase in the quantity of hardware resources needed for implementing the desired cryptosystem on the same device.

An integrated circuit (reconfigurable) on which is implemented an asymmetric cryptosystem can lack hardware resources for a new implementation of the same cryptosystem but with larger operands that guarantee a greater level of security. To maintain the per-

formance of other applications and upgrade to a greater level of security, the designer has to acquire a new integrated circuit larger than his/her previous one in use. For most designers, this solution is not desirable owing to its financial cost.

Designing hardware accelerators for asymmetric cryptosystems with resource utilization adaptable to the resources available on an integrated circuit is a solution to the mentioned problem. We refer to such accelerators as *flexible* hardware accelerators. In RNS implementations of asymmetric cryptosystems, such accelerators can be achieved by using *fewer* channels than normally needed to perform computations on the large operands. In this case, we refer to these fewer channels as physical channels (PCs) to differenciate them from the in-number normally needed channels that we name virtual channels (VCs).

The work presented in this thesis primarily targets ECC applications. However, some of the contributions are also adaptable to other asymmetric cryptosystems such as RSA.

To evaluate our propositions and compare our solutions with the state-of-the-art ones whenever the latter are existent, we implemented the propositions on *field programmable gate arrays* (FPGAs) using *high-level synthesis* (HLS) tools. FPGAs are integrated circuits configurable after manufacture. Their reconfigurability and lower costs (in design complexity and time, financial cost per unit [1]) compared with ASICs [2] are reasons that motivate their use for hardware implementations—generally, and particularly in the work presented in this thesis—despite their lower performance (see, for example [ST12]). HLS allows to use a high-level description (for example in C or C++) to automatically describe a register transfer level (RTL) design in a hardware description language (HDL). Compared with HDLs, HLS facilitates fast configurations of FPGAs; more configuration details are handled by the HLS tools.

Contributions

We have investigated two aspects of the BE. The choice of BE as our primary subject of investigation is motivated by the fact that the BE is the most important operation in the ECSM when targeting RNS implementations. One investigated aspect of the BE is its speed. A new BE algorithm with a theoretical cost smaller than that of the state-of-the-art

1. According to [ST12], the manufacturing cost of FPGAs is paid off by the numerous clients owing to the large quantity in which they are usually produced. For ASICs [2] to become profitable, they must be produced in extremely large quantity because of the important cost in human resources, equipement, design complexity and time they require.

2. ASICs stand for application specific integrated circuits.

algorithm will ultimately result in performance improvements of RNS implementations of the ECSM. The other invertigated aspect is the flexibility of the BE and the translation of the latter to the ECSM. In RNS implementations of the ECSM, the BE is not only the most important operation, but also the most complex one. Implementing a flexible BE is therefore essential in implementing an RNS-flexible ECSM.

The first contribution of this thesis is a new BE algorithm [DBT19], named *hierarchical base extension* (HBE). HBE relies on a hierarchical approach for computing the Chinese remainder theorem (CRT). The approach comprises two phases. In the first phase, the input residues are combined by pairs through computations of partial CRTs in the input RNS base. In the second phase, the remaining of the CRT computation is proceeded on the results of these partial CRTs in the output RNS base. The theoretical cost of the HBE algorithm is smaller than that of the BE algorithm proposed in [KKSS00], named KBE (in this thesis) and largely regarded as the state-of-the-art BE. This cost reduction translates to a cost reduction of the RNS MR operation using HBE compared with the one using KBE. To exploit the inherent parallelism of RNS well preserved in the *cox-rower* architecture proposed in [KKSS00], the latter is adapted to support HBE. HLS implementations on FPGA of the two algorithms for various finite field sizes and channel widths show that HBE solutions are always faster *and* in nearly all cases smaller than KBE ones. The area *vs.* time trade-off is *always* in favor of HBE. As a consequence, similar performance improvements are expected in ECSM implementations using HBE compared with the ones using KBE; this expectation is verified in the next contribution.

[DBT19] L. Djath, K. Bigou and A. Tisserand. Hierarchical approach in RNS base extension for asymmetric cryptography. In *26th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 46–53. IEEE, 2019.
DOI: 10.1109/ARITH.2019.00016
Also available from `https://hal.archives-ouvertes.fr/hal-02096353`

The second contribution of this thesis is two RNS-flexible hardware accelerators for the ECSM on FPGA. The BE operation, either HBE [DBT19] or KBE [KKSS00], is demonstrated to be *flexible*. In other words, for a given size of finite field, each of the two BEs can be implemented using a flexible quantity of hardware resources. An architecture is derived for each of the flexible HBE and KBE. The number of PCs in each architecture is flexible. The two BEs are implemented on FPGA with HLS for various numbers of PCs ranging between the divisors of the number of VCs. Implementation solutions show

that when the number of used PCs increases from q to kq with $k > 1$, the solution time decreases with a factor less than k, regardless of the BE. This observation results from the constancy of the number of wait cycles between pipelined loops independently of the number of used PCs. The solutions with the best area *vs.* time trade-offs are obtained with implementations using a few PCs. HBE solutions remain faster with better area *vs.* time trade-offs than KBE ones.

The new flexible architectures are then used to implement two flexible ECSMs, one using HBE [DBT19] and the other KBE [KKSS00]. These flexible ECSM implementations use adaptable quantities of hardware resources. The used quantities of hardware resources depend on the number of PCs chosen (at design time) for the RNS implementation of the ECSMs. Similarly to the flexible BEs, our flexible ECSM solution time decreases with a factor less than k when the number of used PCs increases from q to kq, regardless of the BE in use. Again, implementations using a few PCs present the best solutions in area *vs.* time trade-offs. Also, the ECSM solutions using HBE are faster *and* present better area *vs.* time trade-offs than the ones using KBE. Last, though comparable in area *vs.* time trade-offs with the best implementation results from the literature, most of our flexible ECSM solutions are much smaller. Therefore, our flexible ECSM solutions can be implemented on integrated circuits with limited hardware resources. The proposed RNS-flexible architectures can also be used to implement multi-level security hardware accelerators. We plan to submit this contribution in the near future.

The last contribution of this thesis is an auxiliary project started in the early months of the PhD. We studied how to efficiently perform modular multiplications and accumulations (typical operations in RNS implementations of asymmetric cryptosystems) on FPGA using HLS tools. HLS favors fast explorations of different parameters for various implementation constraints (for example, loop unrolling). Parameters considered in our study include the size and the shape (arbitrary, specific) of the moduli as well as the type of series (with and without intermediate MR) for the modular operations. The report of this study, presented in [DZBT19] in French, is put in Appendix A.

[DZBT19] L. Djath, T. Zijlstra, K. Bigou and A. Tisserand. Comparaison d'algorithmes de réduction modulaire en HLS sur FPGA. In *Conférence d'informatique en Parallélisme, Architecture et Système* - Compas'19, 2019.
Also available from `https://hal.archives-ouvertes.fr/hal-02129095`

Outline

This thesis comprises three chapters. In the first chapter, we detail the context of this thesis, namely ECC. The ECSM and computations involved in its computation are also introduced. Besides ECC in the first chapter, we present the RNS and its inherent properties as well as a survey of BEs and RNS MRs. The first and the second contributions of this thesis are respectively presented in the second and the third chapters.

Chapter 1

State of the Art

1.1 Asymmetric Cryptography

1.1.1 Overview of Asymmetric Cryptography

Asymmetric cryptography is based on the idea of publicizing a key. In asymmetric cryptography, each individual possesses two keys: a private key, and a public key usually computed from the private one. The private key is known only by its owner while the public key is revealed.

The safety of revealing the public key without giving information about the private key is associated with the hardness of inverting one-way functions. A function $f : x \mapsto f(x)$ is one way if computing $f(x)$ from x is *easy* while computing x from $f(x)$ is *hard*. Easy means computing $f(x)$ from x can be performed in polynomial time. Hard means succeeding in computing x from $f(x)$ in polynomial time for all attempting probabilistic algorithms is extremely unlikely. However, proving the hardness of inverting functions is difficult in itself. In practice, functions largely regarded as computationally difficult to invert are used. A practical way to view the hardness: If computing x from $f(x)$ takes considerable amount of time (say, hundreds of years) with the best known algorithms and a huge computation power, then the function f is largely regarded as hard to invert.

In asymmetric cryptography, the hardness should only exists for an eavesdropper. An authorized individual should be able to efficiently compute x from $f(x)$. This ability is usually provided by a trap door that the authorized individual possesses. For instance, the private key of the authorized individual is used as the trap door in an asymmetric cryptosystem.

Applications of asymmetric cryptography include key exchange [DH76] and digital signature [RSA78, Elg85]. Applications of asymmetric cryptography are used in many practical contexts such as in establishing secure communications with an e-commerce website or in authenticating to a server.

Diffie-Hellman key-exchange protocol and the RSA cryptosystem, two notable advancements in asymmetric cryptography, are presented in the following subsections. Elliptic curve cryptography (ECC) is then described.

Diffie-Hellman Key-Exchange Protocol [DH76]

The safety of the Diffie-Hellman key-exchange protocol [DH76] is based on the assumption that the discrete logarithm problem (DLP) in finite field is computationally difficult for well chosen parameters. The parameters to consider include, among others the size of the field, and tailored properties against known attacks such as the Pohlig-Hellman attack [PH78]. Current recommended sizes are a few thousands bits (say, 2048 or 3072 bits); see for example [oST13].

Let p be a prime, g be a primitive element of the finite field $\mathbb{F}_p$. The DLP can be stated: Knowing $n = g^x \bmod p$, find x. The function

$$f : x \mapsto n = g^x \bmod p$$

can be viewed as a one-way function associated with the DLP.

Two individuals, Alice and Bob, want to share a common key over a network. The Diffie-Hellman key-exchange protocol [DH76] is described in Algorithm 1. At the end of the exchange Alice and Bob have a common secret key $K_{AB} = K_{BA}$ since $n_B^{x_A} \bmod p = n_A^{x_B} \bmod p = g^{x_A x_B} \bmod p$. An eavesdropper has to solve the DLP if he/she intends to recover the secret operands x_A or x_B from n_A or n_B. Besides, computing the shared key K_{AB} from n_A or n_B without knowledge of respectively x_B or x_A involves solving the DLP.

Algorithm 1: Diffie-Hellman key-exchange protocol [DH76].

Input: p a prime, and g a primitive element of $\mathbb{F}_p$
Output: Alice and Bob have a common key $K_{AB} = K_{BA} = g^{x_A x_B} \bmod p$

1. Alice chooses a random number $x_A \in \mathbb{F}_p$ and keeps it secret
2. Alice computes $n_A = g^{x_A} \bmod p$ and sends it to Bob
3. Bob chooses a random number $x_B \in \mathbb{F}_p$ and keeps it secret
4. Bob computes $n_B = g^{x_B} \bmod p$ and sends it to Alice
5. Alice receives n_B and computes $K_{AB} = n_B^{x_A} \bmod p$
6. Bob receives n_A and computes $K_{BA} = n_A^{x_B} \bmod p$

RSA Cryptosystem [RSA78]

The RSA cryptosystem [RSA78] was the first published work allowing two individuals to exchange a message over a network using a pair of keys (private and public). In RSA, the safety of keeping secret the private key in spite of revealing the public key relies on the considered difficulty of factoring large integers.

The problem of the factorization can be put: Knowing an integer n, factorize n. The function

$$f : (p, q) \mapsto n = p \times q$$

can be viewed as a one-way function associated with the problem of the factorization. Computing n is straightforward (through a multiplication algorithm) but computing p and q from n requires considerably more effort. In practice, an attacker has to factorize a number n of a few thousands bits. For example, the recommended sizes for the RSA modulus n in [oST13] are 1024, 2048 and 3072 bits.

Alice wants to generate her private key (d, n) and her public key (e, n). The simplified procedure is presented in Algorithm 2. For an eavesdropper, computing the decryption key d from the encryption key e requires to factor n.

Bob wants to send a message (also called plaintext) m, $0 \leq m \leq n - 1$ to Alice. Bob gets the public key (e, n) of Alice. The encryption and decryption procedures are described in Algorithm 3.

At the end, Alice gets m since $c^d \bmod n = (m^e)^d \mod n = m$. An eavesdropper who gets the ciphertext c has to solve the following problem: Find the eth root of $c \bmod n$. This problem is also considered to be computationally difficult. By assumption, the eavesdropper does not possess the decryption key d of Alice. Strictly speaking, an eavesdropper able to solve one of the factorization problem and the eth root mod n problem can recover the plaintext m.

Algorithm 2: Private- and public-key generation in RSA [RSA78].

Output: private key (d, n) and public key (e, n)

1. Alice chooses two random large primes p and q and computes $n = pq$
2. Alice chooses a random large integer d coprime with $(p - 1)(q - 1)$
3. Alice computes e, the inverse of d mod $(p - 1)(q - 1)$
4. Alice keeps (d, n) secret and publicizes (e, n)

Algorithm 3: Encryption and decryption procedures in RSA [RSA78].

Input: Bob gets the public key (e, n) of Alice with the intention of sending her a plaintext m

Output: Alice recovers m

1 **Encryption**

2 Bob computes $c = m^e \bmod n$ and sends c to Alice

3 **Decryption**

4 Alice receives c and computes $c^d \bmod n = m$

1.1.2 Elliptic Curve Cryptography

Elliptic Curves

Part "Elliptic Curves" of the current subsection is a synthesis of basic concepts on elliptic curves from [Hus04, CF06, Sil09] useful in the work presented in this thesis.

An elliptic curve is a nonsingular cubic curve along with a stated base point. In this thesis the following simplified definition is considered, though restrictive on the characteristic of the underlying field of the curve.

Definition 1 (Elliptic Curve). Let K be a field of characteristic $\neq 2, 3$. An *elliptic curve* E_K defined over K is the set of elements $(x, y) \in K^2$ satisfying the equation

$$E_K : \quad y^2 = x^3 + ax + b \tag{1.1}$$

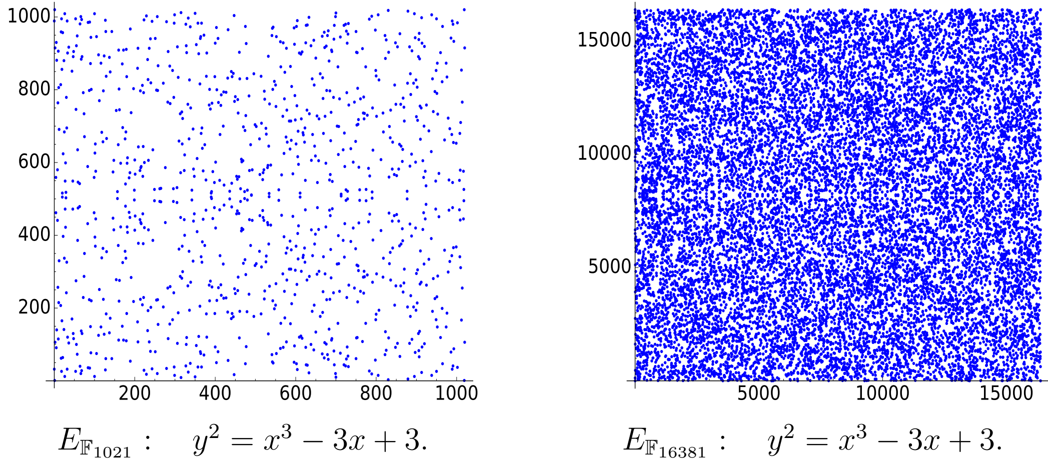
where $a, b \in K$ are such that $-16(4a^3 + 27b^2) \neq 0$.

The quantity $\Delta = -16(4a^3 + 27b^2)$ is the discriminant of Equation 1.1 and $\Delta \neq 0$ conveys the nonsingularity of the curve E_K.

The elements of the elliptic curve E_K are called points. Figure 1.1 depicts the points of an elliptic curve defined over the finite fields $\mathbb{F}_{1021}$ and $\mathbb{F}_{16381}$.

Equation 1.1 is the short Weierstraß equation of an elliptic curve. The generalized Weierstraß equation of an elliptic curve defined over a field of arbitrary characteristic can be found in [Hus04, CF06, Sil09], and is presented in Appendix B.

Two points Q and Q' of an elliptic curve E_K can be added by drawing a line (QQ') through them and taking the symmetric (in relation to the x-axis) of the third point of intersection between the line (QQ') and the curve E_K. The point addition of Q and Q' is denoted $Q+Q'$. The addition of Q and Q is referred to as point doubling and denoted $[2]Q$. The construction of the point $[2]Q$ is similar to that of the point $Q+Q'$, the tangent at Q



$E_{\mathbb{F}_{1021}}: \quad y^2 = x^3 - 3x + 3.$ $E_{\mathbb{F}_{16381}}: \quad y^2 = x^3 - 3x + 3.$

Figure 1.1 – Example of an elliptic curve defined over two finite fields.

to the curve replacing the line (QQ'). Constructions of point addition and doubling are illustrated on the curve $E_{\mathbb{R}} : y^2 = x^3 - 3x + 3$ in Figure 1.2.

Point addition $Q+Q'$ and point doubling $[2]Q$ are also obtained through formulas

$$\begin{cases} x_{Q+Q'} = \lambda^2 - x_Q - x_{Q'} \\ y_{Q+Q'} = \lambda(x_Q - x_{Q+Q'}) - y_Q \end{cases} \quad \text{and} \quad \begin{cases} x_{[2]Q} = \mu^2 - 2x_Q \\ y_{[2]Q} = \mu(x_Q - x_{[2]Q}) - y_Q \end{cases} \tag{1.2}$$

where $\lambda = \dfrac{y_Q - y_{Q'}}{x_Q - x_{Q'}}$ and $\mu = \dfrac{3x_Q^2 + a}{2y_Q}$.

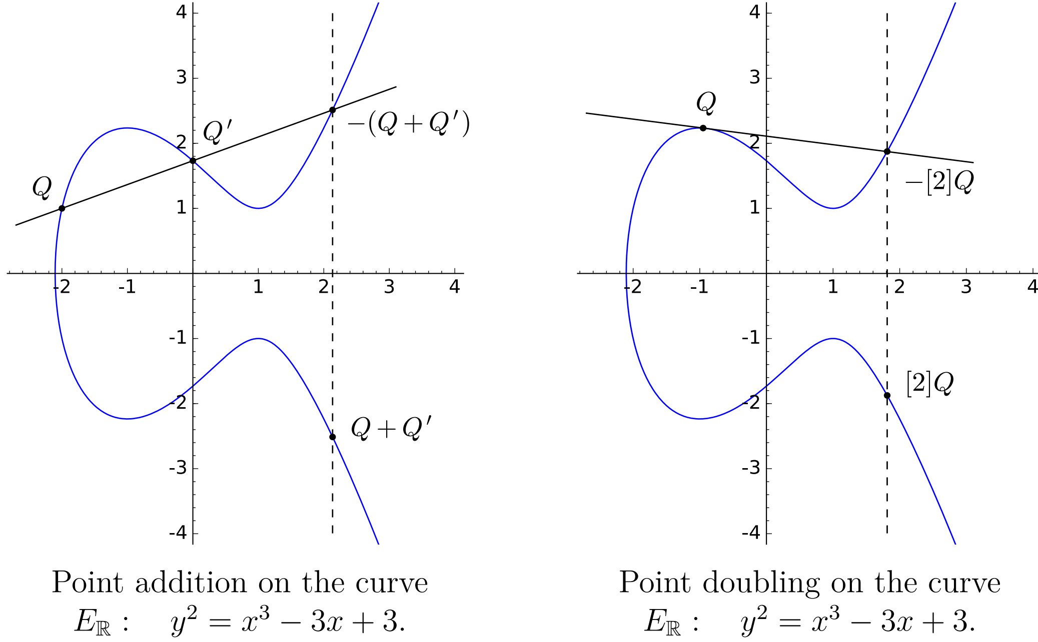
Formulas in Equation 1.2 include divisions (for the introduced values [1] λ and μ). In general, divisions are difficult to perform. A solution to avoid computing the inherent divisions of Equation 1.2 is to homogenize the equation of the curve E_K, that is, to express it in projectives coordinates. By substituting $x = X/Z$ and $y = Y/Z$ into Equation 1.1 we obtain

$$E_K : \quad Y^2Z = X^3 + aXZ^2 + bZ^3. \tag{1.3}$$

Using projective coordinates to express the formulas in Equation 1.2 yields

$$\begin{cases} X_{Q+Q'} = BC \\ Y_{Q+Q'} = A(X_Q Z_{Q'} B^2 - C) - Y_Q Z_{Q'} B^3 \\ Z_{Q+Q'} = Z_Q Z_{Q'} B^3 \end{cases} \quad \text{and} \quad \begin{cases} X_{[2]Q} = 2FH \\ Y_{[2]Q} = E(4G - H) - 8Y_Q^2 F^2 \\ Z_{[2]Q} = 8F^3 \end{cases} \tag{1.4}$$

1. The values λ and μ are the slopes of the line (QQ') and the tangent at Q to the curve E_K respectively.



Point addition on the curve $E_{\mathbb{R}}: \quad y^2 = x^3 - 3x + 3.$

Point doubling on the curve $E_{\mathbb{R}}: \quad y^2 = x^3 - 3x + 3.$

Figure 1.2 – Geometric description of point addition and point doubling.

where $A = Y_{Q'}Z_Q - Y_Q Z_{Q'}$, $B = X_{Q'}Z_Q - X_Q Z_{Q'}$, $C = Z_Q Z_{Q'} A^2 - (X_{Q'}Z_Q + X_Q Z_{Q'})B^2$; $E = 3X_Q^2 + aZ_Q^2$, $F = Y_Q Z_Q$, $G = X_Q Y_Q^2 Z_Q$, $H = E^2 - 8G$.

Besides projective coordinates, there exist other systems of coordinates wherein the computation of the inherent divisions of Equation 1.2 can be avoided. The reader is referred to [CF06] for a presentation of some of these systems of coordinates and the formulas of point addition and doubling within them.

The point $\mathcal{O} = (0 : 1 : 0)$ is the base point (also called point at infinity) expressed in projective coordinates.

Lemma 1 (**Group Law on Elliptic Curve**). *Let E_K be an elliptic curve defined over a field K. The points of E_K provided with the operation $+$ is a group of neutral element the point $\mathcal{O}$.*

A proof of Lemma 1 can be found in [Sil09].

Definition 2 (**Elliptic Curve Scalar Multiplication**). Let E_K be an elliptic curve

defined over a field K. The *elliptic curve scalar multiplication (ECSM)* is the function

$$\begin{aligned} f : (\mathbb{Z}_+ \backslash \{0\}) \times E_K &\to E_K \\ (u, Q) &\mapsto [u]Q = \underbrace{Q + Q + \cdots + Q}_{u \text{ terms}}. \end{aligned}$$

Definition 2 of the ECSM is generalized to all $u \in \mathbb{Z}$ by writing

$$[u]Q = \begin{cases} \mathcal{O} & \text{if } u = 0 \\ Q + Q + \cdots + Q & \text{if } u > 0 \\ (-Q) + (-Q) + \cdots + (-Q) & \text{otherwise,} \end{cases}$$

where the point $-Q$ is the symmetric of the point Q in relation to the x-axis.

In the following, ECC is presented through the lens of the elliptic curve discrete logarithm problem (ECDLP).

Elliptic Curve Discrete Logarithm Problem

Let $E_{\mathbb{F}_q}$ be an elliptic curve defined over a finite field $\mathbb{F}_q$, and Q a point of $E_{\mathbb{F}_q}$ of order v. The ECDLP is stated: Knowing the point $[u]Q$ of the curve $E_{\mathbb{F}_q}$ (and the point Q), find the unique scalar u, $0 \leq u \leq v - 1$ by which the point Q is multiplied. The function

$$f_Q : u \mapsto [u]Q$$

can be viewed as a one-way function associated with the ECDLP. Computing $[u]Q$ from u is easily performed using one of the various state-of-the-art algorithms (for example *double and add* in [HMV04]) while computing u from $[u]Q$ is difficult for well-chosen parameters. The involved parameters are the underlying finite field of the curve and the curve itself.

When choosing the underlying finite field $\mathbb{F}_q$ of the curve $E_{\mathbb{F}_q}$ the aim is to have a field on which the arithmetic is efficient while the curve still ensures a good level of security (meaning the ECDLP should be hard to solve). Several types of fields (prime fields, fields of prime characteristic, binary fields, etc.) have been studied and some fields have been standardized; see, for example the ones in [ndlsdsd11] (from ANSSI [2]) and in [oST13]

2. ANSSI stands for *Agence nationale de la sécurité des systèmes d'information* (France).

(from NIST [3]). The reader should bear in mind that regardless of the type of fields from which is selected the underlying finite field $\mathbb{F}_q$, the cardinality q of $\mathbb{F}_q$ should never be small. The theorem of Hasse says that the cardinality $\sharp E_{\mathbb{F}_q}$ of the curve is bounded as follows:

$$q + 1 - 2\sqrt{q} \leq \sharp E_{\mathbb{F}_q} \leq q + 1 + 2\sqrt{q}. \tag{1.5}$$

If q is small, then $\sharp E_{\mathbb{F}_q}$ is small. The consequence is that the ECDLP on such a curve $E_{\mathbb{F}_q}$ can be solved through exhaustive search attack.

Choosing the curve is tricky and two types of attacks on the ECDLP have to be considered when doing so. The first type consists of attacks on arbitrary curves such as the Pohlig-Hellman attack [PH78] and the Pollard Rho method [Pol78]. The Pohlig-Hellman attack is performed by computing u modulo each prime factor (raised to its maximum power within the prime decomposition) of $\sharp E_{\mathbb{F}_q}$ and recovering u through the Chinese remainder theorem (CRT). The u-modulo-prime-factor step becomes easy if the mentioned prime factors are small. Choosing the curve such that v, the order of the point Q and hence a factor of $\sharp E_{\mathbb{F}_q}$, be a large prime prevents this attack. The Pollard rho method can be viewed as a randomized equivalent of the baby-step giant-step method. Despite its effectiveness, the Pollard Rho method still runs in exponential time.

The second type of attacks is about attacks on specific curves. Examples of these attacks are the Menezes-Okamoto-Vanstone attack [MOV93] and the Frey-Rück attack [FR94] which exploit the fact that, under some circumstances, the ECDLP can be reduced to the DLP in some extension field $\mathbb{F}_{q^l}$ (of the underlying field $\mathbb{F}_q$ of the curve). The DLP in the extension field can be solved if l is small. Examples of curves subject to these attacks are supersingular curves and curves $E_{\mathbb{F}_q}$ of cardinality $q-1$. These attacks are prevented by choosing the curves such that v is not a factor of $q^l - 1$ for all small l. Menezes [Men01] suggested that, when $v > 2^{160}$, checking this condition for $1 \leq l \leq 20$ is enough. Another example of attacks on specific curve is the attack on curves $E_{\mathbb{F}_p}$ of cardinality p for which authors of [Sem98] and [SA98] prove that the ECDLP can be solved in polynomial time. This attack is prevented by ensuring the cardinality of the curve is not p when choosing the curve.

The idea of ECC, proposed in [Mil85, Kob87], is to use asymmetric cryptographic applications based on the ECDLP. The preference of the ECDLP to the DLP (in finite field) is motivated by the fact that there is no known algorithm solving the ECDLP for general cases in sub-exponential time, contrary to the DLP in finite field (for example,

3. NIST stands for National Institute of Standards and Technology (United States).

the index calculus). An inherent consequence is that the cryptographic operands in ECC are smaller than the ones in asymmetric cryptographic applications based on the DLP for the same level of security. For example, the elliptic curve digital signature algorithm (ECDSA [4]) with public-key size of 256 bits is considered providing roughly the same level of security (approximately 128 bits of security) as the digital signature algorithm (DSA [5]) with public-key size of at least 2048 bits; see [oST13] for recommendations on parameter sizes for signature algorithms based on the DLP and the ECDLP.

Protocols of asymmetric cryptographic applications (key exchange, digital signature, public-key encryption) based on the DLP are adaptable to elliptic curves. As an example, Algorithm 4 describes the elliptic curve version (ECDH) [Mil85] of the Diffie-Hellman key-exchange protocol presented in Subsection 1.1.1. An eavesdropper who gets $[u_A]Q$ or $[u_B]Q$ has to solve the ECDLP to recover the (secret) scalars u_A or u_B. Moreover, solving the ECDLP is a crucial part of forging the agreed key $[u_A u_B]Q$ from $[u_A]Q$ or $[u_B]Q$ without knowledge of u_A or u_B.

Algorithm 4: Elliptic curve version of the Diffie-Hellman key-exchange protocol [Mil85].

Input: $E_{\mathbb{F}_q}$ is an elliptic curve defined over $\mathbb{F}_q$, and Q a point of $E_{\mathbb{F}_q}$ of order v
Output: Alice and Bob have a common key
$[u_A u_B]Q = [u_A]([u_B]Q) = [u_B]([u_A]Q) = [u_B u_A]Q$

1. Alice chooses a random number u_A, $0 \leq u_A \leq v - 1$ and keeps it secret
2. Alice computes $[u_A]Q$ and sends it to Bob
3. Bob chooses a random number u_B, $0 \leq u_B \leq v - 1$ and keeps it secret
4. Bob computes $[u_B]Q$ and sends it to Alice
5. Alice receives $[u_B]Q$ and computes $[u_A]([u_B]Q) = [u_A u_B]Q$
6. Bob receives $[u_A]Q$ and computes $[u_B]([u_A]Q) = [u_B u_A]Q$

In the remaining of this thesis we assume the curve is defined over a prime field $\mathbb{F}_p$. Our assumption is motivated by the fact that in practice, curves over prime fields have proven to be more resistant (to attacks) than curves defined over other fields. Among others, curves defined over binary fields have been subject to attacks; see, for example [GHS02b, GHS02a, Hes03, MTW04].

4. The ECDSA is based on the ECDLP.
5. The DSA is based on the DLP.

Computations Involved in an ECSM

The ECSM results from several point doublings and additions (operations in $E_{\mathbb{F}_q}$), which in turn are obtained through operations in the finite field $\mathbb{F}_p$. This hierarchical description is presented in Figure 1.3.

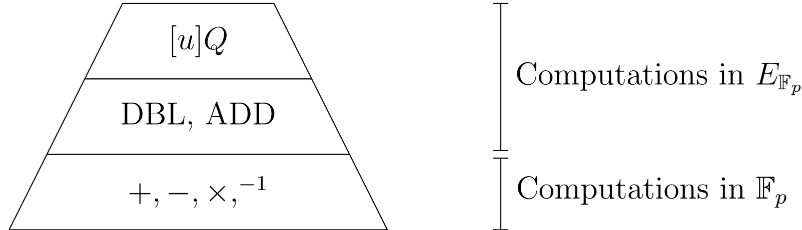


Figure 1.3 – Hierarchical description of computations involved in an ECSM.

There exist several algorithms in the literature to compute the ECSM. The elementary *double-and-add* algorithm (see, for example [HMV04, p. 97]) is presented in Algorithm 5. The *double-and-add* algorithm is an elliptic curve adaptation of the *square-and-multiply* algorithm for exponentiation (for example $x^5 = ((1^2 \times x)^2)^2 \times x$). Other algorithms for exponentiation such as the *sliding window method* have also been adapted to elliptic curves. The reader is referred to [HMV04] for some of these algorithms.

Algorithm 6 presents the *Montgomery ladder* [Mon87] which was initially proposed for curves of equations of the form $by^2 = x^3 + ax^2 + x$. Thanks to [BJ02, IT02], the use of this algorithm has been extended to curves of equation in short Weierstraß form. The use of Algorithm 6 presents two major advantages. First, the Y-coordinate (in projective

---

Algorithm 5: *Double-and-add* algorithm (see, for example [HMV04]).

Input: Q a point of the elliptic curve $E_{\mathbb{F}_p}$
u an integer written in binary representation as $u_{s-1}u_{s-2}...u_0$
Output: $R = [u]Q$

1. $R \leftarrow \mathcal{O}$
2. **for** $i \leftarrow s-1$ ***to*** 0 **do**
3. $\quad R \leftarrow [2]R$
4. $\quad$**if** $u_i = 1$ **then**
5. $\quad\quad R \leftarrow R + Q$
6. **return** R

Algorithm 6: Montgomery ladder algorithm [Mon87].

Input: Q a point of the elliptic curve $E_{\mathbb{F}_p}$
u an integer written in binary representation as $u_{s-1}u_{s-2}...u_0$
Output: $Q_1 = [u]Q$
1 $Q_1 \leftarrow \mathcal{O}$; $Q_2 \leftarrow Q$
2 **for** $i \leftarrow s-1$ ***to*** 0 **do**
3   **if** $u_i = 0$ **then**
4         $Q_1 \leftarrow [2]Q_1$; $Q_2 \leftarrow Q_1 + Q_2$
5   **else**
6         $Q_2 \leftarrow [2]Q_2$; $Q_1 \leftarrow Q_1 + Q_2$
7 **return** Q_1

system of coordinates) does not need to be computed during the ECSM. Therefore, the cost of the algorithm is reduced. Note that the Y-coordinate can be recovered at the end of the algorithm if necessary. Second, regardless of the bit of the scalar u, a point doubling and a point addition are performed. In other words, the number, the type and the order of operations are the same regardless of the value of the bit of u. This constancy of the number, the type and the order of operations per bit of u constitutes an advantage in terms of protection against some side-channel attacks such as simple power analysis (SPA).

Point doublings and additions are computed through formulas elaborated depending on the parameters of the curve (underlying field, curve equation) and the system of coordinates in use. The reader is referred to [CF06] for some of these formulas. Regardless of the chosen formulas, they involve operations in $\mathbb{F}_p$. Therefore, using an efficient arithmetic in $\mathbb{F}_p$ is crucial in computing the ECSM.

In the work presented in this thesis, we choose to use the *residue number system* (RNS) to perform operations in $\mathbb{F}_p$ because RNS provides several efficiency advantages described in Subsection 1.3.2. The hierarchical description of ECSM computations in Figure 1.3 is extended by adding an extra layer at its basis; see Figure 1.4. The contributions of this thesis reside in this extra layer about computations in $\mathbb{F}_p$ using RNS.

The existence of operations in $\mathbb{F}_p$ implies a need for efficient modular reductions (MRs). Some MR algorithms from the literature are presented before delving into RNS.

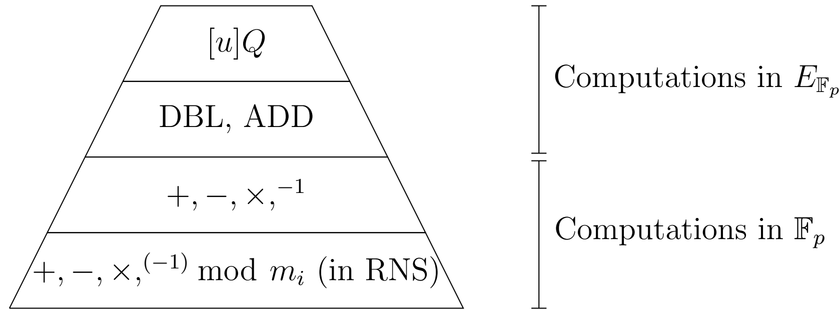


Figure 1.4 – Hierarchical description of computations involved in an ECSM with RNS as chosen number system.

1.2 Modular Reduction

Computations of point doublings and additions, useful in ECSM computations involve numerous MRs since operands are in finite fields. For example, over a finite field $\mathbb{F}_p$ of 256-bit elements, one ECSM requires a few thousands mod p reductions. Therefore, it is crucial to ensure MRs are efficient. In this section some MR algorithms from the literature are presented.

1.2.1 Montgomery Reduction

Montgomery [Mon85] introduces a MR algorithm that avoids computing the costly naive division. The idea is to use an auxiliary number r by which the MR and the division are straightforward. For example, in binary representation, mod 2^l reduction and (integer) division by 2^l are respectively performed by selecting the l least significant bits (LSBs) and by taking all the bits except these l LSBs of the considered numbers. The MR idea is described in Algorithm 7. Note that the output of the algorithm on input x is $xr^{-1} \bmod p$ and not $x \bmod p$.

Besides its efficiency, Algorithm 7 is consistent with multiplications and additions/subtractions. Indeed,

$$xr^{-1} \times x'r^{-1} = (xx'r^{-1})r^{-1} \bmod p \quad \text{and} \quad xr^{-1} + x'r^{-1} = (x + x')r^{-1} \bmod p.$$

In other words, there is a correspondence between the product and the sum/difference of operands in the input space on one hand, and the product and the sum/difference of operands in the output space on the other hand. Owing to this correspondence in

Algorithm 7: Montgomery reduction algorithm [Mon85].

Input: x such that $x < pr$
Precomp.: p, r coprime with p such that $r > p$ and $r^{-1} < p$
p' such that $p' < r$ and $rr^{-1} - pp' = 1$
Output: $y = xr^{-1} \bmod p < 2p$
1 $z \leftarrow (x \bmod r)p' \bmod r$
2 $t \leftarrow x + zp$
3 $y \leftarrow t/r$

addition to its efficiency, Algorithm 7 is suitable for protocols of asymmetric cryptography applications where numerous modular multiplications (MMs) and additions/subtractions are needed, in particular for protocols of ECC applications. If successive MMs have to be performed, for example in the case of a modular exponentiation or a progressive modular accumulation of products, another way to exploit this correspondence is to repeatedly input the products in the form $(xr \bmod p)(x'r \bmod p)$ instead of xx' to Algorithm 7. The repeated outputs will be of the form $xx'r \bmod p$, that is, in the same space as the input factors. Hence, the change of space is avoided between successive MMs. In such cases, the changes of spaces only happen before the start ($x \to xr \bmod p$) and after the end ($xr \bmod p \to x$) of the successive MMs.

Another remark on Algorithm 7 is the range of its output: If x is inputted, the output is $xr^{-1} \bmod p < 2p$. In successive MMs, an intermediate product x may grow in such a way that x no longer verifies the input condition $x < pr$ for the next run of Algorithm 7. Usually, the auxiliary number r is adjusted beforehand to prevent this situation for happening. A typical example: Suppose a modular exponentiation on elements of $\mathbb{F}_p$ is being performed using Algorithm 7 for the reduction, with $p < r < 2p$. During the modular exponentiation, it may occur that an output y_1 of Algorithm 7 is such that $p < y_1 < 2p$. In such a case and for an element $y_2 \in \mathbb{F}_p$, the product $x = y_1 \times y_2$ cannot be guaranteed to be less than pr, which is the input condition of Algorithm 7. For our typical example, taking (beforehand) r such that $r > 2p$ ensures that this input condition remains true during the whole modular exponentiation.

1.2.2 Barrett Reduction

Barrett [Bar86] proposes an algorithm to perform the modular reduction by approximating the quotient of the integer division. Let p and $x < p^2$ be numbers. The aim is to compute

$x \bmod p$. The quotient of the integer division $\lfloor x/p \rfloor$ is approximated by

$$\left\lfloor \left\lfloor \frac{x}{2^w} \right\rfloor \frac{p'}{2^w} \right\rfloor,$$

where $p' = \left\lfloor \frac{2^{2w}}{p} \right\rfloor$. Since $x \bmod p = x - p \left\lfloor \frac{x}{p} \right\rfloor$, the idea is to compute

$$\tilde{x} = x - p \left\lfloor \left\lfloor \frac{x}{2^w} \right\rfloor \frac{p'}{2^w} \right\rfloor. \tag{1.6}$$

The computed value $\tilde{x} < 3p$ [Bar86]. The value $x \bmod p$ is recovered from $\tilde{x}$ through at most two subtractions of p.

1.2.3 On Pseudo-Mersenne Numbers

Numbers of the form $p = 2^w - c$ where $0 \leq c < 2^{w/2}$ are said to be pseudo-Mersenne, in reference to Mersenne numbers (of the form $p = 2^w - 1$).

Reductions modulo pseudo-Mersenne numbers are efficiently performed. The main idea is to exploit the form of these numbers and use bit selection and multiplication by a small constant to perform the reduction. Let x be a number of binary size $2w$ (say, the product of two numbers of size w). Start by remarking that x can be written

$$x = x_1 \times 2^w + x_0, \tag{1.7}$$

where x_0 and x_1 are less than 2^w. Note also that $2^w \equiv c \pmod{p}$.
Therefore,

$$x \bmod p \equiv x_1 c + x_0 \pmod{p}. \tag{1.8}$$

The number $\check{x} = x_1 c + x_0$ is of at most $3n/2$ bits and can be rewritten similarly to Equation 1.7 into

$$\check{x} = \check{x}_1 \times 2^w + \check{x}_0. \tag{1.9}$$

The binary size of $\check{x}_0$ is w while that of $\check{x}_1$ is at most $w/2$. From the congruence at Equation 1.8 and the equality at Equation 1.9, is deduced

$$x \bmod p \equiv \check{x}_1 c + \check{x}_0 \pmod{p}. \tag{1.10}$$

The number $\check{x}_1 c + \check{x}_0$ is of at most $n+1$ bits. The possible last reduction mod$p$ is performed by at most one subtraction of p.

In the general case where the target operations are in arbitrary fields, this algorithm cannot be used for reductions in these fields.

1.3 Residue Number System

1.3.1 Overview of the Residue Number System

This subsection presents a simplified description of RNS, first proposed in [Val56, Gar59]. The reader interested in more details about RNS is referred to [ST67]. The concept of congruence will be used throughout the remaining of this thesis, and therefore introduced at the start.

Definition 3 (**Congruence**). Let $m > 0$ be an integer. Two integers x and y are *congruent modulo* m if $x - y$ is a multiple of m. We write

$$x \equiv y \pmod{m} \tag{1.11}$$

and we say that y is the residue of x modulo m. The integer m is called the modulus.

This definition still holds when the modulus $m < 0$. In the remaining of this document all the moduli are assumed positive for convenience.

The Residue Number System [Val56, Gar59]

In RNS, numbers are represented by their residues over a set $\mathcal{M} = \{m_1, \ldots, m_n\}$ of moduli m_i. The moduli m_i are selected pairwise coprime and the set $\mathcal{M}$ they compose is called an RNS base.

Example 1. To represent the number 117 in the RNS base $\{5, 6, 7\}$, we compute the residues of 117 modulo 5, 6 and 7 respectively.

$$\begin{aligned} 117 &\equiv 2 \pmod{5} \\ 117 &\equiv 3 \pmod{6} \\ 117 &\equiv 5 \pmod{7} \end{aligned}$$

The tuple $(2, 3, 5)$ is the RNS representation of the number 117 in the base $\{5, 6, 7\}$.

In practice, the sizes of the moduli used for RNS implementations of ECC applications are typical sizes of DSP slices or machines words, that is, in [10, 64] bits.

RNS can be viewed as a number system where large numbers (a few hundreds of bits) are transformed into a set of small numbers (a few dozen bits). Computations on large numbers are then replaced by computations on small numbers, the latter being usually performed in parallel.

For RNS to be well defined as a number system it is crucial that the moduli m_i (elements of the RNS base) be pairwise coprime. By *well defined* we mean there is a one-on-one correspondence between numbers in $[0, \prod_{i=1}^{n} m_i - 1]$ and their RNS representation. Otherwise, some representations will not correspond to any number in $[0, \prod_{i=1}^{n} m_i - 1]$. Moreover, in such case at least two (more precisely, $\gcd(m_1, \dots, m_n)$) numbers will have the same RNS representation.

Example 2. Let suppose in Example 1 that 5, 6, and 8 were chosen as moduli. These moduli are not pairwise coprime since $\gcd(6, 8) = 2$. The representation $(1, 4, 3)$ does not correspond to any number in $[0, 239]$. Besides, the numbers 117 and 237 have the same representation $(2, 3, 5)$.

We ease the notation of the congruence by writing

$$y = |x|_m \quad \text{to say that} \quad x \equiv y \pmod{m}.$$

Definition 4. Let $\mathcal{M} = \{m_1, \dots, m_n\}$ be an RNS base. A number x is represented in RNS according to the base $\mathcal{M}$ by the tuple

$$(x_{m_1}, \dots, x_{m_n}),$$

where $x_{m_i} = |x|_{m_i}$ for $i = 1, \dots, n$.

Whenever there is no ambiguity on the RNS base in which x is represented, the notation of the RNS representation of x is further simplified by simply writing $(x_1, \dots, x_n)$.

Arithmetic Operations in RNS

Let $\mathcal{M} = \{m_1, \dots, m_n\}$ be an RNS base and $M = \prod_{i=1}^{n} m_i$. Let $\diamond$ stands for any of the operations $\times$, $+$ or $-$ with operands in RNS representation. Let x, y be two integers represented in RNS according to the base $\mathcal{M}$ by $(x_1, \dots, x_n)$ and $(y_1, \dots, y_n)$ respectively.

Let also assume the result of multiplication, addition and subtraction of x and y is less than M. The RNS computation of the multiplication, addition and subtraction between x and y is performed by

$$(x_1, \ldots, x_n) \diamond (y_1, \ldots, y_n) = (|x_1 \diamond y_1|_{m_1}, \ldots, |x_n \diamond y_n|_{m_n}). \tag{1.12}$$

If y is coprime with M, then the notation $\diamond$ is extended to include division and Equation 1.12 still holds.

Example 3. The numbers 117 and 68 are represented in RNS according to the base $\{5, 6, 7\}$ by $(2, 3, 5)$ and $(3, 2, 5)$ respectively. The number 185, sum of 117 and 68, is represented by $(|2 + 3|_5, |3 + 2|_6, |5 + 5|_7) = (0, 5, 3)$.

Conversion from Positional Number System to RNS and *Vice Versa*

Knowing how to convert numbers from positional system to RNS and *vice versa* is justified by the fact that the use of positional systems is widespread. For instance, most digital systems use the weighted number system of radix 2. A case where the conversions become handy is when one of two (or more) individuals in a network uses the RNS while the other(s) a positional system. An RNS-to-positional or the reverse conversion has to be performed for each exchange.

The conversion of a number from positional representation to RNS is performed by computing the residues of this number over the moduli of the RNS base. The reader can recognize this method in Example 1.

A number represented in RNS is converted into a positional system by using the Chinese remainder theorem (CRT) presented in Theorem 2.

Theorem 2 (Chinese Remainder Theorem). *Let $m_1, \ldots, m_n$ be n positive integers that are pairwise coprime. The system of congruences*

$$\begin{cases} x_1 = |x|_{m_1} \\ x_2 = |x|_{m_2} \\ \quad \vdots \\ x_n = |x|_{m_n} \end{cases} \tag{1.13}$$

has a unique solution modulo $M = \prod_{i=1}^{n} m_i$.

This solution is given by

$$x = \left| \sum_{i=1}^{n} |x_i {M_i}^{-1}|_{m_i} M_i \right|_M , \tag{1.14}$$

where $M_i = \dfrac{M}{m_i}$ *for* $i = 1, \ldots, n$.

Theorem 2 about the CRT is proven in Appendix C.

The CRT is also used to perform base extensions, that is, conversions from one RNS base to another; see Subsection 1.3.3.

1.3.2 RNS Inherent Properties and Complexity Measurement

RNS Inherent Properties

In RNS, the basic operations *multiplication*, *addition* and *subtraction* are held independently on each modulus of the RNS base. This independence means the mentioned computations are carry-free between the channels. Consequently, multiplications, additions and subtractions can be performed in parallel.

RNS is a nonpositional representation system, that means, the value of a number does not change if its residues in a given base are reordered differently. We emphasize that a reordering of the residues, if done, must be followed by a similar reordering in the used RNS base. For example, $(2, 3, 5)$ in the RNS base $\{5, 6, 7\}$ and $(3, 2, 5)$ in the (same but reordered) RNS base $\{6, 5, 7\}$ represent the exact same number 117. However, in the RNS base $\{5, 6, 7\}$ the tuple $(3, 2, 5)$ represents the number 68.

The major drawback of RNS is the difficulty to perform position-related operations such as comparisons, divisions by integers not coprime with the range M of the RNS base, and MRs. This difficulty results from that of evaluating the order of magnitude of operands in nonpositional representation systems. For example, comparisons are difficult because we cannot just rely on the position of digits (as in a usual positional system) of two (or more) integers in RNS to compare them. An alternative is to convert the to-be-compared integers into a positional system such as *mixed-radix system* (MRS) before the comparison. These conversions are costly as we see in Subsection 1.3.3.

Exact divisions by integers not coprime with M are also difficult in RNS for the same reason. Exact divisions by such integers might require to convert those integers to their positional representation before performing the divisions. As mentioned, those conversions are costly. Another way to perform divisions the reader might think of is to subtract

the divisor iteratively. However this method requires a comparison per subtraction to determine the end of the division process. As explained, the comparisons are difficult.

Another difficult operation in RNS is the mod p reduction, where p is a (large) integer not coprime with the range M of the RNS base. Since operations in protocols of asymmetric cryptographic applications are usually performed in rings or fields, the mod p reduction is a crucial operation. Some ways to perform MRs from the literature are presented in Section 1.2.

How Complexity is Measured?

Deciding on a complexity unit is always tricky, and even more when hardware aspects are involved in addition to theoretical ones. It is critical to choose the metric of the input size and define what the basic operations are. The input size is the number of residues, which is the same as the number of moduli of the considered RNS base. The size itself of the moduli is almost hidden in the complexity. In hardware implementations, it is common to choose the size of the moduli such that the residues (operands of the various RNS computations) fit in the multipliers embedded in the DSP slices [6].

We consider our basic operations to be modular multiplications, additions and subtractions, the modulus being an element of the RNS base. For a historical interest, we point out that Szabo and Tanaka [ST67, pp. 140–150] reported mixed-radix conversion as a basic operation besides multiplication, addition and subtraction. The reason behind including mixed-radix conversion is that the latter is used in sign-determination related operations such as relative-magnitude determination. In most asymmetric cryptography applications and particularly in ECC applications, we are not interested in sign determination.

Complexity-unit candidates for asymmetric cryptosystems in RNS are *elementary modular multiplication* (EMM) and *elementary modular addition* (EMA). An EMM and an EMA are respectively a modular multiplication and a modular addition/subtraction in a channel. The EMM is more significant than the EMA within the cost of RNS computations because the elementary multiplication (without the modular reduction step) usually costs more than the elementary addition while the modular reduction algorithm costs the same in EMM as in EMA at worst. Besides, additions are usually performed by adders/accumulators located next to the multipliers embedded in the DSP slices. Examples of references where the complexity is expressed in terms of the number of EMMs include

6. DSP slices stand for digital signal processing slices.

[KKSS00, BDK01, GLP+12, BT15].

1.3.3 Base Extension

A base extension (BE) is a conversion of a number x from its representation in an RNS base $\mathcal{M} = \{m_1, \ldots, m_n\}$ to its representation in another RNS base $\mathcal{M}' = \{m'_1, \ldots, m'_{n'}\}$. The notation $BE_{\mathcal{M}\rightarrow\mathcal{M}'}(x)$ indicates the computation of the residues $(x_{m'_1}, \ldots, x_{m'_{n'}})$ of a number x in $\mathcal{M}'$, knowing the residues $(x_{m_1}, \ldots, x_{m_n})$ of x in $\mathcal{M}$.

An intuitive approach would be to convert x from the RNS base $\mathcal{M}$ to its positional representation using the CRT (formula given in Equation 1.14), then convert x from its positional representation to the RNS base $\mathcal{M}'$. However, this approach is troublesome owing to the variable sizes of the operands. Therefore, the conversion to positional representation is usually avoided.

Base extensions (BEs) from the literature can be classified in two major types: the ones that go through a representation in MRS, therefore requiring an RNS-to-MRS conversion, and the others that use the CRT formula (without converting the number into a positional representation). These two types of base extension are presented hereafter.

Base Extensions Requiring an RNS-to-MRS Conversion

Definition 5 (**Mixed-radix system; from [ST67, pp. 41–47]**)**.** Let $\mathcal{M} = \{m_1, \ldots, m_n\}$ be a set of numbers not necessarily coprime. A number x is said to be represented in MRS according to the radices m_i if

$$x = \underline{x_n} \prod_{i=1}^{n-1} m_i + \underline{x_{n-1}} \prod_{i=1}^{n-2} m_i + \cdots + \underline{x_2} m_1 + \underline{x_1} \tag{1.15}$$

The tuple $(\underline{x_1}, \ldots, \underline{x_n})$ is the MRS representation of x in the MRS base $\mathcal{M}$.

BEs using an RNS-to-MRS conversion comprise two parts: a conversion of x from the RNS base $\mathcal{M}$ to the associated MRS base $\mathcal{M}$, and a conversion of x from the MRS base $\mathcal{M}$ to the RNS base $\mathcal{M}'$.

The earliest algorithm converting a number x from RNS to MRS appeared in [Val56] according to [ST67, pp. 41–51], and in [Gar59]. Let $\mathcal{M} = \{m_1, \ldots, m_n\}$ be an RNS base (the moduli m_i are coprime), and $(x_1, \ldots, x_n)$ be the RNS representation of x in $\mathcal{M}$. Let $(\underline{x_1}, \ldots, \underline{x_n})$ denote the MRS representation of x in $\mathcal{M}$. Their idea of the RNS-to-MRS conversion (that is, $(x_1, \ldots, x_n)$ into $(\underline{x_1}, \ldots, \underline{x_n})$) is described hereafter.

Let start with the MRS representation of x in $\mathcal{M}$

$$x = \underline{x_n} \prod_{i=1}^{n-1} m_i + \underline{x_{n-1}} \prod_{i=1}^{n-2} m_i + \cdots + \underline{x_3} m_1 m_2 + \underline{x_2} m_1 + \underline{x_1}. \tag{1.16}$$

By reducing modulo m_1 the left and the right side of Equation 1.16 we obtain $\underline{x_1} = x_1$ (since $x_1 = |x|_{m_1}$). Next, substracting $\underline{x_1} = x_1$ from each side of Equation 1.16 and multiplying the consequent results by ${m_1}^{-1}$ yield

$$(x - \underline{x_1}){m_1}^{-1} = \underline{x_n} \prod_{i=2}^{n-1} m_i + \underline{x_{n-1}} \prod_{i=2}^{n-2} m_i + \cdots + \underline{x_3} m_2 + \underline{x_2}. \tag{1.17}$$

By reducing the left and the right side of Equation 1.17 we obtain $\underline{x_2} = \left|(x - x_1){m_1}^{-1}\right|_{m_2}$. This process is repeated until all the MRS digits $\underline{x_i}$ of x are computed. Algorithm 8 summarizes the idea.

Algorithm 8: RNS-to-MRS conversion algorithm [Val56, Gar59].

Input: $x_1, \ldots, x_n$
Output: $\underline{x_1}, \ldots, \underline{x_n}$
1 **Precomputations:** $|m_j^{-1}|_{m_i}$, for all i, $2 \leq i \leq n$ and j, $1 \leq j \leq i-1$
2 $\underline{x_1} = x_1$
3 **for** $i \leftarrow 2$ **to** n **do**
4     $\underline{x_i} \leftarrow x_i$
5   **for** $j \leftarrow 1$ **to** $i-1$ **do**
6         $\underline{x_i} \leftarrow |(\underline{x_i} - \underline{x_j}) m_j^{-1}|_{m_i}$

The reader can remark that computations in Algorithm 8 are performed mod m_i. Therefore, the MRS possesses the advantage over other positional systems that the size of the operands remains the same during the conversion from RNS.

The second part of the BE, namely the conversion of x from the MRS base $\mathcal{M}$ to the RNS base $\mathcal{M}'$, is equivalent to computing, for all $j = 1, \ldots, n$,

$$\left| \underline{x_n} \prod_{i=1}^{n-1} m_i + \underline{x_{n-1}} \prod_{i=1}^{n-2} m_i + \cdots + \underline{x_2} m_1 + \underline{x_1} \right|_{m'_j}. \tag{1.18}$$

At least two ways to perfom this conversion exist. One way is reported in [ST67, pp. 47–51] where the authors exemplify the idea with one modulus of $\mathcal{M}'$. Let assume this modulus is m'_1. The idea of the conversion of x from $\mathcal{M}$ to $\{m'_1\}$ is to run Algorithm 8 on inputs

$x_{m_1}, \ldots, x_{m_n}, 0$ (in $\{m_1, \ldots, m_n\} \cup \{m'_1\}$) while maintaining the same bounds for the loop of Algorithm 8, that is, i from 2 to n. The RNS digit $x_{m'_1}$ is solution of an equation mod m'_1 of the first degree as its only unknown variable; the other variables of the equation are the digit in $\{m'_1\}$ outputted by the run of Algorithm 8 and some precomputations. This method can be used to perform the conversion from $\mathcal{M}$ to $\mathcal{M}'$ by one run of Algorithm 8 on inputs $x_{m_1}, \ldots, x_{m_n}, 0, \ldots, 0$ (in $\{m_1, \ldots, m_n\} \cup \{m'_1, \ldots, m'_n\}$) while the bounds of the loop of Algorithm 8 are kept the same (i from 2 to n). Each RNS digit $x_{m'_j}$ is solution of an equation of the first degree mod m'_j as its only unknown variable. However, this method requires the number of simultaneously-running inputs to be equal to the sum of the sizes of the two RNS bases $\mathcal{M}$ and $\mathcal{M}'$. When targeting hardware implementations, this method leads to a used hardware resources twice the ones needed by the *cox-rower* architecture [KKSS00] (see below in Part "Base Extension Using the CRT formula") without a gain in time.

Another way to perform the MRS-to-RNS conversion is to compute Equation 1.18 directly. For example, authors of [BKP09] proceed this way and prove that the computation at Equation 1.18 can be performed through shifts and additions/substractions if the selected moduli m_i and m'_j are pseudo-Mersenne numbers $2^w - c$ with c of small Hamming weight. However, their method involves a lot of data dependencies between the channels, and hence hinders the ability to exploit the inherent parallelism of RNS. In the work presented in this thesis, we are interested in preserving as much as possible the inherent parallelism of RNS. Therefore, the base architecture selected for this thesis is the *cox-rower* [KKSS00] (see below in Part "Base Extension Using the CRT formula") because it exploits the inherent parallelism of RNS. In other words, the architectures implemented in the work presented in this thesis are adaptations of the *cox-rower* architecture.

Overall, the presented BE using the RNS-to-MRS conversion costs $\frac{n(3n+1)}{2}$ EMMs.

To summarize, the two notable properties of the BE using conversions through MRS are the fact that the RNS-to-MRS conversion is sequential accross the moduli of $\mathcal{M}$ on one hand, and either the need for twice the size of an RNS base in area or the involvement of a lot of data dependencies on the other hand.

Base Extension Using the CRT Formula

Let $\mathcal{M} = \{m_1, \ldots, m_n\}$ and $\mathcal{M}' = \{m'_1, \ldots, m'_n\}$ be two RNS bases, $(x_{m_1}, \ldots, x_{m_n})$ and $(x_{m'_1}, \ldots, x_{m'_n})$ the RNS representations of a number x in $\mathcal{M}$ and $\mathcal{M}'$ respectively. We

recall the CRT formula:

$$x = \left| \sum_{i=1}^{n} |x_{m_i} {M_i}^{-1}|_{m_i} M_i \right|_M , \quad (1.19)$$

where $M_i = \dfrac{M}{m_i}$ for $i = 1, \dots, n$. From Definition 3 of congruence, there is an integer h such that

$$x = \left(\sum_{i=1}^{n} |x_{m_i} {M_i}^{-1}|_{m_i} M_i \right) - hM. \quad (1.20)$$

The main idea of BEs using the CRT is to compute the integer h or its approximation and use the obtained result in the computation of the various $x_{m'_j}$ by reducing each side of Equation 1.20 modulo m'_j, for $j = 1, \dots, n$.

The earliest BE using the CRT is proposed in [SK89]. The idea is to add to $\mathcal{M}$ an extra modulus $m_{n+1} \geq n$ wherein $h = |h|_{m_{n+1}}$ is computed using the fact that $h < n$ and

$$|h|_{m_{n+1}} = \left| M^{-1} \left(\sum_{i=1}^{n} |x_{m_i} {M_i}^{-1}|_{m_i} M_i - x \right) \right|_{m_{n+1}} . \quad (1.21)$$

The integer h in Equation 1.20 can also be computed by approximation, in which case there is no need for an extra modulus. Posch and Posch [PP93] provide an algorithm which computes an approximation of

$$h = \left\lfloor \sum_{i=1}^{n} \frac{|x_{m_i} {M_i}^{-1}|_{m_i}}{m_i} \right\rfloor , \quad (1.22)$$

and introduce for x the bound $[0, (1 - \epsilon_{\max})M[$ (where $\epsilon_{\max} \ll 1$) in order for the approximation to be exact. Kawamura *et al.* [KKSS00] propose to compute

$$\tilde{h} = \left\lfloor \sum_{i=1}^{n} \frac{\operatorname{trunc}(|x_{m_i} {M_i}^{-1}|_{m_i})}{2^w} + \sigma \right\rfloor \quad (1.23)$$

which is an approximation of h, w being the binary size of the moduli m_i and σ a chosen value to counteract the error in the approximation. The function trunc approximates $|x_{m_i} {M_i}^{-1}|_{m_i}$ by its t most significant bits (MSB), $t < w$, the $w - t$ remaining bits put at 0. Algorithm 9 describes their BE. Under conditions specified in Theorem 1 and Theorem 2 in [KKSS00], the value of $\tilde{h}$ computed in Algorithm 9 is respectively h on one hand, either h or $h - 1$ on the other hand. In the RNS MR algorithm from [PP95] where two BEs are performed, these theorems help decide on the input value of σ to get the correct reduction results.

Algorithm 9: Base extension $BE_{\mathcal{M}\to\mathcal{M}'}(x)$ [KKSS00].

Input: $(x_{m_1},\ldots,x_{m_n})$, σ
Precomp.: $|M_i^{-1}|_{m_i}$, for all i, $1 \le i \le n$
$|M_i|_{m'_j}$, for all i, $1 \le i \le n$ and j, $1 \le j \le n$
$|-M|_{m'_j}$, for all j, $1 \le j \le n$
Output: $(x_{m'_1},\ldots,x_{m'_n})$

1 **for** $i \leftarrow 1$ **to** n **parallel do**
2 $\quad \widehat{x}_{m_i} \leftarrow \left| x_{m_i} \times |M_i^{-1}|_{m_i} \right|_{m_i}$
3 **for** $i \leftarrow 1$ **to** n **do**
4 $\quad \sigma \leftarrow \sigma + \frac{\mathrm{trunc}(\widehat{x}_{m_i})}{2^w}$
5 $\quad h_i \leftarrow \lfloor \sigma \rfloor$
6 $\quad \sigma \leftarrow \sigma - h_i$
7 $\quad$ **for** $j \leftarrow 1$ **to** n **parallel do**
8 $\quad\quad x_{m'_j} \leftarrow \left| x_{m'_j} + \widehat{x}_{m_i} \times |M_i|_{m'_j} + h_i \times |-M|_{m'_j} \right|_{m'_j}$

Algorithm 9 costs $n^2 + n$ EMMs. Our count does not include the multiplications $h_i \times |-M|_{m'_j}$ since they are selections between 0 or $|-M|_{m'_j}$ (the bit h_i being 0 or 1). Operations from lines 4–6 are also not included in the cost because they are efficiently performed using the small accumulator on t bits (usually, $t \in [4, 8]$) called the *cox* [KKSS00].

The *cox-rower* architecture proposed in [KKSS00] and adapted to ECC in [Gui10] is presented in Figure 1.5. Computations at line 2 of Algorithm 9 are performed in parallel by the n *rowers*. The large multiplexer outputs the results of these computations by the left and right buses. The right bus broadcasts these results to the various *rowers* which now perform computations at line 8 of Algorithm 9 in parallel. The left bus transfers the results of the computations at line 2 to the *cox* unit which performs computations at lines 4–6 of Algorithm 9 and send their results to the various *rowers* where line 8 are being performed.

BEs using the CRT possess the advantage over BEs using an RNS-to-MRS conversion of finishing in $n+1$ steps while using n (or $n+1$ moduli for [SK89]) channels. The notable advantage of Algorithm 9 is that the main computations, at lines 2 and 8, are performed in parallel. Unlike in Algorithm 8, there is no carry from one modulus to another in these crucial steps. Hence, Algorithm 9 exploits the inherent independence (in RNS) of multiplication, addition and subtraction among the moduli.

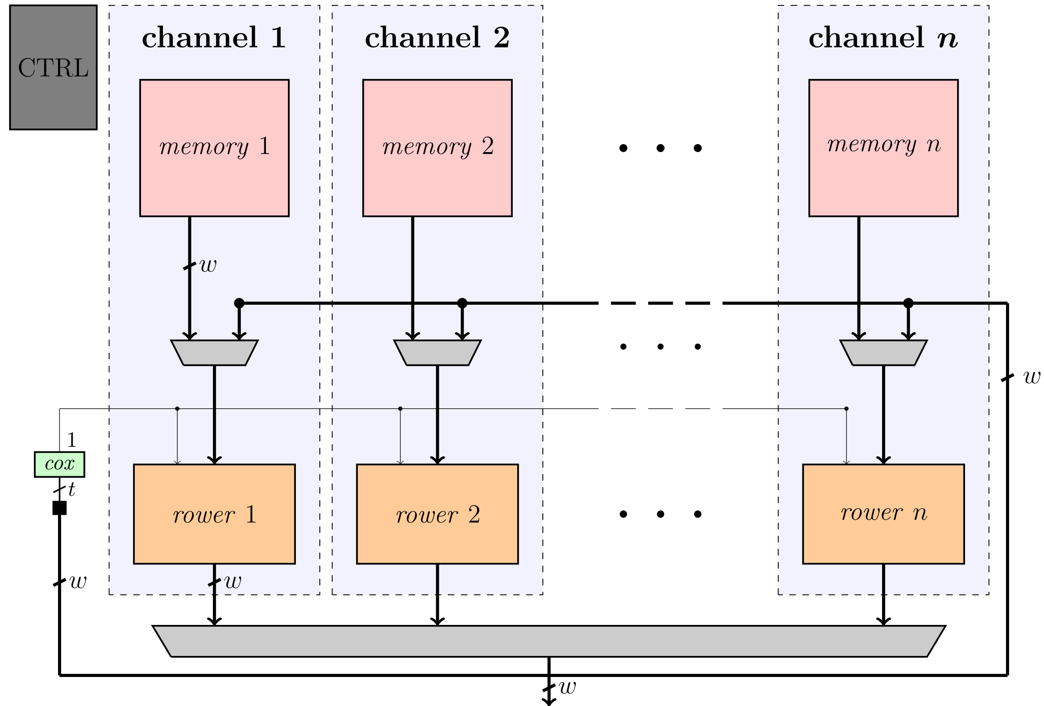


Figure 1.5 – The *cox-rower* architecture in [Gui10], adapted from [KKSS00].

1.3.4 RNS Modular Reduction

The importance of MRs at the field level is mentioned in Section 1.2. Moreover, RNS by definition introduces MRs at another level, the various reductions mod m_i. In this section we present some algorithms from the literature that efficiently perform the RNS MRs.

RNS Montgomery Reduction

The efficiency of the Montgomery reduction presented in Algorithm 7 motivates Posch and Posch [PP95] and later Kawamura *et al.* [KKSS00] to propose an RNS adapted version of the algorithm. Their algorithm is presented in Algorithm 10. Let $\mathcal{M}$ and $\mathcal{M}'$ be two RNS bases such that each modulus of $\mathcal{M}'$ is coprime with all moduli of $\mathcal{M}$. Let also assume $x < MM'$ (for example, the product of two numbers). It is important that x be represented in the two bases because of its range.

Since the reduction mod M is easy in the RNS base $\mathcal{M}$, the auxiliary number r for the Montgomery reduction in RNS is chosen to be M (the RNS equivalent of the 2^l for binary representation). By saying reduction mod M is easy in the base $\mathcal{M}$, we mean the residues

Algorithm 10: RNS Montgomery reduction algorithm [PP95].

Input: $x_{\mathcal{M}}$, $x_{\mathcal{M}'}$
Precomp.: $p_{\mathcal{M}'}$, $(-p^{-1})_{\mathcal{M}}$, $(M^{-1})_{\mathcal{M}'}$
Output: $s_{\mathcal{M}}$ and $s_{\mathcal{M}'}$ where $s = (xM^{-1}) \bmod p$

1. $q_{\mathcal{M}} \leftarrow x_{\mathcal{M}} \times (-p^{-1})_{\mathcal{M}}$
2. $q_{\mathcal{M}'} \leftarrow BE_{\mathcal{M}\rightarrow\mathcal{M}'}(q_{\mathcal{M}})$
3. $r_{\mathcal{M}'} \leftarrow x_{\mathcal{M}'} + q_{\mathcal{M}'} \times p_{\mathcal{M}'}$
4. $s_{\mathcal{M}'} \leftarrow r_{\mathcal{M}'} \times (M^{-1})_{\mathcal{M}'}$
5. $s_{\mathcal{M}} \leftarrow BE_{\mathcal{M}'\rightarrow\mathcal{M}}(s_{\mathcal{M}'})$

(in $\mathcal{M}$) of a number remain unchanged when this number is taken modulo M. However, the RNS division by M cannot be performed in base $\mathcal{M}$ because M as the product of the various m_i is not coprime with them, hence not invertible mod m_i. The BE at line 2 allows to go from the base $\mathcal{M}$ to the base $\mathcal{M}'$ wherein the RNS division by M can be performed. This RNS division is only possible if the moduli of $\mathcal{M}'$ are coprime with the ones of $\mathcal{M}$ (this has been assumed). The second BE (line 5 of Algorithm 10) returns the result of the division by M back to the base $\mathcal{M}$. This BE is also necessary for having the MR result in the two bases. Lines 1, 3 and 4 of Algorithm 10 are the RNS equivalents of lines 1, 2 and 3 of Algorithm 7 respectively.

Algorithm 10 costs $2n^2 + 5n$ EMMs. This cost can be improved, for example by combining some precomputations and reducing the number of times the input residues are multiplied by these precomputations [GLP$^+$12, Gui10]. Gandino *et al.* [GLP$^+$12] improve the cost to $2n^2 + 2n$ EMMs by further reorganizing some internal computations.

RNS Barrett Reduction

The RNS Barrett reduction idea [SS13] is to use RNS to compute

$$\tilde{x} = x - p \left\lfloor \left\lfloor \frac{x}{2^w} \right\rfloor \frac{p'}{2^w} \right\rfloor . \tag{1.24}$$

where $\tilde{x} < 3p$ [Bar86]. Equation 1.24 is the Equation 1.6 already mentioned in Subsection 1.2.2, the exact value $x \bmod p$ being recoverable from $\tilde{x}$ through at most two subtractions of p. The multiplications and subtraction in the formula at Equation 1.24 are now performed in RNS, that is, modular multiplications and subtraction on RNS digits. The operations of the type $\left\lfloor \frac{x}{2^w} \right\rfloor$ are replaced by a check of divisibility by 2^w, a subtraction

of a computed offset value if divisibility is not possible, and a multiplication by the precomputed inverse of 2^w over the moduli of the RNS base besides an additional modulus. In the overall complexity, these operations introduce a parameter that depends on the number of cycles taken by a MM, making difficult the comparison to other algorithms. Nevertheless, the authors will later acknowledge in [SS14] (wherein an architecture to support their algorithm is proposed) that the proposed algorithm require more EMMs than RNS Montgomery reduction.

On Pseudo-Mersenne Numbers and RNS

In Subsection 1.2.3 we mention that the algorithm tailored for pseudo-Mersenne cannot be used for reductions in arbitrary fields. However, moduli m_i of the RNS base can be chosen to be small pseudo-Mersenne numbers to speed the various computations mod m_i, regardless of the modular reduction algorithm used at the field level.

1.4 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are used for many hardware implementations of asymmetric cryptosystems in the literature. FPGAs are presented in this section as well as some RNS implementations of asymmetric cryptosystems from the literature. High-level synthesis (HLS), as the chosen tool in this thesis to assist in the implementation on FPGAs, is also introduced.

1.4.1 Overview of FPGAs

Some of the material in this subsection is a synthesis of the introduction to FPGAs from [ST12]. The used vocabulary is from Xilinx FPGAs because their 7-series devices constitute our implementation target. Similar concepts are available for FPGAs from other manufacturers under other names.

FPGAs are integrated circuits whose functionalities are programmable after manufacture. The functionality of an FPGA is programmed with the design; the FPGA is said to be configured. FPGAs are usually reconfigurable. They serve in many applications such as signal processing, computer vision and hardware security.

An FPGA can be described as a matrix of configurable logic blocks (CLBs). The CLBs usually comprise fast memories with low capacity called look-up tables (LUTs) combined

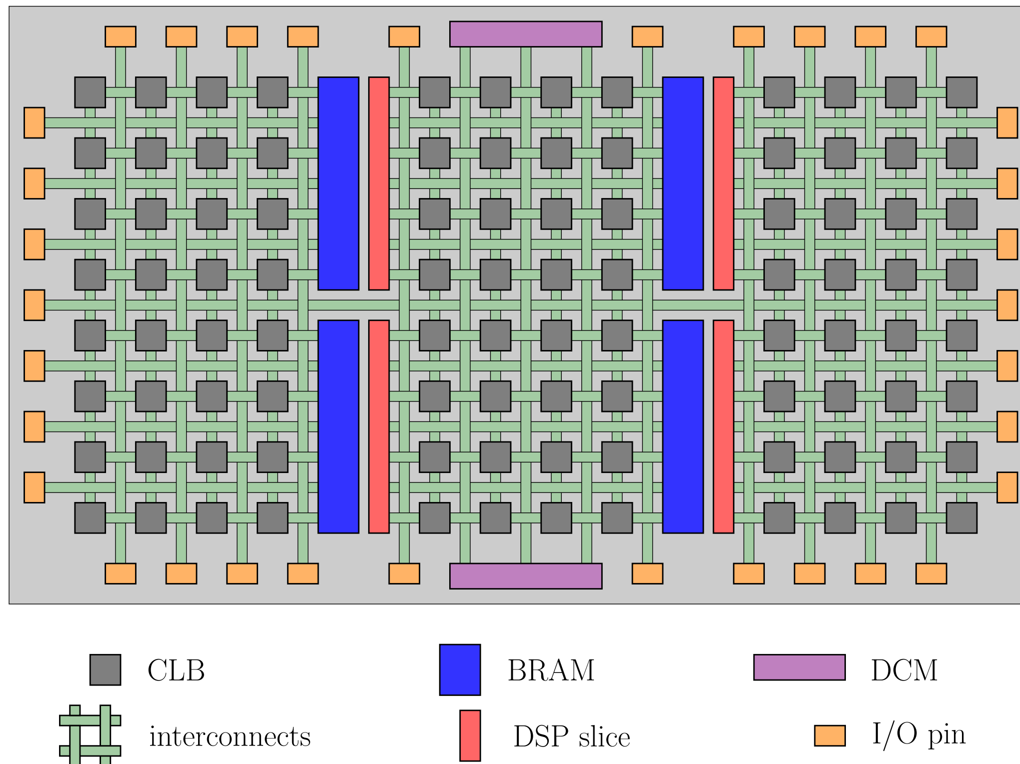


Figure 1.6 – Simplified view of an FPGA.

with flip-flops (FFs) and a small multiplexer. A LUT allows to carry out combinatorial functions such as 4-input bits or 6-input bits to 1-output bit functions depending on the number of its inputs (4-input LUT or 6-input LUT respectively). A common view of a CLB can be found in [ST12]. Interconnects are programmable and they link the CLBs to each other as well as to other elements of the FPGA. Figure 1.6 presents a simplified view of an FPGA.

Other elements such as block RAMs (BRAMs) for data storage, digital signal processing (DSP) slices for arithmetic processing, digital clock manager (DCM) and input/output (I/O) pins are usually present in the modern FPGAs. One of the most complex element in modern FPGAs is probably the DSP slice which comprises a multiplier and an accumulator, and can perform elementary arithmetic functions. For example, in the Xilinx 7-series FPGAs used for our implementations, the DSP slice (DSP48E1) embeds a 18×25 multiplier in 2's complement and a 48-bit accumulator. Figure 1.7 presents the basic functionalities of this DSP slice. It can perform operations of the form $(A+D) \times B + C$, where A, D, B and C are respectively of size 30, 25, 18 and 48 bits. The reader is referred to

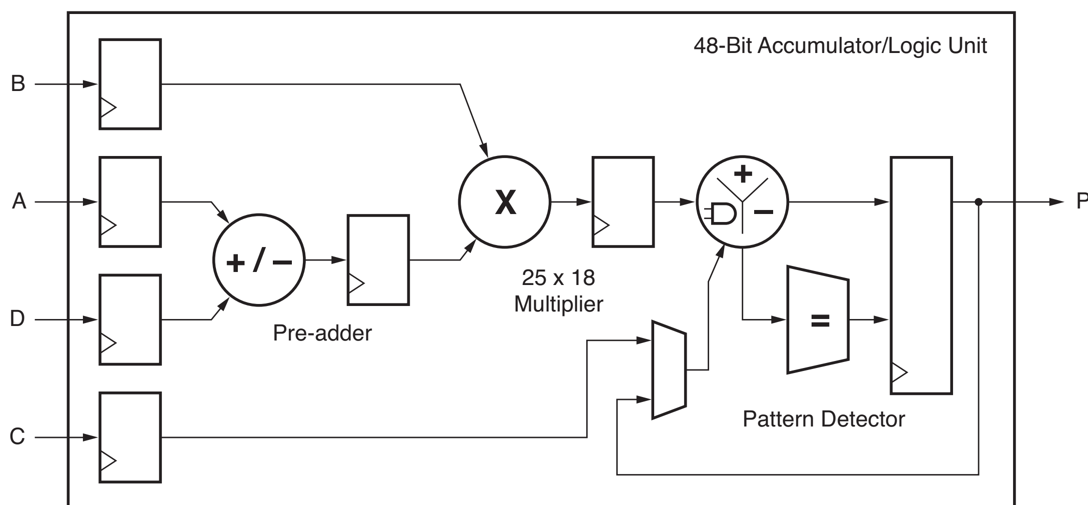


Figure 1.7 – Basic functionalities of the DSP slice in 7-series FPGAs from Xilinx (from [Xil18a]).

the user guide [Xil18a] for more details about this DSP slice.

The implementation on FPGAs is commonly performed using computer-aided design (CAD) tools and is described as follows. First, the functionality of the FPGA is implemented in a hardware description language (HDL), allowing to describe the desired architecture at a register-transfer level (RTL). It is also possible to describe the functionality of the FPGA in a high-level language such as C, C++ or SystemC. In this case, an HLS tool is used to automatically generate the HDL description of the RTL from the description in high-level language. In the work presented in this thesis, we describe the functionality in C and use Vivado HLS 2019.2 from Xilinx as HLS tool. Second, the CAD tool converts the HDL description of the RTL into a configuration stream, usually called *bitstream*. Last, the generated bitstream is used to configure the various elements of the FPGA, notably the CLBs, and the interconnects which will route the signal between the CLBs, and between the CLBs and other elements of the FPGA.

FPGAs present numerous advantages, notably their (re)configurability, compared with other integrated circuits such as application-specific integrated circuits (ASICs). Besides, the financial cost per unit and the time to market of FPGAs are lower than the ones of ASICs because ASIC designs usually require a large design team, complex and costly tools, considerable design and manufacture time [ST12]. However, the FPGA performance is lower (lower speed and more power consumption) than the ASIC one due mainly to

the programmable interconnects (and their layout) which ensure FPGA configurations. Still, FPGAs remain performant for applications they are used in. To summarize, FPGAs provide good hardware performance in addition to being configurable.

1.4.2 Some RNS Implementations of Asymmetric Cryptosystems on FPGA from the Literature

RNS implementations of asymmetric cryptosystems have gained interest in the two latest decades. Table 1.1 presents some of these implementations on FPGA from the literature. In references reported in Tabe 1.1, most authors implemented the core operation of the cryptosystem, for example the modular exponentiation for RSA or the ECSM for ECC. The ECSM solution by Bajard and Merkiche [BM14] is currently the fastest one among RNS implementations of the ECSM over an underlying finite field of 256-bit elements. However, several parameters have to be considered for a fair comparison of FPGA implementations, causing the comparison to be difficult. An important parameter to consider is the FPGA technology. The same code is likely to give better implementation results on a recent FPGA than an older one (say, 10 or more years older FPGA). Besides, the area metrics such as the number of DSPs cannot be directly compared if the multipliers embedded in the DSPs do not handle the same input size. For example, DSPs embedding 9×9 multipliers cannot be fairly compared in number to the ones embedding 18×25 multiplier in 2's complement. Also, the number of combinatorial functions implementable by a 6-input LUT is greater than the one implementable by a 4-input LUT. Therefore, the numbers of slices are difficult to compare between two implementations using different technologies of FPGA (the slices comprise LUTs among other elements). We see in Chapter 3 that we compare our implementation results with the ones from the literature for FPGA technologies which are the closest to ours.

Another parameter to consider in the comparison is the chosen algorithm for computing the core operation. For example, a sliding window (without any protection) is likely to be faster than a double-and-add or a Montgomery ladder. For ECC, the type of the underlying field (binary, prime, etc.) and if prime field, the form of the prime (pseudo-Mersenne or arbitrary) are also parameters to consider. The protection of the implementation, if any, is also to be considered in the comparison because there exists an extra cost if the implementation is protected. In addition, the extra cost is dependent

Table 1.1 – Some RNS implementations of asymmetric cryptosystems on FPGA from the literature. The size$^{(\star)}$ is the operand size (for example the modulus size for RSA or the size of the underlying field of the curve for ECC).

reference	proc./journal	FPGA	cryptosystem	size$^{(\star)}$
[CNPQ03]	MWSCAS	Virtex 2	RSA	1024
[SKS06]	MELECON	Virtex 2 Pro	ECC	160
[SFM+09]	IEEE TCAS I	Virtex E	ECC	160, 192, 224, 256
[Gui10]	CHES	Stratix Stratix II	ECC	160, 192, 256, 384, 512
[CDF+11]	CHES	Cyclone II Stratix II & III Virtex 6	Pairings	126, 128, 192
[YFCV12]	Pairing	Virtex 4 Virtex 6	Pairings	126, 128
[PITM13]	DSD	Spartan 3	RSA	1024
[ESJ+13]	IEEE TVLSI	Virtex E Virtex 2 Pro Stratix II	ECC	160, 192, 224, 256
[BM14]	CARDIS	Kintex 7	ECC	256, 521
[FKSK15]	CS2	Virtex 2 Virtex 5	ECC	192
[BEMP15]	ARITH	Virtex 5 Kintex 7	Lattice-based cryptography	64, 128
[BT15]	CHES	Spartan 6 Virtex 5 Kintex 7	ECC	192, 384, 512

on the type of protection (for example against SPA or DPA [7]). Overall, comparison of FPGA implementations of cryptosystems are tricky and the reported performance results are insufficient in themselves to make a fair comparison.

To close this subsection, we mention that many RNS implementations (for example [NMSK01, Gui10, BM14, BT15]) are based on adaptations of the *cox-rower* architecture by Kawamura *et al.* [KKSS00] because the latter exploits the inherent parallelism of RNS. The architectures implemented in our contributions at Chapters 2 and 3 are also adapted from the *cox-rower*.

7. DPA stands for differential power analysis.

1.4.3 High-Level Synthesis

HLS allows to automatically generate a register-transfer level (RTL) design described in a hardware description language (HDL) from a high-level description (usually in C, C++ or SystemC). The behavioral specification (for example the C code) and the implementation constraints (for example *loop pipelining*) are inputted to HLS which automatically generates the HDL description of the RTL and outputs the performance estimations of this desciption.

The main advantage of HLS is the reduction of the design time since HLS allows to focus mainly on the functionality; the consequence is a gain in productivity for designers. Besides, HLS favors fast explorations of the design space for different parameters and constraints. However, one should not expect HLS to automatically explore the design space without any guidance and provide excellent results. Our experiments suggest that the coding style of the high-level description has to be as close as possible to a behavioral specification in HDL in order to guide HLS to achieve decent results compared with a handwritten HDL description.

Figure 1.8 presents an excerpt of C code for HLS (as a toy example) to perform multiplications of arrays, element by element. There are `2N` multiplications to be performed and two channels are used. Each channel is the hardware support of `N` multiplications, and the channels run in parallel. Each function describes a component. The component `prod` allows to perform the `2N` multiplications and calls the components `mul0` and `mul1` for this purpose. Each of `mul0` and `mul1` is associated with a channel and performs one multiplication between a precomputed multiplicand (stored internally) and an input multiplier. The data types `word` and `dword`[8] represent unsigned integers of respective sizes 8 and 16 bits. The multiplicand and the multiplier are of data type `word`, and the output product is of data type `dword`. Attention should be paid to cast the type of the multiplicand and the multiplier to `dword` in the writing of the multiplication (lines 11 and 15). The reason behind this need for casting is that in C programming, a product of two values of data

8. The types `word` and `dword` are not built-in data types of C programming and the sizes 8 and 16 bits are specific to the example in Figure 1.8. The data types to use in an implementation have to be specified by the designer regarding the size of the manipulated data and should not be limited to usual types and related sizes of C programming (such as 16 or 32 bits for an `int` or an `unsigned int` depending on the platform). For example, choosing the data type `word` to represent unsigned integers of size 17 bits allows to fully exploit the 18×25 multipliers in 2's complement embedded in the DSP slices of Xilinx 7-series FPGAs (asymmetric widths for DSP operands are not considered in this thesis). The product of two numbers of data type `word` is a number of data type `dword`, the latter representing in this case unsigned integers of size 34 bits.

```
 1. void prod(word x0[N],word x1[N],dword res0[N],dword res1[N]){
 2.     counter i;
 3.     for (i=0; i<N; i++){
 4. #pragma HLS PIPELINE
 5.         mul0(i, x0[i], &res0[i]);
 6.         mul1(i, x1[i], &res1[i]);
 7.     }
 8. }

 9. void mul0(counter i,word x,dword *r){
10.     word precomp[N] = {59, 60, 61, 62, 63};
11.     *r = ((dword) (x)) * ((dword) (precomp[i]));
12. }
13. void mul1(counter i,word x,dword *r){
14.     word precomp[N] = {54, 55, 56, 57, 58};
15.     *r = ((dword) (x)) * ((dword) (precomp[i]));
16. }
```

Figure 1.8 – Excerpt of C code for HLS to explain the multiplication of arrays elements on two channels. Each of the components `mul0` and `mul1` is dedicated to a channel and performs a multiplication of a value received as input by a precomputed value stored internally. The description made in the component `prod` allows the components `mul0` and `mul1` to run in parallel. `#pragma HLS PIPELINE` allows to pipeline the operations within the loop of the component `prod`.

type `word` is a value of data type `word`. Last, the loop at line 3 is pipelined by adding the adequate directive (line 4) to achieve a better throughput.

1.5 Conclusion

In this chapter, we presented ECC which is the context of this thesis. The ECSM, as the core operation in protocols of ECC applications, has been described and examples of state-of-the-art algorithms to compute it have been stated. We also described the RNS, the number system we choose to perform the ECSM because of its exploitable inherent advantages. The crucial operation BE has been presented. In particular, the state-of-the-art BE proposed in [KKSS00] has been introduced, with which we compare the first contribution of this thesis in the next chapter. We also introduced FPGAs as well as HLS which we use to implement on FPGA our RNS solutions presented in the following

chapters.

Chapter 2

Hierarchical Base Extension

Modular reductions (MRs) are an important part of computations of the elliptic curve scalar multiplication (ECSM), the core operation in protocols of ECC applications. In RNS implementations of the ECSM, the cost of an MR is substantially the cost of the two base extensions (BE) composing it. Therefore, the BE is a crucial operation in RNS implementations for ECC.

In this chapter, we present our BE [DBT19] which uses a hierarchical approach for computing the CRT formula. The main idea is to combine two (three or four) residues of a given number (the input of the BE) in one base to form *super-residues*, and then proceed with the computation of the CRT formula on these *super-residues* in the other base.

Much of the material presented in this chapter comes from [DBT19], a paper we presented at ARITH-26 [1] in June 2019. We draw the attention of the reader to the fact that the implementations, that provide the results presented in this chapter, were perfomed using Vivado HLS 2019.2, on the contrary to Vivado HLS 2017.4 for implementation results in [DBT19]. The implementation codes and optimization files as well as the target FPGA are exactly the same in this chapter as in [DBT19]. These re-implementations are solely motivated by updating the results presented in [DBT19].

Additional notations used throughout the rest of the thesis are introduced in Section 2.1. The BE by Kawamura *et al.* (KBE) [KKSS00], largely regarded as the state-of-the-art BE, and its associated *cox-rower* architecture are detailed in Section 2.2. Then, we present our *hierarchical base extension* (HBE) [DBT19] in Section 2.3, along with a comparison of the theoretical cost of HBE with that of KBE. We also describe an architecture, adapted from the *cox-rower*, to support HBE. In Section 2.4 the benefit from using HBE instead of KBE in RNS modular multiplications is evaluated. Implementation results of HBE and KBE are presented in Section 2.5. This chapter is concluded in Section 2.6.

1. ARITH-26 stands for 26th IEEE Symposium on Computer Arithmetic.

2.1 Notations

RNS bases and residues (of a number) are represented in two dimensions. Two reasons motivate the preference of the two-dimension notations to the common one-dimension notations. We want to better exhibit the hierarchy of the computations toward the CRT and show the intermediate results that we name *super-residues*.

- An RNS base $\mathcal{M}$ of $n = r \times c$ moduli is written $\mathcal{M} = \begin{Bmatrix} m_{1,1} & \dots & m_{1,c} \\ \vdots & \dots & \vdots \\ m_{r,1} & \dots & m_{r,c} \end{Bmatrix}$.

Let i, j be integers such that $1 \le i \le r$ and $1 \le j \le c$.

- $M = \prod_{i=1}^{r} \prod_{j=1}^{c} m_{i,j}$.
- $M_{i,j} = \dfrac{M}{m_{i,j}}$.
- $M^{(i)} = \prod_{j=1}^{c} m_{i,j}$. The $M^{(i)}$, $1 \le i \le r$, are called *super-moduli*.
- $\overline{M^{(i)}} = \dfrac{M}{M^{(i)}}$.
- $\overline{m_{i,j}^{(i)}} = \dfrac{M^{(i)}}{m_{i,j}}$.
- A number x is represented in the RNS base $\mathcal{M}$ by $x_{\mathcal{M}} = \begin{pmatrix} x_{m_{1,1}} & \dots & x_{m_{1,c}} \\ \vdots & \dots & \vdots \\ x_{m_{r,1}} & \dots & x_{m_{r,c}} \end{pmatrix}$, where $x_{m_{i,j}} = |x|_{m_{i,j}}$.

2.2 Kawamura Base Extension

In this section the BE proposed by Kawamura *et al.* [KKSS00] and mentioned in Subsection 1.3.3 is detailed using the two-dimension notations introduced in Section 2.1.

2.2.1 Overview of KBE

The Chinese remainder theorem (CRT) formula, mentioned in Theorem 2 (Chapter 1), is recalled. Let $\mathcal{M}$ and $\mathcal{M}'$ be two RNS bases, $x_{\mathcal{M}}$ and $x_{\mathcal{M}'}$ the RNS representations of a

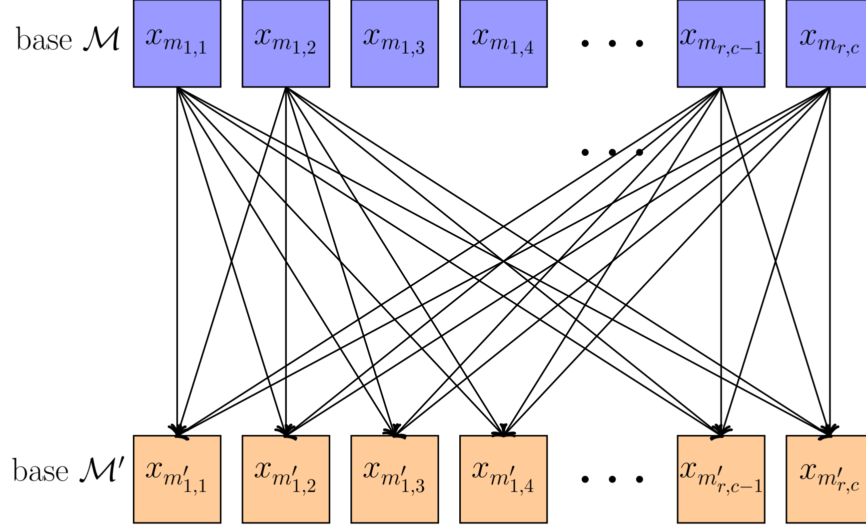


Figure 2.1 – Simplified description of KBE.

number x in $\mathcal{M}$ and $\mathcal{M}'$ respectively. The number x is given by:

$$x = \left| \sum_{i=1}^{r} \sum_{j=1}^{c} \left| x_{m_{i,j}} M_{i,j}^{-1} \right|_{m_{i,j}} M_{i,j} \right|_{M}, \tag{2.1}$$

$$= \left(\sum_{i=1}^{r} \sum_{j=1}^{c} \left| x_{m_{i,j}} M_{i,j}^{-1} \right|_{m_{i,j}} M_{i,j} \right) - hM. \tag{2.2}$$

Like most BEs based on the CRT, the idea of KBE is to compute the CRT formula of Equation 2.2 in the base $\mathcal{M}'$. Figure 2.1 depicts a simplified description of the CRT computation in KBE. The computation of the CRT is performed on all input residues in each modulus of the output RNS base. In other words, for every output residue $x_{m'_{k,l}}$ computed, all input residues $x_{m_{i,j}}$ are multiplied by some precomputed values (not represented in Figure 2.1) and accumulated modulo the corresponding modulus of the output RNS base. The main contribution of [KKSS00] is that the integer h in Equation 2.2 can be approximated by the integer $\tilde{h}$ in Equation 2.3, and the term hM can be subtracted iteratively from the results of the CRT in every modulus $m'_{k,l}$ of the output RNS base.

$$\tilde{h} = \left\lfloor \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{\mathrm{trunc}(|x_{m_{i,j}} M_{i,j}^{-1}|_{m_{i,j}})}{2^{w}} + \sigma \right\rfloor \tag{2.3}$$

2.2.2 KBE Algorithm

Description of KBE Algorithm Using a Two-Dimension Notation

Algorithm 11 is Algorithm 9 of KBE [KKSS00] (Subsection 1.3.3) rewritten using the two-dimension notations presented in Section 2.1. Since r and c are chosen such that $n = rc$, we replace loops bounded by n with nested loops bounded by r and c. The computations performed within the various loops of Algorithm 9 remain unchanged in Algorithm 11; the various indices are adjusted to take into account the two-dimension notations.

The term $\sum_{i=1}^{r}\sum_{j=1}^{c} |x_{m_{i,j}} M_{i,j}^{-1}|_{m_{i,j}} M_{i,j}$ of Equation 2.2 is computed at line 3 and within line 11 of Algorithm 11. The correctness of $x_{\mathcal{M}'}$ depends on the accuracy of $\tilde{h}$, the approximation of h. The value of $\tilde{h}$ is computed in lines 6–8 of Algorithm 11; $\tilde{h}$ is the sum of the bits $h_{i,j}$ computed at line 7. The function trunc approximates $|x_{m_{i,j}} M_{i,j}^{-1}|_{m_{i,j}}$ by its t most significant bits (MSBs), $t < w$, the $w - t$ remaining bits set to 0. In practice, the function trunc selects the t MSBs of $|x_{m_{i,j}} M_{i,j}^{-1}|_{m_{i,j}}$ and the denominator 2^w is replaced by 2^t. Choosing $t \in [4, 8]$ is usually sufficient for RNS implementations of ECC applications.

Validation of KBE

Theorems 3 and 4, from [KKSS00], provide conditions that ensure the approximation $\tilde{h}$ (in Equation 2.3) is h or at worst $h - 1$. Two values d and e are defined below before presenting the theorems.

Definition 6 (from [KKSS00])**.** The values d and e are given by

$$d = \max_{1 \le i \le r,\ 1 \le j \le c} \left(\frac{\widehat{x}_{m_{i,j}} - \mathrm{trunc}(\widehat{x}_{m_{i,j}})}{m_{i,j}} \right) \quad \text{and} \quad e = \max_{1 \le i \le r,\ 1 \le j \le c} \left(\frac{2^w - m_{i,j}}{2^w} \right).$$

Theorem 3 (from [KKSS00])**.** *If* $0 \le rc(d + e) \le \sigma < 1$ *and* $0 \le x < (1 - \sigma)M$*, then* $\tilde{h} = h$*.*

Theorem 4 (from [KKSS00])**.** *If* $\sigma = 0$*,* $0 \le rc(d + e) < 1$ *and* $0 \le x < M$*, then* $\tilde{h} = h$ *or* $\tilde{h} = h - 1$*.*

The approximation of $\widehat{x}_{m_{i,j}}$ and $m_{i,j}$ by respectively $\mathrm{trunc}(\widehat{x}_{m_{i,j}})$ and 2^w introduces an error upperly bounded by $rc(d + e)$. From Theorem 3, the output of KBE is exact if the base $\mathcal{M}$ is chosen such that $rc(d + e) \le \sigma < 1$ and $x < (1 - \sigma)M$. In practice,

Algorithm 11: Kawamura base extension $BE_{\mathcal{M}\to\mathcal{M}'}(x)$ adapted from [KKSS00]

Input: $x_{m_{i,j}}$ for all i, $1 \le i \le r$ and j, $1 \le j \le c$; $\sigma = 0$ or 0.5
Precomp.: $|M_{i,j}^{-1}|_{m_{i,j}}$, $|M_{i,j}|_{m'_{k,l}}$, $|-M|_{m'_{k,l}}$,
for all i, $1 \le i \le r$; j, $1 \le j \le c$; k, $1 \le k \le r$; l, $1 \le l \le c$
Output: $x_{m'_{k,l}}$ for all k, $1 \le k \le r$ and l, $1 \le l \le c$

1 **for** $i \leftarrow 1$ **to** r **parallel do**
2   **for** $j \leftarrow 1$ **to** c **parallel do**
3     $\widehat{x}_{m_{i,j}} \leftarrow \left| x_{m_{i,j}} \times |M_{i,j}^{-1}|_{m_{i,j}} \right|_{m_{i,j}}$
4 **for** $i \leftarrow 1$ **to** r **do**
5   **for** $j \leftarrow 1$ **to** c **do**
6     $\sigma \leftarrow \sigma + \frac{\mathrm{trunc}(\widehat{x}_{m_{i,j}})}{2^w}$
7     $h_{i,j} \leftarrow \lfloor \sigma \rfloor$
8     $\sigma \leftarrow \sigma - h_{i,j}$
9   **for** $k \leftarrow 1$ **to** r **parallel do**
10   **for** $l \leftarrow 1$ **to** c **parallel do**
11         $x_{m'_{k,l}} \leftarrow \left| x_{m'_{k,l}} + \widehat{x}_{m_{i,j}} \times |M_{i,j}|_{m'_{k,l}} + h_{i,j} \times |-M|_{m'_{k,l}} \right|_{m'_{k,l}}$

we use $\sigma = 0.5$. From Thorem 4, the output of KBE is $x_{\mathcal{M}'}$ or $(x + M)_{\mathcal{M}'}$ (that is, x or $x + M$ in the base $\mathcal{M}'$) if $\sigma = 0$, $rc(d + e) < 1$ and $x < M$. Two BEs are used in the RNS Montgomery reduction algorithm [PP95, KKSS00]. The first of the two BEs can be inexact because the correctness of the result mod p is not impacted. The second BE must be exact for the result mod p to be correct. Therefore, it is common to choose RNS bases such that conditions of Theorems 4 and 3 are respectively satisfied for the first and the second BE.

Cost of KBE

Line 3 of Algorithm 11 is performed rc times. The cost of lines 6–8 is negligible because these lines are efficiently performed by a small accumulator on t bits (usually, $t \in [4, 8]$). This accumulator is called the *cox* unit; see Subsection 2.2.3. Line 11 is performed r^2c^2 times. In total, Algorithm 11 costs $r^2c^2 + rc$ EMMs, which is equal to the $n^2 + n$ EMMs for the cost of KBE [KKSS00] mentioned in the state of the art (Subsection 1.3.3).

2.2.3 The *Cox-Rower* Architecture

Kawamura *et al.* [KKSS00] provide the *cox-rower* architecture to perform KBE. This architecture (or its variant) has been used to efficiently implement the RSA (for example [NMSK01]) and the ECSM (for example, [Gui10, BM14]). Figure 2.2 presents a two-dimension-notation version of the *cox-rower* in [Gui10], adapted from [KKSS00].

There is one *rower* per channel, dedicated to computations over one modulus per RNS base. Each *rower* embeds a multiplier and an accumulator, and is able to perform computations of the form $g \leftarrow (g + a \times b + f) \bmod m_{i,j}$ or mod $m'_{i,j}$, where the operands a, b, f and g are of size the width w of the channels. All *rowers* run in parallel. The execution of KBE using the *cox-rower* architecture is described as follows. The precomputations are held in the memory. The multiplications in base $\mathcal{M}$ at line 3 of Algorithm 11 are performed in the *rowers*. These multiplications are not represented in the simplified description of KBE in Figure 2.1. The large multiplexer receives the results of these multiplications from

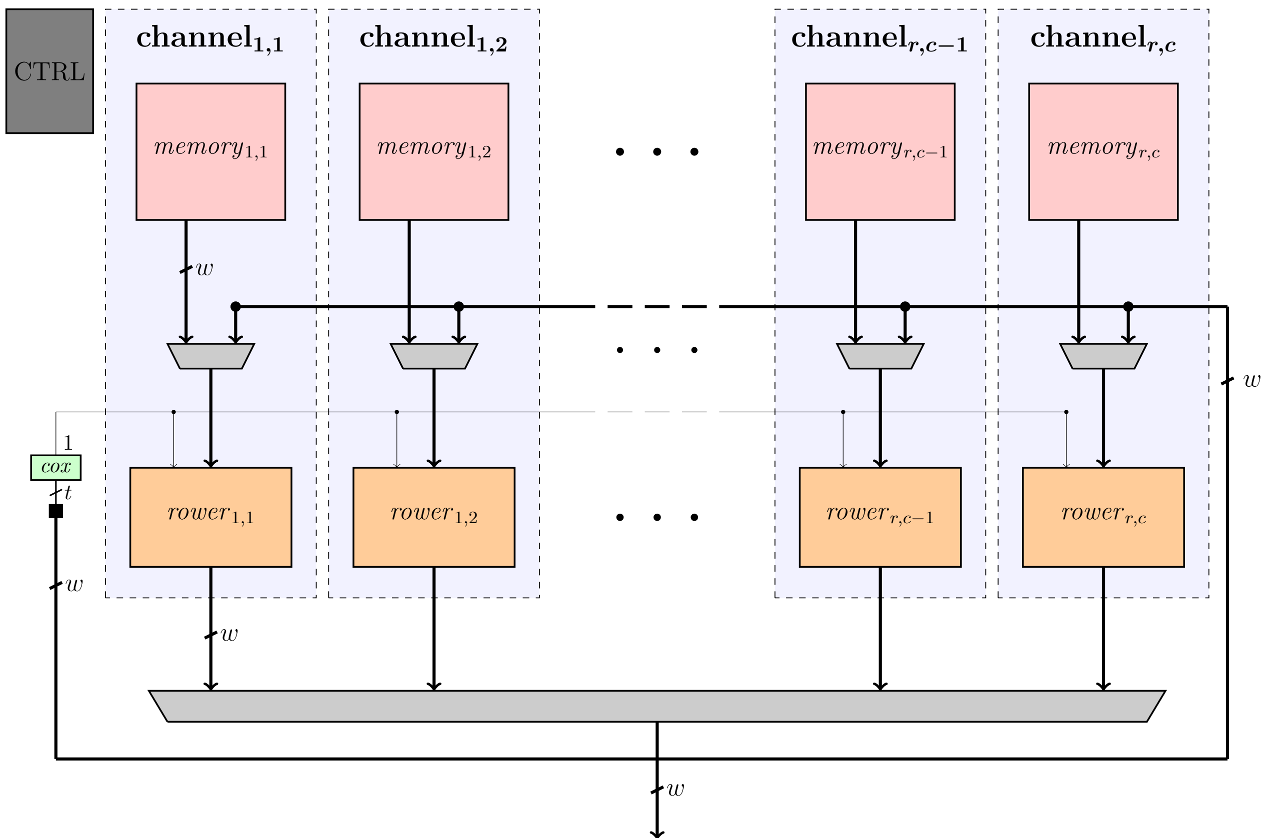


Figure 2.2 – The *cox-rower* architecture in [Gui10], adapted from [KKSS00].

the *rowers* and outputs two buses (left and right). The right bus broadcasts the results to the *rowers* in order to proceed with the computation of the CRT in base $\mathcal{M}'$ (line 11 of Algorithm 11). This broadcast of each result to the various *rowers* corresponds to the arrows in the simplifed description of KBE presented in Figure 2.1. The left bus transfers the results (of the multiplications) to the *cox* unit which computes the bits $h_{i,j}$ (lines 6–8 of Algorithm 11). The *cox* then sends the bits $h_{i,j}$ (0 or 1) to the various *rowers*. Once received in the *rowers*, these bits activate the subtraction of 0 or M in the *rowers*. These subtractions correspond to adding $h_{i,j} \times |-M|_{m'_{k,l}}$ (in line 11 of Algorithm 11).

2.3 Hierarchical Base Extension

In [DBT19] we proposed a hierarchical approach to perform the BE operation.

Hierarchical approaches are not new in RNS. For example, RNS can be used recursively when inside-channel computations are performed in a small RNS base; see [Yas92, HRdH+18]. The performance using this method is not improved significantly compared with using traditional RNS methods. However, this method presents additional properties related to protection against some physical attacks.

Authors of [SA99, Tom11] propose to use RNS bases of three or four moduli, where some of these moduli are factorizable into smaller ones. They target signal processing applications, wherein the used integers (of a few dozens of bits) are smaller than in asymmetric cryptography applications. For this approach to lead to efficient implementations of asymmetric cryptosystems, the MR must be efficient in the (sub)channels using these moduli and their factorizations. Our proposed idea in [DBT19] is a hierarchical approach for computing the BE (through a CRT computation and we target cryptographic applications) rather than an RNS hierarchical representation.

Hierarchical approaches with partial CRTs are proposed in [BLS03, vdH17] to improve the performance of the CRT computation in an RNS-to-positional conversion. To our knowledge, HBE [DBT19] is the first BE that uses a hierarchical approach for full RNS computations.

2.3.1 Overview of HBE

The main idea of HBE can be described in two phases: First, we combine the input residues per group of c (that is, per row) through computations of small and partial CRTs

in the base $\mathcal{M}$. The results of these partial-CRT computations are named *super-residues*. These combinations are depicted with some simplifications in Figure 2.3 for $c = 2$. The residues $x_{m_{i,1}}$ and $x_{m_{i,2}}$ (in blue) are combined into the *super-residues* $\widehat{x}_{M^{(i)}}$ (in blue). In the second phase, we proceed with the computation of the CRT on these *super-residues* in the base $\mathcal{M}'$. In Figure 2.3, this phase is simplified into the arrows going from the *super-residues* $\widehat{x}_{M^{(i)}}$ (in blue) to the residues $x_{m'_{i,j}}$ (in orange).

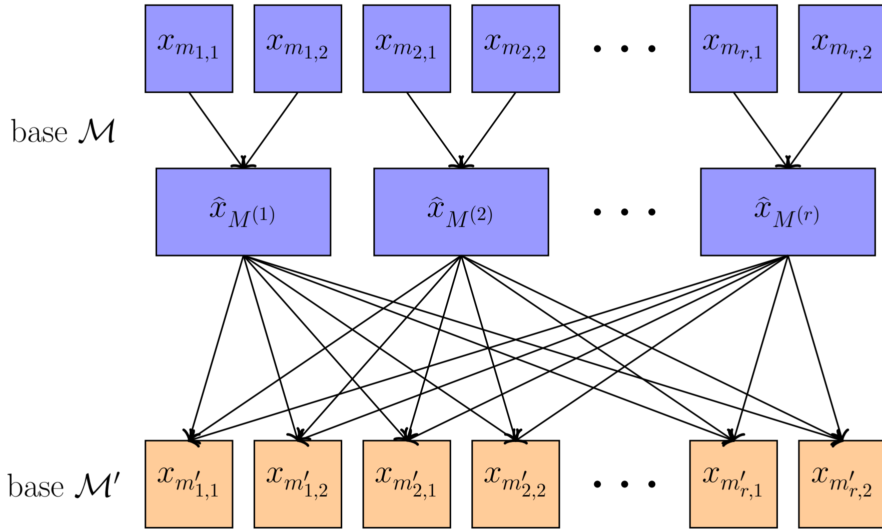


Figure 2.3 – Simplified description of HBE $c = 2$.

2.3.2 HBE Algorithm

Desription of the Algorithm

HBE is presented in Algorithm 12. Like KBE and generally BEs based on the CRT, HBE computes the CRT formula of Equation 2.2 in the RNS base $\mathcal{M}'$. Lines 1–3 are the same in Algorithm 12 as in Algorithm 11. Lines 4–7 correspond to the computation of partial CRTs on each row, resulting in the *super-residues* $\widehat{x}_{M^{(i)}}$ (one *super-residue* per row). The *super-residues* are not reduced at this step. Therefore, $\widehat{x}_{M^{(i)}} < cM^{(i)}$ for all i, $1 \leq i \leq r$.

Lines 9–11 of Algorithm 12 of HBE are an adaptation of lines 6–8 of Algorithm 11 of KBE. This time, the values h_i are of size $1 + \lceil \log_2 c \rceil$ bits. Under conditions of Theorems 3 and 4 initially specified for KBE, we prove that the approximation of h

Algorithm 12: Hierarchical base extension $BE_{\mathcal{M}\to\mathcal{M}'}(x)$ [DBT19]

Input: $x_{m_{i,j}}$ for all i, $1 \leq i \leq r$ and j, $1 \leq j \leq c$; $\sigma = 0$ or 0.5
Precomp.: $|M_{i,j}^{-1}|_{m_{i,j}}$, $|M_{i,j}|_{m'_{k,l}}$, $|-M|_{m'_{k,l}}$,
for all i, $1 \leq i \leq r$; j, $1 \leq j \leq c$; k, $1 \leq k \leq r$; l, $1 \leq l \leq c$
Output: $x_{m'_{k,l}}$ for all k, $1 \leq k \leq r$ and l, $1 \leq l \leq c$

1 **for** $i \leftarrow 1$ **to** r **parallel do**
2   **for** $j \leftarrow 1$ **to** c **parallel do**
3         $\widehat{x}_{m_{i,j}} \leftarrow \left| x_{m_{i,j}} \times |M_{i,j}^{-1}|_{m_{i,j}} \right|_{m_{i,j}}$
4 **for** $i \leftarrow 1$ **to** r **parallel do**
5     $\widehat{x}_{M^{(i)}} \leftarrow 0$
6   **for** $j \leftarrow 1$ **to** c **do**
7         $\widehat{x}_{M^{(i)}} \leftarrow \widehat{x}_{M^{(i)}} + \widehat{x}_{m_{i,j}} \times \overline{m_{i,j}^{(i)}}$ (without modular reduction)
8 **for** $i \leftarrow 1$ **to** r **do**
9     $\sigma \leftarrow \sigma + \frac{\text{trunc}(\widehat{x}_{M^{(i)}})}{2^{w\times c}}$
10     $h_i \leftarrow \lfloor \sigma \rfloor$
11     $\sigma \leftarrow \sigma - h_i$
12   **for** $k \leftarrow 1$ **to** r **parallel do**
13   **for** $l \leftarrow 1$ **to** c **parallel do**
14             $\widehat{x}_{m'_{k,l,i}} \leftarrow |\widehat{x}_{M^{(i)}}|_{m'_{k,l}}$
15             $x_{m'_{k,l}} \leftarrow \left| x_{m'_{k,l}} + \widehat{x}_{m'_{k,l,i}} \times \left| \overline{M^{(i)}} \right|_{m'_{k,l}} + h_i \times |-M|_{m'_{k,l}} \right|_{m'_{k,l}}$

reaches the same result, that is, $\sum_{i=1}^{n} h_i = h$ or $h-1$. The proof is provided in the part “Validation of HBE” below.

Lines 14 and 15 are the most costly lines of Algorithm 12 since they are performed r^2c times. The modular reduction $\widehat{x}_{M^{(i)}} \bmod m'_{k,l}$ is performed at line 14. Line 15 of Algorithm 12 corresponds to line 11 of Algorithm 11. Since the factors $\overline{m_{i,j}^{(i)}}$ are inserted during the computation of the *super-residues* $\widehat{x}_{M^{(i)}}$, multiplying the results of line 14 by $\overline{M^{(i)}}$ is sufficient to get the CRT computed at line 15.

To summarize, the input residues are combined per row through partial CRTs to get the *super-residues*. Then the computation of the CRT in the output base is proceeded on the *super-residues* with an approximation similar to that of KBE [KKSS00]. The main computation (lines 14 and 15 of Algorithm 12) of HBE is performed r^2c times while that of KBE (line 11 of Algorithm 11) is perfomed r^2c^2 times. However, HBE introduces

$w \times (c-1)w$ bit multiplications (line 7 of Algorithm 12) and reductions of $cw + \lceil \log_2 c \rceil$ bits by w-bit integers (line 14 of Algorithm 12).

The first three lines of Algorithm 11 of KBE can be hidden in the computation of the RNS MR from [PP95] by combining some precomputations of Algorithm 11 of KBE with some of Algorithm 10 of the RNS MR; see [GLP+12, Gui10, KKSY18]. As a result, the implementation of the RNS MR is made faster. Algorithm 12 of HBE is also compatible with this optimization.

We choose $c \in \{2, 3, 4\}$ small for RNS implementations of ECC applications to keep the inherent parallelism of RNS as much as possible. The case $c = 2$ reveals some simplifications, notably in the architecture; see Subsection 2.3.3.

Validation of HBE

The CRT formula is recalled.

$$x = \left(\sum_{i=1}^{r} \sum_{j=1}^{c} \left| x_{m_{i,j}} M_{i,j}^{-1} \right|_{m_{i,j}} M_{i,j} \right) - hM. \tag{2.4}$$

To validate HBE presented in Algorithm 12, we start by proving that the term in parentheses in Equation 2.4 is computed within the execution of Algorithm 12. We then discuss the correctness of h approximated by computing the values h_i in Algorithm 12.

The term in parentheses in Equation 2.4 is

$$\begin{aligned}
s_x &= \sum_{i=1}^{r} \sum_{j=1}^{c} \left| x_{m_{i,j}} M_{i,j}^{-1} \right|_{m_{i,j}} M_{i,j} \\
&= \sum_{i=1}^{r} \sum_{j=1}^{c} \widehat{x}_{m_{i,j}} M_{i,j} \\
&= \sum_{i=1}^{r} \sum_{j=1}^{c} \widehat{x}_{m_{i,j}} \frac{M}{m_{i,j}} = \sum_{i=1}^{r} \sum_{j=1}^{c} \widehat{x}_{m_{i,j}} \frac{M^{(i)}}{m_{i,j}} \times \frac{M}{M^{(i)}} \\
&= \sum_{i=1}^{r} \left(\sum_{j=1}^{c} \widehat{x}_{m_{i,j}} \overline{m_{i,j}^{(i)}} \right) \overline{M^{(i)}} \\
&= \sum_{i=1}^{r} \widehat{x}_{M^{(i)}} \times \overline{M^{(i)}}
\end{aligned} \tag{2.5}$$

The reader can recognize that Equation 2.5 is computed (mod $m'_{k,l}$) at line 14 and the

first two terms at line 15 of Algorithm 12.

Theorems 3 and 4 from [KKSS00] give an upper bound for the error made by approximating h by $\tilde{h}$, using Algorithm 11 of KBE. Since $\mathrm{trunc}(\widehat{x}_{m_{i,j}}) \leq \widehat{x}_{m_{i,j}}$ and $2^w > m_{i,j}$ for all i, $1 \leq i \leq r$ and j, $1 \leq j \leq c$, we obtain

$$\sum_{i=1}^{r}\sum_{j=1}^{c}\frac{\mathrm{trunc}(\widehat{x}_{m_{i,j}})}{2^w} < \sum_{i=1}^{r}\sum_{j=1}^{c}\frac{\widehat{x}_{m_{i,j}}}{m_{i,j}}. \tag{2.6}$$

Besides, from the proofs of Theorems 3 and 4 in [KKSS00],

$$\sum_{i=1}^{r}\sum_{j=1}^{c}\frac{\widehat{x}_{m_{i,j}}}{m_{i,j}} - \sum_{i=1}^{r}\sum_{j=1}^{c}\frac{\mathrm{trunc}(\widehat{x}_{m_{i,j}})}{2^w} < rc(d+e). \tag{2.7}$$

Our discussion of the correctness of h for HBE with $c \in \{2,3,4\}$ is built upon Theorems 3 and 4 from [KKSS00] with respect to KBE. Clearly, if the base $\mathcal{M}$ is chosen verifying Theorems 3 and 4 for KBE, then $\mathcal{M}$ can be used for HBE, $c \in \{2,3,4\}$.

Let assume $m_{1,1} < m_{1,j}$ for all j, $2 \leq j \leq c$, and $m_{1,j} < m_{i,j}$ for all i, $2 \leq i \leq r$ and j, $1 \leq j \leq c$. One can rearrange the moduli in the two-dimension notation of the base $\mathcal{M}$ to get these inequalities.

With respect to Algorithm 12 of HBE, let the values $d'_{M^{(i)}}$, $e'_{M^{(i)}}$, d' and e' be defined by

$$d'_{M^{(i)}} = \frac{\widehat{x}_{M^{(i)}} - \mathrm{trunc}(\widehat{x}_{M^{(i)}})}{M^{(i)}}, \qquad e'_{M^{(i)}} = \frac{2^{cw} - M^{(i)}}{2^{cw}}$$

and

$$d' = \max_{1\leq i\leq r}\left(d'_{M^{(i)}}\right), \qquad e' = \max_{1\leq i\leq r}\left(e'_{M^{(i)}}\right).$$

The function trunc now approximates $\widehat{x}_{M^{(i)}}$ (of $cw + \lceil\log_2 c\rceil$ bits) by its $t + \lceil\log_2 c\rceil$ MSBs, the $2w-t$ remaining bits set to 0. The super-modulus $M^{(i)}$ is approximated by 2^{cw}. To ease the notation and improve the clarity of the argument, we assume $c = 2$—the arguments for the cases $c = 3, 4$ being analogous to that of $c = 2$. From this assumption, the function trunc approximates $\widehat{x}_{M^{(i)}}$ by its $t+1$ MSBs, and the super-modulus $M^{(i)}$ is approximated by 2^{2w}. The introduced values $d'_{M^{(i)}}$, $e'_{M^{(i)}}$, d' and e' are rewritten

$$d'_{M^{(i)}} = \frac{\widehat{x}_{M^{(i)}} - \mathrm{trunc}(\widehat{x}_{M^{(i)}})}{M^{(i)}}, \qquad e'_{M^{(i)}} = \frac{2^{2w} - M^{(i)}}{2^{2w}} \tag{2.8}$$

and

$$d' = \frac{\widehat{x}_{M^{(1)}} - \mathrm{trunc}(\widehat{x}_{M^{(1)}})}{M^{(1)}}, \qquad e' = \frac{2^{2w} - M^{(1)}}{2^{2w}}. \tag{2.9}$$

Remark 1 (**On d and d' values**). The various $\widehat{x}_{m_{i,j}}$ and $\widehat{x}_{M^{(i)}}$ as well as their truncated values cannot be predicted in the numerator of d in Definition 6 and that of d' in Equation 2.9. Therefore, we assume the upper bounds

$$d = \frac{2^{w-t} - 1}{m_{1,1}}, \qquad d' = \frac{2^{2w-t} - 1}{M^{(1)}}. \tag{2.10}$$

We want to prove that

$$\sum_{i=1}^{r} \frac{\mathrm{trunc}(\widehat{x}_{M^{(i)}})}{2^{2w}} < \sum_{i=1}^{r} \sum_{j=1}^{2} \frac{\widehat{x}_{m_{i,j}}}{m_{i,j}} \tag{2.11}$$

and

$$\sum_{i=1}^{r} \sum_{j=1}^{2} \frac{\widehat{x}_{m_{i,j}}}{m_{i,j}} - \sum_{i=1}^{r} \frac{\mathrm{trunc}(\widehat{x}_{M^{(i)}})}{2^{2w}} < 2r(d+e). \tag{2.12}$$

For all i, $1 \le i \le r$,

$$\sum_{j=1}^{2} \frac{\widehat{x}_{m_{i,j}}}{m_{i,j}} = \frac{\widehat{x}_{m_{i,1}} m_{i,2} + \widehat{x}_{m_{i,2}} m_{i,1}}{m_{i,1} m_{i,2}} = \frac{\widehat{x}_{M^{(i)}}}{M^{(i)}}.$$

Since $\mathrm{trunc}(\widehat{x}_{M^{(i)}}) \le \widehat{x}_{M^{(i)}}$ and $2^{2w} > M^{(i)}$, we obtain for all i, $1 \le i \le r$,

$$\frac{\mathrm{trunc}(\widehat{x}_{M^{(i)}})}{2^{2w}} < \sum_{j=1}^{2} \frac{\widehat{x}_{m_{i,j}}}{m_{i,j}},$$

and consequently

$$\sum_{i=1}^{r} \frac{\mathrm{trunc}(\widehat{x}_{M^{(i)}})}{2^{2w}} < \sum_{i=1}^{r} \sum_{j=1}^{2} \frac{\widehat{x}_{m_{i,j}}}{m_{i,j}},$$

which is Equation 2.11.

Let us bound $\sum_{i=1}^{r} \sum_{j=1}^{2} \frac{\widehat{x}_{m_{i,j}}}{m_{i,j}} - \sum_{i=1}^{r} \frac{\mathrm{trunc}(\widehat{x}_{M^{(i)}})}{2^{2w}}$.

From Equation 2.8, we do have on one hand $\mathrm{trunc}(\widehat{x}_{M^{(i)}}) = \widehat{x}_{M^{(i)}} - M^{(i)} \times d'_{M^{(i)}}$, and on the other hand $e'_{M^{(i)}} = 1 - \frac{M^{(i)}}{2^{2w}}$, which implies $1 - e'_{M^{(i)}} = \frac{M^{(i)}}{2^{2w}}$, that is, $\frac{1}{2^{2w}} = \frac{1 - e'_{M^{(i)}}}{M^{(i)}}$.

We get

$$\begin{aligned}
\sum_{i=1}^{r} \frac{\operatorname{trunc}(\widehat{x}_{M^{(i)}})}{2^{2w}} &= \sum_{i=1}^{r} \left(\widehat{x}_{M^{(i)}} - M^{(i)} \times d'_{M^{(i)}}\right) \times \frac{1 - e'_{M^{(i)}}}{M^{(i)}} \\
&= \sum_{i=1}^{r} \frac{\widehat{x}_{M^{(i)}}}{M^{(i)}} - \sum_{i=1}^{r} e'_{M^{(i)}} \frac{\widehat{x}_{M^{(i)}}}{M^{(i)}} - \sum_{i=1}^{r} (1 - e'_{M^{(i)}})\, d'_{M^{(i)}} \\
&= \sum_{i=1}^{r} \frac{\widehat{x}_{M^{(i)}}}{M^{(i)}} - \sum_{i=1}^{r} e'_{M^{(i)}} \frac{\widehat{x}_{M^{(i)}}}{M^{(i)}} - \sum_{i=1}^{r} \frac{M^{(i)}}{2^{2w}} d'_{M^{(i)}}. \qquad (2.13)
\end{aligned}$$

The terms at the right of the equal sign in Equation 2.13 verify

$$\sum_{i=1}^{r} \frac{\widehat{x}_{M^{(i)}}}{M^{(i)}} = \sum_{i=1}^{r} \sum_{j=1}^{2} \frac{\widehat{x}_{m_{i,j}}}{m_{i,j}};$$

$$\sum_{i=1}^{r} e'_{M^{(i)}} \frac{\widehat{x}_{M^{(i)}}}{M^{(i)}} < 2re' \text{ because } \frac{\widehat{x}_{M^{(i)}}}{M^{(i)}} < 2, \text{ and } e'_{M^{(i)}} \leq e' \text{ by definition;}$$

$$\sum_{i=1}^{r} \frac{M^{(i)}}{2^{2w}} d'_{M^{(i)}} < rd' \text{ because } \frac{M^{(i)}}{2^{2w}} < 1, \text{ and } d'_{M^{(i)}} \leq d' \text{ by definition.}$$

It follows from Equation 2.13 that

$$\sum_{i=1}^{r} \frac{\operatorname{trunc}(\widehat{x}_{M^{(i)}})}{2^{2w}} > \sum_{i=1}^{r} \sum_{j=1}^{2} \frac{\widehat{x}_{m_{i,j}}}{m_{i,j}} - 2re' - rd',$$

that is,

$$\sum_{i=1}^{r} \sum_{j=1}^{2} \frac{\widehat{x}_{m_{i,j}}}{m_{i,j}} - \sum_{i=1}^{r} \frac{\operatorname{trunc}(\widehat{x}_{M^{(i)}})}{2^{2w}} < 2re' + rd'. \qquad (2.14)$$

Proving $e' + 2^{-1}d' < e + d$ suffices to get Equation 2.12 from Equation 2.14. We have

$$\begin{aligned}
d - 2^{-1}d' &= \frac{2^{w-t} - 1}{m_{1,1}} - \frac{2^{-1}\left(2^{2w-t} - 1\right)}{M^{(1)}} \\
&= \frac{m_{1,2}\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)}{M^{(1)}} \qquad \left(\text{recall } M^{(1)} = m_{1,1} \times m_{1,2}\right) \qquad (2.15)
\end{aligned}$$

and

$$
\begin{aligned}
e' - e &= \frac{2^{2w} - M^{(1)}}{2^{2w}} - \frac{2^w - m_{1,1}}{2^w} \qquad \text{(to get the value of } e \text{ from Definition 6,} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{use the fact that } \forall\, i,j,\, m_{1,1} \leq m_{i,j}) \\
&= 1 - \frac{M^{(1)}}{2^{2w}} - \left(1 - \frac{m_{1,1}}{2^w}\right) = \frac{m_{1,1}}{2^w} - \frac{M^{(1)}}{2^{2w}} \\
&= \frac{m_{1,1} \times 2^w - M^{(1)}}{2^{2w}} = \frac{m_{1,1}\left(2^w - m_{1,2}\right)}{2^{2w}} \\
&= \frac{m_{1,1} \times \varepsilon_{1,2}}{2^{2w}} \qquad (2.16)
\end{aligned}
$$

From Equations 2.15 and 2.16, we obtain

$$
\begin{aligned}
\frac{d - 2^{-1}d}{e' - e} &= \frac{m_{1,2}\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)}{M^{(1)}} \times \frac{2^{2w}}{m_{1,1} \times \varepsilon_{1,2}} \\
&= \frac{m_{1,2}\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)}{m_{1,1} \times \varepsilon_{1,2}} \times \frac{2^{2w}}{M^{(1)}} \qquad (2.17)
\end{aligned}
$$

Lemma 5. *If* $t < \frac{w}{2} - 1$ *then* $\frac{m_{1,2}\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)}{m_{1,1} \times \varepsilon_{1,2}} > 1$.

Proof. The inequality $m_{1,2} > m_{1,1}$ implies $m_{1,2} \geq m_{1,1} + 1$ and $\varepsilon_{1,2} \leq \varepsilon_{1,1} - 1$, which leads to

$$
m_{1,2}\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right) \geq \left(m_{1,1} + 1\right)\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)
$$

and

$$
m_{1,1} \times \varepsilon_{1,2} \leq m_{1,1}\left(\varepsilon_{1,1} - 1\right).
$$

Subsequently,

$$
\frac{m_{1,2}\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)}{m_{1,1} \times \varepsilon_{1,2}} \geq \frac{\left(m_{1,1} + 1\right)\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)}{m_{1,1}\left(\varepsilon_{1,1} - 1\right)}. \qquad (2.18)
$$

We have

$$\begin{aligned}(m_{1,1}+1)(2^{w-t}-1) - 2^{-1}(2^{2w-t}-1) - m_{1,1}(\varepsilon_{1,1}-1)\\ &= m_{1,1}\times 2^{w-t} + 2^{w-t} - m_{1,1} - 1 - 2^{2w-t-1} + 2^{-1} - m_{1,1}(\varepsilon_{1,1}-1)\\ &= (m_{1,1}\times 2^{w-t} - 2^{2w-t-1}) + (2^{w-t}+2^{-1}) - (m_{1,1}\times\varepsilon_{1,1}+1). \qquad (2.19)\end{aligned}$$

The first term at the right of the equal sign in Equation 2.19 verifies

$$m_{1,1}\times 2^{w-t} - 2^{2w-t-1} > 2^{2w-t-1} - 2^{\frac{3w}{2}-t}. \qquad (2.20)$$

Indeed, $\varepsilon_{1,1} < 2^{\frac{w}{2}}$ implies $-\varepsilon_{1,1} > -2^{\frac{w}{2}}$, which leads to $m_{1,1} = 2^w - \varepsilon_{1,1} > 2^w - 2^{\frac{w}{2}}$, and subsequently

$$\begin{aligned}m_{1,1}\times 2^{w-t} > 2^{w-t}\left(2^w - 2^{\frac{w}{2}}\right) &= 2^{2w-t} - 2^{\frac{3w}{2}-t}\\ &= 2\times 2^{2w-t-1} - 2^{\frac{3w}{2}-t}.\end{aligned}$$

It follows that $m_{1,1}\times 2^{w-t} - 2^{2w-t-1} > 2^{2w-t-1} - 2^{\frac{3w}{2}-t}$.

The third term at the right of the equal sign in Equation 2.19 verifies

$$2^{2w-t-1} - 2^{\frac{3w}{2}-t} > m_{1,1}\times\varepsilon_{1,1} + 1 = m_{1,1}(\varepsilon_{1,1} + m_{1,1}^{-1}). \qquad (2.21)$$

Indeed, $2^{2w-t-1} - 2^{\frac{3w}{2}-t}$ can be rewritten

$$2^{2w-t-1} - 2^{\frac{3w}{2}-t} = 2^w \times 2^{\frac{w}{2}} \times 2^{-t}\left(2^{\frac{w}{2}-1} - 1\right), \qquad (2.22)$$

where the factors of the right of the equal sign verify inequalities detailed as follows.

- The factor 2^w in Equation 2.22 verifies $2^w > m_{1,1}$.
- The factor $2^{\frac{w}{2}}$ in Equation 2.22 verifies $2^{\frac{w}{2}} > \varepsilon_{1,1} + m_{1,1}^{-1}$. This inequality comes from $2^{\frac{w}{2}} \geq \varepsilon_{1,1}+1$ (since $\varepsilon_{1,1} < 2^{\frac{w}{2}}$) and $\varepsilon_{1,1}+1 > \varepsilon_{1,1} + m_{1,1}^{-1}$ (since $m_{1,1}^{-1} < 1$, due to the fact that the size w of $m_{1,1}$ is in [10, 64], that is, $m_{1,1} > 1$).
- The factor $2^{-t}\left(2^{\frac{w}{2}-1} - 1\right)$ in Equation 2.22 verifies $2^{-t}\left(2^{\frac{w}{2}-1} - 1\right) \geq 1$. Since $t < \dfrac{w}{2} - 1$, we have $2^t < 2^{\frac{w}{2}-1}$, which implies $2^t \leq 2^{\frac{w}{2}-1} - 1$, and consequently $1 \leq 2^{-t}\left(2^{\frac{w}{2}-1} - 1\right)$.

From Equations 2.20 and 2.21, we obtain

$$m_{1,1} \times 2^{w-t} - 2^{2w-t-1} > m_{1,1} \times \varepsilon_{1,1} + 1,$$

which implies

$$(m_{1,1} \times 2^{w-t} - 2^{2w-t-1}) - (m_{1,1} \times \varepsilon_{1,1} + 1) > 0.$$

It follows, *a fortiori*, that

$$(m_{1,1} \times 2^{w-t} - 2^{2w-t-1}) + (2^{w-t} + 2^{-1}) - (m_{1,1} \times \varepsilon_{1,1} + 1) > 0.$$

Considering Equation 2.19, we get

$$(m_{1,1} + 1)(2^{w-t} - 1) - 2^{-1}(2^{2w-t} - 1) - m_{1,1}(\varepsilon_{1,1} - 1) > 0,$$

that is,

$$\frac{(m_{1,1} + 1)\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)}{m_{1,1}\left(\varepsilon_{1,1} - 1\right)} > 1.$$

From Equation 2.18 we then obtain

$$\frac{m_{1,2}\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)}{m_{1,1} \times \varepsilon_{1,2}} > 1.$$

□

In practice, the condition $t < \frac{w}{2} - 1$ of Lemma 5 is verified for RNS implementations of asymmetric cryptosystems where w is usually in [10, 64]. For $w \geq 11$ we can always find t in [4, 8] verifying this condition. For $w = 10$, it suffices to take $t = 3$ to ensure this condition. It follows from Lemma 5 that

$$\frac{m_{1,2}\left(2^{w-t} - 1\right) - 2^{-1}\left(2^{2w-t} - 1\right)}{m_{1,1} \times \varepsilon_{1,2}} > 1.$$

Since $\frac{2^{2w}}{M^{(1)}} > 1$, Equation 2.17 then implies

$$\frac{d - 2^{-1}d'}{e' - e} > 1.$$

It follows that $d - 2^{-1}d' > e' - e$, that is,

$$e' + 2^{-1}d' < e + d \tag{2.23}$$

We conclude from Equations 2.14 and 2.23 that

$$\sum_{i=1}^{r}\sum_{j=1}^{2}\frac{\widehat{x}_{m_{i,j}}}{m_{i,j}} - \sum_{i=1}^{r}\frac{\mathrm{trunc}(\widehat{x}_{M^{(i)}})}{2^{2w}} < 2r(e+d), \tag{2.24}$$

which is Equation 2.12.

Cost of HBE

Some operands at lines 7 and 14 of Algorithm 12 are of size greater than the width w of the channels. At line 7, there is a product of w-bit and $(c-1)w$-bit integers as well as an accumulation of cw-bit integers. The result of line 7, a $(cw + \lceil \log_2 c \rceil)$-bit integer, is reduced modulo a w bit integer at line 14. We introduce three additional notations to represent the cost of these operations. Our goal is to compare the cost of these operations to the usual metric, namely EMM. From there, we can fairly evaluate the HBE cost against the KBE one.

- $\mathrm{CMUL}(w, w')$: cost of a multiplication of a w-bit integer by a w'-bit integer.
- $\mathrm{CADD}(w, w')$: cost of an addition/subtraction of a w-bit integer to a w'-bit integer.
- $\mathrm{CMR}(w, w')$: cost of a w-bit integer mod a w'-bit integer reduction.

Similarly to Algorithm 11 of KBE, lines 1–3 of Algorithm 12 of HBE cost rc EMMs. Lines 4–7 cost $rc(\mathrm{CMUL}(w,(c-1)w) + \mathrm{CADD}(cw, cw))$. As for the cost of lines 6–8 of Algorithm 11 of KBE, the cost of lines 9–11 of Algorithm 12 of HBE is not counted. Indeed, the cost of these lines is negligible since the operations at these lines are performed efficiently by the *cox* unit, a small accumulator of $t + \lceil \log_2 c \rceil$ bits. Finally, lines 14 and 15 are operated r^2c times. Line 14 costs r^2c $\mathrm{CMR}(cw + \lceil \log_2 c \rceil, w)$. Line 15 costs $r^2c(\mathrm{EMM} + 2\mathrm{CADD}(w, w))$.

Overall, Algorithm 12 of HBE costs

$$\begin{aligned} &r^2c\,(\mathrm{EMM} + \mathrm{CMR}(cw + \lceil \log_2 c \rceil, w) + 2\ \mathrm{CADD}(w,w)) \\ &\quad + rc\,(\mathrm{EMM} + \mathrm{CMUL}(w,(c-1)w) + \mathrm{CADD}(cw,cw))\,. \end{aligned} \tag{2.25}$$

Assuming $c = 2$, we obtain for the cost of HBE

$$\begin{aligned} 2r^2 &(\mathrm{EMM} + \mathrm{CMR}(2w+1, w) + 2\ \mathrm{CADD}(w, w)) \\ &+ 2r\,(\mathrm{EMM} + \mathrm{CMUL}(w, w) + \mathrm{CADD}(2w, 2w)) \,. \end{aligned}$$

In DSP slices of modern FPGAs, additions are usually hidden in the pipeline computing the multiplications. Therefore, we concentrate the cost on the modular multiplications, similarly to most works in the literature. With this simplification, we obtain for the cost of HBE

$$2r^2\ (\mathrm{EMM} + \mathrm{CMR}(2w+1, w)) + 2r\,(\mathrm{EMM} + \mathrm{CMUL}(w, w)) \,.$$

Overestimating a $\mathrm{CMUL}(w, w)$ to be equal to an EMM, we obtain that HBE ($c = 2$) costs

$$2r^2\ \mathrm{EMM} + 2r^2\ \mathrm{CMR}(2w+1, w) + 4r\ \mathrm{EMM},$$

that is,

$$\frac{n^2}{2}\ \mathrm{EMM} + \frac{n^2}{2}\ \mathrm{CMR}(2w+1, w) + 2n\ \mathrm{EMM}. \tag{2.26}$$

Recall that KBE costs

$$n^2\ \mathrm{EMM} + n\ \mathrm{EMM}. \tag{2.27}$$

The comparison of the HBE ($c = 2$) cost with the KBE one is made difficult by the presence of the term $\frac{n^2}{2}\ \mathrm{CMR}(2w+1, w)$ in Equation 2.26. HBE ($c = 2$) may cost less than KBE if $\mathrm{CMR}(2w + 1, w) \ll \mathrm{EMM}$. In order to compare a $\mathrm{CMR}(2w + 1, w)$ with an EMM, we implemented the two operations on a Xilinx XCV7020 FPGA (using Vivado HLS 2019.2) for various widths w of channels. The implementation results are reported in Table 2.1. In most cases, an EMM costs about twice a $\mathrm{CMR}(2w + 1, w)$ in time and/or in area. Therefore, we estimate $\mathrm{EMM}/4 \leq \mathrm{CMR}(2w + 1, w) \leq \mathrm{EMM}/2$.

Estimating one EMM to be in $\{2, 3, 4\}\ \mathrm{CMR}(2w + 1, w)$, we evaluate the theoretical cost ratio HBE ($c = 2$) / KBE of one BE for various numbers n of moduli. The result of this evaluation is shown in Figure 2.4. The theoretical cost reduction of HBE ($c = 2$) against KBE is up to 35% using these estimations.

Table 2.1 – FPGA implementation results for $\mathrm{CMR}(2w+1, w)$ and EMM operations on a Xilinx XC7Z020 FPGA.

operations	$\mathrm{CMR}(2w+1, w)$				EMM			
w (bits)	17	20	24	28	17	20	24	28
nb. slices	1	1	24	29	1	23	1	39
nb. DSPs	2	2	1	1	3	3	4	5
nb. cycles	1	1	2	2	2	2	2	3
time (ns)	2.2	2.2	9.3	9.5	10.3	7.8	7.8	18.7

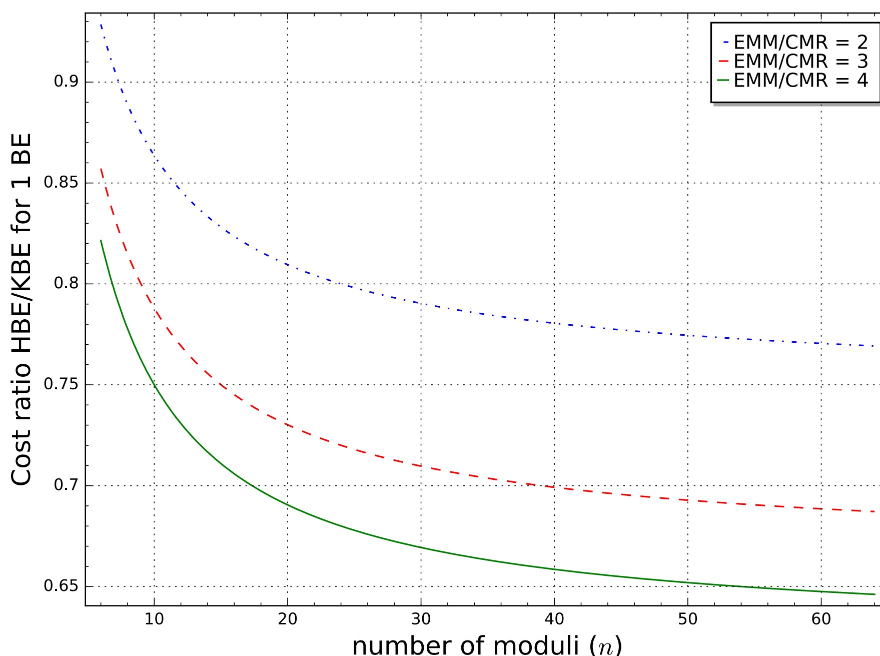


Figure 2.4 – Theoretical cost ratio HBE/KBE of one BE for various estimations EMM/$\mathrm{CMR}(2w+1, w)$ and numbers n of moduli.

2.3.3 A *Cox-Rower* Architecture Adapted for HBE

The *cox-rower* architecture [KKSS00], initially designed for KBE, has been adapted for HBE to exploit its efficiency. The adapted architecture for $c = 2$ is presented in Figure 2.5.

Algorithm 12 of HBE consists of three primary (groups of) loops. The first and the third loops can be performed on the HBE architecture (Figure 2.5) as well as on the KBE

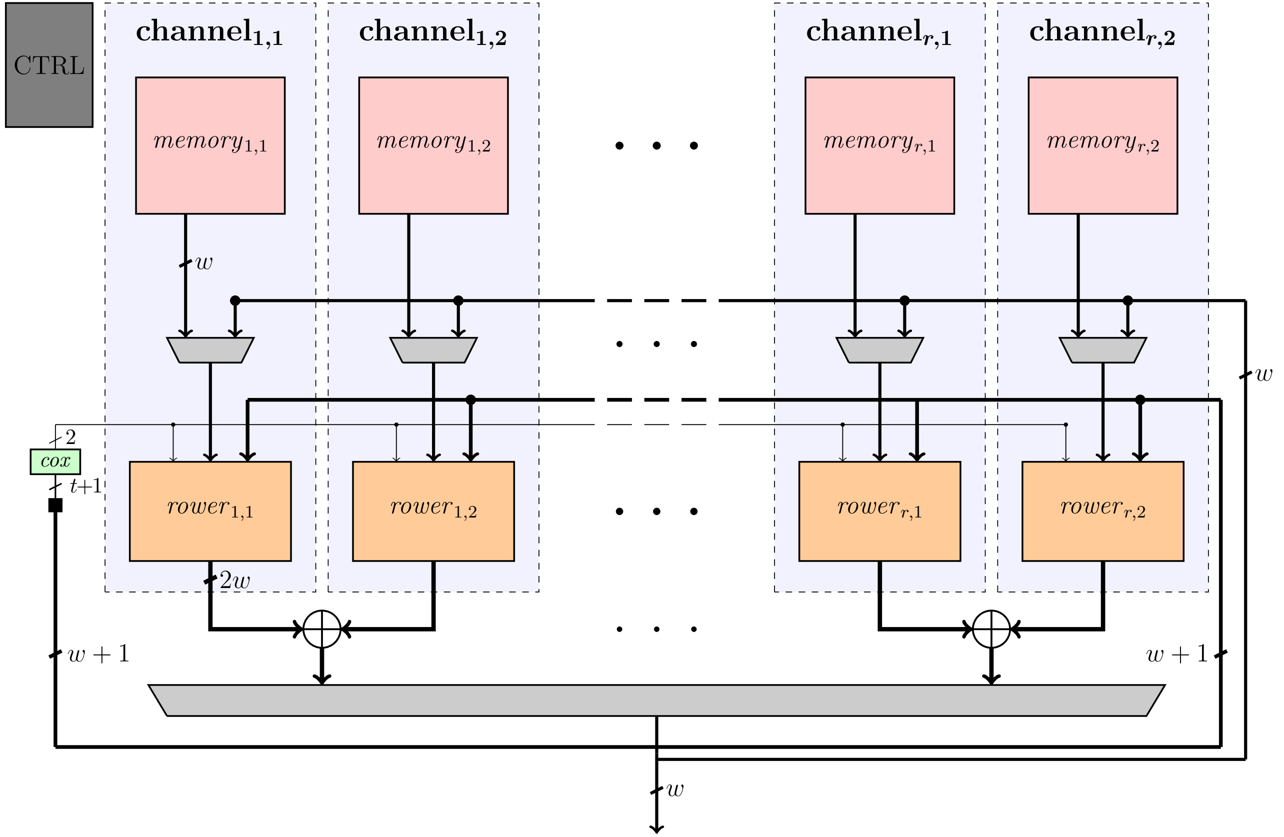


Figure 2.5 – Architecture of HBE $c = 2$.

one (the original *cox-rower* at Figure 2.2) since computations within these loops (except those about the h_i) run in parallel on the $rc = n$ channels. If $c > 2$, the architecture has to be adapted to support reductions of integers larger than $2w + 1$ bits. The second loop performs in parallel $r = n/c$ accumulations of products. Therefore, the value of c is inversely related to the quality of the parallelism. For $c = 2$, the lost in parallelism is slight because $r = n/2$ accumulations of products are performed in parallel. The size of the accumulator of the second loop is $cw + \lceil \log_2 c \rceil$. For $c = 2$, the value of the accumulator can be easily reduced in the third loop, with a reduction similar to the one performed in the third loop of KBE algorithm.

The *cox* unit for HBE is now a small accumulator of size $t + \lceil \log_2 c \rceil$ bits, unlike in the original *cox-rower* architecture for KBE [KKSS00] where the corresponding small accumulator is of size t bits. For $c = 2$, the size of the small accumulator is $t + 1$. The output h_i of the *cox* is of 2-bit size in HBE $c = 2$. The value h_i sent to the various

rowers now activates the subtractions of 0, M, $2M$ and $3M$ in the *rowers* (line 15 of Algorithm 12).

The behavior of the HBE $c = 2$ architecture is described as follows. The precomputations are held in the memory, as in the original *cox-rower*. The $2r$ multiplications of lines 1–3 of Algorithm 12 are performed by the $2r$ *rowers* running in parallel. These multiplications are not represented in the simplified description of HBE in Figure 2.3. The *super-residues* are computed through $2r$ multiplications and additions, r operations running in parallel at a time. This step corresponds to the arrows from $x_{m_{i,1}}$ and $x_{m_{i,2}}$ (in blue) to $\widehat{x}_{M^{(i)}}$ (in blue) in the simplifed description of HBE $c = 2$ presented in Figure 2.3. The *super-residues* are broadcasted to the channels to perform lines 14 and 15 of Algorithm 12 of HBE. This broadcast is operated by the two right buses at the output of the large multiplexer. We use two buses to be able to employ one of them to broadcast w-bit results when necessary, for example the $\widehat{x}_{m_{i,j}}$ computed at line 3 of Algorithm 12. The broadcast of the *super-residues* to the various *rowers* corresponds to the arrows from the $\widehat{x}_{M^{(i)}}$ (in blue) to all the $x_{m'_{i,j}}$ in (orange) in the simplifed description of HBE $c = 2$ presented in Figure 2.3. The left bus at the output of the large multiplexer transfers the $w+1$ MSBs of the various *super-residues* to the *cox* unit. At its turn, the *cox* unit computes the values h_i and sends them to the various *rowers* to execute the subtractions at line 15 of Algorithm 12.

2.4 Application to RNS Modular Multiplications

The modular reduction (MR) is performed numerous times during an elliptic curve scalar multiplication (ECSM). The cost of Algorithm 10 about the RNS MR from [PP95], is mainly the cost of the two BEs comprised in the algorithm. Two KBEs cost $2n^2+2n$ EMMs while the RNS MR costs $2n^2 + 5n$ EMMs. The cost of the RNS MR drops to $2n^2 + 2n$ when optimizations of Gandino *et al.* [GLP+12] (combination of precomputations and reordering of internal operations) are applied. One RNS modular multiplication (MM) costs $2n^2 + 4n$ EMMs ($2n$ EMMs for the multiplications in the two RNS bases and $2n^2 + 2n$ EMMs for the RNS MR). When KBE is replaced with HBE ($c = 2$) in the RNS MM, the cost of the latter drops to $\frac{3n^2}{2} + 6n$ and $\frac{5n^2}{4} + 6n$ with the assumptions 1 EMM = 2 and 4 CMR$(2w + 1, w)$ respectively.

Figure 2.6 depicts the theoretical cost ratio HBE/KBE for one RNS MM using optimizations from [GLP+12]. The cost is presented for estimating 1 EMM to be in $\{2, 3, 4\}$

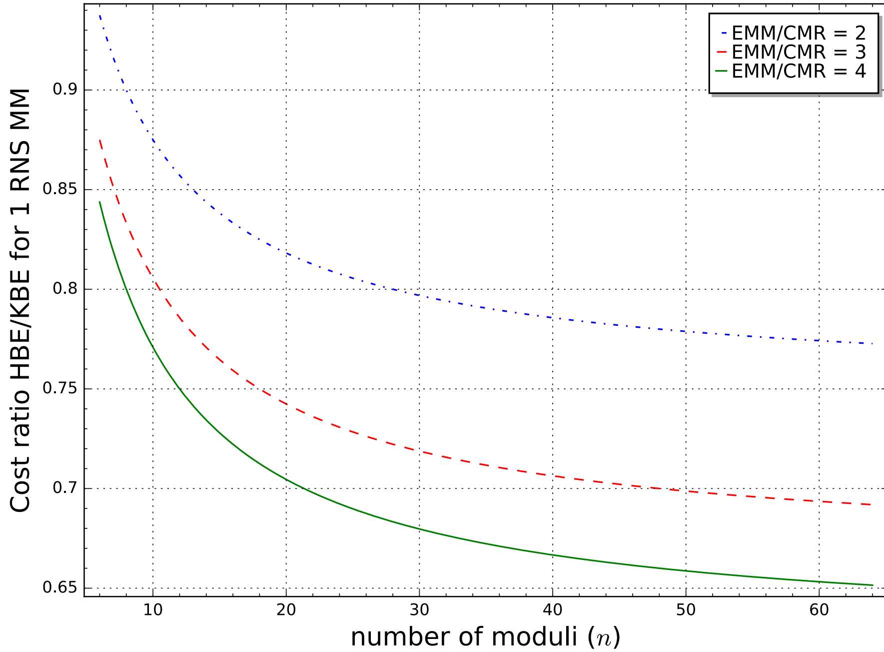


Figure 2.6 – Theoretical cost ratio HBE/KBE of one RNS MM, with optimizations from [GLP+12], for various estimations EMM/CMR$(2w + 1, w)$ and numbers n of moduli.

CMR$(2w + 1, w)$. For a typical example, if we target a finite field $\mathbb{F}_p$ of 256-bit elements for ECC and use channels of 17-bit width (to fit into the hardwired integer multipliers embedded in the Xilinx DSP slices; see Section 2.5), we need $n = 16$ moduli for each RNS base. In such an example, HBE leads to a 17–27% gain in the theoretical cost of the RNS MM compared with KBE.

The theoretical cost ratio HBE/KBE for one RNS MM using another approach for ECC [BT15, BT16] is also evaluated. The proposed idea in [BT15], and generalized in [BT16], is to use p with properties similar to pseudo-Mersenne and blend RNS with a polynomial representation of small degree d. We refer to MM algorithms in [BT15] and [BT16] as HPR $d = 2$ and HPR $d = 4$. Figures 2.7 and 2.8 depict theoretical cost ratios HBE/KBE for one RNS MM using HPR $d = 2$ and HPR $d = 4$ respectively. Typically, if $n = 16$, HBE leads to a theoretical cost gain from 8% to 17% in RNS MM with HPR $d = 2$ compared with KBE. With HPR $d = 4$ and for $n = 16$ moduli, the cost gain of HBE is up to 5%.

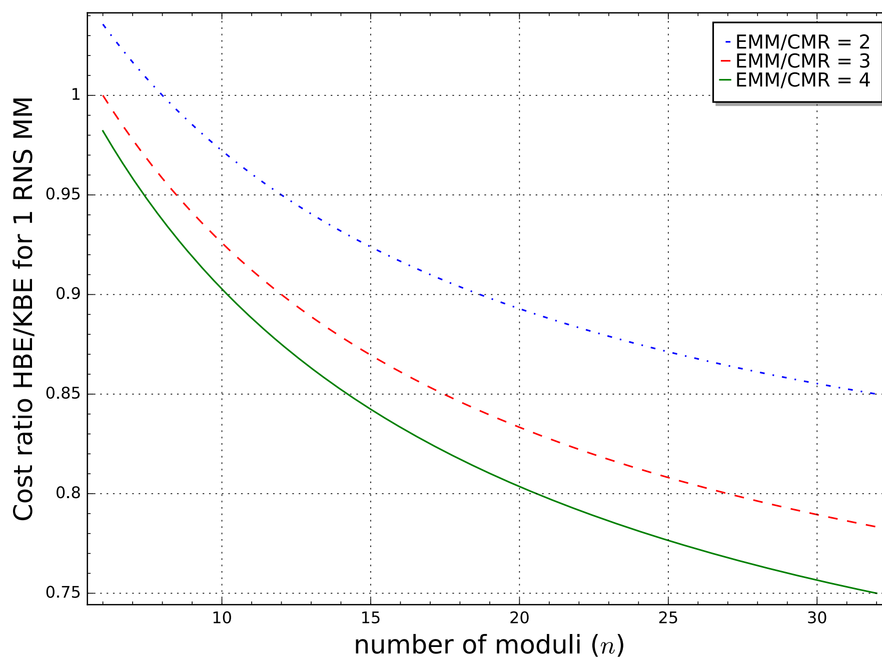


Figure 2.7 – Theoretical cost ratio HBE/KBE of one RNS MM with *HPR* $d = 2$ [BT15] for various estimations EMM/CMR$(2w + 1, w)$ and numbers n of moduli.

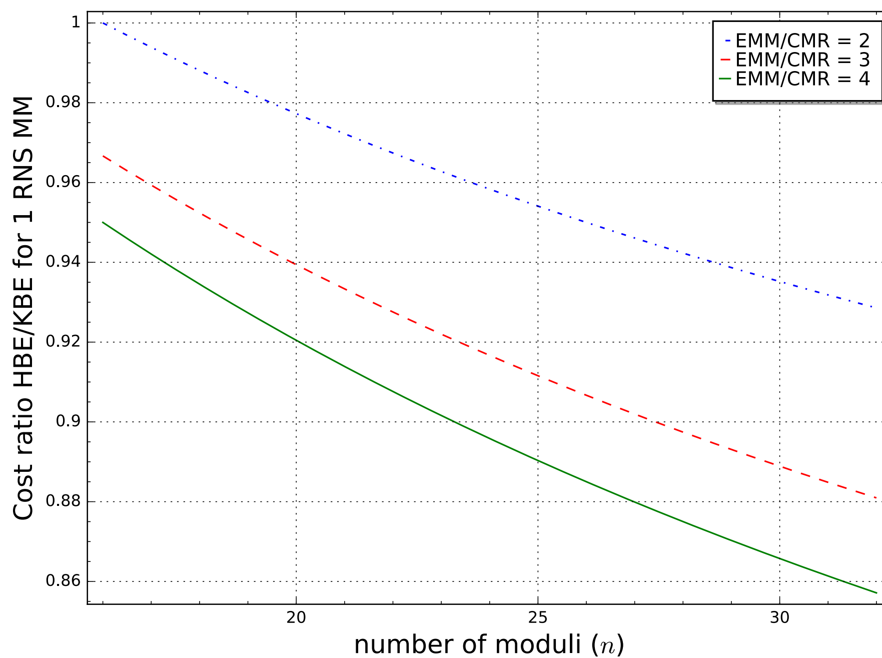


Figure 2.8 – Theoretical cost ratio HBE/KBE of one RNS MM with *HPR* $d = 4$ [BT16] for various estimations EMM/CMR$(2w + 1, w)$ and numbers n of moduli.

2.5 FPGA Implementation Results

We implemented Algorithm 12 of HBE (for $c = 2$) and Algorithm 11 of KBE [KKSS00] on a XC7Z020 FPGA from Xilinx using Vivado HLS 2019.2. The same environment and optimization effort (for example, *loop pipelining*) were used for the implementation of the two BE algorithms to fairly compare them. The implementation of the two algorithms is also motivated by the lack of results for standalone BE implementations in the literature.

We use high-level synthesis (HLS) to reduce the design time of the two BEs and explore several implementation parameters such as the number and the width of the RNS channels. However, we were unable to use HLS to allocate a memory per channel as described in Figures 2.2 and 2.5 of KBE and HBE architectures respectively. A memory containing all the precomputations and feeding all the channels has been used instead. With this method, the parallelism of the channels is limited by the number of their accesses per cycle to the memory. Nevertheless, this situation *does not negate* the fairness of our comparison since the two BE implementations are both subject to the situation. Allocating a memory per channel will improve the results of the two BE implementations. For instance, in Chapter 3 where we were able to appropriately allocate a memory per channel, the results of the BE implementations show significant improvements (see BE results for $\mathbb{F}_p$ of 256-bit elements and channel width w of 17 bits in Table 2.2, and the BE results for 16 physical channels (PCs) in Table 3.3—the FPGAs are both from the Xilinx 7-series family though being different).

The RNS bases are selected for underlying finite fields $\mathbb{F}_p$ (of ECC) of 256- and 384-bit elements. The number n of RNS channels is the smallest even integer such that $nw > \log_2 p$ (because $c = 2$ must divide n). The widths w of the RNS channels examined are 17, 20, 24 and 28 bits. The hardwired integer multipliers embedded in the DSP slices in Xilinx FPGAs support 18×18 and 18×25 bit multiplications in 2's complement. The DSP slices are fully exploited with 17-bit operands for unsigned integers; see [Xil19b, Xil18a]. In our implementations, we do not use asymmetric widths for the DSP operands (for example, 17×24 bits).

The implementation results are reported in Table 2.2. The "time" stands for the product of the period and the number of cycles to compute the BE. The time, the numbers of DSPs and slices of the two algorithms are plotted in Figure 2.9. HBE solutions are faster *and* smaller than KBE solutions in most instances. For example, for $\mathbb{F}_p$ of 256-bit elements and channels of 24-bit width, HBE is 20% faster, 19% smaller in DSPs and 6% smaller

Table 2.2 – HLS implementation results on a XC7Z020 FPGA for HBE $c = 2$ (from [DBT19]) and KBE (from [KKSS00]) algorithms for two widths of prime field elements and four RNS channel widths w.

$\mathbb{F}_p$ width (bits)	BE algo.	KBE	HBE	KBE	HBE	KBE	HBE	KBE	HBE
	w (bits)	17		20		24		28	
256	nb. slices	409	359	962	652	611	575	708	673
	nb. DSPs	51	44	45	39	52	42	77	61
	nb. BRAM	1	1	1	1	1	1	1	1
	period (ns)	9.7	9.8	9.8	9.5	9.6	8.4	9.9	9.6
	nb. cycles	98	91	88	83	89	81	77	71
	time (ns)	950.6	891.8	862.4	788.5	854.4	680.4	762.3	681.6
384	nb. slices	458	468	1270	830	1041	994	903	1062
	nb. DSPs	75	64	63	54	76	60	105	81
	nb. BRAM	1	1	1	1	1	1	1	1
	period (ns)	8.2	8.8	8.7	9.5	9.1	7.6	9.9	9.6
	nb. cycles	165	143	140	122	163	132	103	93
	time (ns)	1353.0	1258.4	1218.0	1159.0	1483.3	1003.2	1019.7	892.8

in slices than KBE. The gain in DSPs amounts to 21% for $w = 28$ bits, in which case the reduction is 11% in computation time and 5% in slices. The only case where HBE costs more than KBE for one of the resource metrics, specifically the number of slices, is for an $\mathbb{F}_p$ of 384-bit elements and channels of 28-bit width. The lost in slices is 15%. However, this lost in slices is balanced by a 23% gain in DSPs and a 12% gain in computation time, leading to an area *vs.* time trade-off again in favor of HBE.

2.6 Conclusion

In this chapter, we have presented the *hierarchical base extension* (HBE) [DBT19]. HBE operates partial computations of small CRTs on the input residues to form *super-residues* in the input RNS base, and proceed with the CRT on these *super-residues* in the output RNS base. On the design of the RNS bases, HBE does not introduce additional constraints compared with the state-of-the-art base extension (KBE) [KKSS00]. HBE shows a gain of up to 35% in the theoretical cost compared with KBE.

We have also presented the architecture proposed to support HBE $c = 2$ [DBT19] inpired by the *cox-rower* architecture for KBE [KKSS00]. The inherent parallelism of RNS is maintained with a slightly deeper pipeline at the *rower* level. FPGA implementations using HLS tools demonstrate that HBE solutions are faster (up to 20%), and in nearly

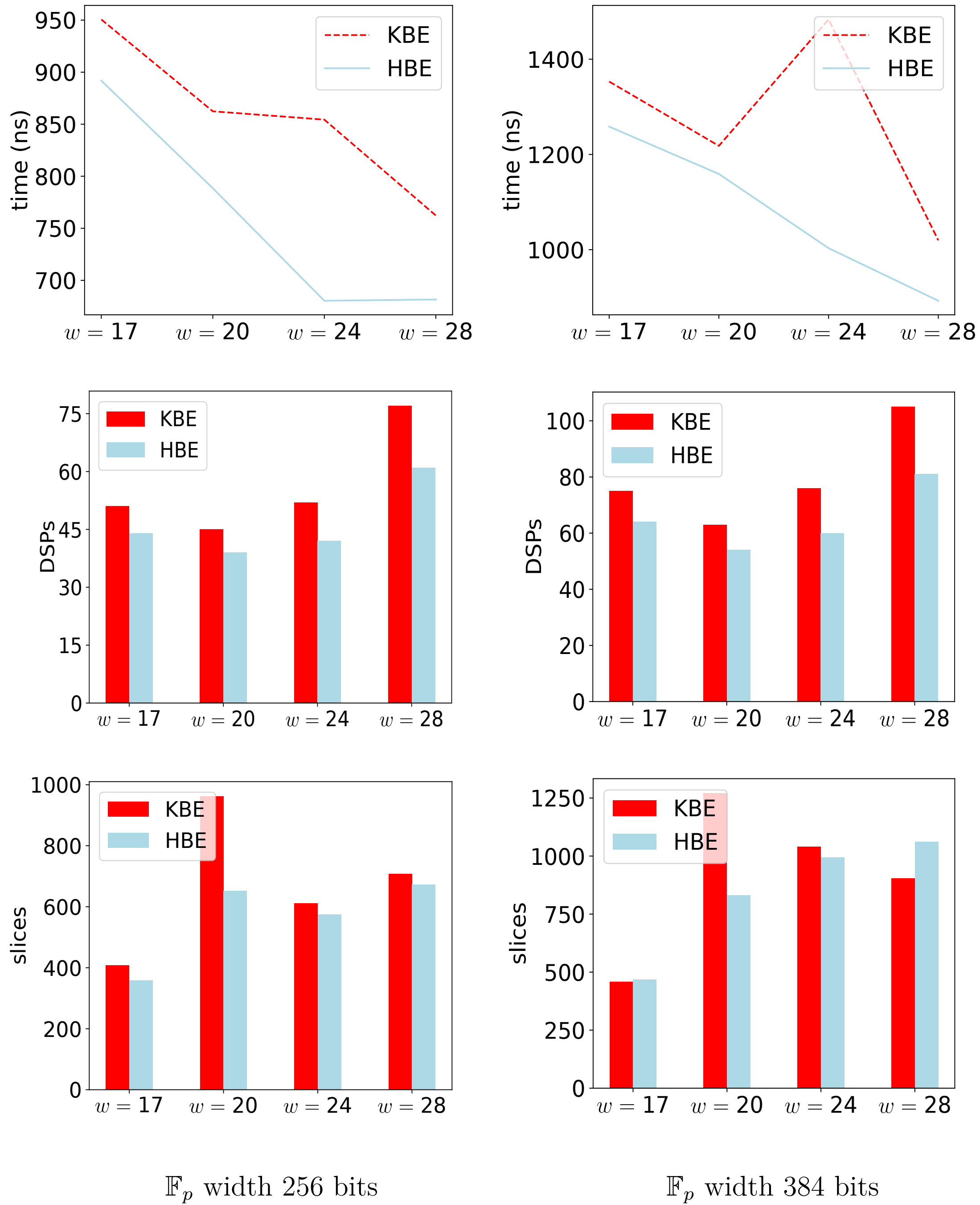


Figure 2.9 – Comparison of the time, the numbers of DSPs and slices between HBE $c = 2$ (from [DBT19]) solutions and KBE (from [KKSS00]) ones.

all cases smaller (up to 23%) than KBE. The area *vs.* time trade-off is always in favor of HBE.

In the next chapter flexible ECSMs using HBE ($c = 2$) and KBE are implemented and their results compared.

Perpective

We intend to study other types of decompositions (for example, $c = 3, 4$) and design architectures to support the inherent optimizations of these decompositions.

Chapter 3

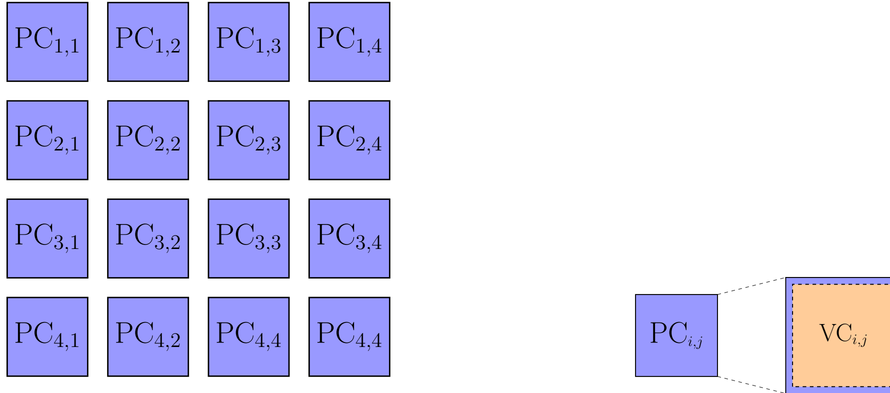
RNS-Flexible Hardware Accelerators for ECC

Elliptic curve cryptography (ECC) serves widely in digital communications where asymmetric cryptography is needed. The elliptic curve scalar multiplication (ECSM) is the core operation in protocols of ECC applications. Two bases of n moduli are needed in RNS implementations of the ECSM. For each base, the number n of moduli is chosen so that the range[1] of the base is greater than the size of the underlying finite field of the curve. A virtual channel (VC) is associated with one modulus per RNS base. Therefore, the RNS implementations of the ECSM require n VCs. Physical channels (PCs) are hardware supports of the RNS computations. In current RNS implementations of the ECSM, the used number q of PCs is equal to the number n of VCs; see, for example [NMSK01, SFM+09, Gui10, BM14]. In other words, each PC is the hardware support of RNS computations over one modulus per RNS base. RNS-flexible implementations of the ECSM are proposed in this chapter. The *flexibility* is to say that the ECSM is implemented using q PCs, q varying between all divisors of n.

In the literature of hardware implementations of ECC over prime fields, the term *flexibility* embodies two meanings that are different from the one given in this chapter. The first meaning concerns hardware implementations able of supporting two different asymmetric cryptosystems such as ECC and RSA (for example [BBMO04]) or ECC and hyperelliptic curve cryptography (HECC) (for example [BMPV06]). The second meaning concerns hardware implementations of ECC with two or more underlying prime fields; see, for example [OP01, AR14].

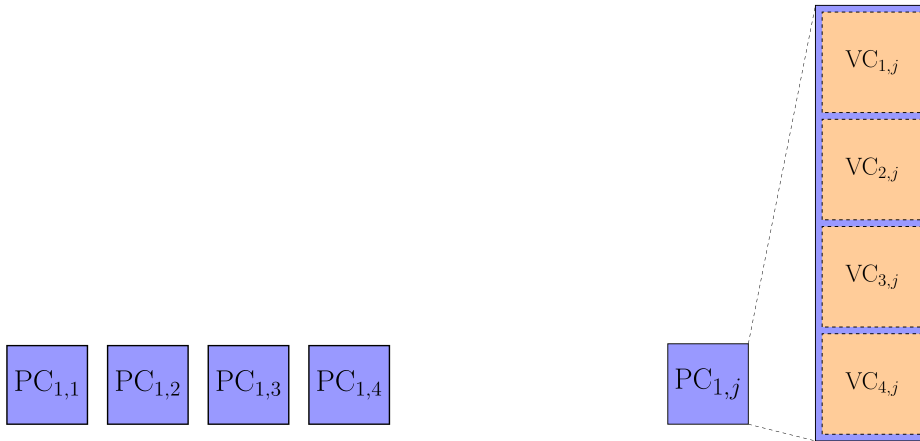
The meaning of the term *flexibility* in this chapter is different from the ones mentioned above and is put this way: For a given underlying prime field, the ECSM is implemented using an adaptable (that is, *flexible*) quantity of hardware resources. The flexibility of the hardware-resource quantity is achieved by using q PCs, where q is a divisor of n.

1. The range of an RNS base $\{m_1, \ldots, m_n\}$ is the number $M = \prod_{i=1}^{n} m_i$; see Section 1.3 about RNS.



The $q = 16$ PCs run in parallel. One VC is mapped onto each PC.

Figure 3.1 – Decomposition of the numbers n of VCs and q of PCs in usual RNS implementations of the ECSM, that is, $q = n$. In this example, $q = n = 16$. Computations related to one VC is performed on each PC.



The $q = 4$ PCs run in parallel. $n/q = 4$ VCs are mapped onto each PC.

Figure 3.2 – Example of decomposition of the numbers n of VCs and q of PCs in RNS implementations of the ECSM when $q = 4$ and $n = 16$. Computations related to $n/q = 4$ VCs are performed on each PC. At a time, each PC deals with one VC, and the 4 VCs are processed one after another.

The number n of VCs does not change when using fewer PCs. That means n/q VCs are mapped onto each PC. Let consider a typical example of a prime field of 256-bit elements and two RNS bases of 16 moduli, each of size 17 bits. 16 VCs are needed to cover the full size of the field. The ECSM can be implemented using $q \in \{1, 2, 4, 8, 16\}$ PCs. Figure 3.1 depicts the decomposition of the numbers of VCs and PCs in usual RNS

implementations, that is, when 16 PCs are used. The two-dimension notations introduced in Chapter 2 are used. The left figure shows the 16 PCs (in blue) running in parallel. The right figure depicts one VC (in orange) mapped onto its PC. In Figure 3.2, we give an example of decomposition of the numbers of VCs and PCs when 4 PCs are used for the RNS implementation. The 4 PCs (in blue) run in parallel and are shown in the left figure. The right figure shows 4 VCs (in orange) mapped onto their PC. Each PC runs the computations related to its 4 VCs, one VC at a time and one after another.

The adaptability of the hardware-resource quantity used for the flexible ECSM implementation makes the latter particularly suitable for integrated circuits with limited hardware resources. Besides, the RNS flexibility of the ECSM implementation provides circuit designers with several hardware resources *vs.* performance trade-offs to choose from.

This chapter is organized as follows. Additional notations are introduced in Section 3.1. The flexibility of the BE by Kawamura *et al.* (KBE) [KKSS00] and the hierarchical base extension (HBE) [DBT19] is shown in Sections 3.2 and 3.3 respectively. Then, the flexible ECSM is presented in Section 3.4. The chapter is concluded in Section 3.5.

3.1 Notations and Definitions

Definitions and additional notations are introduced besides the two-dimension notations presented in Chapter 2.

- The number n of VCs is equal to the number of moduli per RNS base. This number is the same independently of the performed flexible implementation. Let us recall that $n = rc$.
- The number of PCs used for a specific implementation of the BE or the ECSM is denoted q, q being a divisor of the number n of VCs (hence $q \leq n$). The number q of PCs varies with the performed flexible implementation.
- The term *flexibility* means an adaptable number q of PCs can be used to implement a BE (either KBE or HBE) or an ECSM. If n is a perfect power, that is, $n = s^a$, then our RNS implementation can be performed on $q \in \{1, s, s^2, \ldots, s^a\}$ PCs.

3.2 Flexible Kawamura Base Extension

This section presents how we implemented the flexible KBE. We first show that Algorithm 11 of KBE [KKSS00] can be adapted to run on a flexible number q of PCs. Then, the adaptation of the *cox-rower* architecture [KKSS00] to use q PCs is presented. FPGA implementations using high-level synthesis (HLS) are presented at last.

3.2.1 Algorithmical Description of the Flexible KBE

Algorithm 13 describes the mapping of the rc VCs onto the q PCs from Algorithm 11 of KBE. Triple nested loops bounded by rc/q, q/c and c at lines 1–3 and 10–12 in Algorithm 13 now replace the double nested loops bounded by r and c at lines 1–2 and 9–10 in Algorithm 11. The nested loops bounded by q/c and c at lines 2–3 and 11–12 of Algorithm 13 describe the q PCs running in parallel. The loops bounded by rc/q at lines 1 and 10 describe the mapping of rc/q VCs onto each of the q PCs, one VC at a time. The computations performed within the various loops of Algorithm 13 and Algorithm 11 are the same. Lines 5–9 of Algorithm 13 are exactly lines 4–8 of Algorithm 11 and correspond to the computations performed by the *cox* unit. These lines are not affected by the new description (q PCs are used instead of rc PCs) because the behavior of the *cox* is independent of the number of PCs.

3.2.2 Architecture of the Flexible KBE

The architecture of the flexible KBE is adapted from the *cox-rower* architecture [KKSS00]. Figure 3.3 depicts the architecture of the flexible KBE for $q = 4$ PCs. Each $memory_{i,j}$ ($i = 1$, 2 and $j = 1$, 2 in our example) stores the precomputations corresponding to rc/q VCs. An additional signal (not represented in Figure 3.3) indicates the running VC on the PC at a time. There is one *rower* per PC, dedicated to computations over rc/q moduli per RNS base, one at a time. The computations performed by the *rowers* are of the form $g \leftarrow (g + a \times b + f) \bmod m_{i,j}$ or mod $m'_{i,j}$ as in the original *cox-rower*, with the operands a, b, f and g being of size the width w of the PCs. The arithmetic processing performed by the *rower* is depicted in Figure 3.4. The first multiplier and adder allow to perform $g+a\times b+f$, the value of g being added only when an accumulation is being performed (for example line 13 of Algorithm 13). The remaining multipliers and adders allow to reduce mod $m_{i,j}$ or mod $m'_{i,j}$ the result of $g + a \times b + f$ using the reduction method specific to

Algorithm 13: Flexible KBE $BE_{\mathcal{M}\to\mathcal{M}'}(x)$ adapted from [KKSS00]. The function $f(v,i)$ is given by $f(v,i) = \frac{q}{c}(v-1)+i$.

Input: $x_{m_{i,j}}$ for all i, $1 \le i \le r$ and j, $1 \le j \le c$; $\sigma = 0$ or 0.5
Precomp.: $|M_{i,j}^{-1}|_{m_{i,j}}$, $|M_{i,j}|_{m'_{k,l}}$, $|-M|_{m'_{k,l}}$, for all i,k, $1 \le i,k \le r$ and j,l, $1 \le j,l \le c$
Output: $x_{m'_{k,l}}$ for all k, $1 \le k \le r$ and l, $1 \le l \le c$

1 **for** $v \leftarrow 1$ **to** rc/q **do**
2   **for** $i \leftarrow 1$ **to** q/c **parallel do**
3   **for** $j \leftarrow 1$ **to** c **parallel do**
4       $\widehat{x}_{m_{f(v,i),j}} \leftarrow \left| x_{m_{f(v,i),j}} \times \left| M_{f(v,i),j}^{-1} \right|_{m_{f(v,i),j}} \right|_{m_{f(v,i),j}}$
5 **for** $i \leftarrow 1$ **to** r **do**
6   **for** $j \leftarrow 1$ **to** c **do**
7     $\sigma \leftarrow \sigma + \frac{\mathrm{trunc}(\widehat{x}_{m_{i,j}})}{2^w}$
8     $h_{i,j} \leftarrow \lfloor \sigma \rfloor$
9     $\sigma \leftarrow \sigma - h_{i,j}$
10   **for** $v \leftarrow 1$ **to** rc/q **do**
11   **for** $k \leftarrow 1$ **to** q/c **parallel do**
12   **for** $l \leftarrow 1$ **to** c **parallel do**
13           $x_{m'_{f(v,k),l}} \leftarrow \left| x_{m'_{f(v,k),l}} + \widehat{x}_{m_{i,j}} \times |M_{i,j}|_{m'_{f(v,k),l}} + h_{i,j} \times |-M|_{m'_{f(v,k),l}} \right|_{m'_{f(v,k),l}}$

pseudo-Mersenne numbers described in Subsection 1.2.3.

The execution of KBE using the flexible architecture is described as follows. The rc multiplications in base $\mathcal{M}$ at line 4 of Algorithm 13 are performed by the q *rowers* running in parallel. Each *rower* is in charge of rc/q multiplications, one at a time. Similarly to the original *cox-rower*, the large multiplexer receives the results of the multiplications from the *rowers* and outputs two buses (left and right). The right bus broadcasts the results inputted to the large multiplexer to the q rowers in order to proceed with the computation of the CRT in base $\mathcal{M}'$, with q executions of line 13 (Algorithm 13) at a time. The left bus transfers the results inputted to the large multiplexer to the *cox* unit which computes

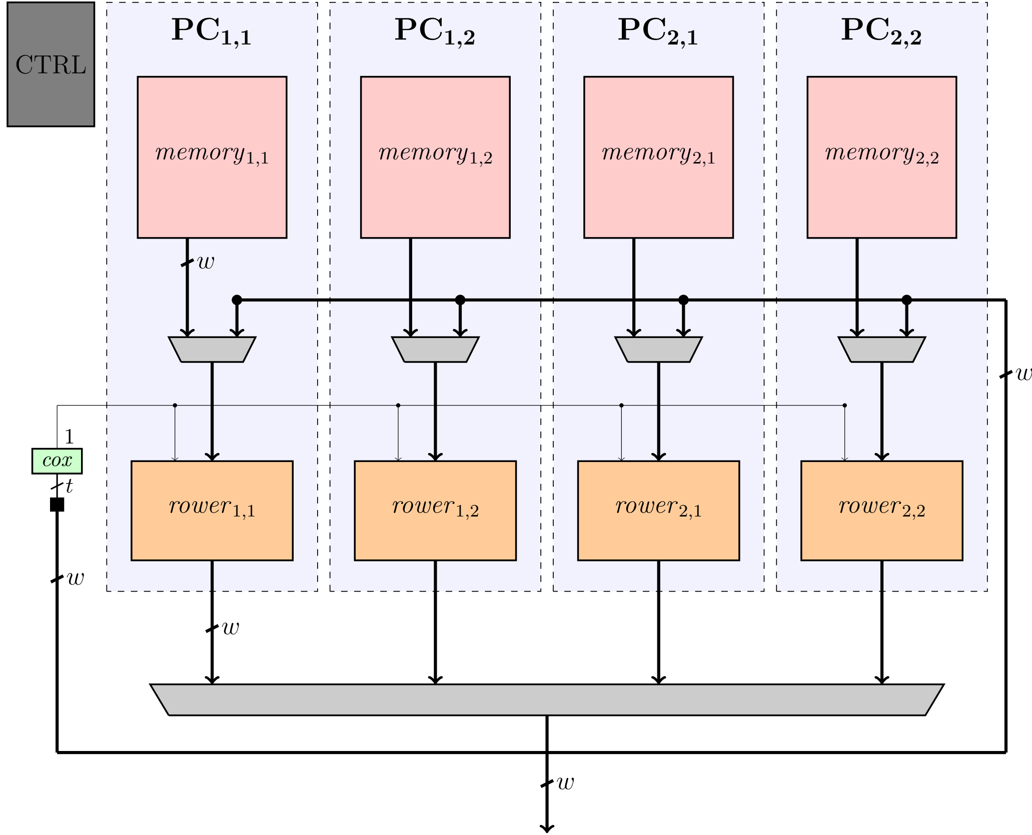


Figure 3.3 – The architecture for the flexible KBE (adapted from the *cox-rower* architecture [KKSS00]): example with $q = 4$ PCs and $c = 2$.

the rc bits $h_{i,j}$ and sends them to the q *rowers*, q bits $h_{i,j}$ at a time, for subtraction of 0 or M.

3.2.3 FPGA Implementation Results

Implementation Environment

We implemented Algorithm 13 on a XCZU7EV-FFVC1156 FPGA from Xilinx using Vivado HLS 2019.2. The finite field $\mathbb{F}_p$ is of 256-bit elements. The PCs are chosen to be 17-bit wide in order to fully exploit the DSP slices in Xilinx FPGAs. Let us recall that the hardwired integer multipliers embedded in the DSP slices in Xilinx FPGAs support 18×18 and 18×25 bit multiplications in 2's complement. Asymmetric widths for the DSP operands are not supported in implementations leading to results presented in this

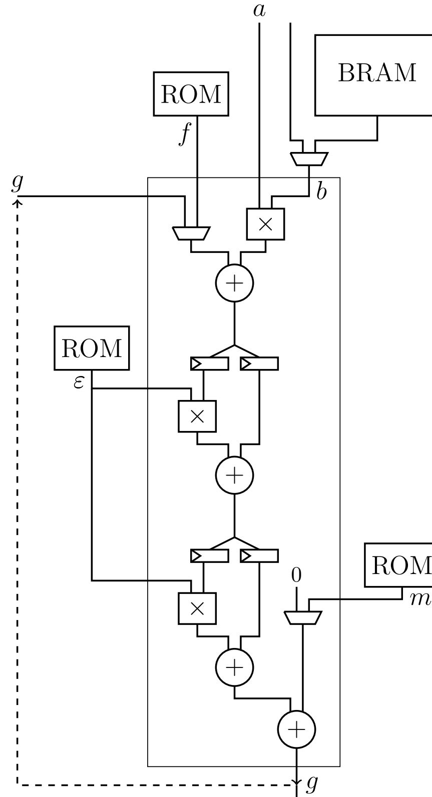


Figure 3.4 – Structure of the arithmetic processing performed by a *rower*. The *rower* performs the operation $g \leftarrow (g + a \times b + f) \bmod m$, where $m = 2^w - \varepsilon$ (with $\varepsilon < 2^{w/2}$), and a, b, f and g are of size w bits.

chapter, similarly to the ones in Chapter 2. Each RNS base needs 16 moduli to represent the elements of $\mathbb{F}_p$ since the size of the moduli is the width of the PCs (17 bits). Algorithm 13 was implemented for all $q \in \{1, 2, 4, 8, 16\}$ PCs using the same environment and optimization effort.

Utilization of Hardware Resources

The FPGA implementation results are reported in Table 3.1. The hardware resources and the performance vary depending on the number of PCs used for the flexible KBE implementation. The variations of hardware resources and performance in relation to the increase of the number of PCs are presented in Figure 3.5. The base of comparison taken in this figure is the implementation results for one PC. The numbers of DSPs and BRAMs increase linearly with the number of PCs. In other words, an increase of the number of

Table 3.1 – HLS implementation results on a XCZU7EV-FFVC1156 FPGA for the flexible KBE (adapted from [KKSS00]) using $q \in \{1, 2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the width of the PCs is 17 bits.

nb. PCs	1	2	4	8	16
nb. slices	107	162	223	341	623
nb. DSPs	3	6	12	24	48
nb. BRAMs	1	2	4	8	16
period (ns)	3.9	3.9	4.1	4.6	4.6
nb. cycles	280	153	88	56	37
time (ns)	1092	596.7	360.8	257.6	170.2

PCs from q to kq induces an increase with the same factor k in the numbers of DSPs and BRAMs. For example, the KBE implementation with 4 PCs is performed using 12 DSPs, which is 4×3 DSPs, the implementation with one PC using 3 DSPs. On the other hand, the number of slices increases less than linearly with the number of PCs, that is, the increase factor is less than k when the number of PCs is increased from q to kq. For example, an increase of the number of PCs from $q = 1$ to $q = 4$ results in an increase factor of only 2.1 in the number of slices (107 and 223 slices for $q = 1$ and 4 PCs respectively).

Number of Cycles and Time Analyses

Let consider the example of the first (group of) loop bounded by 16 (16 VCs corresponding to the 16 moduli per base) in Algorithm 13. Each PC handles $16/q$ VCs (q PCs are used). Figure 3.6 shows the number of cycles needed by a PC to perform all computations related to the $16/q$ VCs it handles. For $q = 2$ and 4, each PC handles 8 and 4 VCs respectively, and runs them one after another. These number of VCs are represented by the 8 columns at the left and 4 columns at the right in Figure 3.6. The 6 boxes of each column symbolize the number of cycles needed by a PC to perform all computations related to one VC. These 6 cycles are typical value of the pipeline depth for our implementation of the arithmetic processing (performed by the *rower*) presented in Figure 3.4. The loop is pipelined. Hence, at each cycle, the operations related to a new VC starts on the PC; this is symbolized by the one-box height gap between the columns in each of the left and right figures (in Figure 3.6).

For the left figure in Figure 3.6, after 8 cycles (from top to bottom), the computations related to the 8 VCs on one PC have all started. We name these cycles operation cycles

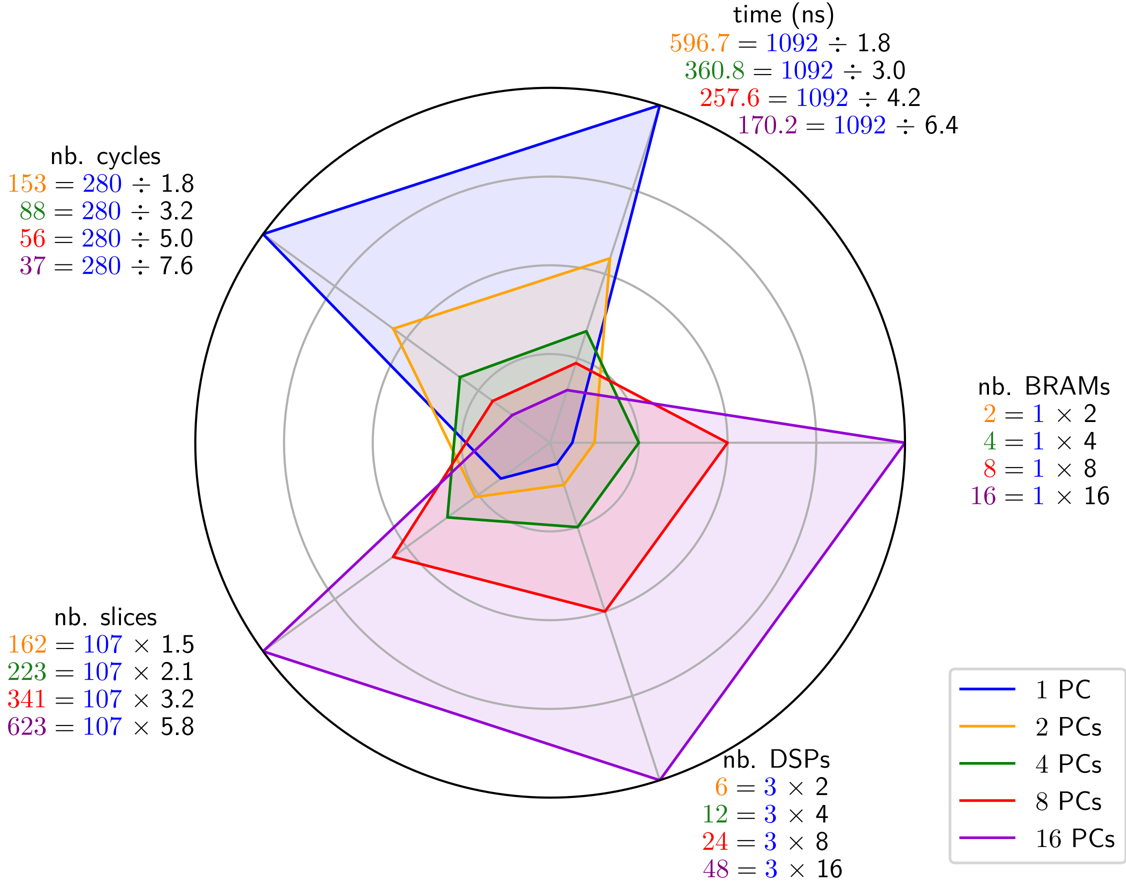
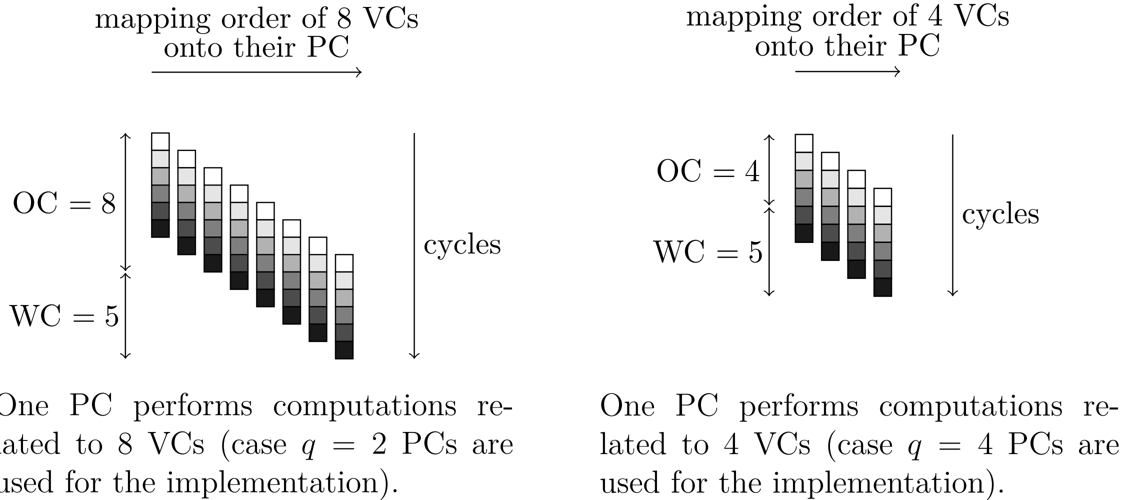


Figure 3.5 – Comparison of FPGA implementation results of the flexible KBE (adapted from [KKSS00]) for $q \in \{1, 2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the width of the PCs is 17 bits.

and their number OC. Therefore, OC = 8. The reader can see that at the right figure in Figure 3.6, OC = 4. Additional cycles are needed by the PC to terminate all computations on its ongoing VCs. These cycles are named wait cycles and their number is denoted WC. In Figure 3.6, WC = 5 in the left and the right figures. This case is the worst scenario, that is, the case where the next loop cannot start until all computations (related to all the VCs) of the current loop are teminated. Ideally, we want WC = 0 because it leads to the minimal value of the total number of cycles for the implementation. However, in our HLS implementations, the achieved WC for the different loops are always > 0 though usually better than the worst case.

When the number of PCs increased from q to kq, the OC decreases with a factor k, but the WC remains the same regardless of the number of PCs used. This constancy of



One PC performs computations related to 8 VCs (case $q = 2$ PCs are used for the implementation).

One PC performs computations related to 4 VCs (case $q = 4$ PCs are used for the implementation).

Figure 3.6 – Example of numbers cycles used by one PC to perform all computations related to the $16/q$ VCs it handles. The PC sequentially 'uploads' the $16/q$ VCs. Computations on the same PC are pipelined. In this typical example of loops at lines 1–4 of Algorithm 13, the computations related to the $rc = 16$ VCs are equally shared by the q PCs in use. At the left figure, $q = 2$ PCs are used and each PC handles 8 VCs. At the first cycle (top to bottom), computations related to the first VC start. At the next cycle, computations related to the next VC start, and so on until computations related to all 8 VCs (mapped onto the PC) start. Meanwhile, it is possible that the PC has terminated the computations related to the VCs that started earlier. Still, a few cycles are needed by the PC to terminate the remaining computations related to its ongoing VCs, that is, to empty out the pipeline. At the right figure, $q = 4$ PCs are used and each PC handles 4 VCs. The filling and emptying of the pipeline are similar to the ones at the left figure.

the WC prevents the implementations from achieving a reduction with a factor k in the total number of cycles. Consequently, this number of cycles decreases less than linearly, that is, the decrease factor is less than k when the number of PCs increases from q to kq. For example, using $q = 4$ PCs instead of one PC decreases the number of cycles by a factor of 3.2 (88 cycles for four PCs and 280 cycles for one PC).

The "time" is the product of the period and the number of cycles (to compute the BE), similarly to Section 2.5. The periods achieved by the implementations on the various numbers q of PCs are relatively close. Therefore, the less-than-linear decrease in the total number of cycles (when the number of PCs increases linearly) results in a less-than-linear decrease in the time.

3.3 Flexible Hierarchical Base Extension

A description of how we implemented the flexible HBE is presented in this section, similarly to Section 3.2 about the flexible KBE. In addition, the flexible HBE solutions are compared with the flexible KBE ones.

3.3.1 Algorithmical Description of the Flexible HBE

Algorithm 14 presents the mapping of the rc VCs onto the q PCs from Algorithm 12 of HBE. Triple nested loops bounded by rc/q, q/c and c at lines 1–3 and 14–16 in Algorithm 14 replace double nested loops bounded by r and c at lines 1–2 and 12–13 in Algorithm 12. Within these triple loops, the inner loops bounded by q/c and c at lines 2–3 and 15–16 describe the q PCs running in parallel. The outer loops bounded by rc/q at lines 1 and 14 describe the mapping of rc/q VCs on each PC, one VC at a time. For the computation of the *super-residues*, double nested loops bounded by rc/q and q/c at lines 5 and 6 of Algorithm 14 replace the single loop bounded by r at line 4 of Algorithm 12. The *super-residues* are now computed by q/c parallel executions of lines 7–9 (of Algorithm 14) at a time, rc/q times. Lines 10–13 of Algorithm 14 remain unchanged regarding lines 8–11 of Algorithm 12 because computations at these lines are not affected by the change of the number of PCs from rc to q. Indeed, these computations are performed by the *cox* unit which is independent of the number of used PCs, as in Algorithm 13 of the flexible KBE.

3.3.2 Architecture of the Flexible HBE

The architecture of the flexible HBE is adapted from that of HBE ($c = 2$) (presented in Subsection 2.3.3) to use q PCs, q being a divisor of rc. Each PC supports rc/q VCs. Figure 3.7 depicts the architecture of the HBE for $q = 4$ PCs. The number of *rowers* is reduced to the desired number q of PCs, one *rower* per PC. The computation of the *super-residues* involves an accumulation without reduction; see lines 5–9 of Algorithm 14. To be able to perform these accumulations, we split the arithmetic processing performed by the *rower* of KBE (in Figure 3.4) into two parts: an upper and a lower part. The upper part, depicted in Figure 3.8, performs the operation $G \leftarrow G + a \times b + f$, that is, without reduction. G is of size at most $2w + 1$ bits; a, b and f are of size the width w of the PCs. In addition to operations at lines 1–4, 10–16 and 18 in Algorithm 14 which are existent in

Algorithm 14: Flexible HBE $BE_{\mathcal{M}\rightarrow\mathcal{M}'}(x)$ adapted from [DBT19]. The function $f(v,i)$ is given by $f(v,i) = \frac{q}{c}(v-1)+i$.

Input: $x_{m_{i,j}}$ for all i, $1 \le i \le r$ and j, $1 \le j \le c$; $\sigma = 0$ or 0.5
Precomp.: $|M_{i,j}^{-1}|_{m_{i,j}}$, $|M_{i,j}|_{m'_{k,l}}$, $|-M|_{m'_{k,l}}$, for all i,k, $1 \le i,k \le r$ and j,l, $1 \le j,l \le c$
Output: $x_{m'_{k,l}}$ for all k, $1 \le k \le r$ and l, $1 \le l \le c$

1 **for** $v \leftarrow 1$ **to** rc/q **do**
2 **for** $i \leftarrow 1$ **to** q/c **parallel do**
3   **for** $j \leftarrow 1$ **to** c **parallel do**
4    $\widehat{x}_{m_{f(v,i),j}} \leftarrow \left| x_{m_{f(v,i),j}} \times \left| M_{f(v,i),j}^{-1} \right|_{m_{f(v,i),j}} \right|_{m_{f(v,i),j}}$

5 **for** $v \leftarrow 1$ **to** rc/q **do**
6 **for** $i \leftarrow 1$ **to** q/c **parallel do**
7   $\widehat{x}_{M^{(f(v,i))}} \leftarrow 0$
8   **for** $j \leftarrow 1$ **to** c **do**
9    $\widehat{x}_{M^{(f(v,i))}} \leftarrow \widehat{x}_{M^{(f(v,i))}} + \widehat{x}_{m_{f(v,i),j}} \times \overline{m_{f(v,i),j}^{(f(v,i))}}$ (without modular reduction)

10 **for** $i \leftarrow 1$ **to** r **do**
11 $\sigma \leftarrow \sigma + \frac{\text{trunc}(\widehat{x}_{M^{(i)}})}{2^{w\times c}}$
12 $h_i \leftarrow \lfloor \sigma \rfloor$
13 $\sigma \leftarrow \sigma - h_i$
14 **for** $v \leftarrow 1$ **to** rc/q **do**
15   **for** $k \leftarrow 1$ **to** q/c **parallel do**
16   **for** $l \leftarrow 1$ **to** c **parallel do**
17     $\widehat{x}_{m'_{f(v,k),l,i}} \leftarrow |\widehat{x}_{M^{(i)}}|_{m'_{f(v,k),l}}$
18     $x_{m'_{f(v,k),l}} \leftarrow \left| x_{m'_{f(v,k),l}} + \widehat{x}_{m'_{f(v,k),l,i}} \times \left| \overline{M^{(i)}} \right|_{m'_{f(v,k),l}} + h_i \times |-M|_{m'_{f(v,k),l}} \right|_{m'_{f(v,k),l}}$

Algorithm 13 of the flexible KBE, the upper part is able to perform the computation of the *super-residus*. In our HLS implementations, the upper part is carried out by one DSP. The lower part, depicted in Figure 3.9, performs the operation $g \leftarrow G \bmod m$, where

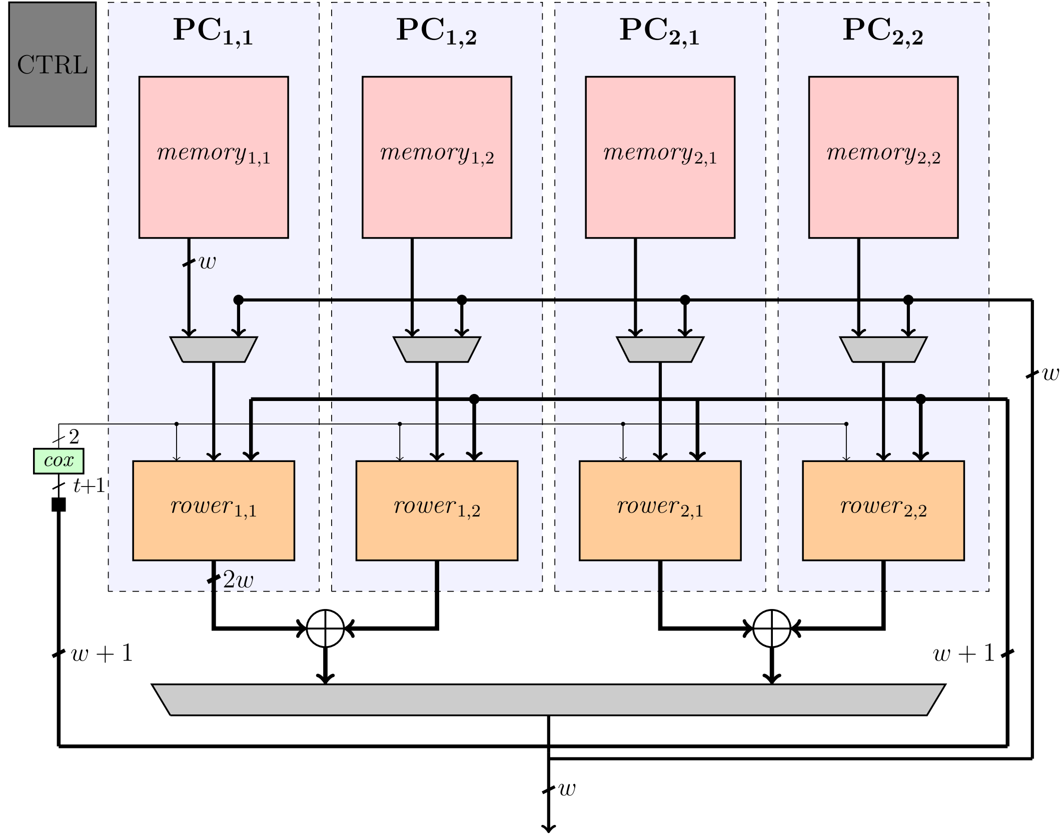


Figure 3.7 – The flexible architecture for HBE (adapted from the HBE architecture in [DBT19], itself inspired by the *cox-rower* architecture [KKSS00]): example with $q = 4$ PCs and $c = 2$.

$m = 2^w - \varepsilon$ and G is of size at most $2w + 1$ bits. At lines 4 and 18 of Algorithm 14, the lower part is used after the upper part in order to perform the needed reduction. At line 17, the lower part is used alone (without the upper part) since only the modular reduction is needed at this line. At line 9 of Algorithm 14 where the *super-residues* are computed, the lower part is not used to avoid performing a modular reduction. In our HLS implementations, the lower part is carried out by one DSP at the expense of some extra slices.

The behavior of the architecture of the flexible HBE is similar to that of HBE and is described as follows. Line 4 of Algorithm 14 is performed on the q PCs, q parallel computations at a time, rc/q times. Then, the *super-residues* are computed through the rc multiplications and additions, q/c operations running in parallel at a time. The *super-residues* are broadcasted to the *rowers* by the two right buses at the output of the large

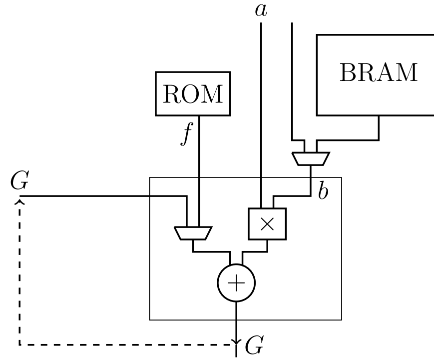


Figure 3.8 – Structure of the upper part of the arithmetic processing performed by a *rower* in the flexible HBE ($c = 2$). The upper part of the *rower* performs the operation $G \leftarrow G + a \times b + f$, where a, b and f are of size w bits, and G of size at most $2w+1$ bits.

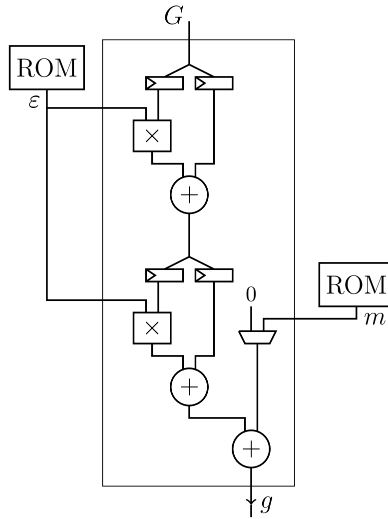


Figure 3.9 – Structure of the lower part of the arithmetic processing performed by a *rower* in the flexible HBE ($c = 2$). The lower part of the *rower* performs the operation $g \leftarrow G \bmod m$, where $m = 2^w - \varepsilon$ (with $\varepsilon < 2^{w/2}$), g is of size w bits, and G of size at most $2w + 1$ bits.

multiplexer, similarly to the broadcast in HBE architecture (see Subsection 2.3.3). The left bus sends the *super-residues* values to the *cox* unit which computes the values h_i and send them to the *rowers* for the subtractions at line 18 (0, M, $2M$ and $3M$ when $c = 2$). The rc computations of lines 17 and 18 of Algorithm 14 are performed on the q PCs, q parallel computations at a time, rc/q times.

3.3.3 FPGA Implementation Results

Implementation Environment

The flexible HBE ($c = 2$) was implemented with the same environment and optimization effort as the flexible KBE was (Subsection 3.2.3). The considered finite field $\mathbb{F}_p$ is also of 256-bit elements and the PCs 17-bit wide. Algorithm 14 of the flexible HBE was implemented using $q \in \{2, 4, 8, 16\}$ PCs. Owing to the architecture of HBE, the number of PCs needs to be greater or equal to c (the *super-residues* are formed by combining c residues). Therefore, using one PC for the implementation of the flexible HBE ($c = 2$) was not possible.

Utilization of Hardware Resources and Time

The FPGA implementation results are reported in Table 3.2. Figure 3.10 depicts the evolution of the numbers of DSPs, BRAMs, slices and cycles, and the time. The base of comparison is the implementation results obtained for $q = 2$ PCs (the flexible HBE cannot be implemented for $q = 1$ PC). An increase in the number of PCs from q to kq results in an increase of factor k in the numbers of DSPs and BRAMs. For example, HBE solution with four PCs uses 8 DSPs and 4 BRAMs, which is twice the number of DSPs and BRAMs used by HBE solution with two PCs (4 DSPs and 2 BRAMs). The number of slices increases less than linearly in relation to the increase of the number of PCs. Similarly to the flexible KBE and for the same reason (WC > 0; see Subsection 3.2.3), the number of cycles of the flexible HBE solutions decreases less than linearly in relation to the increase of the number of PCs. This less-than-linear decrease in the number of

Table 3.2 – HLS implementation results on a XCZU7EV-FFVC1156 FPGA for the flexible HBE (adapted from [DBT19]) using $q \in \{2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the width of the PCs is 17 bits.

nb. PCs	2	4	8	16
nb. slices	179	289	526	609
nb. DSPs	4	8	16	32
nb. BRAMs	2	4	8	16
period (ns)	4.2	4.2	4.4	4.3
nb. cycles	103	67	49	36
time (ns)	432.6	281.4	215.6	154.8

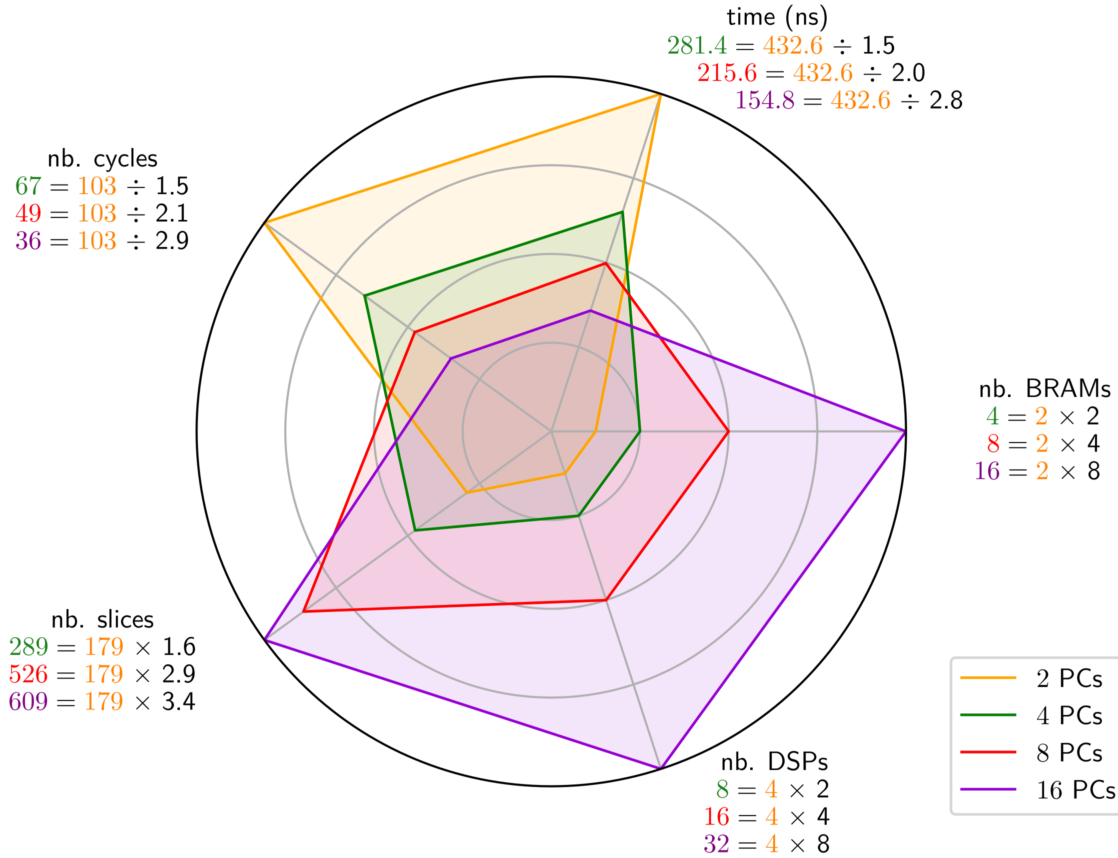


Figure 3.10 – Comparison of FPGA implementation results of the flexible HBE (adapted from [DBT19]) for $q \in \{2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the width of the PCs is 17 bits.

cycles is translated into a less-than-linear decrease in the time. For example, the HBE solution with four PCs is only 1.5 faster than the one with two PCs.

Comparison Flexible KBE *vs.* Flexible HBE

The comparison of the flexible HBE and KBE is reported in Table 3.3. The comparison is performed for $q \in \{2, 4, 8, 16\}$ PCs. No comparison is possible for $q = 1$ PC since no implementation result is available for HBE ($c = 2$) on one PC. In Figure 3.11 are plotted the time, the numbers of cycles, DSPs and slices.

The HBE solution is faster than the KBE one for all $q \in \{2, 4, 8, 16\}$ PCs. For example, for $q = 4$ PCs, the HBE solution is 22% faster than the KBE one. The gain in speed reaches 28% for $q = 2$ PCs. For a given number q of PCs, the number of BRAMs is the

Table 3.3 – Comparison of FPGA implementation results of the flexible HBE (adapted from [DBT19]) with those of the flexible KBE (adapted from [KKSS00]) for various number of PCs. The flexible HBE is not implemented for one PC. Finite fields are of 256-bit elements and the width of the PCs is 17 bits.

BE algos.	KBE	HBE	KBE	HBE	KBE	HBE	KBE	HBE	KBE	HBE
nb. PCs	1		2		4		8		16	
nb. slices	107	-	162	179	223	289	341	526	623	609
nb. DSPs	3	-	6	4	12	8	24	16	48	32
nb. BRAMs	1	-	2		4		8		16	
period (ns)	3.9	-	3.9	4.2	4.1	4.2	4.6	4.4	4.6	4.3
nb. cycles	280	-	153	103	88	67	56	49	37	36
time (ns)	1092	-	596.7	432.6	360.8	281.4	257.6	215.6	170.2	154.8

same for the two BEs. For all $q \in \{2, 4, 8, 16\}$, HBE is 33% smaller than KBE in DSPs. However, in most cases, HBE solutions require more slices than KBE ones; this is due to the fact that only one DSP is used in our HLS implementations of the lower part of the arithmetic processing performed by the *rower* at Figure 3.9 (see Subsection 3.3.2). For example, for $q = 4$ PCs, the HBE solution costs 23% more slices than the KBE one. This extra cost in slices amounts to 35% for $q = 8$ PCs. Note also that for $q = 16$ PCs, the HBE solution is 2% smaller in slices besides being 33% smaller in DSPs and 9% faster than the KBE one. Overall, the extra costs in slices of HBE solutions are well compensated by the gains in DSPs. Besides, HBE solutions are faster than KBE ones for all $q \in \{2, 4, 8, 16\}$ PCs.

When increasing the number of PCs from q to kq, the decrease factor in the number of cycles of HBE is smaller than that of KBE. For example, by using four PCs instead of two (that is, double the number of PCs), the number of cycles of the HBE solution is reduced by 35% while that of the KBE solution is reduced by 42%. This result is to be expected since in Algorithm 14 of HBE an additional (group of) loop is introduced (compared with KBE)—the loop for the computation of the *super-residues*—, resulting in an augmentation of the total WC. The consequence is that the gain in time of HBE compared with KBE decreases as the number of PCs increases. In other words, the fewer PCs are used, the greater is the time gain of HBE solutions compared with KBE ones. Nevertheless, HBE solutions remain faster than KBE ones for all considered numbers of PCs.

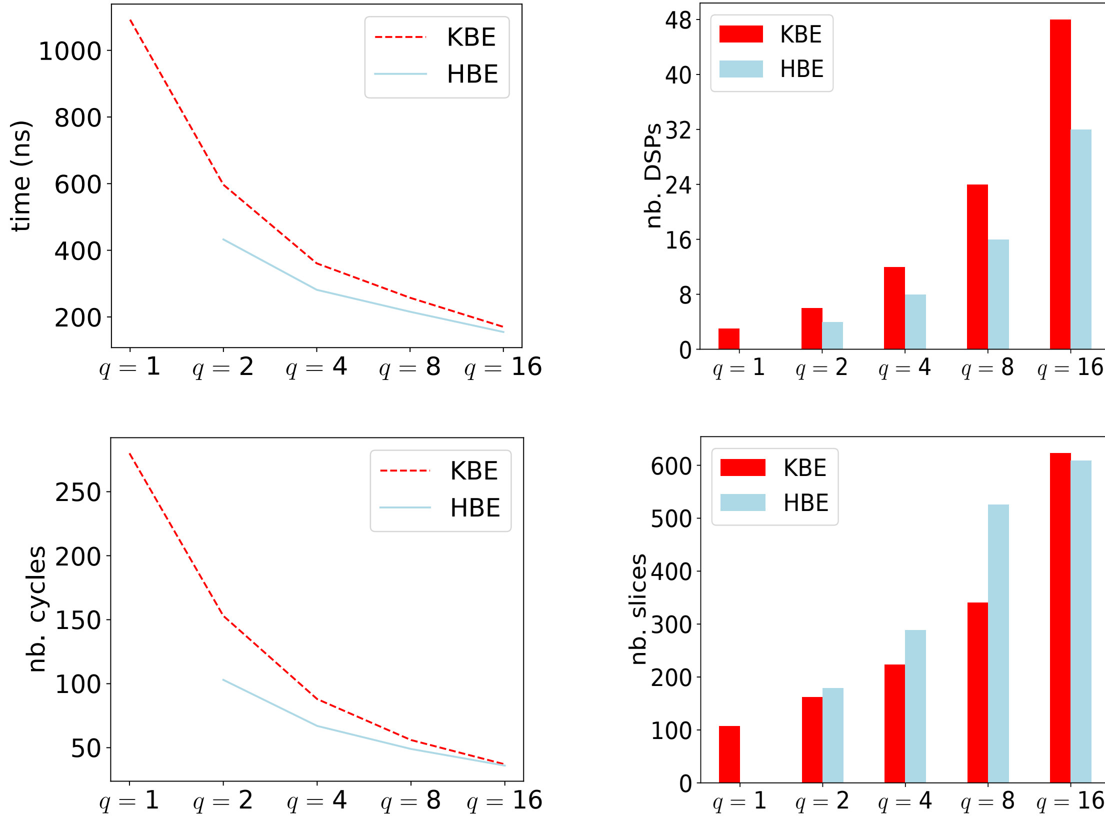


Figure 3.11 – Comparison of the time, the numbers of cycles, DSPs and slices between the flexible HBE solutions and the flexible KBE ones for various numbers of PCs. HBE is not implemented for one PC.

3.4 Flexible Elliptic Curve Scalar Multiplication

This section presents how we implemented the flexible ECSMs from the flexible BEs presented in the two previous sections. First, the main steps between the ECSM and the BE are described. Then, FPGA implementation results of the flexible ECSMs using KBE and HBE are presented. The results of the two flexible ECSMs (with KBE and HBE) are compared with each other, and with FPGA implementation results of the ECSM from the literature.

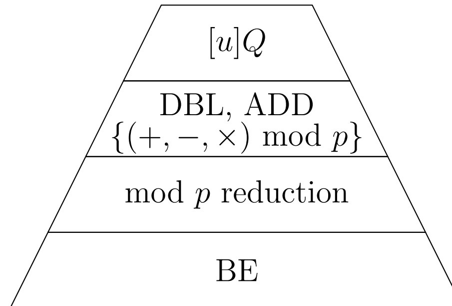


Figure 3.12 – Hierarchical description of RNS computations of the ECSM. The ECSM is computed from a few hundreds point doublings and additions. If Algorithm 15 of the Mongomery ladder [Mon87] is used, each iteration of its loop requires two dozens basic operations and a dozen RNS mod p reductions. Each mod p reduction is made of two BEs and a few basic operations. The cost of the ECSM is mainly driven by that of the mod p reduction, which is itself driven by that of the BEs.

3.4.1 Overview of the Flexible ECSM

Hierarchy of the ECSM

The RNS computation of the ECSM is performed through operations that are presented in an hierarchical way in Figure 3.12. From top to bottom, the RNS computation of the ECSM defined over $\mathbb{F}_p$ requires an ECSM algorithm, point addition and doubling formulas, an efficient RNS MR algorithm (for the mod p reduction), and a BE algorithm.

The chosen algorithm for computing the ECSM in this thesis is the Montgomery ladder [Mon87], recalled in Algorithm 15. At each bit of the scalar, a point doubling and addition are performed. In other words, the arithmetic operations at each iteration are the same regardless of the value of the scalar bit. The constancy of the number of operations is an advantage in terms of computation protection [2]. There is an extra cost attached to Montgomery ladder compared with other ECSM algorithms such as the *double-and-add* algorithm. This extra cost is due to the fact that a point doubling and addition are performed at each iteration. The number of point doublings and additions needed to perform an ECSM depends on the bit size of the underlying finite field of the curve. In the current standards (see, for example [ndlsdsd11, oST13]), a few hundreds point doublings

2. The computation protection is not sufficient for a complete secure implementation. Indeed, other parameters of hardware implementations such as the number of control signals or the storage addresses of iteration results have to be considered. For example, if the storage addresses of the results of an iteration depend on the bit value of the scalar and no further step is taken, then the information leak is likely to differ depending on the bit value of the scalar.

Algorithm 15: Montgomery ladder algorithm [Mon87]

Input: Q a point of the elliptic curve $E_{\mathbb{F}_p}$
u an integer written in binary representation as $u_{s-1}u_{s-2}...u_0$
Output: $Q_1 = [u]Q$

```
1  Q1 ← O; Q2 ← Q
2  for i ← s − 1 to 0 do
3      if u_i = 0 then
4          Q1 ← [2]Q1; Q2 ← Q1 + Q2
5      else
6          Q2 ← [2]Q2; Q1 ← Q1 + Q2
7  return Q1
```

Table 3.4 – Generalized Montgomery curve formulas to curves in short Weierstraß equation [BJ02, IT02]. The formulas are in projective coordinates. The value $\mathrm{x_Q}$ denotes the affine x-coordinate of Q, the input point of Algorithm 15 about the Montgomery ladder.

$Q_1 + Q_2$	$[2]Q_1$
$X_{Q_1+Q_2} = -4bZ_{Q_1}Z_{Q_2}(X_{Q_1}Z_{Q_2} + X_{Q_2}Z_{Q_1}) + (X_{Q_1}X_{Q_2} - aZ_{Q_1}Z_{Q_2})^2$ $Z_{Q_1+Q_2} = \mathrm{x_Q}(X_{Q_1}Z_{Q_2} - X_{Q_2}Z_{Q_1})^2$	$X_{[2]Q_1} = ({X_{Q_1}}^2 - a{Z_{Q_1}}^2)^2 - 8bX_{Q_1}{Z_{Q_1}}^3$ $Z_{[2]Q_1} = 4Z_{Q_1}({X_{Q_1}}^3 + aX_{Q_1}{Z_{Q_1}}^2 + b{Z_{Q_1}}^3)$

Table 3.5 – Steps to compute point addition and doubling formulas from [BJ02, IT02] reported in Table 3.4.

$Q_1 + Q_2$	$[2]Q_1$
$R = X_{Q_1}Z_{Q_2} + X_{Q_2}Z_{Q_1}$	$E = 2X_{Q_1}Z_{Q_1}$
$S = X_{Q_1}X_{Q_2}$	$F = {X_{Q_1}}^2$
$L = Z_{Q_1}Z_{Q_2}$	$G = {Z_{Q_1}}^2$
$D = RL$	$H = -4bG$
$J = R^2 - 4SL$	$I = aG$
$X_{Q_1+Q_2} = -4bD + (S - aL)^2$	$X_{[2]Q_1} = EH + (F - I)^2$
$Z_{Q_1+Q_2} = \mathrm{x}_Q J$	$Z_{[2]Q_1} = 2E(F + I) - GH$

and additions are needed to perform an ECSM using the Montgomery ladder.

The formulas used for point addition and doubling in our ECSM implementations are from [BJ02, IT02] (in projective coordinates) because they are applicable to arbitrary curves in short Weierstraß equation. These formulas and the steps to compute them are respectively reported in Tables 3.4 and 3.5. The reader can see from the computation

Algorithm 16: RNS Montgomery reduction algorithm [PP95]

Input: $x_{\mathcal{M}}$, $x_{\mathcal{M}'}$
Precomp.: $p_{\mathcal{M}'}$, $(-p^{-1})_{\mathcal{M}}$, $(M^{-1})_{\mathcal{M}'}$
Output: $s_{\mathcal{M}}$ and $s_{\mathcal{M}'}$ where $s = (xM^{-1}) \bmod p$
1 $q_{\mathcal{M}} \leftarrow x_{\mathcal{M}} \times (-p^{-1})_{\mathcal{M}}$
2 $q_{\mathcal{M}'} \leftarrow BE_{\mathcal{M}\to\mathcal{M}'}(q_{\mathcal{M}})$
3 $r_{\mathcal{M}'} \leftarrow x_{\mathcal{M}'} + q_{\mathcal{M}'} \times p_{\mathcal{M}'}$
4 $s_{\mathcal{M}'} \leftarrow r_{\mathcal{M}'} \times (M^{-1})_{\mathcal{M}'}$
5 $s_{\mathcal{M}} \leftarrow BE_{\mathcal{M}'\to\mathcal{M}}(s_{\mathcal{M}'})$

steps in Table 3.5 that each iteration of the loop of Algorithm 15 of the Montgomery ladder is made of multiplications, additions and subtractions mod p (the operations are performed in the underlying finite field $\mathbb{F}_p$ of the curve). The main cost of these operations in RNS comes from the mod p reduction. Indeed, the cost of the RNS mod p reduction is quadratic in EMMs while that of multiplications, additions and subtractions is linear in EMMs; see Section 1.3.

We choose to perform the mod p reduction with the state-of-the-art algorithm from Posch and Posch [PP95], recalled in Algorithm 16. This algorithm comprises two BEs (at lines 2 and 5), and some multiplications and additions (at lines 1, 3 and 4). The cost of Algorithm 16 comes mainly from the two BEs. Some precomputations of Algorithm 16 can be combined with some of the BE algorithm in use, as have shown Gandino *et al.* [GLP+12] and Guillermin [Gui10]. These combinations allow to reduce the number of multiplications between the input residues and the precomputations. We use these optimizations in our implementations.

In summary, the ECSM can be viewed as composed of BEs and basic operations (multiplications, additions and subtractions) outside the BEs. Therefore, a flexible implementation of the ECSM is performed through flexible BEs and flexible basic operations.

Algorithmical Description

Since the ECSM is made of BEs and basic operations, an algorithmical description of the flexible basic operations is provided in addition to the algorithmical descriptions of the flexible BEs (KBE and HBE) presented in Subsections 3.2.1 and 3.3.1. Let x and y be two large integers. Algorithm 17 describes the RNS computation in base $\mathcal{M}$ of $s = x \diamond y$ using q PCs, where the symbol $\diamond$ stands for any of the basic operations—multiplication, addition and subtraction. The inner loops bounded by q/c and c at lines 2 and 3 describe

the q PCs running in parallel to compute the basic operation. The outer loop bounded by rc/q at line 1 describes the mapping of rc/q VCs on each PC, one VC at a time.

Algorithm 17 is useful in computations of the formulas in Table 3.5 (without the mod p reduction) and in computations of lines 1, 3 and 4 of Algorithm 16 (for the mod p reduction). For RNS computations of basic operations in base $\mathcal{M}'$, the same algorithm is used by replacing $\mathcal{M}$ with $\mathcal{M}'$.

Algorithm 17: Flexible basic operation in the RNS base $\mathcal{M}$. The function $f(v,i)$ is given by $f(v,i) = \frac{q}{c}(v-1)+i$.

Input: $x_{m_{i,j}}$ and $y_{m_{i,j}}$ for all i, $1 \leq i \leq r$ and j, $1 \leq j \leq c$
Output: $s_{m_{i,j}}$ for all i, $1 \leq i \leq r$ and j, $1 \leq j \leq c$

1 **for** $v \leftarrow 1$ **to** rc/q **do**
2   **for** $i \leftarrow 1$ **to** q/c **parallel do**
3   **for** $j \leftarrow 1$ **to** c **parallel do**
4             $s_{m_{f(v,i),j}} = \left| x_{m_{f(v,i),j}} \diamond y_{m_{f(v,i),j}} \right|_{m_{f(v,i),j}}$

Architecture

The architectures used for the flexible ECSMs are the ones at Figures 3.3 and 3.7 used for the flexible KBE and HBE respectively, depending on the BE used in the flexible ECSM. Additional precomputations related to the RNS MR (Algorithm 16) and the point doubling and addition formulas (curve related values such as $-a$ and $-4b$) in Table 3.5 are stored in the memories. The arithmetic processing performed by the *rower* remains unchanged: the one at Figure 3.4 if the flexible ECSM uses KBE, and the ones at Figures 3.8 and 3.9 if the flexible ECSM uses HBE. The basic-operation computations described in Algorithm 17 are performed using the architecture corresponding to the chosen BE. If KBE is used in the ECSM implementation, the basic operations (outside the BEs) are performed by the architecture at Figure 3.3 with the *rower* performing the arithmetic processing described in Figure 3.4. If HBE is used in the ECSM implementation, the same basic operations are performed by the architecture at Figure 3.7 with the *rower* performing the combination of the upper and the lower part of the arithmetic processing depicted in Figures 3.8 and 3.9.

On Flexible Multi-Level-Security ECSMs

Depending on its usage, an application may need two (or more) security levels. For example, the application may need a lower security level for a frequent usage and a higher one for a less frequent usage. In such a situation, two underlying finite fields (of the ECSM), corresponding to the two desired security levels, are needed. The designer is forced to choose between performing two implementations of the ECSMs over the two finite fields (one ECSM per finite field) or one implementation of the ECSM over the largest finite field (to ensure the highest security level by default). In the first case, the two implementations use separate hardware resources, leading to a used quantity of hardware resources greater than the one needed by each implementation. In the second case, the solution frequently uses more time than normally needed (the lower security level is more frequently used by the application). The flexible architectures at Figures 3.7 and 3.3 can be employed to implement the two ECSMs (over the two finite fields) on the same hardware resources with appropriate performance for each ECSM. The term "flexible multi-level-security ECSMs" indicates two (or more) ECSMs over finite fields of different sizes can be implemented using the same hardware resources, and this quantity of hardware resources is flexible.

Let us consider the case of two finite fields $\mathbb{F}_{p_1}$ and $\mathbb{F}_{p_2}$ where p_1 and p_2 are two large prime such that the bitsize of p_2 is twice that of p_1. If n VCs are needed for the flexible ECSM over $\mathbb{F}_{p_1}$, then $2n$ VCs can be used for the flexible ECSM over $\mathbb{F}_{p_2}$ (the same width of VCs is considered for the ECSM implementations on $\mathbb{F}_{p_1}$ and $\mathbb{F}_{p_2}$). The same architecture can be used to implement the flexible ECSMs over $\mathbb{F}_{p_1}$ and $\mathbb{F}_{p_2}$ by mapping respectively n/q and $2n/q$ VCs onto each PC. An additional signal, that indicates which one of the flexible ECSM over $\mathbb{F}_{p_1}$ or over $\mathbb{F}_{p_2}$ is being performed, has to be added to the architecture. Mainly, the core of the architecture remains the same, that is, the arithmetic processing performed by each *rower* is unchanged. For a given number of PCs, this constancy of the architecture core would lead to the same number of DSPs used regardless of the underlying finite field ($\mathbb{F}_{p_1}$ or $\mathbb{F}_{p_2}$) of the ECSM.

On the memory of each PC are stored the precomputations needed for the implementation of the ECSM over $\mathbb{F}_{p_1}$ *and* over $\mathbb{F}_{p_2}$. If the two sets of precomputations per memory (of each PC) can be stored in a single BRAM, then the solutions of the flexible multi-level-security ECSMs would use the same area (in DSPs and BRAMs) as the solutions of a "single" flexible ECSM (without targeting the multi-level-security aspect). This situation is likely to occur since BRAMs in modern FPGAs have large storage capacities which are usually underexploited in RNS implementations of asymmetric cryptosystems;

see, for example Tables 3.6 and 3.8 for the occupancies of BRAMs in our flexible ECSM implementations. Although using the same hardware resources (in DSPs and BRAMs), the solutions of the flexible ECSMs over $\mathbb{F}_{p_1}$ and $\mathbb{F}_{p_2}$ would not present the same performance. Indeed, the solution of the flexible ECSM over $\mathbb{F}_{p_2}$ would be slower since more computations have to be performed per PC (twice VCs are mapped onto each PC).

The flexible multi-level-security ECSMs were not implemented. The FPGA implementation results presented in the next subsection only concern the flexible ECSMs over a single finite field.

3.4.2 FPGA Implementation Results

We implemented the flexible ECSMs using the same environment as the flexible BEs. The ECSMs using KBE and HBE are respectively called ECSM-KBE and ECSM-HBE. Similarly to the flexible BEs, the considered finite field $\mathbb{F}_p$ is of 256-bit elements and the PCs are 17-bit wide. Also, the flexible ECSM-KBE was implemented for $q \in \{1, 2, 4, 8, 16\}$ PCs and the flexible ECSM-HBE for $q \in \{2, 4, 8, 16\}$ PCs. Similarly to the flexible HBE, the flexible ECSM-HBE was not implemented for $q = 1$ PC owing to HBE architecture (at least 2 PCs are required to implement HBE $c = 2$).

Flexible ECSM-KBE

FPGA implementation results of the flexible ECSM-KBE are reported in Table 3.6. The hardware resources and the performance vary depending on the number of PCs used for the implementation. The resource and performance variations in relation to the number of PCs are depicted in Figure 3.13. These variations are similar to the ones observed for the flexible KBE. The numbers of DSPs and BRAMs increase linearly with the number of PCs, that is, they increase with a factor k when the number of PCs increases from q to kq. On the other hand the number of slices increases less than linearly with the number of PCs, that is, it increases with a factor less than k when the number of PCs increases from q to kq. For example, using four PCs instead of one PC causes a 4 times increase in the numbers of DSPs (12 instead of 3) and BRAMs (4 instead of 1) but only a 1.4 times increase in the number of slices (4432 instead of 3120).

The variations of the number of cycles and the time are also similar to the ones observed for the flexible KBE. The number of cycles and the time decrease less than linearly in relation to the increase of the number of PCs. In other words, the number of cycles and

Table 3.6 – HLS implementation results on a XCZU7EV-FFVC1156 FPGA for the ECSM-KBE using $q \in \{1, 2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the PCs 17-bit wide.

nb. PCs	1	2	4	8	16
nb. slices	3120	3317	4432	5841	13621
nb. DSPs	3	6	12	24	48
nb. BRAMs (occupancy)	1 (38%)	2 (19%)	4 (10%)	8 (5%)	16 (3%)
period (ns)	4.4	4.5	4.7	4.9	5.1
nb. cycles	2541429	1376748	790575	481648	311713
time (ms)	11.2	6.2	3.7	2.4	1.6

the time decrease with a factor less than k when the number of PCs increases from q to kq. As explained for the flexible KBE, the less-than-linear decrease in the number of cycles results from the various WCs being > 0 in our HLS implementations (see Part "Number of Cycles and Time analyses" in Subsection 3.2.3). Since the periods of the implementation solutions (for the various numbers of PCs) are close, the less-than-linear decrease in the number of cycles results in a less-than-linear decrease in the time. For instance, when the number of PCs is increased from one to four, the time decreases by a factor 3.0, from 11.2 ms to 3.7 ms.

One BRAM is allocated to each PC. Each BRAM contains precomputations related to the rc/q VCs that are mapped onto the PC it is allocated to. The total number of precomputations related to all the rc VCs does not vary with the number of PCs. The precomputations are equally shared between the BRAMs of the PCs in a way that each precomputation can be accessed locally, that is, by the *rower* of the same PC. The more PCs are used, the lesser each BRAM is occupied. The occupancy of one BRAM is reported in Table 3.6 for the various number of used PCs. These occupancies are the ceils of the values computed by the formula

$$\frac{\text{nb. stored bits}}{36000 \text{ bits}} \times 100. \tag{3.1}$$

The denominator is the BRAM size in the Xilinx 7-series FPGAs, that is, 36 Kb [Xil19a, p. 14] (two RAMs of 18 Kb each). The reader can see that the BRAMs are not fully occupied regardless of the number of PCs.

Since the used hardware resources differ depending on the number of PCs of the

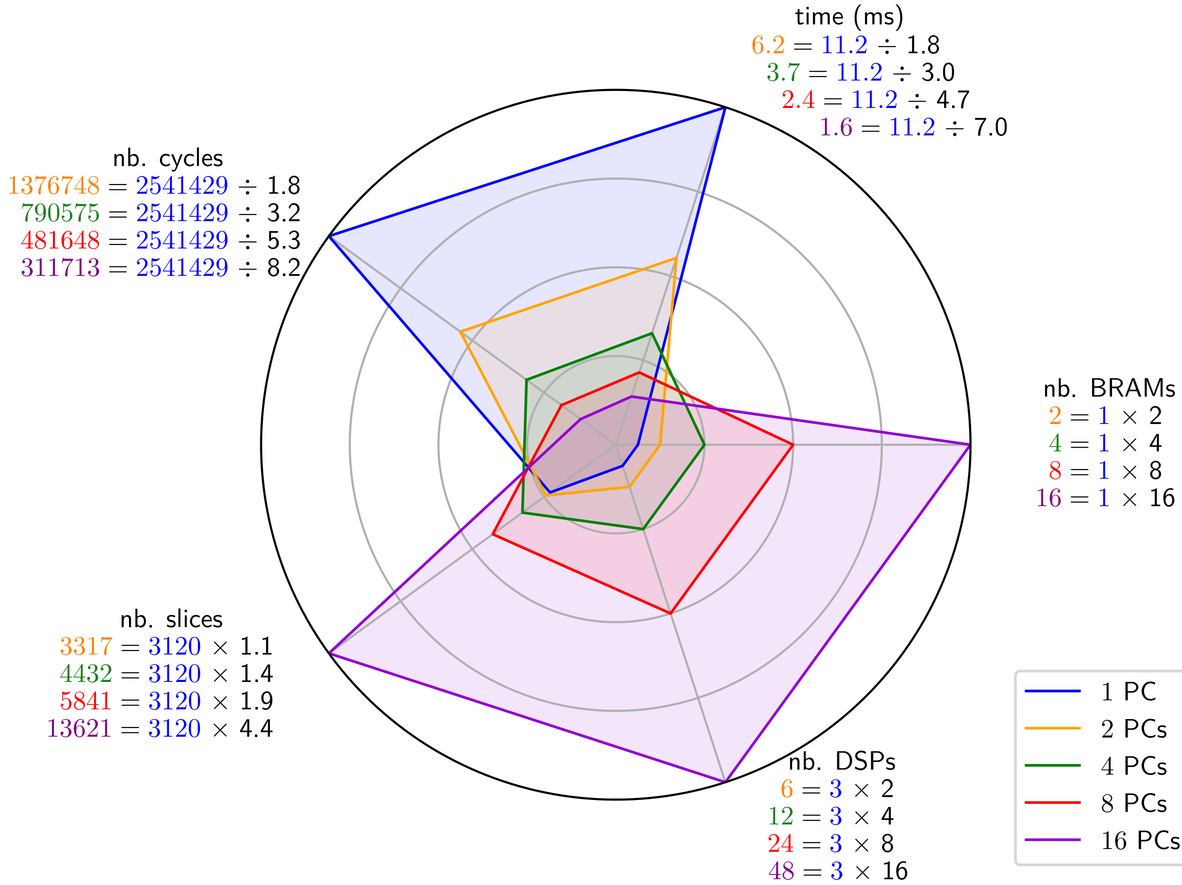


Figure 3.13 – Comparison of FPGA implementation results of the flexible ECSM-KBE for $q \in \{1, 2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the PCs 17-bit wide.

implementation, we introduce additional metrics to assess the obtained gain in time when increasing hardware resources. These metrics are DSPs × time, slices × time and BRAMs × time, and they provide information about area *vs.* time trade-offs. The HLS implementation results according to these metrics are reported in Table 3.7. The DSPs × time and BRAMs × time trade-offs are the best (smallest) for $q = 1$ PC, and increase with the number of PCs while the contrary is observed for the slices × time trade-off (except for $q = 16$ PCs). Overall, the area *vs.* time trade-offs are better when a few PCs are used for the ECSM-KBE implementation. This result is to be expected since the WCs being > 0 have lesser impact on the time when the flexible ECSM-KBE implementation is performed on fewer PCs.

In summary, the used hardware resources increase with the number of PCs (linearly for the DSPs and BRAMs and less than linearly for the slices). On the other hand, the

Table 3.7 – HLS implementation results of the ECSM-KBE according to introduced metrics of area *vs.* time trade-offs for $q \in \{1, 2, 4, 8, 16\}$ PCs.

nb. PCs	1	2	4	8	16
DSPs × time	33.6	37.2	44.4	57.6	76.8
slices × time	34944	20565.4	16398.4	14018.4	21793.6
BRAMs × time	11.2	12.4	14.8	19.2	25.6

time decreases less than linearly in relation to the increase of the number of PCs. This less-than-linear decrease in the time is mainly due to the various WCs being > 0 in the flexible ECSM-KBE implementations. As a result, ECSM-KBE implementations with fewer PCs present the best area *vs.* time trade-offs.

Flexible ECSM-HBE

FPGA implementation results of the flexible ECSM-HBE are reported in Table 3.8. Similarly to the flexible ECSM-KBE, the hardware resources and the performance vary depending on the number of PCs used for the implementation. Figure 3.14 depicts these variations. Similarly to the flexible HBE, the comparison base of the variations is the implementation results obtained for two PCs (ECSM-HBE implementation is not possible for $q = 1$ PC). Again, the numbers of DSPs and BRAMs increase linearly with the number of PCs while the number of slices increases less than linearly. For instance, using four PCs instead of two PCs results in a 2 times increase in the numbers of DSPs (8 instead of 4) and BRAMs (4 instead of 2) but only a 1.1 times increase in the number of slices (4851 instead of 4307).

Similarly to the flexible BEs and ECSM-KBE, the number of cycles and the time of the flexible ECSM-HBE solutions decrease less than linearly in relation to the increase of the number of PCs. The various WCs > 0 account for the less-than-linear decrease in the number of cycles (see Part "Number of Cycles and Time analyses" in Subsection 3.2.3). At its turn, the less-than-linear decrease in the number of cycles results in a less-than-linear decrease in the time (the periods of the implementation solutions for the various numbers of PCs are close). For example, when the number of PCs increases from two to four PCs, the time decreases by a factor 1.4 (4.3 ms for two PCs and 3.0 ms for four PCs).

One BRAM is allocated to each PC, similarly to the flexible ECSM-KBE. The BRAM of each PC contains precomputations related to the rc/q VCs mapped onto the PC. All

Table 3.8 – HLS implementation results on a XCZU7EV-FFVC1156 FPGA for the ECSM-HBE using $q \in \{2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the PCs 17-bit wide.

nb. PCs	2	4	8	16
nb. slices	4307	4851	6145	14599
nb. DSPs	4	8	16	32
nb. BRAMs (occupancy)	2 (14%)	4 (7%)	8 (4%)	16 (2%)
period (ns)	4.3	4.8	4.9	5.1
nb. cycles	1002615	637235	444325	305763
time (ms)	4.3	3.0	2.2	1.5

precomputations needed for the ECSM-HBE computation are equally shared between the BRAMs, regardless of the number of PCs in use. The occupancy of one BRAM is reported in Table 3.8 for the different number of used PCs. The BRAM occupancy is computed using the formula at Equation 3.1, similarly to the flexible ECSM-KBE. Again, the BRAMs are also not fully occupied in the flexible ECSM-HBE implementation.

The flexible ECSM-HBE implementation results according to the introduced metrics of area *vs* time trade-offs are reported in Table 3.9. Similarly to the flexible ECSM-KBE results, the flexible ECSM-HBE trade-offs in DSPs $\times$ time and BRAMs $\times$ time are better when implementations are performed on fewer PCs. The reverse is true for the slices $\times$ time trade-off, except for $q = 16$ PCs. For $q = 16$ PCs, the HLS tool uses much more LUTs, leading to much more slices and resulting in a much greater slices $\times$ time trade-off (we recall that a slice is made of LUTs, FFs and other logic elements such as a small multiplexer). Overall, the area *vs.* time trade-offs are better when the ECSM-HBE is implemented with fewer PCs. As for the flexible ECSM-KBE, this result is not surprising since the WCs being > 0 have less impact on the time of on-few-PCs implementations than on that of on-many-PCs implementations.

To summarize, when the number of PCs increases, the hardware resources also increase (linearly in DSPs and BRAMs and less than linearly in slices) while the time decreases less than linearly. The flexible ECSM-HBE solutions have their best area *vs.* time trade-offs when they are implemented using few PCs.

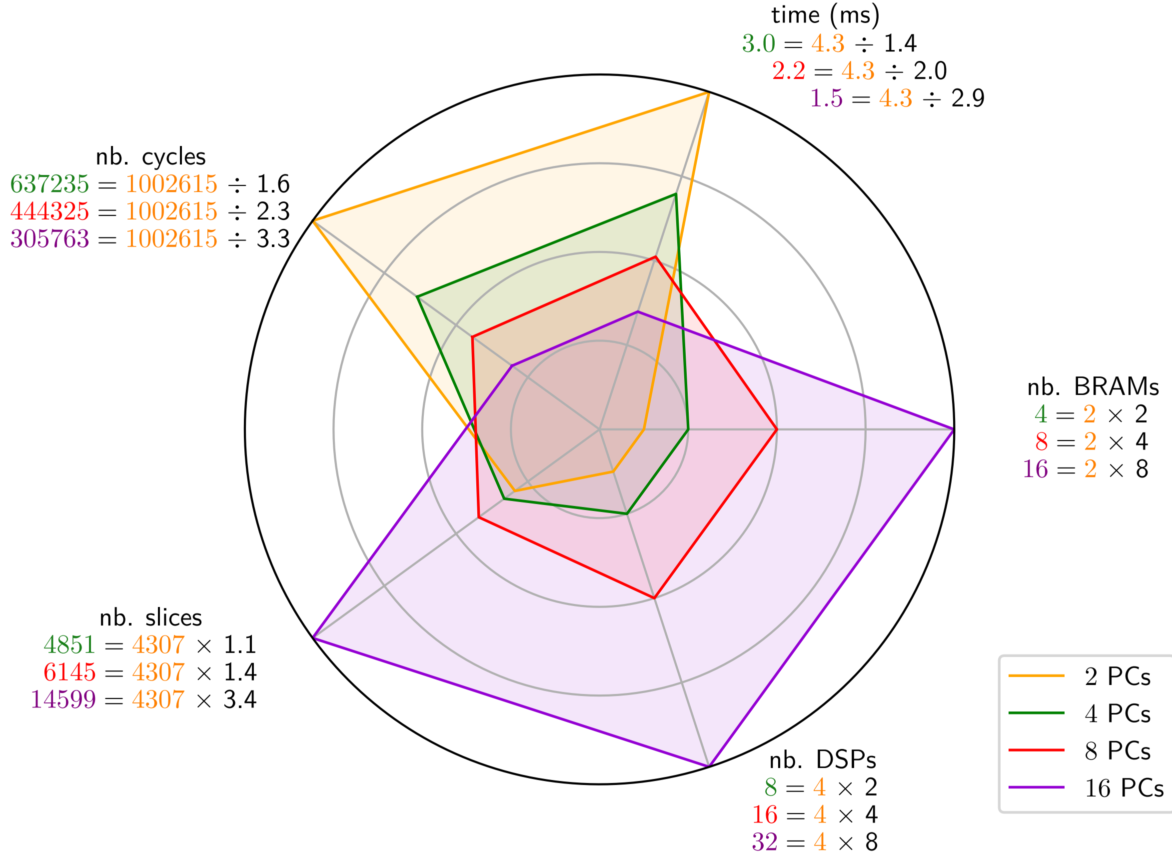


Figure 3.14 – Comparison of FPGA implementation results of the flexible ECSM-HBE for $q \in \{2, 4, 8, 16\}$ PCs. Finite fields are of 256-bit elements and the PCs 17-bit wide.

Table 3.9 – HLS implementation results of the ECSM-HBE according to introduced metrics of area *vs.* time trade-offs for $q \in \{2, 4, 8, 16\}$ PCs.

nb. PCs	2	4	8	16
DSPs × time	17.2	24	35.2	48
slices × time	18520.1	14553	13519	21898.5
BRAMs × time	8.6	12	17.6	24

Comparison Flexible ECSM-KBE *vs.* Flexible ECSM-HBE

In Table 3.10 flexible ECSM-KBE solutions and flexible ECSM-HBE ones, reported respectively in Tables 3.6 and 3.8, are put together. The comparison of the flexible ECSMs is performed for $q \in \{2, 4, 8, 16\}$ PCs since the ECSM-HBE is not implemented using one PC. In Figure 3.15 are plotted the time, the numbers of cycles, DSPs and slices.

Table 3.10 – Comparison of flexible ECSM-HBE solutions with flexible ECSM-KBE ones for the various numbers of PCs. ECSM-HBE is not implemented for one PC.

nb. PCs	1		2		4		8		16	
BE algorithms	KBE	HBE	KBE	HBE	KBE	HBE	KBE	HBE	KBE	HBE
nb. slices	3120	-	3317	4307	4432	4851	5841	6145	13621	14599
nb. DSPs	3	-	6	4	12	8	24	16	48	32
nb. BRAMs	1	-	2		4		8		16	
period (ns)	4.4	-	4.5	4.3	4.7	4.8	4.9	4.9	5.1	5.1
nb. cycles	2541429	-	1376748	1002615	790575	637235	481648	444325	311713	305763
time (ms)	11.2	-	6.2	4.3	3.7	3.0	2.4	2.2	1.6	1.5

ECSM-HBE solutions are faster than ECSM-KBE ones for all $q \in \{2, 4, 8, 16\}$ PCs. For example, the ECSM-HBE solution is 19% faster than the ECSM-KBE one for $q = 4$ PCs. For $q = 2$ PCs, the gain in time amounts to 31%. The number of used BRAMs remains the same for ECSM-HBE and ECSM-KBE. Similarly to what has been observed for the BEs, ECSM-HBE solutions are 33% smaller than ECSM-KBE ones in DSPs for all $q \in \{2, 4, 8, 16\}$ PCs. However, there is a small extra cost in slices for ECSM-HBE. For example, ECSM-HBE costs 9% more slices than ECSM-KBE for $q = 4$ PCs. The maximum extra cost in slices for ESCM-HBE solutions is 23%, reached for $q = 2$ PCs. Globally, the extra costs in slices for ECSM-HBE solutions are overly compensated by their gains in DSPs, in addition to ECSM-HBE solutions being faster than ECSM-KBE ones for all $q \in \{2, 4, 8, 16\}$ PCs. Consequently, the area *vs.* time trade-offs are in favor of the flexible ECSM-HBE for all $q \in \{2, 4, 8, 16\}$ PCs.

Let us recall that the decrease factor in the number of cycles in HBE is smaller than the one in KBE when the number of PCs increases from q to kq (see Part "Comparison Flexible KBE *vs.* Flexible HBE" in Subsection 3.3.3). This result induces a decrease factor in the number of cycles in ECSM-HBE solutions smaller than the one in ECSM-KBE solutions when the number of PCs increases. For example, by using four PCs instead of two PCs, the number of cycles of the ECSM-HBE solution is decreased by 36% while that of the ECSM-KBE solution is decreased by 43%. Consequently, the gain in time of ECSM-HBE solutions compared with ECSM-KBE ones decreases when the number of PCs increases. In other words, the fewer the PCs, the greater the gain in time of ECSM-HBE solutions compared with ECSM-KBE ones. Considering that the best area *vs.* time trade-offs are obtained for ECSM implementations using fewer PCs regardless of the chosen BE, this result can also be put in this way: The better is the area *vs.* time trade-off of the ECSM for each BE, the faster are ECSM-HBE solutions compared with ECSM-KBE ones.

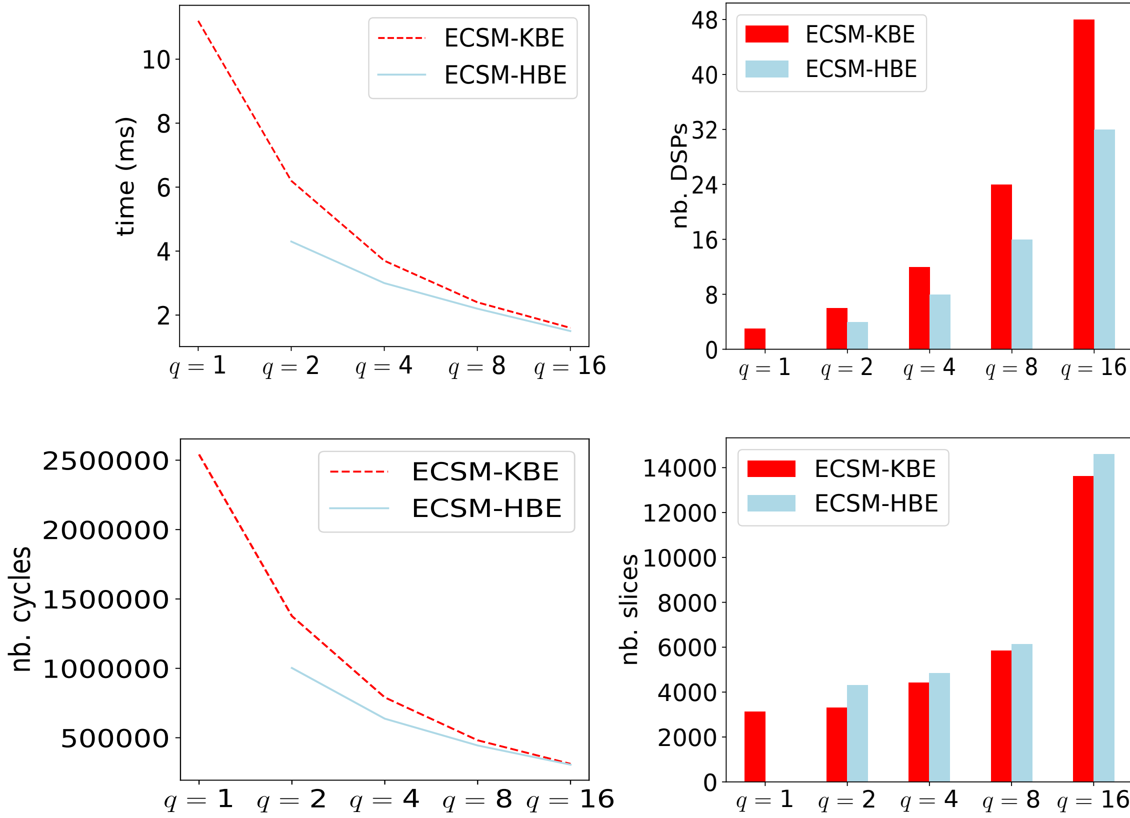


Figure 3.15 – Comparison of the time, the numbers of cycles, DSPs and slices between the flexible ECSM-HBE solutions and the flexible ECSM-KBE ones for various numbers of PCs. ECSM-HBE is not implemented for one PC.

Comparison with Results from the Literature

Comparing implementations results of various works is always tricky because of the numerous parameters that have to be considered. For the comparison to be relevant, parameters have to be the same or close enough. These parameters include the ECSM algorithm as well as the point doubling and addition formulas, the FPGA and its configuration tool. In our case, for each to-be-compared-with result from the literature, the flexible ECSM-KBE and ECSM-HBE should be implemented on the same FPGA with the same tool using the same formulas in order to enable a fair comparison. This work is not possible in the time frame allowed by the PhD grant. Besides, we choose to use HLS for our FPGA implementations because it enhances fast configurations of FPGAs (by focusing almost exclusively on the functionality), and numerous FPGA implementations are required in

our demonstration of the RNS flexibility. The selected HLS tool is not compatible with most of the FPGAs used in ECSM implementations from the literature. Therefore, we select a few implementations results that we consider to be the most significant from the literature and show that our results are comparable with them. The FPGA configurations in all selected references were described using a hardware description language (HDL). Configuring an FPGA using an HDL requires much more time than using HLS.

The papers from the literature chosen for the comparison are the following ones, listed below in reverse chronological order. Our ECSM implementations target arbitrary underlying prime field of the curve, similarly to all selected references except [GP08].

- The first paper by Bajard and Merkiche [BM14] is instructive for our comparison because the authors use RNS for operations in the underlying finite field and they choose the Montgomery ladder as the ECSM algorithm (the same as we do). The main difference compared with our work comes from the architecture. The reduction algorithm chosen in their work to perform the mod m_i in the various *rowers* is the Montgomery reduction algorithm [Mon85]; this is to support their main contribution: avoiding the constraints related to choosing pseudo-Mersenne for the RNS base moduli. For example, there is not enough pseudo-Mersenne moduli of size 17 bits to form the two bases needed to perform modular operation in an underlying finite field of 521-bit elements. Another advantage of using this paper in our comparison is that their implementation was performed on an FPGA from the Xilinx 7-series FPGAs, as ours. Their ECSM solution is the fastest among the ones using RNS.
- In the second paper, Ma *et al.* [MLPJ13] do not use RNS, but usual positional number system for operations in the underlying finite field. The ECSM is computed using the *window* method with randomized jacobian coordinates, as proposed in [Mö01]. The main advantage of using this method is that the ECSM is computed with less iterations compared for example to a Montgomery ladder. Besides, the number of point additions is considerably reduced depending on the chosen size of the window. As far as we know, their solution is currently the fastest one for the ECSM over arbitrary prime field reported in the literature.
- The contribution of Guillermin [Gui10] is also instructive in our comparison because RNS is used for operations on the underlying finite field as well as the Montgomery ladder for the ECSM. Besides, the overall core of the architecture in Figure 3.3 used for the flexible ECSM-KBE implementation comes from this paper, itself adapted

from the one in [KKSS00]. However, although the arithmetic operation of the *rower* is the same in [Gui10] and this work, we were unable to manually place the registers for pipeline stages (performed automatically by the HLS tool) as in their arithmetic processing due to limitations of the current HLS tools. In addition, the author uses the point addition and doubling formulas from [BDE13] whereas we use the ones in [BJ02, IT02].

- The last paper is by Güneysu and Paar [GP08]. RNS is not used for operations on the underlying finite field. Besides, the authors target NIST pseudo-Mersenne prime fields and design their operators for this purpose; the mod p reductions are then faster than in arbitrary prime fields. The ECSM was implemented using the *double-and-add* algorithm. The *double-and-add* algorithm usually costs less point additions than the Montgomery ladder. For fairly chosen scalars, the gain in point additions is about half the bit size of the underlying finite field of the curve. The authors also estimate the results if the *window* method were used. To keep a fair comparison with real implementation results from the literature and ours, this estimation is not reported in our comparison. The reported results are from Tables 1 and 2 of [GP08].

Table 3.11 contains our ECSM implementation results and the ones in the mentioned papers from the literature. Since the used hardware resources differ from one implementation to another, the metrics slices × time, DSPs × time and BRAMs × time are added. The added metrics provide a close evaluation of the gain in time when using more area or *vice versa*. The smaller the value, the better the area *vs.* time trade-off. The symbol $^{(\star)}$ in the frequency column of Table 3.11 indicates that the frequencies are the ceils of the period inverses for our implementations, and the ceils of the frequencies for the results from the literature.

The area and the area × time trade-offs of our ECSM solutions, the one in [BM14] and the best one in [MLPJ13] at line 12 of Table 3.11 are plotted in Figure 3.16. We choose to plot the results in these two references because their FPGA technologies are the closests to ours. Since [BM14] and [MLPJ13] do not target flexible implementations, their implementation results are plotted near ours for $q = 16$ PCs.

Our ECSM solutions for all $q \in \{1, 2, 4, 8\}$ are smaller in DSPs and BRAMs than the ones from [BM14] and [MLPJ13]. This result demonstrates the usefulness of our flexible ESCM-KBE and ECSM-HBE. The gain in area is compensated by a lost in time, leading to mixed results in area × time trade-offs. For example, our ECSM-HBE solutions for

Table 3.11 – Comparison of our FPGA implementation results of the ECSM with the ones from the literature.

reference (FPGA)	used BE if RNS	slices	DSPs	BRAMs	freq.$^{(\star)}$ (MHz)	time (ms)	slices × time / DSPs × time / BRAMs × time
ours (XCZU7EV-FFVC1156)	KBE (1 PC)	3120	3	1	228	11.2	34944 / 33.6 / 11.2
	KBE (2 PCs)	3317	6	2	223	6.2	20565.4 / 37.2 / 12.4
	HBE (2 PCs)	4307	4	2	233	4.3	18520.1 / 17.2 / 8.6
	KBE (4 PCs)	4432	12	4	213	3.7	16398.4 / 44.4 / 14.8
	HBE (4 PCs)	4851	8	4	209	3.0	14553 / 24 / 12
	KBE (8 PCs)	5841	24	8	205	2.4	14018.4 / 57.6 / 19.2
	HBE (8 PCs)	6145	16	8	205	2.2	13519 / 35.2 / 17.6
	KBE (16 PCs)	13621	48	16	197	1.6	21793.6 / 76.8 / 25.6
	HBE (16 PCs)	14599	32	16	197	1.5	21898.5 / 48 / 24
[BM14] (Kintex-7)	KBE	1630	46	16	282	0.612	997.56 / 28.152 / 9.792
[MLPJ13] (XC4VLX100-12FF1148)		4655	37	11	250	0.44	2048.2 / 16.28 / 4.84
[MLPJ13] (XC5LX110T-3FF1136)		1725	37	10	291	0.38	655.5 / 14.06 / 3.8
[Gui10] (EP1S60)	KBE	16200 LE	125	0	91	1.17	18954 (LE·ms) / 146.25 / 0
[Gui10] (EP2S30)	KBE	9177 ALM	96	0	158	0.68	6240.36 (ALM·ms) / 65.28 / 0
[GP08] (XC4VFX12-12)		1715	32	11	490	0.62	1063.3 / 19.84 / 6.82

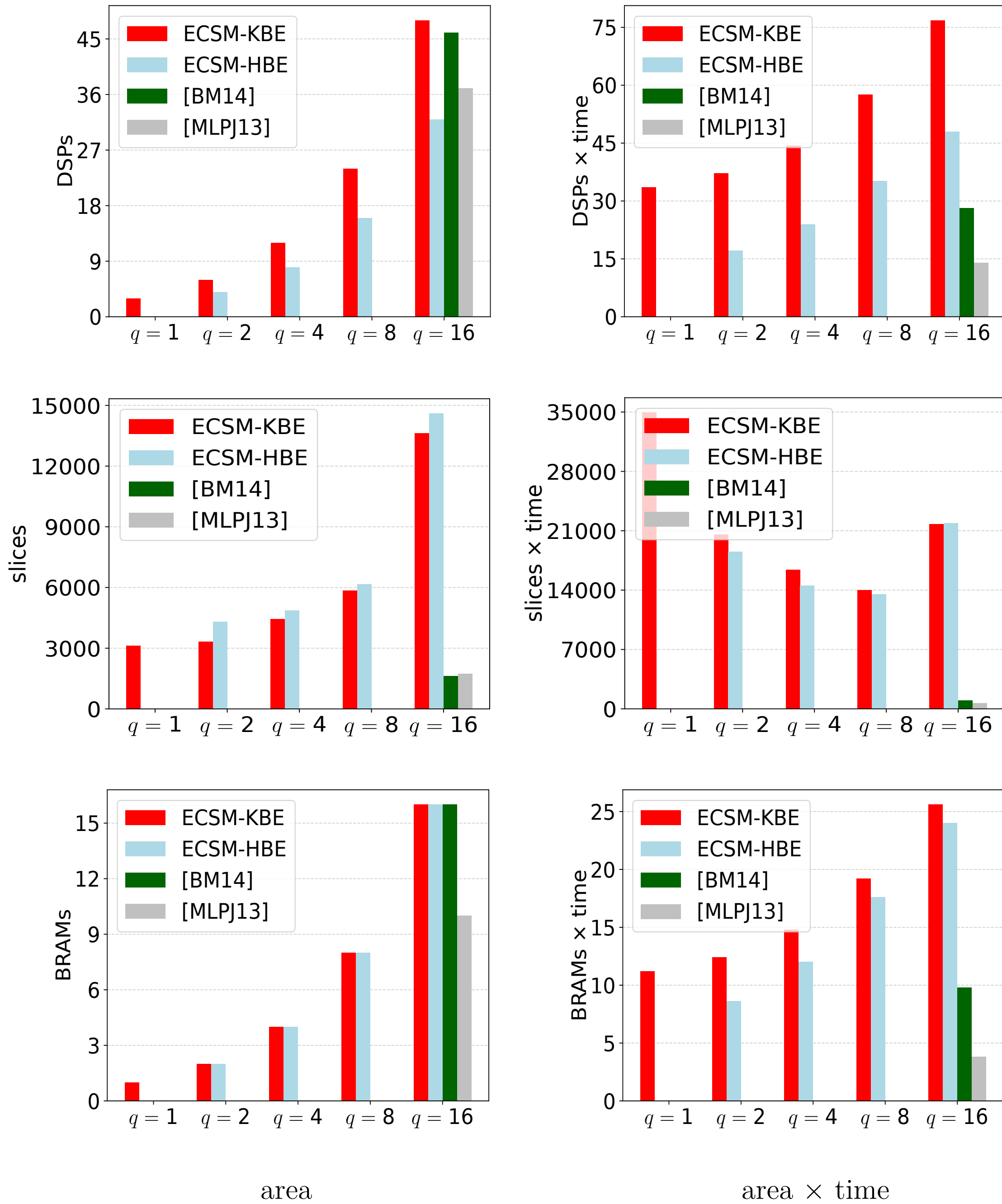


Figure 3.16 – Comparison of the area and area × time trade-offs of our ECSM implementations with the one in [BM14] and the best one in [MLPJ13]. ECSM implementations from [BM14, MLPJ13], and generally from the literature, are not *flexible*.

$q = 2$ PCs are 39% better in DSPs × time trade-off than the ones in [BM14], but 18% not as good as the ones in [MLPJ13] regarding the same metric. In BRAMs × time trade-off,

our ECSM-HBE solutions for $q = 2$ PCs are 12% better than the ones in [BM14], but 55% not as good as the ones in [MLPJ13]. Our ECSM solutions are larger in slices than the ones in [BM14, MLPJ13], resulting in slices × time trade-offs in their favor. Nevertheless, our ECSM-KBE and ECSM-HBE are flexible, and our best ECSM solutions are comparable in area × trade-offs with the best ones from the literature, notably [BM14, MLPJ13].

The implementations results from [Gui10, GP08] are instructive but the comparison with our results is difficult. The FPGA used in our work is not from the same manufacturer as the one used in [Gui10]. Therefore, any conclusion is difficult to draw, for example from the gain in DSPs × time (their DSPs are smaller) or the loss in BRAMs × time. It is also hard to conclude from the results in [GP08] because they restrict the underlying finite field $\mathbb{F}_p$ of the curve to be NIST prime fields where the mod p reduction is faster than in arbitrary prime fields; the mod p reduction is costly (approximately two BEs) in our implementations. Nevertheless, it seems safe to assume that the results from [Gui10, GP08] are equivalent to the ones from [BM14, MLPJ13] with which we compared our results.

Overall, most of our ECSM solutions are *smaller* than the ones from the literature. The comparison results are mixed in terms of area *vs.* time trade-offs. Our best ECSM solutions are comparable with the best ones from the literature, in addition to our implementations being *flexible*.

3.5 Conclusion

The RNS *flexibility* of the BE has been proven in this chapter. We have demonstrated that the BE operation can be implemented using a flexible number of PCs, leading to an adaptable utilization of hardware resources. The flexible architectures allow to choose the desired numbers of DSPs and BRAMs to be used for the implementation; their numbers vary linearly with the number of PCs. HLS implementations of the flexible HBE [DBT19] and KBE [KKSS00] provide several area *vs.* time trade-offs for the comparison of the two flexible BEs. HBE solutions are faster with better area *vs.* time trade-offs in all comparable cases.

We have also shown that the ECSM operation is *flexible*. The flexible ECSM can be implemented using either the flexible HBE or the flexible KBE. With variations of the numbers of DSPs and BRAMs similar to the ones of the flexible BEs, HLS implementations of the flexible ECSM-HBE and ECSM-KBE also provide several area *vs.* time trade-offs. ECSM-HBE solutions are always faster and present better area *vs.* time trade-offs than

ECSM-KBE ones.

Although our ECSM solutions are slower than the best ones from the literature (most of our solutions are smaller), our best solutions are comparable with the latter in terms of area *vs.* trade-offs. For example, our ECSM-HBE solutions are 39% smaller than results in [BM14] but 18% larger than results in [MLPJ13] in DSPs × time. Our ECSM implementations present the main advantage of being *flexible* compared with the ones from the literature. Consequently, the ECSM can be implemented so that the quantity of used hardware resources is adaptable and small. Therefore, such implementations can be performed on integrated circuits with limited available hardware resources. Nearly all our solutions are *smaller* than the ones from the literature, demonstrating the benefit of the *flexibility*.

Conclusion

The work presented in this thesis aims at designing RNS-flexible hardware accelerators for the ECSM operation on FPGA using HLS. In Chapter 1 ECC as the context of this thesis has been presented. We have also presented the ECSM which is the core operation in protocols of ECC applications. The computation of the ECSM over $\mathbb{F}_p$ involves numerous multiplications, additions and subtractions in $\mathbb{F}_p$, and hence numerous mod p reductions. RNS has also been presented in Chapter 1 as well as its advantages such as the independence of multiplications, additions and subtractions between the moduli of an RNS base. Besides, the base extension (BE) has been described and its importance in the RNS mod p reduction has been demonstrated.

Contributions

In Chapter 2 we presented our *hierarchical base extension* (HBE) proposed in [DBT19]. HBE proceeds by first computing super-residues which are results of partial CRTs on the residues in the input RNS base. The continuation of the CRT is then carried out on the super-residues in the output RNS base. Compared with the state-of-the-art base extension (KBE) [KKSS00], no additional contraint is needed to form the RNS bases. The theoretical gain introduced by HBE compared with KBE is up to 35%. We also proposed an architecture for HBE [DBT19], adapted from the *cox-rower* [KKSS00]. Our architecture preserves the inherent parallelism of RNS with a marginally deeper pipeline than the *cox-rower*. FPGA implementations using HLS show that HBE solutions are always faster (up to 20%), *and* in nearly all cases smaller (up to 23%) than KBE solutions. HBE solutions present the best area *vs.* time trade-off in all cases.

In Chapter 3 the *flexible HBE* and *KBE* are presented before showing how to use them to obtain flexible ECSMs. We first demonstrated that HBE and KBE are flexible, that is, they can be implemented using a flexible number of PCs. The architectures of HBE [DBT19] and KBE [KKSS00] are adapted to support this flexibility. The flexibility of the architectures allow us to select the desired numbers of DSPs and BRAMs to be used for the BE implementation. FPGA implementations using HLS show that the flexible HBE solutions are always faster (up to 28%) *and* 33% smaller in DSPs that the flexible KBE

ones. Though most flexible HBE solutions are larger in slices (up to 35%) than the flexible KBE ones, the area *vs.* time trade-off is in favor of the flexible HBE solutions in all cases.

We showed how to derive flexible ECSMs from the flexible HBE and KBE in the rest of Chapter 3. The derived *flexible ECSM-HBE* and *ECSM-KBE* can be implemented using the architectures described for the flexible HBE and KBE respectively. The hardware resources of our flexible ECSMs are adaptable owing to the flexibility of the architectures, similarly to the flexible BEs. FPGA implementations using HLS show that the flexible ECSM-HBE solutions are always faster (up to 31%) *and* 33% smaller in DSPs than the flexible ECSM-KBE ones. The flexible ECSM-HBE solutions are slightly larger in slices (up to 23%) than the flexible ECSM-KBE ones. Similarly to the flexible BEs, the area *vs.* time trade-off is always in favor of the flexible ECSM-HBE solutions. For ECSM-HBE solutions and ECSM-KBE ones, the best area *vs.* time trade-offs are given by implementations with fewer PCs. Compared with the best FPGA implementation results from the literature, nearly all our ECSM-HBE and ECSM-KBE solutions are smaller in DSPs and BRAMs. Although our ECSM solutions are slower than the best ones from the literature (most of our solutions are smaller), the area $\times$ time trade-offs of our best solutions are comparable with the latter. For instance, our ECSM-HBE solutions are 39% smaller than the ECSM ones in [BM14] but 18% larger than the ECSM ones in [MLPJ13] in DSPs $\times$ time. The smallness and adaptability of area utilizations in our solutions, obtained thanks to the *flexibility*, imply that ECSM-HBE and ECSM-KBE can be implemented on integrated circuits with limited available hardware resources.

Perspectives

In Chapter 2 the super-residues in our HBE implementations are results of partial CRTs on $c = 2$ input residues. The partial CRTs are also possible on other numbers c of input residues. We plan to study the cases $c = 3$ and 4 and design optimized architectures for HBE.

We mention in Chapter 3 (Subsection 3.4.1) that our flexible ECSM architectures can be used to implement flexible multi-level-security ECSMs. For example, using the same architecture, and hence the same numbers of DSPs and BRAMs, we could implement ECSM-KBE and ECSM-HBE over two different finite fields: one of 256-bit elements and the other of 512-bit elements. The number of VCs to be mapped onto each PC for an ECSM over the finite field of 512-bit elements would be twice the one for an ECSM over the finite field of 256-bit elements. We also plan to study this aspect of the flexibility and

how to optimize the architectures to support ECSMs over two or more finite fields.

The ECSM algorithm used in our flexible implementations is the Montgomery ladder without any additional protections (for example against DPA). How will behave countermeasures against various side-channel attacks in a context of flexible ECSM implementations is worth a study.

Appendix A

Comparaison d'algorithmes de réduction modulaire en HLS sur FPGA

This appendix contains the unabridged contribution (in French) from [DZBT19].

Comparaison d'algorithmes de réduction modulaire en HLS sur FPGA

Libey Djath[1], Timo Zijlstra[2], Karim Bigou[1], et Arnaud Tisserand[2]

[1,2]Lab-STICC UMR 6285, [2]CNRS, [1]Univ. Bretagne Occidentale, Univ. Bretagne Sud

Résumé

Dans ce travail, nous comparons différents algorithmes de réduction modulaire implantés en synthèse de haut niveau sur FPGA pour des applications de cryptographie asymétrique. Nous étudions comment effectuer les réductions modulaires en fonction des tailles et formes (particulières/quelconques) des moduli, du type et du nombre des autres opérations arithmétiques impliquées. Pour cela, nous développons une bibliothèque C, qui sera distribuée sous licence libre, d'arithmétique modulaire pour la cryptographie asymétrique.

Mots-clés : arithmétique modulaire, conception en synthèse de haut niveau, exploration d'architectures et d'algorithmes, circuit FPGA.

1. Introduction

Les implantations en cryptographie asymétrique nécessitent un support d'*arithmétique modulaire* de plus en plus avancé. RSA utilise des carrés et multiplications modulo un nombre de quelques milliers de bits. Les cryptosystèmes actuels nécessitent des séquences

d'opérations plus complexes [1] mais sur de plus petites tailles : des centaines de bits pour les courbes elliptiques (ECC) [HMV04] et hyper-elliptiques (HECC) [CF06] ; ou des dizaines de bits pour la cryptographie post-quantique (PQC) sur des réseaux euclidiens (RE).

La représentation modulaire des nombres, ou RNS pour *residue number system* [Gar59, ST67, Big14], amène un plus grand *parallélisme* interne permettant d'accélérer les calculs comme dans RSA et ECC, mais il nécessite un niveau supplémentaire de *réductions modulaires*. RNS découpe les nombres en petits morceaux dans une base de moduli premiers entre eux deux à deux notés m_i. Les calculs s'effectuent sur les restes modulo les m_i dans un *canal* propre à chaque m_i. Ces « petites » réductions modulo chaque m_i s'ajoutent aux réductions modulaires à un plus haut niveau sur les nombres complets (p. ex. le modulo p pour des éléments de $\mathrm{GF}(p)$). Dans ce papier, nous nous intéressons seulement aux réductions modulo les m_i dans le cadre de l'utilisation de RNS.

Dans le cadre PQC, on utilise des *petits corps finis* (p. ex. éléments entre 13 et 23 bits) mais pour des calculs sur des *polynômes* (p. ex. degrés entre 256 et 1024) et des *petites matrices* (souvent de tailles 2×2, 3×3 , 4×4).

Dans ce papier, nous nous intéressons uniquement aux réductions modulaires pour des tailles de quelques dizaines de bits (pour les moduli de RNS et pour les petits corps utilisés dans PQC-RE). Nous ne traitons pas des réductions pour des plus grands nombres comme ceux utilisés pour RSA et ECC (ceci fera l'objet de travaux futurs).

Les outils actuels de synthèse de circuits n'offrent pas de support très avancé pour la réduction modulaire. Ainsi, nous développons une *bibliothèque C* dédiée à l'*arithmétique modulaire* en cryptographie asymétrique utilisable en synthèse de haut niveau (HLS pour *high level synthesis*). Elle sera distribuée sous licence libre une fois suffisamment complétée, validée et documentée.

Nous utilisons la HLS pour *explorer de nombreux compromis* entre les représentations des nombres, les algorithmes de calcul et les architectures matérielles (ce qui est très coûteux en synthèse VHDL ou Verilog). Pour un modulo quelconque (c.-à-d. avec une écriture dense et sans structure comme $5101963 = (10011011101100110001011)_2$), on utilise principalement les réductions de Montgomery [Mon85] et de Barrett [Bar84]. Pour RE, on utilise plutôt un modulo spécifique avec une décomposition binaire très creuse (comme $8380417 = 2^{23} - 2^{13} + 1$) car la réduction se simplifie beaucoup (voir p. ex. [Sol99]). En RNS, on utilise souvent des moduli de la forme $m_i = 2^w \pm c_i$ où w est la taille des mots dans les canaux et les c_i des petites constantes, denses, pour des questions de performances

1. Addition, soustraction, carré, multiplication, multiplication par des constantes, inversion.

(voir p. ex. [Cra92]). Ici aussi, cela conduit à des réductions plus efficaces que pour des moduli quelconques. En pratique, il y a un compromis entre la forme du modulo, ou des moduli, et le nombre de telles valeurs utilisables.
Dans ce papier, nous présentons les résultats de notre bibliothèque pour la *réduction modulaire de petits nombres* (c.-à-d. d'au plus quelques dizaines de bits, pour RNS ou PQC-RE). Nous comparons expérimentalement l'impact des principaux algorithmes de réduction pour différentes formes de moduli. Les principaux motifs de calcul de nos applications, utilisés dans ce papier, sont la somme réduite et la somme réduite de produits. Nous évaluons comment se comportent les outils de HLS sur ces motifs de calcul et expérimentons différentes techniques d'exploration pour différentes contraintes arithmétiques et architecturales.

2. Définitions et notations

Pour la suite du papier, nous définissons et utilisons :

— m le modulo de taille w bits pouvant avoir plusieurs formes :
  — MQ : *modulo quelconque* avec une écriture dense et sans structure particulière ;
  — MSC : *modulo spécifique à écriture binaire très creuse* (p. ex. 3 bits non nuls sur w) ;
  — MSR : *modulo spécifique pour RNS* de forme $2^w \pm c$ (où c est petit, $c < 2^{w/2}$) ;
— la réduction modulaire $x \bmod m$ ou des opérations modulaires $(x \diamond y) \bmod m$ avec l'opération $\diamond \in \{\pm, \times\}$;
— le motif de calcul: une séquence d'opérations arithmétiques faisant intervenir des réductions modulaires sur des vecteurs de taille N, les motifs étudiés ici sont:
  — M1 : $\sum_{i=1}^{N} x_i \bmod m$;
  — M2 : $\sum_{i=1}^{N} x_i \times y_i \bmod m$;
— deux stratégies sont comparées pour la réduction de séquence d'opérations modulaires :
  — RIS : Réduction Intermédiaire Systématique, p. ex. $\left(\sum_{i=1}^{N} (x_i \times y_i \bmod m)\right) \bmod m$;
  — RSF : Réduction Seulement à la Fin, p. ex. $\left(\sum_{i=1}^{N} x_i \times y_i\right) \bmod m$;
— l'opérande de la réduction modulaire peut avoir plusieurs tailles (p. ex. w, $w + \lceil \log_2 N \rceil$, $2w$ ou $2w + \lceil \log_2 N \rceil$ bits dans nos applications) ;
— des données x et y de taille w bits pour les vecteurs des motifs ;
— TM le temps total de calcul (en ns) pour obtenir le résultat d'un motif.

3. Bibliothèque développée

Nous souhaitons aider les concepteurs d'implantations matérielles en cryptographie asymétrique à explorer différents compromis algorithmes/représentations des nombres/architectures de calcul. Dans ce papier, nous traitons uniquement de la réduction modulo m, un entier d'au plus quelques dizaines de bits, et de son utilisation dans quelques motifs de calcul typiques. Nous travaillons d'abord sur des m « petits » pour PQC et RNS, mais nous compléterons notre bibliothèque pour des moduli plus grands dans l'avenir (p. ex. éléments de GF(p) de quelques centaines de bits). La forme de m influençant beaucoup les algorithmes et les performances, nous avons choisi de supporter :

- des moduli quelconques (MQ), des moduli spécifiques très creux (MSC) utilisés pour PQC et des moduli spécifiques à RNS (MSR) ;
- les algorithmes de réduction de Montgomery [Mon85] et de Barrett [Bar84] pour MQ, et des algorithmes spécifiques pour MSC et MSR [Cra92, Sol99].

Nous comparons nos résultats avec l'algorithme « natif » employé par Vivado HLS lors de l'utilisation de l'opérateur `%` du langage C. Nos résultats montrent qu'il s'agit probablement d'une division euclidienne itérative dont on conserve le reste final [EL03, EL94]. En tout, nous comparons 5 algorithmes de réduction modulaire différents dans le même cadre.

Toutes nos implantations sont génériques pour des opérandes de taille w bits fixée à la conception. Pour RNS et PQC, w est de quelques dizaines de bits au plus. La taille N des vecteurs dans les motifs est aussi générique. Pour Barrett et Montgomery, il faut précalculer à la conception des constantes internes (p. ex. un inverse modulaire qui dépend de m et de w).

Nous définissons et utilisons des types de données spécifiques pour chaque taille nécessaire et des macros de transtypage (*cast*) pour adapter correctement les tailles [2]. Par exemple pour M2 en version RSF (c.-à-d. réduction seulement à la fin), il faut pouvoir réduire l'accumulation des N produits sur $2w + \lceil \log_2 N \rceil$ bits. Pour ce motif, nous avons aussi évalué une réduction intermédiaire systématique (RIS) à chaque itération.

La figure A.1 présente un extrait de code C de notre bibliothèque et un exemple de code devant être produit par l'utilisateur. La partie haute de la figure est le code de la réduction modulaire avec l'algorithme de Barrett de la bibliothèque. Les types comme `word` et `sumdword` doivent être définis par l'utilisateur selon un « *template* » fourni avec

2. On rappelle que la sémantique du langage C est assez peu mathématique, p. ex. le produit de 2 mots est un mot de même taille que les opérandes.

la bibliothèque. Par exemple, le type `word` est de taille de w bits (code pour $w = 13$: `typedef uint13 word;`), le type `dword` est de taille double pour les produits de deux nombres de w bits (code pour $w = 13$: `typedef uint26 word;`), le type `sumdword` est pour les accumulateurs sur $2w + \lceil \log_2 N \rceil$ bits, etc. L'utilisateur doit aussi définir des macros de transtypage (*cast*) pour chacun des types définis. Par exemple, la macro `W` est définie par `#define W(x) ((word) (x))`. Le *template* fournit la liste de l'ensemble des macros à définir selon les usages. Toutes ces définitions spécifiques à l'application doivent être codées par l'utilisateur en complétant le *template* `parameters.h` inclus en ligne 1. Ce fichier définit aussi la constante N choisie pour les vecteurs.

La partie basse de la figure A.1 présente un exemple de code utilisateur pour effectuer le motif M2 avec la stratégie RSF. L'utilisateur doit :

- tout d'abord inclure le fichier `parameters.h` en ligne 1 afin de « configurer » la bibliothèque pour son application ;
- spécifier ses entrées (ici 2 tableaux de N données de taille w bits en `word`) et ses sorties (ici la somme réduite donc aussi en `word`) en ligne 4 ;
- initialiser l'accumulateur avec la bonne taille en ligne 6 ;
- effectuer son calcul (ici la somme des produits selon la stratégie RSF, donc sans réduction intermédiaire) en lignes 7 et 8 ;
- et enfin effectuer la réduction en appelant la fonction de réduction choisie (ici `barrett(res)` en ligne 9).

L'utilisation est assez simple, puisqu'une fois les spécifications de l'application définies (préparation du fichier `parameters.h` à partir d'un *template* fourni), il suffit de faire quelques appels de fonctions, utiliser les bons types et éventuellement quelques *casts* pour s'assurer de la bonne taille des données intermédiaires. Les étiquettes comme `acc` sur la boucle d'accumulation en ligne 7 du code de la fonction `m2_rsf` sont utilisées pour spécifier les cibles des directives d'optimisations particulières de l'outil HLS (pipeline, déroulage de boucle, etc.).

Nos implantations ont été réalisées avec Vivado HLS 2017.4 [Xil18b] sur un FPGA Artix-7 (xc7a15) de Xilinx. Les résultats présentés ci-dessous ont comme contraintes : une taille de modulo de $w \in \{13, 17, 23, 30\}$ bits, une taille de vecteurs $N \in \{10, 20, 40, 100\}$, une période d'horloge cible de 3 ns et un effort d'optimisation par défaut (moyen). Nous avons aussi exploré l'impact de directives de pipeline et de déroulage de boucle de l'outil (voir [Xil17]). Nous obtenons ainsi plus de 2400 résultats d'implantations différentes. Nous présentons ci-dessous un sous-ensemble représentatif pour des raisons de place (d'autres

Code (de la bibliothèque) pour la réduction modulaire avec l'algorithme de Barrett :

```
1  #include "parameters.h"
2  #include "arithmod_internal.h"
3
4  word barrett(sumdword x)
5  {
6      sumword x1 = SUM_W(x >> width);
7      sumword q = SUM_W((RSW(x1) * RSW(R_const)) >> (shift - width));
8      word x0 = W(x);
9      counter c = 0;
10     if (x0 > M) c = 2;
11     else if (x0 != 0) c = 1;
12     q = q + c;
13     sumdword z = SUM_DW(q) * SUM_DW(m);
14     signword res = x - z;
15     if (res < 0) res = res + M;
16     if (res < 0) res = res + M;
17     return W(res);
18 }
```

Code (utilisateur) pour le motif M2 RSF avec réduction de Barrett :

```
1  #include "parameters.h"
2  #include "arithmod.h"
3
4  word m2_rsf(word A[N], word B[N])
5  {
6      sumdword res=0;
7      acc: for(counter i=0; i<N; i++)
8          res += DW(A[i]) * DW(B[i]);
9      return barrett(res);
10 }
```

Figure A.1 – Extraits de codes C de notre bibliothèque (haut) et de son utilisation (bas).

résultats pour d'autres contraintes seront disponibles dans la documentation de la bibliothèque). Notre attention portera principalement sur le motif M2, car il est crucial pour RNS et PQC.

4. Résultats d'implantation

4.1. Comparaison des différents algorithmes de réduction

Nous commençons par comparer les différents algorithmes de réduction modulaire (le `%` de Vivado et nos implantations de Barrett, Montgomery, MSC et MSR) pour différentes

tailles w de modulo dans le cas du motif M2 en version RSF avec $N = 20$. La figure A.2 présente les résultats pour le meilleur temps de calcul obtenu pour les différentes combinaisons de directives d'optimisations testées. Pour chaque métrique (temps, surface et compromis surface $\times$ temps), les résultats sont normalisés par rapport à la plus grande valeur pour la métrique.

Nous observons que l'algorithme natif utilisé par l'outil HLS lors de l'appel de l'opérateur `%` est bien moins performant que nos implantations des algorithmes de la littérature. Il n'utilise pas de bloc DSP pour la réduction (uniquement pour l'accumulation des produits $\sum_{i=1}^{N} x_i \times y_i$), mais il nécessite de nombreux cycles de calcul. Il semble utiliser une boucle avec un nombre d'itérations dépendant de la taille de l'opérande dans nos expérimentations. Il présente donc peu d'intérêt pour nos applications cryptographiques car nos implantations sont toutes bien meilleures.

Les meilleurs résultats obtenus sont clairement pour MSR et MSC. Ceci est parfaitement logique puisque les algorithmes pour les moduli spécifiques utilisent des propriétés particulières permettant de simplifier les calculs (ce qui n'est pas possible pour des MQ). Ainsi, pour des applications RNS, l'algorithme MSR est plus performant que Barrett et Montgomery. Pour PQC, la même conclusion s'impose pour MSC.

Le nombre de cycles d'horloge de nos 4 implantations (Barrett, Montgomery, MSC et MSR) sont proches (et souvent autour de 50% de moins que pour `%`). Les fréquences obtenues sont proches de la contrainte imposée à l'outil.

Des résultats similaires sont obtenus pour les autres tailles N de vecteurs testées. Notre bibliothèque permet à l'utilisateur d'utiliser le meilleur algorithme de réduction modulaire en fonction du type de modulo utilisé dans son application. Dans le cas d'applications avec des moduli spécifiques, l'utilisation des algorithmes MSC et MSR offre de bien meilleurs résultats. Dans le cas d'applications avec des moduli quelconques, notre bibliothèque confirme que Montgomery est un peu meilleur que Barrett.

Dans la suite, nous ne donnerons plus les résultats pour l'algorithme de réduction natif avec l'opérateur `%` car il est bien trop lent pour nos applications cryptographiques.

4.2. Impact des directives d'optimisation sur la boucle d'accumulation

Nous avons testé plusieurs directives d'optimisation sur la boucle d'accumulation: `pipeline` correspond au cas où les N itérations sont pipelinées ; `unrollk` au cas où (N/k) itérations sont effectuées sur k opérateurs en parallèle.

La table A.1 présente les résultats d'implantation du motif M2 RSF avec $N = 20$ pour

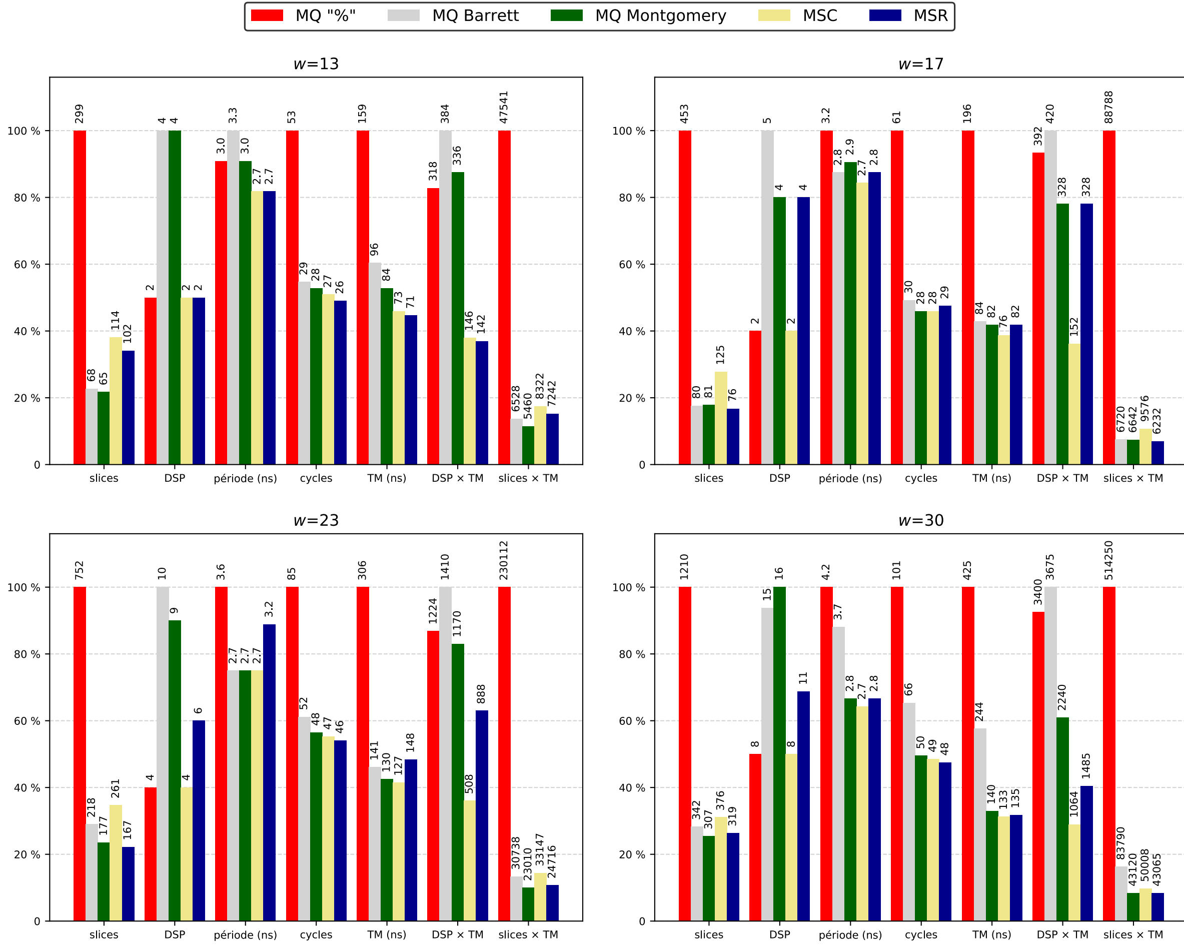


Figure A.2 – Comparaison des différents algorithmes de réduction pour $w \in \{13, 17, 23, 30\}$ bits pour le motif M2 RSF avec $N = 20$.

la réduction spécifique de moduli MSR de taille $w = 23$ bits. Nous comparons différentes directives de l'outil de synthèse HLS sur la boucle d'accumulation. La première ligne du tableau présente comme référence les résultats sans aucune directive. Chaque produit de la boucle d'accumulation requiert 2 blocs DSP car ils ont 2 opérandes de $w = 23$ bits, ce qui est plus grand que la taille du multiplieur disponible dans un bloc DSP du FPGA cible (voir [Xil18b, Xil18a]). La réduction finale requiert également 2 blocs DSP.

On remarque que les différentes implantations atteignent quasiment la période cible. En pipelinant, on arrive à diviser par 3 le temps de calcul TM sans changer le nombre de DSP utilisés. On peut encore réduire de 28 % et 40 % TM en déroulant avec un facteur 2 et 4 respectivement, au prix d'une augmentation du nombre de DSP utilisés. Au-delà d'un facteur 4, on n'observe plus d'amélioration significative du temps, tout en payant le

prix d'une augmentation du nombre de multiplieurs DSP utilisés. Ceci est très certainement dû au nombre croissant d'accès mémoires simultanés nécessaires à l'exploitation du parallélisme.

Enfin, avec une métrique de coût global de type surface×temps, nous observons que les directives `pipeline` et `pipeline` + `unroll2` se démarquent assez nettement des autres. Un niveau de parallélisme modéré semble donc facilement exploitable dans le contexte de notre bibliothèque. Pour pouvoir exploiter plus de parallélisme interne, il nous faudrait probablement changer l'algorithme et la structure du code.

directives	surface		temps (ns, cycles)			surface × temps	
	slices	DSP	période	cycles	TM	DSP × TM	slices × TM
aucune	136	4	3.1	216	670	2680	91120
`pipeline`	142	4	3.2	64	205	820	29110
`pipeline` + `unroll2`	167	6	3.2	46	148	888	24716
`pipeline` + `unroll4`	228	10	3.3	37	123	1230	28044
`pipeline` + `unroll10`	526	22	3.1	39	121	2662	63646

Table A.1 – Impact des directives d'optimisation pour M2 RSF, MSR, $w = 23$ et $N = 20$.

4.3. Impact de la stratégie de réduction

Pour un motif nécessitant de nombreuses opérations modulo m, plusieurs *stratégies* de réduction sont envisageables. Il est possible d'effectuer une réduction intermédiaire systématique (RIS) à chaque itération ; pour M2 cela correspond au calcul :

$$\left(\sum_{i=1}^{N} (x_i \times y_i \bmod m)\right) \bmod m.$$

Il est possible d'effectuer une réduction seulement à la fin (RSF) ; pour M2 cela donne :

$$\left(\sum_{i=1}^{N} x_i \times y_i\right) \bmod m.$$

Dans la version RSF de motifs comme M1 et M2, l'accumulateur doit être plus large pour absorber les retenues de la somme des résultats de chaque itération (avec $\lceil \log_2 N \rceil$ bits en plus). Ceci engendre une réduction finale plus coûteuse car son opérande est plus large.

Nous avons implanté les stratégies RIS et RSF pour les motifs M1 et M2 et nos différents algorithmes de réduction. La table A.2 présente une partie représentative de nos résultats.

motif	algorithme et stratégie	surface		temps (ns, cycles)			surface×temps	
		slices	DSP	période	cycles	TM	DSP×TM	slices×TM
M1	Montgomery RSF	122	5	2.6	31	81	405	9882
	Barrett RSF	110	1	2.9	58	169	169	18502
	MSC RSF	62	0	2.5	17	43	0	2635
	MSR RSF	66	0	2.6	17	45	0	2970
M2	Montgomery RIS	194	12	2.6	60	156	1872	30264
	Montgomery RSF	149	7	2.6	64	167	1165	24794
	Barrett RIS	259	12	2.8	53	149	1781	38436
	Barrett RSF	218	10	2.7	52	141	1404	30608
	MSC RIS	403	4	2.7	55	149	594	59846
	MSC RSF	261	4	2.7	47	127	508	33121
	MSR RIS	146	8	2.6	31	81	645	11768
	MSR RSF	167	6	3.2	46	148	884	24583

Table A.2 – Impact des stratégies de réduction pour $w = 23$ et $N = 20$.

Pour M1, nous donnons uniquement les résultats pour RSF car RIS est toujours bien moins performant (le coût d'une itération de boucle, une addition, est bien trop faible devant celui d'une réduction modulaire). Nous présentons les résultats de M1 RSF pour montrer l'impact de l'itération (par rapport à M2).

Pour M2 avec Barrett et Montgomery, RSF est meilleur en compromis surface×temps (RIS peut être un tout petit plus rapide mais pour une surface plus importante).

Globalement RSF est souvent plus efficace que RIS (pour les N et w testés). Mais il nous reste à explorer ce qui se passe pour des N très grands et des w petits (comme c'est le cas pour RE avec p. ex. $w = 13$ et $N \in [256, 1024]$).

D'autres stratégies intermédiaires sont envisageables, comme réduire de temps en temps de petits accumulateurs partiels. Cela pourrait être intéressant pour des motifs où les calculs dans chaque itération sont plus complexes qu'une seule opération.

4.4. Impact de la taille N des vecteurs

Enfin, nous analysons l'impact de la taille des vecteurs des motifs sur les performances et les coûts. La figure A.3 présente quelques résultats représentatifs pour l'algorithme de réduction MSR avec $w = 23$. Nous observons que pour les N testés, le compromis surface×temps est proche d'une fonction affine de N (résultat général à toutes nos implantations). Cette croissance affine avec N permet d'effectuer des estimations à haut niveau très simplement (avec une marge d'erreur raisonnable).

Cependant, la figure A.3 suggère aussi un phénomène bien plus complexe à anticiper (du moins dans l'état actuel de nos connaissances). Ceci est en lien avec les deux dernières lignes de la table A.2 où le motif RIS est plus efficace que RSF. On observe sur la figure A.3 que le temps de calcul et le nombre de cycles du RIS croissent moins vite que ceux de RSF. Le même phénomène est observé pour les compromis surface×temps, on peut donc en déduire que le gain de RIS sur RSF se fait sur le coût de chaque itération de la boucle d'accumulation. Cependant, d'un point de vue purement arithmétique, RIS calcule $s \leftarrow (s + x_i \times y_i) \bmod m$ à chaque itération, contre $s \leftarrow (s + x_i \times y_i)$ pour RSF, donc RIS effectue plus de calculs par itération. Nous voyons que l'outil arrive à mieux pipeliner les itérations de RIS que celles de RSF dans certains cas. Il est clair qu'il nous reste du travail pour mieux cerner quand et comment (modification de la structure du code) utiliser les différentes directives d'optimisation de l'outil HLS.

5. Conclusion

Notre bibliothèque offre un support, assez simple d'utilisation, d'algorithmes de réduction modulaire avancés qui ne sont pas supportés nativement par les outils de HLS actuels. Elle offre aussi la possibilité de générer des circuits optimisés en temps et en surface pour chaque type de modulo (de forme quelconque ou spécifique). Ceci est particulièrement intéressant pour des applications en cryptographie asymétrique comme ECC en RNS ou PQC.

Nous allons continuer le développement de notre bibliothèque et ajouter d'autres opérations, formes de moduli et motifs de calcul. Nous souhaitons aussi essayer d'autres outils de HLS et fabricants/familles de FPGA.

Remerciements

Ce travail a été financé en partie par le PEC, la DGA et la Région Bretagne. Nous remercions chaleureusement les relecteurs anonymes pour leurs précieuses remarques et corrections.

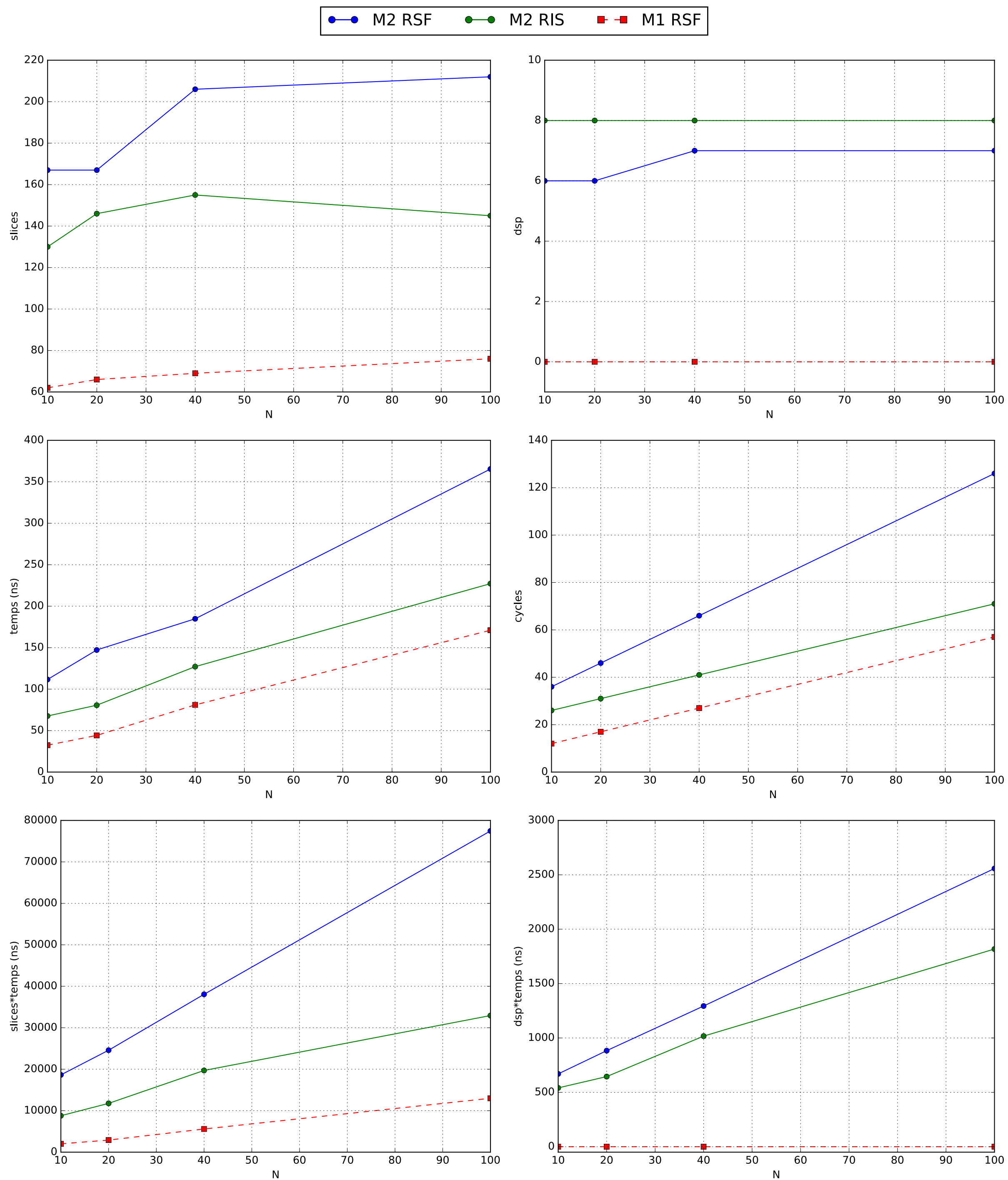


Figure A.3 – Impact de la taille N des vecteurs pour MSR avec $w = 23$.

Appendix B

Generalized Weierstrass Equation of Elliptic Curves

Definition 7 (from [CF06]). An elliptic curve E_K defined over a field K is given by the Weierstraß equation

$$E_K: \quad y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{B.1}$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ and the values

$$b_2 = a_1^2 + 4a_2, \quad b_4 = a_1a_3 + 2a_4$$
$$b_6 = a_3^2 + 4a_6 \quad \text{and} \quad b_8 = a_1^2a_6 - a_1a_3a_4 + 4a_2a_6 + a_2a_3^2 - a_4^2$$

are such that $\Delta = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6 \neq 0$.

Appendix C

Proof of the Chinese Remainder Theorem

In Theorem 2 we presented the Chinese remainder theorem (CRT). The CRT is widespread and a proof of it can be found in most books about elementary number theory; see for example [NZM91, Tat05, Bur11]. We prove the CRT in three steps:

- **The system**

$$(\Sigma) \quad \begin{cases} x_1 = |x|_{m_1} \\ x_2 = |x|_{m_2} \\ \quad \vdots \\ x_n = |x|_{m_n} \end{cases} \tag{C.1}$$

  has a solution.
  For all i, $1 \leq i \leq n$, let C_i be a number such that $|C_i M_i|_{m_i} = 1$. The various C_i exist because for all i, $1 \leq i \leq n$, m_i is coprime with M_i (since m_i is coprime with all m_j, $j \neq i$ and $M_i = \prod_{j \neq i} m_j$). Let us define $y = \sum_{i=1}^{n} x_i C_i M_i$. We do have

$$\begin{aligned} &\text{for all } i,\ |x_i C_i M_i|_{m_i} = x_i \text{ and} \\ &\text{for all } j \neq i,\ |x_i C_i M_i|_{m_j} = 0 \text{ because } m_j \text{ divides } M_i. \end{aligned}$$

  Hence, for all i, $y \equiv x_i \pmod{m_i}$. The number y is a solution of (Σ).
- **The solution of (Σ) is unique modulo M.**
  Let z be another solution of (Σ). By definition of z, for all i, $|z|_{m_i} = x_i$. Therefore, for all i, $|z|_{m_i} = |y|_{m_i} = x_i$ because y is also a solution of (Σ). This equivalence implies, for all i, $|z - y|_{m_i} = 0$ which is equivalent to, for all i, m_i divides $z - y$. We deduce $\prod_{i=1}^{n} m_i$ divides $z - y$, that is, M divides $z - y$. We get $|z|_M = |y|_M$.
- **The value given to x in Theorem 2 is the unique solution of (Σ) between 0 and M.**

The number x is one of the solutions of (Σ), by definition of (Σ). Therefore, x can be written

$$\begin{aligned} x = |y|_M &= \left|\sum_{i=1}^{n} x_i C_i M_i\right|_M \text{ owing to the uniqueness of the } (\Sigma) \text{ solution modulo } M \\ &= \left|\sum_{i=1}^{n} |x_i C_i|_{m_i} M_i\right|_M \\ &= \left|\sum_{i=1}^{n} \left|x_i M_i^{-1}\right|_{m_i} M_i\right|_M \text{ by definition of } C_i. \end{aligned}$$

Résumé substantiel en français

Dans ce manuscrit de thèse, nous présentons des accélérateurs matériels pour la cryptographie asymétrique avec une utilisation flexible de ressources matérielles. La cryptographie asymétrique est implantée dans de nombreux appareils utilisés au quotidien, comme par exemple les ordinateurs personnels et les téléphones intelligents. Elle sert notamment à échanger des clés secrètes, à signer numériquement des documents, ou encore à s'authentifier par exemple, auprès d'un serveur. Les opérandes dans les cryptosystèmes actuels sont de très grands nombres : par exemple, quelques centaines de bits pour la cryptographie basée sur les courbes elliptiques (ECC) [Mil85, Kob87] et quelques milliers de bits pour RSA [RSA78]. Une arithmétique efficace et adaptée aux grands nombres est nécessaire afin d'éviter de trop faibles performances des appareils sur lesquels sont implantés ces cryptosystèmes.

Le système modulaire de représentation des nombres (RNS) [Val56, Gar59] est un système non positionnel dans lequel un grand nombre est représenté par ses restes modulo de petits nombres premiers entre eux deux à deux. L'ensemble de ces petits nombres premiers entre eux constitue une base RNS. Les opérations élémentaires habituelles, à savoir multiplication, addition et soustraction, sont effectuées entre les petits restes de façon indépendante. Autrement dit, les opérations élémentaires entre de grands nombres sont remplacées par des opérations entre de petits restes. De plus, ces opérations élémentaires peuvent être effectuées en parallèle. Il n'y a pas de propagation de retenues entre les différents restes [ST67] à cause de l'indépendance des opérations élémentaires entre les restes. Effectuer des opérations élémentaires est donc très efficace en RNS. Ces avantages du RNS ont entraîné ces deux dernières décennies un gain d'intérêt de son usage dans les implantations matérielles de la cryptographie asymétrique. En guise d'exemples, nous citons les travaux [NMSK01, SFM$^+$09, Gui10, BM14].

Le RNS présente un inconvénient majeur qui est la difficulté à effectuer des réductions modulaires (MR), des divisions et des comparaisons. Cette difficulté est due au caractère non positionnel du RNS (l'ordre de grandeur des opérandes est plus difficile à déterminer que dans une représentation positionnelle "classique"). Pourtant, les MR sont nombreuses dans les calculs des cryptosystèmes asymétriques actuels. Par exemple,

l'opération de base de RSA est l'exponentiation modulaire. Pour ECC, l'opération de base est la multiplication scalaire (ECSM), qui elle-même se calcule à partir de nombreuses opérations dans des corps finis (ce qui necessite de nombreuses MR). L'extension de base de Kawamura *et al.* (KBE) [KKSS00] est utilisée dans l'algorithme de MR de l'état de l'art proposé dans [PP95] afin de réduire le coût de ce dernier. L'extension de base (BE) devient une opération cruciale dans les implantations RNS parce que le coût de la MR provient essentiellement du coût des deux BE qui la composent.

Les canaux sont les supports matériels des opérations élémentaires entre les restes modulo les éléments d'une base RNS. Les implantations RNS de la littérature (par exemple, [NMSK01, SFM+09, Gui10, BM14]) utilisent autant de canaux que d'éléments d'une base nécessaires à la représentation des grandes opérandes. Il en résulte une utilisation d'une quantité de ressources matérielles correspondant à la taille des grandes opérandes. Cette quantité requise constitue un problème dès lors qu'elle est supérieure à celle disponible sur le circuit intégré qu'on veut utiliser. Cette situation peut se présenter lorsque le circuit intégré choisi pour l'implantation du cryptosystème est très petit avec peu de ressources matérielles ou lorsque plusieurs applications non cryptographiques coexistent avec des applications cryptographiques sur un même circuit intégré. En effet, la plupart des circuits intégrés sont principalement utilisés pour des applications non cryptographiques comme par exemple le traitement de signal ou la vision par ordinateur. En tenant compte des ressources matérielles utilisées par ces applications principales, le reste des ressources matérielles sur le circuit intégré peut être insuffisant pour implanter le cryptosystème souhaité.

Concevoir des accélérateurs matériels flexibles pour la cryptographie asymétrique permet de résoudre ce problème. Dans ce manuscrit, nous appelons *accélérateurs matériels flexibles* des accélérateurs matériels dont l'utilisation de ressources matérielles est *flexible*, c'est-à-dire qui peut être adaptée, pour une même taille des grandes opérandes. On peut donc choisir la quantité de ressources matérielles à utiliser par une implantation d'un cryptosystème asymétrique en fonction de celle disponible sur le circuit intégré. Dans les implantations RNS, de tels accélérateurs peuvent être réalisés en les concevant de façon à ce qu'ils utilisent un nombre de canaux plus petit que celui normalement nécessaire pour les calculs sur les grandes opérandes. Nous appelons *canaux physiques* (PC) les canaux effectivement utilisés, et notons q leur nombre. À chaque élément d'une base RNS, nous associons un *canal virtuel* (VC). Le nombre n de VC est alors le nombre d'éléments nécessaires pour représenter les grandes opérandes.

Contributions

La première contribution de cette thèse est un nouvel algorithme de BE nommé *hierarchical base extension* (HBE) [DBT19]. HBE utilise une approche hiérarchique dans le calcul du théorème chinois des restes (CRT). Cette approche se décrit en deux phases. Au cours de la première phase, les restes en entrée sont combinés par paires à l'aide de CRT partiels effectués dans la base d'entrée. Lors de la seconde phase, la suite du calcul du CRT est effectuée sur les résultats de ces CRT partiels dans la base de sortie. Le coût théorique de HBE est jusqu'à 35% plus petit que celui de KBE [KKSS00], largement considéré comme la BE de l'état de l'art.

De plus, nous avons proposé une architecture pour HBE [DBT19]. Cette architecture est adaptée de l'architecture *cox-rower* de KBE [KKSS00]. Notre architecture préserve le parallélisme naturel du RNS avec un *pipeline* légèrement plus profond au niveau des *rowers*. HBE et KBE ont été implantés sur FPGA à l'aide d'outils de synthèse de haut niveau (HLS). Ces implantations visent des tailles de corps typiques de celles utilisées dans les applications de ECC et sont effectuées pour plusieurs tailles de canaux. Les résultats de ces implantations montrent que HBE est dans tous les cas plus rapide que KBE (jusqu'à 20%), et dans presque tous les cas plus petit (jusqu'à 23%) en surface. HBE présente toujours le meilleur compromis surface/temps.

Deux accélérateurs matériels RNS flexibles de l'ECSM constituent la seconde contribution de cette thèse. Dans un premier temps, nous montrons que la BE peut être implantée de façon *flexible*, et cela pour la BE de l'état de l'art (KBE) [KKSS00] et la notre (HBE) [DBT19]. Les architectures flexibles sont dérivées des architectures de base des deux BE. Le nombre q de PC dans chaque architecture flexible est adaptable, et q est choisi à la conception parmi les diviseurs de n (le nombre de VC). Il en résulte que pour une taille donnée de corps fini, la BE (HBE ou KBE) peut être implantée avec une quantité *flexible* de ressources matérielles. Le concepteur peut donc choisir le nombre q de PC à utiliser pour une implantation donnée en fonction de la quantité de ressources matérielles dont il dispose sur son circuit intégré. La *flexibilité* permet d'obtenir plusieurs compromis surface/temps pour l'implantation d'une même opération. HBE et KBE flexibles ont été implantées sur FPGA à l'aide d'outils de HLS. Les résultats d'implantations montrent qu'indépendamment de la BE, lorsque le nombre de PC passe de q à kq, les nombres de DSP et BRAM croissent d'un facteur k. Par contre, le nombre de slices croît d'un facteur plus petit que k. L'augmentation de la surface utilisée induit une diminution du temps, qui décroît d'un facteur plus petit que k. En outre, les résultats de HBE flexible sont

plus rapides (jusqu'à 28%) et 33% plus petits en DSP que ceux de KBE flexible, pour un même nombre de PC utilisés. Par contre, les implantations de HBE flexible utilisent plus de slices que celles de KBE flexible (jusqu'à 35%). Toutefois, le compromis surface/temps est toujours en faveur des résultats de HBE flexible.

Dans un second temps, deux ECSM flexibles nommées *ECSM-HBE* et *ECSM-KBE flexibles* sont implantées à partir des architectures flexibles des deux BE (HBE et KBE respectivement). Comme pour les BE flexibles, les nombres de DSP et BRAM croissent d'un facteur k lorsque le nombre de PC utilisés passe de q à kq. Aussi, les facteurs de croissance du nombre de slices et de décroissance du temps sont plus petits que k. Les résultats de comparaison des deux ECSM flexibles sont similaires à ceux des deux BE flexibles. Pour un même nombre de PC utilisés, les résultats de ECSM-HBE flexible sont toujours plus rapides (jusqu'à 31%) et 33% plus petits en DSP que ceux de ECSM-KBE flexible. Les résultats de ECSM-HBE flexible sont légèrement plus gros en slices (jusqu'à 23%) que ceux de ECSM-KBE flexible. Comme pour les BE flexibles, les résultats de ECSM-HBE flexible présentent les meilleurs compromis surface/temps. Quoiqu'en compromis surface/temps, nos meilleurs résultats d'implantations sont comparables aux meilleurs résultats de la littérature, la plupart de nos implantations présentent des nombres de DSP et de BRAM bien plus petits. La possibilité d'implanter des ECSM sur des corps de base d'une même taille en utilisant une surface plus petite et adaptable constitue l'apport majeur de la *flexibilité*.

La troisième contribution de cette thèse est une étude auxiliaire au projet de thèse et rapportée dans [DZBT19]. Le but est d'étudier les meilleures façons d'effectuer des multiplications modulaires et accumulations sur FPGA à partir d'outils de HLS. Le sujet d'étude est motivé par la présence de nombreuses multiplications modulaires et accumulations dans les implantations RNS des cryptosystèmes asymétriques. Plusieurs paramètres tels que la taille et la forme (générique, spécifique) des moduli ainsi que la stratégie de réduction (réduction intermédiaire ou réduction seulement à la fin) sont étudiés. Nous avons également étudié plusieurs contraintes d'implantation tels que le *pipelining* et l'*unrolling* de boucles en fonction des paramètres.

Bibliography

[AR14] H. Alrimeih and D. Rakhmatov. Fast and flexible hardware support for ECC over multiple standard prime fields. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22:2661–2674, Dec. 2014.

[Bar84] P. Barrett. *Communications Authentication and Security Using Public Key Encryption: A Design for Implementation.* PhD thesis, University of Oxford, 1984.

[Bar86] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology (CRYPTO)*, volume 263 of *LNCS*, pages 311–323. Springer, 1986.

[BBMO04] L. Batina, G. Bruin-Muurling, and S. B. Örs. Flexible hardware design for RSA and elliptic curve cryptosystems. In *Topics in Cryptology (CT-RSA)*, volume 2964 of *LNCS*, pages 250–263. Springer, 2004.

[BDE13] J. C. Bajard, S. Duquesne, and M. Ercegovac. Combining leak–resistant arithmetic for elliptic curves defined over $\mathbb{F}_p$ and RNS representation. *Publications mathématiques de Besançon*, pages 67–87, 2013.

[BDK01] J.-C. Bajard, L.-S. Didier, and P. Kornerup. Modular multiplication and base extensions in residue number systems. In *IEEE 15th Symposium on Computer Arithmetic (ARITH)*, pages 59–65. IEEE, 2001.

[BEMP15] J.-C. Bajard, J. Eynard, N. Merkiche, and T. Plantard. RNS arithmetic approach in lattice-based cryptography. In *IEEE 22nd Symposium on Computer Arithmetic (ARITH)*, pages 113–120. IEEE, 2015.

[Big14] K. Bigou. *Étude théorique et implantation matérielle d'unités de calcul en représentation modulaire des nombres pour la cryptographie sur courbes elliptiques.* PhD thesis, University Rennes 1, Lannion, France, November 2014.

[BJ02] É. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In *Public Key Cryptography*, volume 2274, pages 335–345. Springer, 2002.

[BKP09] J.-C. Bajard, M. Kaihara, and T. Plantard. Selected RNS bases for modular multiplication. In *IEEE 19th Symposium on Computer Arithmetic (ARITH)*, pages 25–32. IEEE, 2009.

[BLS03] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *International symposium on Symbolic and algebraic computation (ISSAC)*, pages 37–44. ACM, Aug. 2003.

[BM14] J.-C. Bajard and N. Merkiche. Double level Montgomery cox-rower architecture, new bounds. In *Proc. 13th Smart Card Research and Advanced Application Conference (CARDIS)*, volume 8968 of *LNCS*, pages 139–153. Springer, 2014.

[BMPV06] L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede. Flexible hardware architectures for curve-based cryptography. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 4839–4842. IEEE, 2006.

[BT15] K. Bigou and A. Tisserand. Single base modular multiplication for efficient hardware rns implementations of ecc. In *Proc. 17th Cryptographic Hardware and Embedded Systems (CHES)*, volume 9293 of *LNCS*, pages 123–140. Springer, Sep. 2015.

[BT16] K. Bigou and A. Tisserand. Hybrid position-residue number system. In *IEEE 23rd Symposium on Computer Arithmetic (ARITH)*, pages 126–133. IEEE, Jul. 2016.

[Bur11] D. M. Burton. *Elementary Number Theory*. The McGraw-Hill Companies, Inc, 7th edition, 2011.

[CDF+11] R. C.C. Cheung, S. Duquesne, J. Fan, N. Guillermin, I. Verbauwhede, and G. X. Yao. FPGA implementation of pairings using residue number system and lazy reduction. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 6917 of *LNCS*, pages 421–441. Springer, 2011.

[CF06] H. Cohen and G. Frey. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2006.

[CNPQ03] M. Ciet, M. Neve, E. Peeters, and J.-J. Quisquater. Parallel fpga implementation of RSA with residue number systems. In *46th Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 806–810. IEEE, Dec. 2003.

[Cra92] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system. US Patent 5159632 A, October 1992.

[DBT19] L. Djath, K. Bigou, and A. Tisserand. Hierarchical approach in RNS base extension for asymmetric cryptography. In *IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 46–53. IEEE, 2019.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, Nov. 1976.

[DZBT19] L. Djath, T. Zijlstra, K. Bigou, and A. Tisserand. Comparaison d'algorithmes de réduction modulaire en HLS sur FPGA. In *Conférence d'informatique en Parallélisme, Architecture et Système - Compas'19*, June 2019.

[EL94] M. D. Ercegovac and T. Lang. *Division and Square-Root Algorithms: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic, 1994.

[EL03] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[Elg85] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, Jul. 1985.

[ESJ+13] M. Esmaeildoust, D. Schinianakis, H. Javashi, T. Stouraitis, and Keivan Navi. Efficient rns implementation of elliptic curve point multiplication over $gf(p)$. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21:1545–1549, Aug. 2013.

[FKSK15] A. P. Fournaris, N. Klaoudatos, N. Sklavos, and C. Koulamas. Fault and power analysis attack resistant RNS based edwards curve point multiplication. In *Proc. Second Workshop on Cryptography and Security in Computing Systems (CS2)*, pages 43–46, Jan. 2015.

[FR94] G. Frey and H.-G. Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Mathematics of Computation*, 62:865–874, Apr. 1994.

[Gar59] H. L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EC-8(2):140–147, June 1959.

[GHS02a] S. D. Galbraith, F. Hess, and N. P. Smart. Extending the GHS weil descent attack. In *Proc. Internat. Conf. Theory and Application of Cryptographic Techniques (EUROCRYPT)*, volume 2332 of *LNCS*, pages 29–44. Springer, 2002.

[GHS02b] P. Gaudry, F. Hess, and N. Smart. Constructive and destructive facets of weil descent on elliptic curves. *Journal of Cryptology*, 15:19–46, 2002.

[GLP+12] F. Gandino, F. Lamberti, G. Paravati, J.-C. Bajard, and P. Montuschi. An algorithmic and architectural study on montgomery exponentiation in rns. *IEEE Transactions on Computers*, 61:1071–1083, Aug. 2012.

[GP08] T. Güneysu and C. Paar. Ultra high performance ecc over nist primes on commercial FPGAs. In *Proc. 10th Cryptographic Hardware and Embedded Systems (CHES)*, volume 5154 of *LNCS*, pages 62–78. Springer, 2008.

[Gui10] N. Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In *Proc. 12th Cryptographic Hardware and Embedded Systems (CHES)*, volume 6225 of *LNCS*, pages 48–64. Springer, 2010.

[Hes03] F. Hess. The GHS attack revisited. In *Proc. Internat. Conf. Theory and Application of Cryptographic Techniques (EUROCRYPT)*, volume 2656 of *LNCS*, pages 374–387. Springer, 2003.

[HMV04] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2004.

[HRdH+18] H. D.L. Hollmann, R. Rietman, S. de Hoogh, L. Tolhuizen, and P. Gorissen. A multi-layer recursive residue number system. In *IEEE International Symposium on Information Theory (ISIT)*, pages 1460–1464. IEEE, Jun. 2018.

[Hus04] D. Husemöller. *Elliptic Curves.* Springer, 2nd edition, 2004.

[IT02] T. Izu and T. Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In *Public Key Cryptography*, volume 2274, pages 280–296. Springer, 2002.

[KKSS00] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-Rower architecture for fast parallel Montgomery multiplication. In *Proc. Internat. Conf. Theory and Application of Cryptographic Techniques (EUROCRYPT)*, volume 1807 of *LNCS*, pages 523–538. Springer, May 2000.

[KKSY18] S. Kawamura, Y. Komano, H. Shimizu, and T. Yonemura. RNS montgomery reduction algorithms using quadratic residuosity. *Journal of Cryptographic Engineering*, page 313–331, 2018.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, Jan. 1987.

[Men01] A. Menezes. Evaluation of security level of cryptography: The elliptic curve discrete logarithm problem (ECDLP). Technical report, Dec. 2001.

[Mil85] V .S. Miller. Use of elliptic curve in cryptography. In *Advances in Cryptology (CRYPTO)*, volume 218, pages 417–426. Springer, 1985.

[MLPJ13] Y. Ma, Z. Liu, W. Pan, and J. Jing. A high-speed elliptic curve cryptographic processor for generic curves over GF(p). In *International Conference on Selected Areas in Cryptography (SAC)*, volume 8282 of *LNCS*, pages 421–437. Springer, 2013.

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

[Mon87] P. L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, January 1987.

[MOV93] A. J. Menezes, T. Okamoto, and S. A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. In *IEEE Transactions on Information Theory*, volume 39, pages 1639–1646. IEEE, Sep. 1993.

[MTW04] A. Menezes, E. Teske, and A. Weng. Weak fields for ECC. In *Topics in Cryptology (CT-RSA)*, volume 2964 of *LNCS*, pages 366–386. Springer, 2004.

[Mö01] B. Möller. Securing elliptic curve point multiplication against side-channel attacks. In *International Conference on Information Security*, volume 2200 of *LNCS*, pages 324–334. Springer, 2001.

[ndlsdsd11] Agence nationale de la sécurité des systèmes d'information. Avis relatif aux paramètres de courbes elliptiques définis par l'Etat français. Technical report, Oct. 2011.

[NMSK01] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura. Implementation of RSA algorithm based on RNS montgomery multiplication. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *LNCS*, pages 364–376. Springer, 2001.

[NZM91] I. Niven, H. S. Zuckerman, and H. L. Montgomery. *An Introduction to the Theory of Numbers*. John Wiley & Sons, Inc, 5th edition, 1991.

[OP01] G. Orlando and C. Paar. A scalable GF(p) elliptic curve processor architecture for programmable hardware. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *LNCS*, pages 348–363. Springer, 2001.

[oST09] National Institute of Standards and Technology. Digital signature standard (DSS). Technical report, June 2009.

[oST13] National Institute of Standards and Technology. Digital signature standard (DSS). Technical report, Jul. 2013.

[PH78] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. In *IEEE Transactions on Information Theory*, volume 24, pages 106–110. IEEE, Jan. 1978.

[PITM13] G. Perin, L. Imbert, L. Torres, and P. Maurine. Electromagnetic analysis on rsa algorithm based on rns. In *Euromicro Conference on Digital System Design (DSD)*, pages 345–352. IEEE, Sep. 2013.

[Pol78] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32:918–924, Jul. 1978.

[PP93] K. C. Posch and R. Posch. Base extension using a convolution sum in residue number systems. *Computing*, 50(2):93–104, June 1993.

[PP95] K. C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6:449–454, May 1995.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, Feb. 1978.

[SA98] T. Satoh and K. Araki. Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves. *Commentarii Mathematici Universitatis Sancti Pauli*, 47:81–92, 1998.

[SA99] A. Skavantzos and M. Abdallah. Implementation issues of the two-level residue number system with pairs of conjugate moduli. *IEEE Transactions on Signal Processing*, 47:826–838, Mar. 1999.

[Sem98] I. A. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p. *Mathematics of Computation*, 67:353–356, Jan. 1998.

[SFM+09] D. M. Schinianakis, A. P. Fournaris, H. E. Michail, A. P. Kakarountas, and T. Stouraitis. An RNS implementation of an $\mathbb{F}_p$ elliptic curve point multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(6):1202–1213, June 2009.

[Sil09] J. H. Silverman. *The Arithmetic of Elliptic Curves.* Springer, 2nd edition, 2009.

[SK89] A. P. Shenoy and R. Kumaresan. Fast base extension using a redundant modulus in RNS. *IEEE Transactions on Computers*, 38(2):292–297, February 1989.

[SKS06] D.M. Schinianakis, A.P. Kakarountas, and T. Stouraitis. A new approach to elliptic curve cryptography: an RNS architecture. In *IEEE Mediterranean Electrotechnical Conference (MELECON)*, pages 1241–1245. IEEE, May 2006.

[Sol99] J. A. Solinas. Generalized Mersenne numbers. Technical Repport CORR-99-39, University of Waterloo, Centre for Applied Cryptographic Research, 1999.

[SS13] D. Schinianakis and T. Stouraitis. An RNS modular multiplication algorithm. In *IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 958–961. IEEE, 2013.

[SS14] D. Schinianakis and T. Stouraitis. An RNS barrett modular multiplication architecture. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2229–2232. IEEE, 2014.

[ST67] N. S. Szabo and R. I. Tanaka. *Residue arithmetic and its applications to computer technology.* McGraw-Hill, 1967.

[ST12] O. Sentieys and A. Tisserand. *Architectures reconfigurables FPGA. Technologies logicielles Architectures des systèmes, H 1 196.* Techniques de l'ingénieur, 2012.

[Tat05] J. J. Tattersall. *Elementary number theory in nine chapters.* Cambridge University Press, 2nd edition, 2005.

[Tom11] T. Tomczak. Hierarchical residue number systems with small moduli and simple converters. *International Journal of Applied Mathematics and Computer Science*, 21:173–192, 2011.

[Val56] M. Valach. Převod čísel ze soustavy zbytkových tříd do polyadické soustavy změnou měřitka periody (the translation of numbers from the system of residual classes to a polyadic system by change of scale of period). In *Stroje na Zpracování Informací, Sborník IV, Nakl. ČSAV, Praha (in Czech)*, 1956.

[vdH17] J. van der Hoeven. Fast Chinese remaindering in practice. In *International Conference on Mathematical Aspects of Computer and Information Sciences (MACIS)*, pages 95–106. Springer, Nov. 2017.

[Xil17] Xilinx. Vivado HLS optimization methodology guide (UG1270). Technical report, December 2017.

[Xil18a] Xilinx. 7 series DSP48E1 slice user guide (UG479). Technical report, March 2018.

[Xil18b] Xilinx. Vivado design suite user guide high-level synthesis (UG902). Technical report, February 2018.

[Xil19a] Xilinx. 7 series FPGAs memory resources user guide (UG473). Technical report, July 2019.

[Xil19b] Xilinx. Vivado design suite user guide high-level synthesis (UG902). Technical report, Jul. 2019.

[Yas92] H.M. Yassine. Hierarchical residue numbering system suitable for VLSI arithmetic architectures. In *IEEE International Symposium on Circuits and Systems*, pages 811–814. IEEE, May 1992.

[YFCV12] G. X. Yao, J. Fan, R. C.C. Cheung, and I. Verbauwhede. Faster pairing coprocessor architecture. In *Pairing-Based Cryptography (Pairing)*, volume 7708 of *LNCS*, pages 160–176. Springer, 2012.

DOCTORAT BRETAGNE LOIRE MATHSTIC

Université de Bretagne Occidentale

Titre : Accélérateurs matériels RNS flexibles pour la cryptographie asymétrique à haute sécurité

Mot clés : arithmétique des ordinateurs, système modulaire de représentation des nombres, flexibilité, implantation matérielle sur FPGA.

Résumé : Les implantations RNS de cryptosystèmes asymétriques actuels utilisent des ressources matérielles correspondant à la taille des opérandes traitées. Dans cette thèse, nous proposons une nouvelle approche dans l'implantation RNS de cryptosystèmes asymétriques qui permet une utilisation flexible de ressources matérielles. Dans un premier temps, un nouvel algorithme d'extension de base est présenté. Les extensions de bases sont, de part leurs coûts, des opérations critiques dans les implantations RNS. Notre nouvel algorithme d'extension de base utilise une approche hiérarchique dans le calcul du théorème chinois des restes. Comparé à l'algorithme d'extension de base de l'état de l'art, il présente un coût théorique réduit, qui se traduit par un gain en surface et en temps dans nos implantations HLS sur FPGA. Ensuite, nous implantons les deux algorithmes d'extension de base à partir de la nouvelle approche d'implantation RNS. Enfin, des multiplications scalaires utilisant chacune des deux extensions de base sont implantées avec la nouvelle approche. Nos implantations HLS sur FPGA utilisent des ressources matérielles en quantité flexible. De plus, quoique comparables en compromis surface/temps à ceux de l'état de l'art, la plupart de nos résultats sont bien plus petits.

Title: RNS-Flexible Hardware Accelerators for High-Security Asymmetric Cryptography

Keywords: computer arithmetic, residue number system, flexibility, FPGA hardware implementation.

Abstract: Asymmetric cryptosystems are implemented in RNS using a quantity of hardware resources corresponding to the size of the cryptographic operands. In this thesis we propose a new approach to perform RNS implementations of asymmetric cryptosystems that leads to a flexible utilization of hardware resources. We start with describing a new method to perform base extensions which are crucial operations in RNS implementations of asymmetric cryptosystems. The proposed base-extension method, based on a hierarchical approach for computing the Chinese remainder theorem, introduces a reduction of the theoretical cost. Our FPGA implementations using HLS show an area and time gain compared with the state-of-the-art method. Then, we demonstrate the practicality of our new RNS-implementation approach on the two base-extension methods. Last, elliptic curve scalar multiplications based on the two base-extension methods are implemented using our RNS-implementation approach. Our FPGA implementations use a flexible quantity of hardware resources. Besides, although comparable with state-of-the-art ones in area *vs.* time trade-offs, most of our solutions are much smaller.